SQLAlchemy Documentation

Release 1.1.0b1

Mike Bayer

January 12, 2016

Contents

1 Overview **3**
1.1 Documentation Overview . . . 4
1.2 Code Examples . . . 4
1.3 Installation Guide . . . 4
1.3.1 Supported Platforms . . . 4
1.3.2 Supported Installation Methods . . . 4
1.3.3 Install via pip . . . 5
1.3.4 Installing using setup.py . . . 5
1.3.5 Installing the C Extensions . . . 5
1.3.6 Installing on Python 3 . . . 5
1.3.7 Installing a Database API . . . 5
1.3.8 Checking the Installed SQLAlchemy Version . . . 6
1.4 1.0 to 1.1 Migration . . . 6

2 SQLAlchemy ORM **7**
2.1 Object Relational Tutorial . . . 7
2.1.1 Version Check . . . 7
2.1.2 Connecting . . . 8
2.1.3 Declare a Mapping . . . 8
2.1.4 Create a Schema . . . 9
2.1.5 Create an Instance of the Mapped Class . . . 10
2.1.6 Creating a Session . . . 11
2.1.7 Adding New Objects . . . 12
2.1.8 Rolling Back . . . 14
2.1.9 Querying . . . 15
Common Filter Operators . . . 17
Returning Lists and Scalars . . . 18
Using Textual SQL . . . 19
Counting . . . 21
2.1.10 Building a Relationship . . . 22
2.1.11 Working with Related Objects . . . 23
2.1.12 Querying with Joins . . . 24
Using Aliases . . . 26
Using Subqueries . . . 26
Selecting Entities from Subqueries . . . 27
Using EXISTS . . . 28
Common Relationship Operators . . . 29
2.1.13 Eager Loading . . . 29

Subquery Load . . . 29
Joined Load . . . 30
Explicit Join + Eagerload . . . 31
2.1.14 Deleting . . . 32
Configuring delete/delete-orphan Cascade . . . 32
2.1.15 Building a Many To Many Relationship . . . 34
2.1.16 Further Reference . . . 38
2.2 Mapper Configuration . . . 38
2.2.1 Types of Mappings . . . 38
Declarative Mapping . . . 38
Classical Mappings . . . 39
Runtime Introspection of Mappings, Objects . . . 40
2.2.2 Mapping Columns and Expressions . . . 41
Mapping Table Columns . . . 41
SQL Expressions as Mapped Attributes . . . 45
Changing Attribute Behavior . . . 48
Composite Column Types . . . 54
2.2.3 Mapping Class Inheritance Hierarchies . . . 57
Joined Table Inheritance . . . 57
Single Table Inheritance . . . 65
Concrete Table Inheritance . . . 66
Using Relationships with Inheritance . . . 67
Using Inheritance with Declarative . . . 69
2.2.4 Non-Traditional Mappings . . . 69
Mapping a Class against Multiple Tables . . . 69
Mapping a Class against Arbitrary Selects . . . 70
Multiple Mappers for One Class . . . 71
2.2.5 Configuring a Version Counter . . . 71
Simple Version Counting . . . 72
Custom Version Counters / Types . . . 73
Server Side Version Counters . . . 73
Programmatic or Conditional Version Counters . . . 74
2.2.6 Class Mapping API . . . 75
2.3 Relationship Configuration . . . 89
2.3.1 Basic Relationship Patterns . . . 89
One To Many . . . 90
Many To One . . . 90
One To One . . . 91
Many To Many . . . 92
Association Object . . . 94
2.3.2 Adjacency List Relationships . . . 95
Composite Adjacency Lists . . . 96
Self-Referential Query Strategies . . . 97
Configuring Self-Referential Eager Loading . . . 98
2.3.3 Linking Relationships with Backref . . . 99
Backref Arguments . . . 100
One Way Backrefs . . . 101
2.3.4 Configuring how Relationship Joins . . . 103
Handling Multiple Join Paths . . . 103
Specifying Alternate Join Conditions . . . 104
Creating Custom Foreign Conditions . . . 105
Using custom operators in join conditions . . . 106
Overlapping Foreign Keys . . . 107
Non-relational Comparisons / Materialized Path . . . 109

Self-Referential Many-to-Many Relationship . . . 109
Composite "Secondary" Joins . . . 111
Relationship to Non Primary Mapper . . . 112
Building Query-Enabled Properties . . . 113
2.3.5 Collection Configuration and Techniques . . . 113
Working with Large Collections . . . 113
Customizing Collection Access . . . 115
Custom Collection Implementations . . . 118
Collection Internals . . . 125
2.3.6 Special Relationship Persistence Patterns . . . 127
Rows that point to themselves / Mutually Dependent Rows . . . 127
Mutable Primary Keys / Update Cascades . . . 129
2.3.7 Relationships API . . . 131
2.4 Loading Objects . . . 140
2.4.1 Loading Columns . . . 140
Deferred Column Loading . . . 140
Column Bundles . . . 144
2.4.2 Relationship Loading Techniques . . . 145
Using Loader Strategies: Lazy Loading, Eager Loading . . . 145
The Importance of Ordering . . . 147
Loading Along Paths . . . 147
Default Loading Strategies . . . 147
Per-Entity Default Loading Strategies . . . 148
The Zen of Eager Loading . . . 148
What Kind of Loading to Use ? . . . 150
Routing Explicit Joins/Statements into Eagerly Loaded Collections . . . 151
Creating Custom Load Rules . . . 153
Relationship Loader API . . . 154
2.4.3 Constructors and Object Initialization . . . 158
2.4.4 Query API . . . 158
The Query Object . . . 158
ORM-Specific Query Constructs . . . 182
2.5 Using the Session . . . 189
2.5.1 Session Basics . . . 189
What does the Session do ? . . . 189
Getting a Session . . . 189
Session Frequently Asked Questions . . . 191
Basics of Using a Session . . . 195
2.5.2 State Management . . . 199
Quickie Intro to Object States . . . 199
Session Attributes . . . 199
Session Referencing Behavior . . . 200
Merging . . . 201
Expunging . . . 204
Refreshing / Expiring . . . 204
2.5.3 Cascades . . . 207
save-update . . . 208
delete . . . 209
delete-orphan . . . 212
merge . . . 212
refresh-expire . . . 212
expunge . . . 212
Controlling Cascade on Backrefs . . . 212
2.5.4 Transactions and Connection Management . . . 213

Managing Transactions . . . 213
Joining a Session into an External Transaction (such as for test suites) . . . 219
2.5.5 Additional Persistence Techniques . . . 220
Embedding SQL Insert/Update Expressions into a Flush . . . 220
Using SQL Expressions with Sessions . . . 221
Forcing NULL on a column with a default . . . 221
Partitioning Strategies . . . 223
Bulk Operations . . . 224
2.5.6 Contextual/Thread-local Sessions . . . 226
Implicit Method Access . . . 227
Thread-Local Scope . . . 227
Using Thread-Local Scope with Web Applications . . . 228
Using Custom Created Scopes . . . 229
Contextual Session API . . . 229
2.5.7 Tracking Object and Session Changes with Events . . . 231
Persistence Events . . . 231
Object Lifecycle Events . . . 233
Transaction Events . . . 236
Attribute Change Events . . . 237
2.5.8 Session API . . . 237
Session and sessionmaker() . . . 237
Session Utilites . . . 255
Attribute and State Management Utilities . . . 256
2.6 Events and Internals . . . 258
2.6.1 ORM Events . . . 258
Attribute Events . . . 258
Mapper Events . . . 262
Instance Events . . . 271
Session Events . . . 276
Query Events . . . 288
Instrumentation Events . . . 289
2.6.2 ORM Internals . . . 290
2.6.3 ORM Exceptions . . . 330
2.6.4 Deprecated ORM Event Interfaces . . . 331
Mapper Events . . . 332
Session Events . . . 334
Attribute Events . . . 335
2.7 ORM Extensions . . . 336
2.7.1 Association Proxy . . . 336
Simplifying Scalar Collections . . . 336
Creation of New Values . . . 338
Simplifying Association Objects . . . 338
Proxying to Dictionary Based Collections . . . 340
Composite Association Proxies . . . 341
Querying with Association Proxies . . . 343
API Documentation . . . 344
2.7.2 Automap . . . 347
Basic Use . . . 348
Generating Mappings from an Existing MetaData . . . 348
Specifying Classes Explicitly . . . 349
Overriding Naming Schemes . . . 350
Relationship Detection . . . 351
Using Automap with Explicit Declarations . . . 354
API Reference . . . 355

2.7.3 Baked Queries . . . 358
Synopsis . . . 359
Performance . . . 360
Rationale . . . 361
Lazy Loading Integration . . . 363
API Documentation . . . 364
2.7.4 Declarative . . . 365
Basic Use . . . 365
Configuring Relationships . . . 367
Table Configuration . . . 369
Inheritance Configuration . . . 372
Mixin and Custom Base Classes . . . 376
Declarative API . . . 385
2.7.5 Mutation Tracking . . . 392
Establishing Mutability on Scalar Column Values . . . 392
Establishing Mutability on Composites . . . 395
API Reference . . . 397
2.7.6 Ordering List . . . 400
API Reference . . . 402
2.7.7 Horizontal Sharding . . . 403
API Documentation . . . 404
2.7.8 Hybrid Attributes . . . 404
Defining Expression Behavior Distinct from Attribute Behavior . . . 406
Defining Setters . . . 407
Working with Relationships . . . 407
Building Custom Comparators . . . 409
Hybrid Value Objects . . . 410
Building Transformers . . . 412
API Reference . . . 414
2.7.9 Alternate Class Instrumentation . . . 415
API Reference . . . 416
2.8 ORM Examples . . . 417
2.8.1 Mapping Recipes . . . 417
Adjacency List . . . 417
Associations . . . 418
Directed Graphs . . . 418
Dynamic Relations as Dictionaries . . . 418
Generic Associations . . . 418
Large Collections . . . 419
Materialized Paths . . . 419
Nested Sets . . . 419
Performance . . . 419
Relationship Join Conditions . . . 423
XML Persistence . . . 424
Versioning Objects . . . 424
Vertical Attribute Mapping . . . 426
2.8.2 Inheritance Mapping Recipes . . . 426
Basic Inheritance Mappings . . . 426
2.8.3 Special APIs . . . 426
Attribute Instrumentation . . . 426
Horizontal Sharding . . . 427
2.8.4 Extending the ORM . . . 427
Dogpile Caching . . . 427
PostGIS Integration . . . 428

3 SQLAlchemy Core **431**
3.1 SQL Expression Language Tutorial . . . 431
3.1.1 Version Check . . . 431
3.1.2 Connecting . . . 432
3.1.3 Define and Create Tables . . . 432
3.1.4 Insert Expressions . . . 434
3.1.5 Executing . . . 434
3.1.6 Executing Multiple Statements . . . 435
3.1.7 Selecting . . . 436
3.1.8 Operators . . . 438
Operator Customization . . . 440
3.1.9 Conjunctions . . . 440
3.1.10 Using Textual SQL . . . 441
Using More Specific Text with `table()`, `literal_column()`, and `column()` . . . 443
Ordering or Grouping by a Label . . . 444
3.1.11 Using Aliases . . . 445
3.1.12 Using Joins . . . 446
3.1.13 Everything Else . . . 447
Bind Parameter Objects . . . 447
Functions . . . 448
Window Functions . . . 450
Unions and Other Set Operations . . . 450
Scalar Selects . . . 452
Correlated Subqueries . . . 452
Ordering, Grouping, Limiting, Offset...ing... . . . 454
3.1.14 Inserts, Updates and Deletes . . . 455
Correlated Updates . . . 457
Multiple Table Updates . . . 457
Parameter-Ordered Updates . . . 458
Deletes . . . 458
Matched Row Counts . . . 459
3.1.15 Further Reference . . . 459
3.2 SQL Statements and Expressions API . . . 459
3.2.1 Column Elements and Expressions . . . 459
3.2.2 Selectables, Tables, FROM objects . . . 507
3.2.3 Insert, Updates, Deletes . . . 567
3.2.4 SQL and Generic Functions . . . 598
3.2.5 Custom SQL Constructs and Compilation Extension . . . 607
Synopsis . . . 607
Dialect-specific compilation rules . . . 608
Compiling sub-elements of a custom expression construct . . . 608
Enabling Autocommit on a Construct . . . 609
Changing the default compilation of existing constructs . . . 610
Changing Compilation of Types . . . 610
Subclassing Guidelines . . . 610
Further Examples . . . 611
3.2.6 Expression Serializer Extension . . . 613
3.3 Schema Definition Language . . . 614
3.3.1 Describing Databases with MetaData . . . 615
Accessing Tables and Columns . . . 615
Creating and Dropping Database Tables . . . 616
Altering Schemas through Migrations . . . 618
Specifying the Schema Name . . . 618
Backend-Specific Options . . . 619

Column, Table, MetaData API . . . 619
3.3.2 Reflecting Database Objects . . . 644
Overriding Reflected Columns . . . 645
Reflecting Views . . . 645
Reflecting All Tables at Once . . . 646
Fine Grained Reflection with Inspector . . . 646
Limitations of Reflection . . . 651
3.3.3 Column Insert/Update Defaults . . . 651
Scalar Defaults . . . 651
Python-Executed Functions . . . 652
SQL Expressions . . . 653
Server Side Defaults . . . 654
Triggered Columns . . . 654
Defining Sequences . . . 655
Default Objects API . . . 656
3.3.4 Defining Constraints and Indexes . . . 660
Defining Foreign Keys . . . 660
UNIQUE Constraint . . . 664
CHECK Constraint . . . 664
PRIMARY KEY Constraint . . . 665
Setting up Constraints when using the Declarative ORM Extension . . . 665
Configuring Constraint Naming Conventions . . . 665
Constraints API . . . 670
Indexes . . . 687
Index API . . . 689
3.3.5 Customizing DDL . . . 692
Custom DDL . . . 692
Controlling DDL Sequences . . . 693
Using the built-in DDLElement Classes . . . 694
DDL Expression Constructs API . . . 696
3.4 Column and Data Types . . . 703
3.4.1 Column and Data Types . . . 703
Generic Types . . . 704
SQL Standard and Multiple Vendor Types . . . 712
Vendor-Specific Types . . . 718
3.4.2 Custom Types . . . 719
Overriding Type Compilation . . . 719
Augmenting Existing Types . . . 720
TypeDecorator Recipes . . . 727
Replacing the Bind/Result Processing of Existing Types . . . 729
Applying SQL-level Bind/Result Processing . . . 730
Redefining and Creating New Operators . . . 732
Creating New Types . . . 733
3.4.3 Base Type API . . . 734
3.5 Engine and Connection Use . . . 739
3.5.1 Engine Configuration . . . 739
Supported Databases . . . 740
Database Urls . . . 740
Engine Creation API . . . 742
Pooling . . . 747
Custom DBAPI connect() arguments . . . 747
Configuring Logging . . . 747
3.5.2 Working with Engines and Connections . . . 748
Basic Usage . . . 748

Using Transactions . . . 750
Understanding Autocommit . . . 751
Connectionless Execution, Implicit Execution . . . 751
Translation of Schema Names . . . 753
Engine Disposal . . . 754
Using the Threadlocal Execution Strategy . . . 755
Working with Raw DBAPI Connections . . . 756
Registering New Dialects . . . 757
Connection / Engine API . . . 758
3.5.3 Connection Pooling . . . 778
Connection Pool Configuration . . . 778
Switching Pool Implementations . . . 778
Using a Custom Connection Function . . . 779
Constructing a Pool . . . 779
Pool Events . . . 780
Dealing with Disconnects . . . 780
Using Connection Pools with Multiprocessing . . . 782
API Documentation - Available Pool Implementations . . . 783
Pooling Plain DB-API Connections . . . 789
3.5.4 Core Events . . . 790
Connection Pool Events . . . 790
SQL Execution and Connection Events . . . 794
Schema Events . . . 808
3.6 Core API Basics . . . 813
3.6.1 Events . . . 813
Event Registration . . . 813
Named Argument Styles . . . 813
Targets . . . 814
Modifiers . . . 815
Event Reference . . . 815
API Reference . . . 815
3.6.2 Runtime Inspection API . . . 817
Available Inspection Targets . . . 818
3.6.3 Deprecated Event Interfaces . . . 818
Execution, Connection and Cursor Events . . . 818
Connection Pool Events . . . 819
3.6.4 Core Exceptions . . . 821
3.6.5 Core Internals . . . 823

4 Dialects **843**
4.1 Included Dialects . . . 843
4.1.1 Firebird . . . 843
DBAPI Support . . . 843
Firebird Dialects . . . 843
Locking Behavior . . . 843
RETURNING support . . . 844
fdb . . . 844
kinterbasdb . . . 845
4.1.2 Microsoft SQL Server . . . 846
DBAPI Support . . . 846
Auto Increment Behavior . . . 846
Collation Support . . . 848
LIMIT/OFFSET Support . . . 848
Nullability . . . 849

Date / Time Handling . . . 849
Large Text/Binary Type Deprecation . . . 849
Legacy Schema Mode . . . 850
Clustered Index Support . . . 850
MSSQL-Specific Index Options . . . 851
Compatibility Levels . . . 852
Triggers . . . 852
Enabling Snapshot Isolation . . . 852
Known Issues . . . 852
SQL Server Data Types . . . 853
PyODBC . . . 857
mxODBC . . . 859
pymssql . . . 859
zxjdbc . . . 860
AdoDBAPI . . . 860
4.1.3 MySQL . . . 860
DBAPI Support . . . 861
Supported Versions and Features . . . 861
Connection Timeouts . . . 861
CREATE TABLE arguments including Storage Engines . . . 861
Case Sensitivity and Table Reflection . . . 862
Transaction Isolation Level . . . 862
AUTO_INCREMENT Behavior . . . 862
Unicode . . . 863
Ansi Quoting Style . . . 864
MySQL SQL Extensions . . . 864
rowcount Support . . . 864
CAST Support . . . 865
MySQL Specific Index Options . . . 865
MySQL Foreign Keys . . . 866
MySQL Unique Constraints and Reflection . . . 867
TIMESTAMP Columns and NULL . . . 867
MySQL Data Types . . . 868
MySQL-Python . . . 879
pymysql . . . 880
MySQL-Connector . . . 880
cymysql . . . 881
OurSQL . . . 881
Google App Engine . . . 882
pyodbc . . . 882
zxjdbc . . . 883
4.1.4 Oracle . . . 883
DBAPI Support . . . 883
Connect Arguments . . . 883
Auto Increment Behavior . . . 884
Identifier Casing . . . 884
LIMIT/OFFSET Support . . . 884
RETURNING Support . . . 885
ON UPDATE CASCADE . . . 885
Oracle 8 Compatibility . . . 885
Synonym/DBLINK Reflection . . . 886
DateTime Compatibility . . . 886
Oracle Table Options . . . 886
Oracle Specific Index Options . . . 887

Oracle Data Types . . . 887
cx_Oracle . . . 890
zxjdbc . . . 894
4.1.5 PostgreSQL . . . 894
DBAPI Support . . . 894
Sequences/SERIAL . . . 894
Transaction Isolation Level . . . 895
Remote-Schema Table Introspection and Postgresql search_path . . . 895
INSERT/UPDATE...RETURNING . . . 897
Full Text Search . . . 898
FROM ONLY 899
Postgresql-Specific Index Options . . . 899
Postgresql Index Reflection . . . 901
Special Reflection Options . . . 901
PostgreSQL Table Options . . . 902
ARRAY Types . . . 902
JSON Types . . . 903
HSTORE Type . . . 903
ENUM Types . . . 903
PostgreSQL Data Types . . . 904
PostgreSQL Constraint Types . . . 918
psycopg2 . . . 919
pg8000 . . . 923
psycopg2cffi . . . 924
py-postgresql . . . 925
zxjdbc . . . 925
4.1.6 SQLite . . . 925
DBAPI Support . . . 926
Date and Time Types . . . 926
SQLite Auto Incrementing Behavior . . . 926
Database Locking Behavior / Concurrency . . . 927
Transaction Isolation Level . . . 928
SAVEPOINT Support . . . 928
Transactional DDL . . . 929
Foreign Key Support . . . 929
Type Reflection . . . 929
Partial Indexes . . . 930
Dotted Column Names . . . 930
SQLite Data Types . . . 932
Pysqlite . . . 934
Pysqlcipher . . . 938
4.1.7 Sybase . . . 939
DBAPI Support . . . 939
python-sybase . . . 939
pyodbc . . . 939
mxodbc . . . 940
4.2 External Dialects . . . 941
4.2.1 Production Ready . . . 941
4.2.2 Experimental / Incomplete . . . 941
4.2.3 Attic . . . 941

5 Frequently Asked Questions **943**
5.1 Connections / Engines . . . 943
5.1.1 How do I configure logging? . . . 943

5.1.2 How do I pool database connections? Are my connections pooled? . . . 943
5.1.3 How do I pass custom connect arguments to my database API? . . . 943
5.1.4 "MySQL Server has gone away" . . . 944
5.1.5 Why does SQLAlchemy issue so many ROLLBACKs? . . . 944
I'm on MyISAM - how do I turn it off? . . . 944
I'm on SQL Server - how do I turn those ROLLBACKs into COMMITs? . . . 944
5.1.6 I am using multiple connections with a SQLite database (typically to test transaction operation), and my test program is not working! . . . 945
5.1.7 How do I get at the raw DBAPI connection when using an Engine? . . . 945
5.1.8 How do I use engines / connections / sessions with Python multiprocessing, or os.fork()? . . . 945
5.2 MetaData / Schema . . . 946
5.2.1 My program is hanging when I say `table.drop()` / `metadata.drop_all()` . . . 947
5.2.2 Does SQLAlchemy support ALTER TABLE, CREATE VIEW, CREATE TRIGGER, Schema Upgrade Functionality? . . . 947
5.2.3 How can I sort Table objects in order of their dependency? . . . 947
5.2.4 How can I get the CREATE TABLE/ DROP TABLE output as a string? . . . 948
5.2.5 How can I subclass Table/Column to provide certain behaviors/configurations? . . . 948
5.3 SQL Expressions . . . 948
5.3.1 How do I render SQL expressions as strings, possibly with bound parameters inlined? . . . 948
5.3.2 Why does `.col.in_([])` Produce `col != col`? Why not `1=0`? . . . 950
5.4 ORM Configuration . . . 951
5.4.1 How do I map a table that has no primary key? . . . 951
5.4.2 How do I configure a Column that is a Python reserved word or similar? . . . 952
5.4.3 How do I get a list of all columns, relationships, mapped attributes, etc. given a mapped class? 952
5.4.4 I'm getting a warning or error about "Implicitly combining column X under attribute Y" . . . 952
5.4.5 I'm using Declarative and setting primaryjoin/secondaryjoin using an `and_()` or `or_()`, and I am getting an error message about foreign keys. . . . 953
5.4.6 Why is `ORDER BY` required with `LIMIT` (especially with `subqueryload()`)? . . . 954
5.5 Performance . . . 956
5.5.1 How can I profile a SQLAlchemy powered application? . . . 956
Query Profiling . . . 956
Code Profiling . . . 957
Execution Slowness . . . 958
Result Fetching Slowness - Core . . . 958
Result Fetching Slowness - ORM . . . 959
5.5.2 I'm inserting 400,000 rows with the ORM and it's really slow! . . . 960
5.6 Sessions / Queries . . . 963
5.6.1 I'm re-loading data with my Session but it isn't seeing changes that I committed elsewhere . 963
5.6.2 "This Session's transaction has been rolled back due to a previous exception during flush." (or similar) . . . 964
But why does flush() insist on issuing a ROLLBACK? . . . 965
But why isn't the one automatic call to ROLLBACK enough? Why must I ROLLBACK again? 966
5.6.3 How do I make a Query that always adds a certain filter to every query? . . . 967
5.6.4 I've created a mapping against an Outer Join, and while the query returns rows, no objects are returned. Why not? . . . 967
5.6.5 I'm using `joinedload()` or `lazy=False` to create a JOIN/OUTER JOIN and SQLAlchemy is not constructing the correct query when I try to add a WHERE, ORDER BY, LIMIT, etc. (which relies upon the (OUTER) JOIN) . . . 967
5.6.6 Query has no `__len__()`, why not? . . . 967
5.6.7 How Do I use Textual SQL with ORM Queries? . . . 968
5.6.8 I'm calling `Session.delete(myobject)` and it isn't removed from the parent collection! 968
5.6.9 why isn't my `__init__()` called when I load objects? . . . 968
5.6.10 how do I use ON DELETE CASCADE with SA's ORM? . . . 968

5.6.11 I set the "foo_id" attribute on my instance to "7", but the "foo" attribute is still `None` - shouldn't it have loaded Foo with id #7? . . . 968
5.6.12 How do I walk all objects that are related to a given object? . . . 970
5.6.13 Is there a way to automagically have only unique keywords (or other kinds of objects) without doing a query for the keyword and getting a reference to the row containing that keyword? . 971

6 Changes and Migration **973**
6.1 Current Migration Guide . . . 973
6.1.1 What's New in SQLAlchemy 1.1? . . . 973
Introduction . . . 973
Platform / Installer Changes . . . 973
New Features and Improvements - ORM . . . 974
New Features and Improvements - Core . . . 977
Key Behavioral Changes - ORM . . . 985
Key Behavioral Changes - Core . . . 985
Dialect Improvements and Changes - Postgresql . . . 985
Dialect Improvements and Changes - MySQL . . . 987
Dialect Improvements and Changes - SQLite . . . 988
Dialect Improvements and Changes - SQL Server . . . 989
Dialect Improvements and Changes - Oracle . . . 990
6.2 Change logs . . . 990
6.2.1 1.1 Changelog . . . 990
1.1.0b1 . . . 990
6.2.2 1.0 Changelog . . . 996
1.0.12 . . . 996
1.0.11 . . . 996
1.0.10 . . . 997
1.0.9 . . . 999
1.0.8 . . . 1000
1.0.7 . . . 1001
1.0.6 . . . 1002
1.0.5 . . . 1003
1.0.4 . . . 1005
1.0.3 . . . 1006
1.0.2 . . . 1008
1.0.1 . . . 1009
1.0.0 . . . 1010
1.0.0b5 . . . 1012
1.0.0b4 . . . 1012
1.0.0b3 . . . 1013
1.0.0b2 . . . 1014
1.0.0b1 . . . 1015
6.2.3 0.9 Changelog . . . 1041
0.9.11 . . . 1041
0.9.10 . . . 1042
0.9.9 . . . 1043
0.9.8 . . . 1047
0.9.7 . . . 1049
0.9.6 . . . 1052
0.9.5 . . . 1052
0.9.4 . . . 1056
0.9.3 . . . 1061
0.9.2 . . . 1064
0.9.1 . . . 1069

0.9.0 . . . 1070
0.9.0b1 . . . 1076
6.2.4 0.8 Changelog . . . 1089
0.8.7 . . . 1089
0.8.6 . . . 1090
0.8.5 . . . 1091
0.8.4 . . . 1094
0.8.3 . . . 1095
0.8.2 . . . 1099
0.8.1 . . . 1102
0.8.0 . . . 1105
0.8.0b2 . . . 1108
0.8.0b1 . . . 1112
6.2.5 0.7 Changelog . . . 1122
0.7.11 . . . 1122
0.7.10 . . . 1123
0.7.9 . . . 1124
0.7.8 . . . 1127
0.7.7 . . . 1128
0.7.6 . . . 1129
0.7.5 . . . 1132
0.7.4 . . . 1135
0.7.3 . . . 1138
0.7.2 . . . 1142
0.7.1 . . . 1145
0.7.0 . . . 1146
0.7.0b4 . . . 1148
0.7.0b3 . . . 1151
0.7.0b2 . . . 1152
0.7.0b1 . . . 1153
6.2.6 0.6 Changelog . . . 1157
0.6.9 . . . 1157
0.6.8 . . . 1159
0.6.7 . . . 1161
0.6.6 . . . 1163
0.6.5 . . . 1166
0.6.4 . . . 1169
0.6.3 . . . 1173
0.6.2 . . . 1174
0.6.1 . . . 1176
0.6.0 . . . 1178
0.6beta3 . . . 1180
0.6beta2 . . . 1182
0.6beta1 . . . 1187
6.2.7 0.5 Changelog . . . 1200
0.5.9 . . . 1200
0.5.8 . . . 1200
0.5.7 . . . 1200
0.5.6 . . . 1202
0.5.5 . . . 1204
0.5.4p2 . . . 1205
0.5.4p1 . . . 1206
0.5.4 . . . 1206
0.5.3 . . . 1209

0.5.2 . . . 1211
0.5.1 . . . 1212
0.5.0 . . . 1214
0.5.0rc4 . . . 1218
0.5.0rc3 . . . 1219
0.5.0rc2 . . . 1221
0.5.0rc1 . . . 1223
0.5.0beta3 . . . 1225
0.5.0beta2 . . . 1227
0.5.0beta1 . . . 1228
6.2.8 0.4 Changelog . . . 1230
0.4.8 . . . 1230
0.4.7p1 . . . 1231
0.4.7 . . . 1231
0.4.6 . . . 1232
0.4.5 . . . 1234
0.4.4 . . . 1238
0.4.3 . . . 1240
0.4.2p3 . . . 1243
0.4.2 . . . 1245
0.4.1 . . . 1249
0.4.0 . . . 1251
0.4.0beta6 . . . 1253
0.4.0beta5 . . . 1254
0.4.0beta4 . . . 1255
0.4.0beta3 . . . 1256
0.4.0beta2 . . . 1256
0.4.0beta1 . . . 1257
6.2.9 0.3 Changelog . . . 1262
0.3.11 . . . 1262
0.3.10 . . . 1263
0.3.9 . . . 1264
0.3.8 . . . 1266
0.3.7 . . . 1268
0.3.6 . . . 1271
0.3.5 . . . 1274
0.3.4 . . . 1276
0.3.3 . . . 1279
0.3.2 . . . 1279
0.3.1 . . . 1280
0.3.0 . . . 1281
6.2.10 0.2 Changelog . . . 1284
0.2.8 . . . 1284
0.2.7 . . . 1285
0.2.6 . . . 1286
0.2.5 . . . 1287
0.2.4 . . . 1288
0.2.3 . . . 1288
0.2.2 . . . 1289
0.2.1 . . . 1290
0.2.0 . . . 1290
6.2.11 0.1 Changelog . . . 1291
0.1.7 . . . 1291
0.1.6 . . . 1292

0.1.5 . . . 1293
0.1.4 . . . 1294
0.1.3 . . . 1295
0.1.2 . . . 1296
0.1.1 . . . 1296
6.3 Older Migration Guides . . . 1297
6.3.1 What's New in SQLAlchemy 1.0? . . . 1297
Introduction . . . 1298
New Features and Improvements - ORM . . . 1298
New Features and Improvements - Core . . . 1306
Key Behavioral Changes - ORM . . . 1312
Key Behavioral Changes - Core . . . 1325
Dialect Improvements and Changes - Postgresql . . . 1330
Dialect Improvements and Changes - MySQL . . . 1333
Dialect Improvements and Changes - SQLite . . . 1335
Dialect Improvements and Changes - SQL Server . . . 1336
Dialect Improvements and Changes - Oracle . . . 1336
6.3.2 What's New in SQLAlchemy 0.9? . . . 1337
Introduction . . . 1337
Platform Support . . . 1337
Behavioral Changes - ORM . . . 1337
Behavioral Changes - Core . . . 1343
New Features . . . 1346
Behavioral Improvements . . . 1353
Dialect Changes . . . 1363
6.3.3 What's New in SQLAlchemy 0.8? . . . 1364
Introduction . . . 1364
Platform Support . . . 1364
New ORM Features . . . 1364
New Core Features . . . 1373
Behavioral Changes . . . 1378
Removed . . . 1385
6.3.4 What's New in SQLAlchemy 0.7? . . . 1385
Introduction . . . 1386
New Features . . . 1386
Behavioral Changes (Backwards Compatible) . . . 1391
Behavioral Changes (Backwards Incompatible) . . . 1395
Deprecated API . . . 1400
Backwards Incompatible API Changes . . . 1401
Previously Deprecated, Now Removed . . . 1402
6.3.5 What's New in SQLAlchemy 0.6? . . . 1402
Platform Support . . . 1402
New Dialect System . . . 1402
Expression Language Changes . . . 1404
C Extensions for Result Fetching . . . 1406
New Schema Capabilities . . . 1407
Logging opened up . . . 1408
Reflection/Inspector API . . . 1409
RETURNING Support . . . 1409
Type System Changes . . . 1410
ORM Changes . . . 1412
Extensions . . . 1416
6.3.6 What's new in SQLAlchemy 0.5? . . . 1416
Major Documentation Changes . . . 1417

Deprecations Source . . . 1417
Requirements Changes . . . 1417
Object Relational Mapping . . . 1417
Extending the ORM . . . 1420
Schema/Types . . . 1420
Connection Pool no longer threadlocal by default . . . 1422
*args Accepted, *args No Longer Accepted . . . 1422
Removed . . . 1423
Renamed or Moved . . . 1425
Deprecated . . . 1425
6.3.7 What's new in SQLAlchemy 0.4? . . . 1425
First Things First . . . 1425
Module Imports . . . 1426
Object Relational Mapping . . . 1426
SQL Expressions . . . 1434
Schema and Reflection . . . 1435
SQL Execution . . . 1436

7 Indices and tables **1437**

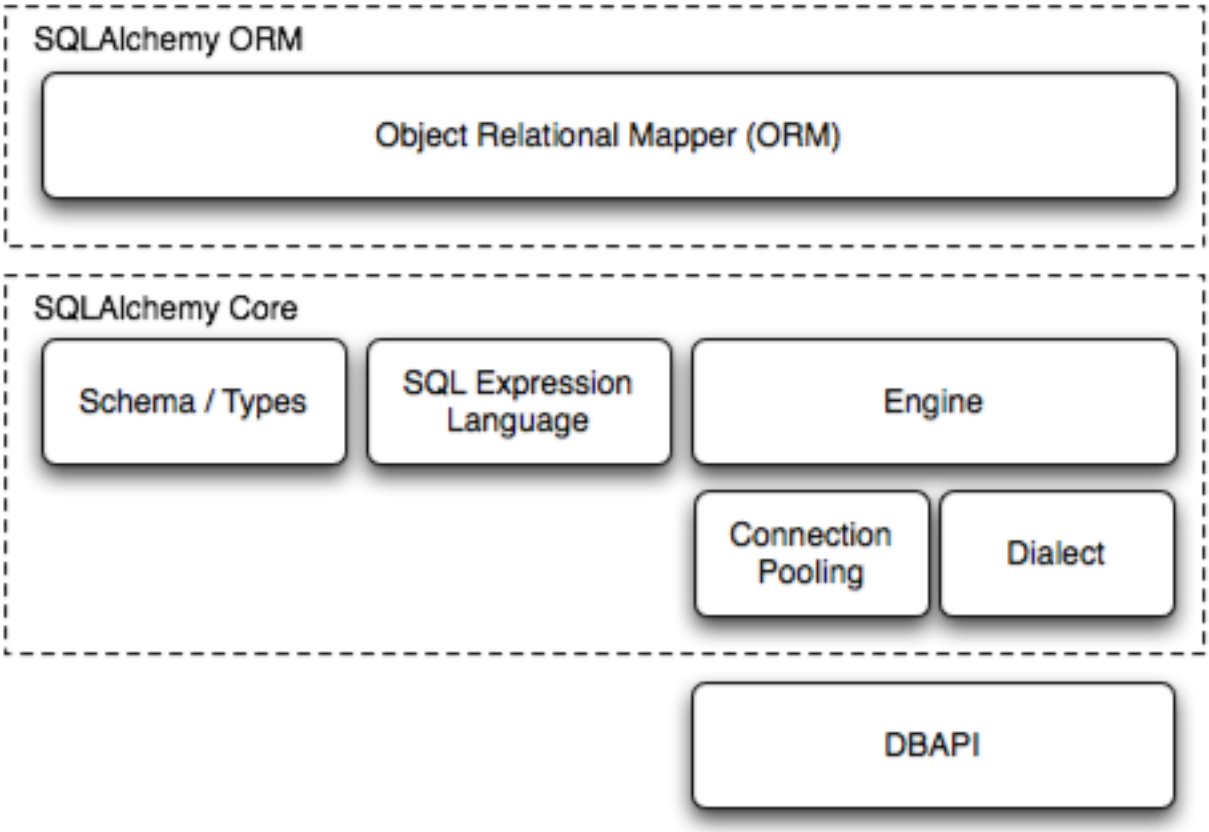
Python Module Index **1439**

Full table of contents. For a high level overview of all documentation, see index_toplevel.

CHAPTER 1

Overview

The SQLAlchemy SQL Toolkit and Object Relational Mapper is a comprehensive set of tools for working with databases and Python. It has several distinct areas of functionality which can be used individually or combined together. Its major components are illustrated in below, with component dependencies organized into layers:



Above, the two most significant front-facing portions of SQLAlchemy are the **Object Relational Mapper** and the **SQL Expression Language**. SQL Expressions can be used independently of the ORM. When using the ORM, the SQL Expression language remains part of the public facing API as it is used within object-relational configurations and queries.

1.1 Documentation Overview

The documentation is separated into three sections: *SQLAlchemy ORM*, *SQLAlchemy Core*, and *Dialects*.

In *SQLAlchemy ORM*, the Object Relational Mapper is introduced and fully described. New users should begin with the *Object Relational Tutorial*. If you want to work with higher-level SQL which is constructed automatically for you, as well as management of Python objects, proceed to this tutorial.

In *SQLAlchemy Core*, the breadth of SQLAlchemy's SQL and database integration and description services are documented, the core of which is the SQL Expression language. The SQL Expression Language is a toolkit all its own, independent of the ORM package, which can be used to construct manipulable SQL expressions which can be programmatically constructed, modified, and executed, returning cursor-like result sets. In contrast to the ORM's domain-centric mode of usage, the expression language provides a schema-centric usage paradigm. New users should begin here with *SQL Expression Language Tutorial*. SQLAlchemy engine, connection, and pooling services are also described in *SQLAlchemy Core*.

In *Dialects*, reference documentation for all provided database and DBAPI backends is provided.

1.2 Code Examples

Working code examples, mostly regarding the ORM, are included in the SQLAlchemy distribution. A description of all the included example applications is at *ORM Examples*.

There is also a wide variety of examples involving both core SQLAlchemy constructs as well as the ORM on the wiki. See Theatrum Chemicum.

1.3 Installation Guide

1.3.1 Supported Platforms

SQLAlchemy has been tested against the following platforms:

- cPython since version 2.6, through the 2.xx series
- cPython version 3, throughout all 3.xx series
- Pypy 2.1 or greater

Changed in version 0.9: Python 2.6 is now the minimum Python version supported.

Platforms that don't currently have support include Jython, IronPython. Jython has been supported in the past and may be supported in future releases as well, depending on the state of Jython itself.

1.3.2 Supported Installation Methods

SQLAlchemy installation is via standard Python methodologies that are based on setuptools, either by referring to `setup.py` directly or by using pip or other setuptools-compatible approaches.

Changed in version 1.1: setuptools is now required by the setup.py file; plain distutils installs are no longer supported.

1.3.3 Install via pip

When `pip` is available, the distribution can be downloaded from Pypi and installed in one step:

```
pip install SQLAlchemy
```

This command will download the latest **released** version of SQLAlchemy from the Python Cheese Shop and install it to your system.

In order to install the latest **prerelease** version, such as `1.1.0b1`, pip requires that the `--pre` flag be used:

```
pip install --pre SQLAlchemy
```

Where above, if the most recent version is a prerelease, it will be installed instead of the latest released version.

1.3.4 Installing using setup.py

Otherwise, you can install from the distribution using the `setup.py` script:

```
python setup.py install
```

1.3.5 Installing the C Extensions

SQLAlchemy includes C extensions which provide an extra speed boost for dealing with result sets. The extensions are supported on both the 2.xx and 3.xx series of cPython.

`setup.py` will automatically build the extensions if an appropriate platform is detected. If the build of the C extensions fails, due to missing compiler or other issue, the setup process will output a warning message, and re-run the build without the C extensions, upon completion reporting final status.

To run the build/install without even attempting to compile the C extensions, the `DISABLE_SQLALCHEMY_CEXT` environment variable may be specified. The use case for this is either for special testing circumstances, or in the rare case of compatibility/build issues not overcome by the usual "rebuild" mechanism:

```
export DISABLE_SQLALCHEMY_CEXT=1; python setup.py install
```

Changed in version 1.1: The legacy `--without-cextensions` flag has been removed from the installer as it relies on deprecated features of setuptools.

1.3.6 Installing on Python 3

SQLAlchemy runs directly on Python 2 or Python 3, and can be installed in either environment without any adjustments or code conversion.

1.3.7 Installing a Database API

SQLAlchemy is designed to operate with a DBAPI implementation built for a particular database, and includes support for the most popular databases. The individual database sections in Dialects enumerate the available DBAPIs for each database, including external links.

1.3.8 Checking the Installed SQLAlchemy Version

This documentation covers SQLAlchemy version 1.1. If you're working on a system that already has SQLAlchemy installed, check the version from your Python prompt like this:

```
>>> import sqlalchemy
>>> sqlalchemy.__version__
1.1.0
```

1.4 1.0 to 1.1 Migration

Notes on what's changed from 1.0 to 1.1 is available here at What's New in SQLAlchemy 1.1?.

CHAPTER 2

SQLAlchemy ORM

Here, the Object Relational Mapper is introduced and fully described. If you want to work with higher-level SQL which is constructed automatically for you, as well as automated persistence of Python objects, proceed first to the tutorial.

2.1 Object Relational Tutorial

The SQLAlchemy Object Relational Mapper presents a method of associating user-defined Python classes with database tables, and instances of those classes (objects) with rows in their corresponding tables. It includes a system that transparently synchronizes all changes in state between objects and their related rows, called a unit of work, as well as a system for expressing database queries in terms of the user defined classes and their defined relationships between each other.

The ORM is in contrast to the SQLAlchemy Expression Language, upon which the ORM is constructed. Whereas the SQL Expression Language, introduced in *SQL Expression Language Tutorial*, presents a system of representing the primitive constructs of the relational database directly without opinion, the ORM presents a high level and abstracted pattern of usage, which itself is an example of applied usage of the Expression Language.

While there is overlap among the usage patterns of the ORM and the Expression Language, the similarities are more superficial than they may at first appear. One approaches the structure and content of data from the perspective of a user-defined domain model which is transparently persisted and refreshed from its underlying storage model. The other approaches it from the perspective of literal schema and SQL expression representations which are explicitly composed into messages consumed individually by the database.

A successful application may be constructed using the Object Relational Mapper exclusively. In advanced situations, an application constructed with the ORM may make occasional usage of the Expression Language directly in certain areas where specific database interactions are required.

The following tutorial is in doctest format, meaning each >>> line represents something you can type at a Python command prompt, and the following text represents the expected return value.

2.1.1 Version Check

A quick check to verify that we are on at least **version 1.1** of SQLAlchemy:

```
>>> import sqlalchemy
>>> sqlalchemy.__version__
1.1.0
```

2.1.2 Connecting

For this tutorial we will use an in-memory-only SQLite database. To connect we use *create_engine()*:

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:', echo=True)
```

The `echo` flag is a shortcut to setting up SQLAlchemy logging, which is accomplished via Python's standard `logging` module. With it enabled, we'll see all the generated SQL produced. If you are working through this tutorial and want less output generated, set it to `False`. This tutorial will format the SQL behind a popup window so it doesn't get in our way; just click the "SQL" links to see what's being generated.

The return value of *create_engine()* is an instance of *Engine*, and it represents the core interface to the database, adapted through a dialect that handles the details of the database and DBAPI in use. In this case the SQLite dialect will interpret instructions to the Python built-in `sqlite3` module.

> **Lazy Connecting**
>
> The *Engine*, when first returned by *create_engine()*, has not actually tried to connect to the database yet; that happens only the first time it is asked to perform a task against the database.

The first time a method like *Engine.execute()* or *Engine.connect()* is called, the *Engine* establishes a real DBAPI connection to the database, which is then used to emit the SQL. When using the ORM, we typically don't use the *Engine* directly once created; instead, it's used behind the scenes by the ORM as we'll see shortly.

See also:

Database Urls - includes examples of *create_engine()* connecting to several kinds of databases with links to more information.

2.1.3 Declare a Mapping

When using the ORM, the configurational process starts by describing the database tables we'll be dealing with, and then by defining our own classes which will be mapped to those tables. In modern SQLAlchemy, these two tasks are usually performed together, using a system known as *Declarative*, which allows us to create classes that include directives to describe the actual database table they will be mapped to.

Classes mapped using the Declarative system are defined in terms of a base class which maintains a catalog of classes and tables relative to that base - this is known as the **declarative base class**. Our application will usually have just one instance of this base in a commonly imported module. We create the base class using the *declarative_base()* function, as follows:

```
>>> from sqlalchemy.ext.declarative import declarative_base

>>> Base = declarative_base()
```

Now that we have a "base", we can define any number of mapped classes in terms of it. We will start with just a single table called `users`, which will store records for the end-users using our application. A new class called `User` will be the class to which we map this table. Within the class, we define details about the table to which we'll be mapping, primarily the table name, and names and datatypes of columns:

```
>>> from sqlalchemy import Column, Integer, String
>>> class User(Base):
...     __tablename__ = 'users'
...
...     id = Column(Integer, primary_key=True)
```

```
...     name = Column(String)
...     fullname = Column(String)
...     password = Column(String)
...
...     def __repr__(self):
...        return "<User(name='%s', fullname='%s', password='%s')>" % (
...                              self.name, self.fullname, self.password)
```

Tip

The `User` class defines a `__repr__()` method, but note that is **optional**; we only implement it in this tutorial so that our examples show nicely formatted `User` objects.

A class using Declarative at a minimum needs a `__tablename__` attribute, and at least one *`Column`* which is part of a primary key [1]. SQLAlchemy never makes any assumptions by itself about the table to which a class refers, including that it has no built-in conventions for names, datatypes, or constraints. But this doesn't mean boilerplate is required; instead, you're encouraged to create your own automated conventions using helper functions and mixin classes, which is described in detail at *Mixin and Custom Base Classes*.

When our class is constructed, Declarative replaces all the *`Column`* objects with special Python accessors known as descriptors; this is a process known as instrumentation. The "instrumented" mapped class will provide us with the means to refer to our table in a SQL context as well as to persist and load the values of columns from the database.

Outside of what the mapping process does to our class, the class remains otherwise mostly a normal Python class, to which we can define any number of ordinary attributes and methods needed by our application.

2.1.4 Create a Schema

With our `User` class constructed via the Declarative system, we have defined information about our table, known as table metadata. The object used by SQLAlchemy to represent this information for a specific table is called the *`Table`* object, and here Declarative has made one for us. We can see this object by inspecting the `__table__` attribute:

```
>>> User.__table__
Table('users', MetaData(bind=None),
            Column('id', Integer(), table=<users>, primary_key=True, nullable=False),
            Column('name', String(), table=<users>),
            Column('fullname', String(), table=<users>),
            Column('password', String(), table=<users>), schema=None)
```

Classical Mappings

The Declarative system, though highly recommended, is not required in order to use SQLAlchemy's ORM. Outside of Declarative, any plain Python class can be mapped to any *`Table`* using the *`mapper()`* function directly; this less common usage is described at *Classical Mappings*.

When we declared our class, Declarative used a Python metaclass in order to perform additional activities once the class declaration was complete; within this phase, it then created a *`Table`* object according to our specifications, and associated it with the class by constructing a *`Mapper`* object. This object is a behind-the-scenes object we normally don't need to deal with directly (though it can provide plenty of information about our mapping when we need it).

The *`Table`* object is a member of a larger collection known as *`MetaData`*. When using Declarative, this object is available using the `.metadata` attribute of our declarative base class.

[1] For information on why a primary key is required, see *How do I map a table that has no primary key?*.

The *MetaData* is a registry which includes the ability to emit a limited set of schema generation commands to the database. As our SQLite database does not actually have a `users` table present, we can use *MetaData* to issue CREATE TABLE statements to the database for all tables that don't yet exist. Below, we call the *MetaData.create_all()* method, passing in our *Engine* as a source of database connectivity. We will see that special commands are first emitted to check for the presence of the `users` table, and following that the actual `CREATE TABLE` statement:

```
>>> Base.metadata.create_all(engine)
SELECT ...
PRAGMA table_info("users")
()
CREATE TABLE users (
    id INTEGER NOT NULL, name VARCHAR,
    fullname VARCHAR,
    password VARCHAR,
    PRIMARY KEY (id)
)
()
COMMIT
```

Minimal Table Descriptions vs. Full Descriptions

Users familiar with the syntax of CREATE TABLE may notice that the VARCHAR columns were generated without a length; on SQLite and Postgresql, this is a valid datatype, but on others, it's not allowed. So if running this tutorial on one of those databases, and you wish to use SQLAlchemy to issue CREATE TABLE, a "length" may be provided to the *String* type as below:

```
Column(String(50))
```

The length field on *String*, as well as similar precision/scale fields available on *Integer*, *Numeric*, etc. are not referenced by SQLAlchemy other than when creating tables.
Additionally, Firebird and Oracle require sequences to generate new primary key identifiers, and SQLAlchemy doesn't generate or assume these without being instructed. For that, you use the *Sequence* construct:

```
from sqlalchemy import Sequence
Column(Integer, Sequence('user_id_seq'), primary_key=True)
```

A full, foolproof *Table* generated via our declarative mapping is therefore:

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, Sequence('user_id_seq'), primary_key=True)
    name = Column(String(50))
    fullname = Column(String(50))
    password = Column(String(12))

    def __repr__(self):
        return "<User(name='%s', fullname='%s', password='%s')>" % (
                                self.name, self.fullname, self.password)
```

We include this more verbose table definition separately to highlight the difference between a minimal construct geared primarily towards in-Python usage only, versus one that will be used to emit CREATE TABLE statements on a particular set of backends with more stringent requirements.

2.1.5 Create an Instance of the Mapped Class

With mappings complete, let's now create and inspect a `User` object:

```
>>> ed_user = User(name='ed', fullname='Ed Jones', password='edspassword')
>>> ed_user.name
'ed'
>>> ed_user.password
'edspassword'
>>> str(ed_user.id)
'None'
```

the `__init__()` method

Our `User` class, as defined using the Declarative system, has been provided with a constructor (e.g. `__init__()` method) which automatically accepts keyword names that match the columns we've mapped. We are free to define any explicit `__init__()` method we prefer on our class, which will override the default method provided by Declarative.

Even though we didn't specify it in the constructor, the `id` attribute still produces a value of `None` when we access it (as opposed to Python's usual behavior of raising `AttributeError` for an undefined attribute). SQLAlchemy's instrumentation normally produces this default value for column-mapped attributes when first accessed. For those attributes where we've actually assigned a value, the instrumentation system is tracking those assignments for use within an eventual INSERT statement to be emitted to the database.

2.1.6 Creating a Session

We're now ready to start talking to the database. The ORM's "handle" to the database is the *`Session`*. When we first set up the application, at the same level as our *`create_engine()`* statement, we define a *`Session`* class which will serve as a factory for new *`Session`* objects:

```
>>> from sqlalchemy.orm import sessionmaker
>>> Session = sessionmaker(bind=engine)
```

In the case where your application does not yet have an *`Engine`* when you define your module-level objects, just set it up like this:

```
>>> Session = sessionmaker()
```

Later, when you create your engine with *`create_engine()`*, connect it to the *`Session`* using *`configure()`*:

```
>>> Session.configure(bind=engine)  # once engine is available
```

Session Lifecycle Patterns

The question of when to make a *`Session`* depends a lot on what kind of application is being built. Keep in mind, the *`Session`* is just a workspace for your objects, local to a particular database connection - if you think of an application thread as a guest at a dinner party, the *`Session`* is the guest's plate and the objects it holds are the food (and the database...the kitchen?)! More on this topic available at *When do I construct a Session, when do I commit it, and when do I close it?*.

This custom-made *`Session`* class will create new *`Session`* objects which are bound to our database. Other transactional characteristics may be defined when calling *`sessionmaker`* as well; these are described in a later chapter. Then, whenever you need to have a conversation with the database, you instantiate a *`Session`*:

```
>>> session = Session()
```

The above *Session* is associated with our SQLite-enabled *Engine*, but it hasn't opened any connections yet. When it's first used, it retrieves a connection from a pool of connections maintained by the *Engine*, and holds onto it until we commit all changes and/or close the session object.

2.1.7 Adding New Objects

To persist our `User` object, we *add()* it to our *Session*:

```
>>> ed_user = User(name='ed', fullname='Ed Jones', password='edspassword')
>>> session.add(ed_user)
```

At this point, we say that the instance is **pending**; no SQL has yet been issued and the object is not yet represented by a row in the database. The *Session* will issue the SQL to persist `Ed Jones` as soon as is needed, using a process known as a **flush**. If we query the database for `Ed Jones`, all pending information will first be flushed, and the query is issued immediately thereafter.

For example, below we create a new *Query* object which loads instances of `User`. We "filter by" the `name` attribute of `ed`, and indicate that we'd like only the first result in the full list of rows. A `User` instance is returned which is equivalent to that which we've added:

```
>>> our_user = session.query(User).filter_by(name='ed').first()
BEGIN (implicit)
INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)
('ed', 'Ed Jones', 'edspassword')
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE users.name = ?
 LIMIT ? OFFSET ?
('ed', 1, 0)
>>> our_user
<User(name='ed', fullname='Ed Jones', password='edspassword')>
```

In fact, the *Session* has identified that the row returned is the **same** row as one already represented within its internal map of objects, so we actually got back the identical instance as that which we just added:

```
>>> ed_user is our_user
True
```

The ORM concept at work here is known as an identity map and ensures that all operations upon a particular row within a *Session* operate upon the same set of data. Once an object with a particular primary key is present in the *Session*, all SQL queries on that *Session* will always return the same Python object for that particular primary key; it also will raise an error if an attempt is made to place a second, already-persisted object with the same primary key within the session.

We can add more `User` objects at once using *add_all()*:

```
>>> session.add_all([
...     User(name='wendy', fullname='Wendy Williams', password='foobar'),
...     User(name='mary', fullname='Mary Contrary', password='xxg527'),
...     User(name='fred', fullname='Fred Flinstone', password='blah')])
```

Also, we've decided the password for Ed isn't too secure, so lets change it:

```
>>> ed_user.password = 'f8s7ccs'
```

The *Session* is paying attention. It knows, for example, that `Ed Jones` has been modified:

```
>>> session.dirty
IdentitySet([<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>])
```

and that three new `User` objects are pending:

```
>>> session.new
IdentitySet([<User(name='wendy', fullname='Wendy Williams', password='foobar')>,
<User(name='mary', fullname='Mary Contrary', password='xxg527')>,
<User(name='fred', fullname='Fred Flinstone', password='blah')>])
```

We tell the *Session* that we'd like to issue all remaining changes to the database and commit the transaction, which has been in progress throughout. We do this via *commit()*:

```
>>> session.commit()
UPDATE users SET password=? WHERE users.id = ?
('f8s7ccs', 1)
INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)
('wendy', 'Wendy Williams', 'foobar')
INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)
('mary', 'Mary Contrary', 'xxg527')
INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)
('fred', 'Fred Flinstone', 'blah')
COMMIT
```

commit() flushes whatever remaining changes remain to the database, and commits the transaction. The connection resources referenced by the session are now returned to the connection pool. Subsequent operations with this session will occur in a **new** transaction, which will again re-acquire connection resources when first needed.

If we look at Ed's `id` attribute, which earlier was `None`, it now has a value:

```
>>> ed_user.id
BEGIN (implicit)
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE users.id = ?
(1,)
1
```

After the *Session* inserts new rows in the database, all newly generated identifiers and database-generated defaults become available on the instance, either immediately or via load-on-first-access. In this case, the entire row was reloaded on access because a new transaction was begun after we issued *commit()*. SQLAlchemy by default refreshes data from a previous transaction the first time it's accessed within a new transaction, so that the most recent state is available. The level of reloading is configurable as is described in Using the Session.

Session Object States

As our `User` object moved from being outside the *Session*, to inside the *Session* without a primary key, to actually being inserted, it moved between three out of four available "object states" - **transient**, **pending**, and **persistent**. Being aware of these states and what they mean is always a good idea - be sure to read *Quickie Intro to Object States* for a quick overview.

2.1.8 Rolling Back

Since the *Session* works within a transaction, we can roll back changes made too. Let's make two changes that we'll revert; `ed_user`'s user name gets set to `Edwardo`:

```
>>> ed_user.name = 'Edwardo'
```

and we'll add another erroneous user, `fake_user`:

```
>>> fake_user = User(name='fakeuser', fullname='Invalid', password='12345')
>>> session.add(fake_user)
```

Querying the session, we can see that they're flushed into the current transaction:

```
>>> session.query(User).filter(User.name.in_(['Edwardo', 'fakeuser'])).all()
UPDATE users SET name=? WHERE users.id = ?
('Edwardo', 1)
INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)
('fakeuser', 'Invalid', '12345')
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE users.name IN (?, ?)
('Edwardo', 'fakeuser')
[<User(name='Edwardo', fullname='Ed Jones', password='f8s7ccs')>, <User(name='fakeuser', fullname='In
```

Rolling back, we can see that `ed_user`'s name is back to `ed`, and `fake_user` has been kicked out of the session:

```
>>> session.rollback()
ROLLBACK


>>> ed_user.name
BEGIN (implicit)
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE users.id = ?
(1,)
u'ed'
>>> fake_user in session
False
```

issuing a SELECT illustrates the changes made to the database:

```
>>> session.query(User).filter(User.name.in_(['ed', 'fakeuser'])).all()
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE users.name IN (?, ?)
('ed', 'fakeuser')
[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]
```

2.1.9 Querying

A *Query* object is created using the *query()* method on *Session*. This function takes a variable number of arguments, which can be any combination of classes and class-instrumented descriptors. Below, we indicate a *Query* which loads `User` instances. When evaluated in an iterative context, the list of `User` objects present is returned:

```
>>> for instance in session.query(User).order_by(User.id):
...     print(instance.name, instance.fullname)
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users ORDER BY users.id
()
ed Ed Jones
wendy Wendy Williams
mary Mary Contrary
fred Fred Flinstone
```

The *Query* also accepts ORM-instrumented descriptors as arguments. Any time multiple class entities or column-based entities are expressed as arguments to the *query()* function, the return result is expressed as tuples:

```
>>> for name, fullname in session.query(User.name, User.fullname):
...     print(name, fullname)
SELECT users.name AS users_name,
        users.fullname AS users_fullname
FROM users
()
ed Ed Jones
wendy Wendy Williams
mary Mary Contrary
fred Fred Flinstone
```

The tuples returned by *Query* are *named* tuples, supplied by the *KeyedTuple* class, and can be treated much like an ordinary Python object. The names are the same as the attribute's name for an attribute, and the class name for a class:

```
>>> for row in session.query(User, User.name).all():
...    print(row.User, row.name)
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
()
<User(name='ed', fullname='Ed Jones', password='f8s7ccs')> ed
<User(name='wendy', fullname='Wendy Williams', password='foobar')> wendy
<User(name='mary', fullname='Mary Contrary', password='xxg527')> mary
<User(name='fred', fullname='Fred Flinstone', password='blah')> fred
```

You can control the names of individual column expressions using the *label()* construct, which is available from any *ColumnElement*-derived object, as well as any class attribute which is mapped to one (such as `User.name`):

```
>>> for row in session.query(User.name.label('name_label')).all():
...    print(row.name_label)
SELECT users.name AS name_label
FROM users
()
ed
```

```
wendy
mary
fred
```

The name given to a full entity such as `User`, assuming that multiple entities are present in the call to *query()*, can be controlled using *aliased()* :

```
>>> from sqlalchemy.orm import aliased
>>> user_alias = aliased(User, name='user_alias')

>>> for row in session.query(user_alias, user_alias.name).all():
...     print(row.user_alias)
SELECT user_alias.id AS user_alias_id,
        user_alias.name AS user_alias_name,
        user_alias.fullname AS user_alias_fullname,
        user_alias.password AS user_alias_password
FROM users AS user_alias
()
<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>
<User(name='wendy', fullname='Wendy Williams', password='foobar')>
<User(name='mary', fullname='Mary Contrary', password='xxg527')>
<User(name='fred', fullname='Fred Flinstone', password='blah')>
```

Basic operations with *Query* include issuing LIMIT and OFFSET, most conveniently using Python array slices and typically in conjunction with ORDER BY:

```
>>> for u in session.query(User).order_by(User.id)[1:3]:
...     print(u)
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users ORDER BY users.id
LIMIT ? OFFSET ?
(2, 1)
<User(name='wendy', fullname='Wendy Williams', password='foobar')>
<User(name='mary', fullname='Mary Contrary', password='xxg527')>
```

and filtering results, which is accomplished either with *filter_by()*, which uses keyword arguments:

```
>>> for name, in session.query(User.name).\
...             filter_by(fullname='Ed Jones'):
...     print(name)
SELECT users.name AS users_name FROM users
WHERE users.fullname = ?
('Ed Jones',)
ed
```

...or *filter()*, which uses more flexible SQL expression language constructs. These allow you to use regular Python operators with the class-level attributes on your mapped class:

```
>>> for name, in session.query(User.name).\
...             filter(User.fullname=='Ed Jones'):
...     print(name)
SELECT users.name AS users_name FROM users
WHERE users.fullname = ?
('Ed Jones',)
ed
```

The *Query* object is fully **generative**, meaning that most method calls return a new *Query* object upon which

further criteria may be added. For example, to query for users named "ed" with a full name of "Ed Jones", you can call *filter()* twice, which joins criteria using AND:

```
>>> for user in session.query(User).\
...         filter(User.name=='ed').\
...         filter(User.fullname=='Ed Jones'):
...     print(user)
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE users.name = ? AND users.fullname = ?
('ed', 'Ed Jones')
<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>
```

Common Filter Operators

Here's a rundown of some of the most common operators used in *filter()*:

- *equals*:

```
query.filter(User.name == 'ed')
```

- *not equals*:

```
query.filter(User.name != 'ed')
```

- *LIKE*:

```
query.filter(User.name.like('%ed%'))
```

- *IN*:

```
query.filter(User.name.in_(['ed', 'wendy', 'jack']))

# works with query objects too:
query.filter(User.name.in_(
        session.query(User.name).filter(User.name.like('%ed%'))
))
```

- *NOT IN*:

```
query.filter(~User.name.in_(['ed', 'wendy', 'jack']))
```

- *IS NULL*:

```
query.filter(User.name == None)

# alternatively, if pep8/linters are a concern
query.filter(User.name.is_(None))
```

- *IS NOT NULL*:

```
query.filter(User.name != None)

# alternatively, if pep8/linters are a concern
query.filter(User.name.isnot(None))
```

- *AND*:

```
# use and_()
from sqlalchemy import and_
query.filter(and_(User.name == 'ed', User.fullname == 'Ed Jones'))

# or send multiple expressions to .filter()
query.filter(User.name == 'ed', User.fullname == 'Ed Jones')

# or chain multiple filter()/filter_by() calls
query.filter(User.name == 'ed').filter(User.fullname == 'Ed Jones')
```

Note: Make sure you use `and_()` and **not** the Python `and` operator!

- `OR`:

```
from sqlalchemy import or_
query.filter(or_(User.name == 'ed', User.name == 'wendy'))
```

Note: Make sure you use `or_()` and **not** the Python `or` operator!

- `MATCH`:

```
query.filter(User.name.match('wendy'))
```

Note: `match()` uses a database-specific `MATCH` or `CONTAINS` function; its behavior will vary by backend and is not available on some backends such as SQLite.

Returning Lists and Scalars

A number of methods on `Query` immediately issue SQL and return a value containing loaded database results. Here's a brief tour:

- `all()` returns a list:

```
>>> query = session.query(User).filter(User.name.like('%ed')).order_by(User.id)
>>> query.all()
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE users.name LIKE ? ORDER BY users.id
('%ed',)
[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>,
      <User(name='fred', fullname='Fred Flinstone', password='blah')>]
```

- `first()` applies a limit of one and returns the first result as a scalar:

```
>>> query.first()
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE users.name LIKE ? ORDER BY users.id
 LIMIT ? OFFSET ?
```

```
('%ed', 1, 0)
<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>
```

- *one()* fully fetches all rows, and if not exactly one object identity or composite row is present in the result, raises an error. With multiple rows found:

```
>>> from sqlalchemy.orm.exc import MultipleResultsFound
>>> try:
...     user = query.one()
... except MultipleResultsFound as e:
...     print(e)
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE users.name LIKE ? ORDER BY users.id
('%ed',)
Multiple rows were found for one()
```

With no rows found:

```
>>> from sqlalchemy.orm.exc import NoResultFound
>>> try:
...     user = query.filter(User.id == 99).one()
... except NoResultFound as e:
...     print(e)
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE users.name LIKE ? AND users.id = ? ORDER BY users.id
('%ed', 99)
No row was found for one()
```

The *one()* method is great for systems that expect to handle "no items found" versus "multiple items found" differently; such as a RESTful web service, which may want to raise a "404 not found" when no results are found, but raise an application error when multiple results are found.

- *one_or_none()* is like *one()*, except that if no results are found, it doesn't raise an error; it just returns `None`. Like *one()*, however, it does raise an error if multiple results are found.
- *scalar()* invokes the *one()* method, and upon success returns the first column of the row:

```
>>> query = session.query(User.id).filter(User.name == 'ed').\
...    order_by(User.id)
>>> query.scalar()
SELECT users.id AS users_id
FROM users
WHERE users.name = ? ORDER BY users.id
('ed',)
1
```

Using Textual SQL

Literal strings can be used flexibly with *Query*, by specifying their use with the *text()* construct, which is accepted by most applicable methods. For example, *filter()* and *order_by()*:

```
>>> from sqlalchemy import text
>>> for user in session.query(User).\
...             filter(text("id<224")).\
...             order_by(text("id")).all():
...     print(user.name)
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE id<224 ORDER BY id
()
ed
wendy
mary
fred
```

Bind parameters can be specified with string-based SQL, using a colon. To specify the values, use the *params()* method:

```
>>> session.query(User).filter(text("id<:value and name=:name")).\
...     params(value=224, name='fred').order_by(User.id).one()
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE id<? and name=? ORDER BY users.id
(224, 'fred')
<User(name='fred', fullname='Fred Flinstone', password='blah')>
```

To use an entirely string-based statement, using *from_statement()*; just ensure that the columns clause of the statement contains the column names normally used by the mapper (below illustrated using an asterisk):

```
>>> session.query(User).from_statement(
...                     text("SELECT * FROM users where name=:name")).\
...                     params(name='ed').all()
SELECT * FROM users where name=?
('ed',)
[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]
```

Or alternatively, specify how the columns map to the *text()* construct explicitly using the *TextClause.columns()* method:

```
>>> stmt = text("SELECT name, id FROM users where name=:name")
>>> stmt = stmt.columns(User.name, User.id)
>>> session.query(User).from_statement(stmt).params(name='ed').all()
SELECT name, id FROM users where name=?
('ed',)
[<User(name='ed', fullname='Ed Jones', password='f8s7ccs')>]
```

We can choose columns to return individually as well, as in any other case:

```
>>> stmt = text("SELECT name, id FROM users where name=:name")
>>> stmt = stmt.columns(User.name, User.id)
>>> session.query(User.id, User.name).\
...           from_statement(stmt).params(name='ed').all()
SELECT name, id FROM users where name=?
('ed',)
```

```
[(1, u'ed')]
```

See also:

Using Textual SQL - The `text()` construct explained from the perspective of Core-only queries.

Changed in version 1.0.0: The `Query` construct emits warnings when string SQL fragments are coerced to `text()`, and `text()` should be used explicitly. See *Warnings emitted when coercing full SQL fragments into text()* for background.

Counting

`Query` includes a convenience method for counting called `count()`:

```
>>> session.query(User).filter(User.name.like('%ed')).count()
SELECT count(*) AS count_1
FROM (SELECT users.id AS users_id,
                users.name AS users_name,
                users.fullname AS users_fullname,
                users.password AS users_password
FROM users
WHERE users.name LIKE ?) AS anon_1
('%ed',)
2
```

Counting on `count()`

`Query.count()` used to be a very complicated method when it would try to guess whether or not a subquery was needed around the existing query, and in some exotic cases it wouldn't do the right thing. Now that it uses a simple subquery every time, it's only two lines long and always returns the right answer. Use `func.count()` if a particular statement absolutely cannot tolerate the subquery being present.

The `count()` method is used to determine how many rows the SQL statement would return. Looking at the generated SQL above, SQLAlchemy always places whatever it is we are querying into a subquery, then counts the rows from that. In some cases this can be reduced to a simpler `SELECT count(*) FROM table`, however modern versions of SQLAlchemy don't try to guess when this is appropriate, as the exact SQL can be emitted using more explicit means.

For situations where the "thing to be counted" needs to be indicated specifically, we can specify the "count" function directly using the expression `func.count()`, available from the `func` construct. Below we use it to return the count of each distinct user name:

```
>>> from sqlalchemy import func
>>> session.query(func.count(User.name), User.name).group_by(User.name).all()
SELECT count(users.name) AS count_1, users.name AS users_name
FROM users GROUP BY users.name
()
[(1, u'ed'), (1, u'fred'), (1, u'mary'), (1, u'wendy')]
```

To achieve our simple `SELECT count(*) FROM table`, we can apply it as:

```
>>> session.query(func.count('*')).select_from(User).scalar()
SELECT count(?) AS count_1
FROM users
('*',)
4
```

The usage of *select_from()* can be removed if we express the count in terms of the User primary key directly:

```
>>> session.query(func.count(User.id)).scalar()
SELECT count(users.id) AS count_1
FROM users
()
4
```

2.1.10 Building a Relationship

Let's consider how a second table, related to User, can be mapped and queried. Users in our system can store any number of email addresses associated with their username. This implies a basic one to many association from the users to a new table which stores email addresses, which we will call addresses. Using declarative, we define this table along with its mapped class, Address:

```
>>> from sqlalchemy import ForeignKey
>>> from sqlalchemy.orm import relationship

>>> class Address(Base):
...     __tablename__ = 'addresses'
...     id = Column(Integer, primary_key=True)
...     email_address = Column(String, nullable=False)
...     user_id = Column(Integer, ForeignKey('users.id'))
...
...     user = relationship("User", back_populates="addresses")
...
...     def __repr__(self):
...         return "<Address(email_address='%s')>" % self.email_address

>>> User.addresses = relationship(
...     "Address", order_by=Address.id, back_populates="user")
```

The above class introduces the *ForeignKey* construct, which is a directive applied to *Column* that indicates that values in this column should be constrained to be values present in the named remote column. This is a core feature of relational databases, and is the "glue" that transforms an otherwise unconnected collection of tables to have rich overlapping relationships. The *ForeignKey* above expresses that values in the addresses.user_id column should be constrained to those values in the users.id column, i.e. its primary key.

A second directive, known as *relationship()*, tells the ORM that the Address class itself should be linked to the User class, using the attribute Address.user. *relationship()* uses the foreign key relationships between the two tables to determine the nature of this linkage, determining that Address.user will be many to one. An additional *relationship()* directive is placed on the User mapped class under the attribute User.addresses. In both *relationship()* directives, the parameter *relationship.back_populates* is assigned to refer to the complementary attribute names; by doing so, each *relationship()* can make intelligent decision about the same relationship as expressed in reverse; on one side, Address.user refers to a User instance, and on the other side, User.addresses refers to a list of Address instances.

Note: The *relationship.back_populates* parameter is a newer version of a very common SQLAlchemy feature called *relationship.backref*. The *relationship.backref* parameter hasn't gone anywhere and will always remain available! The *relationship.back_populates* is the same thing, except a little more verbose and easier to manipulate. For an overview of the entire topic, see the section *Linking Relationships with Backref*.

The reverse side of a many-to-one relationship is always one to many. A full catalog of available *relationship()* configurations is at *Basic Relationship Patterns*.

The two complementing relationships `Address.user` and `User.addresses` are referred to as a bidirectional relationship, and is a key feature of the SQLAlchemy ORM. The section *Linking Relationships with Backref* discusses the "backref" feature in detail.

Arguments to `relationship()` which concern the remote class can be specified using strings, assuming the Declarative system is in use. Once all mappings are complete, these strings are evaluated as Python expressions in order to produce the actual argument, in the above case the `User` class. The names which are allowed during this evaluation include, among other things, the names of all classes which have been created in terms of the declared base.

See the docstring for `relationship()` for more detail on argument style.

Did you know ?

- a FOREIGN KEY constraint in most (though not all) relational databases can only link to a primary key column, or a column that has a UNIQUE constraint.
- a FOREIGN KEY constraint that refers to a multiple column primary key, and itself has multiple columns, is known as a "composite foreign key". It can also reference a subset of those columns.
- FOREIGN KEY columns can automatically update themselves, in response to a change in the referenced column or row. This is known as the CASCADE *referential action*, and is a built in function of the relational database.
- FOREIGN KEY can refer to its own table. This is referred to as a "self-referential" foreign key.
- Read more about foreign keys at Foreign Key - Wikipedia.

We'll need to create the `addresses` table in the database, so we will issue another CREATE from our metadata, which will skip over tables which have already been created:

```
>>> Base.metadata.create_all(engine)
PRAGMA...
CREATE TABLE addresses (
    id INTEGER NOT NULL,
    email_address VARCHAR NOT NULL,
    user_id INTEGER,
    PRIMARY KEY (id),
     FOREIGN KEY(user_id) REFERENCES users (id)
)
()
COMMIT
```

2.1.11 Working with Related Objects

Now when we create a `User`, a blank `addresses` collection will be present. Various collection types, such as sets and dictionaries, are possible here (see *Customizing Collection Access* for details), but by default, the collection is a Python list.

```
>>> jack = User(name='jack', fullname='Jack Bean', password='gjffdd')
>>> jack.addresses
[]
```

We are free to add `Address` objects on our `User` object. In this case we just assign a full list directly:

```
>>> jack.addresses = [
...                 Address(email_address='jack@google.com'),
...                 Address(email_address='j25@yahoo.com')]
```

When using a bidirectional relationship, elements added in one direction automatically become visible in the other direction. This behavior occurs based on attribute on-change events and is evaluated in Python, without using any

SQL:

```
>>> jack.addresses[1]
<Address(email_address='j25@yahoo.com')>

>>> jack.addresses[1].user
<User(name='jack', fullname='Jack Bean', password='gjffdd')>
```

Let's add and commit `Jack Bean` to the database. `jack` as well as the two `Address` members in the corresponding `addresses` collection are both added to the session at once, using a process known as **cascading**:

```
>>> session.add(jack)
>>> session.commit()
INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)
('jack', 'Jack Bean', 'gjffdd')
INSERT INTO addresses (email_address, user_id) VALUES (?, ?)
('jack@google.com', 5)
INSERT INTO addresses (email_address, user_id) VALUES (?, ?)
('j25@yahoo.com', 5)
COMMIT
```

Querying for Jack, we get just Jack back. No SQL is yet issued for Jack's addresses:

```
>>> jack = session.query(User).\
... filter_by(name='jack').one()
BEGIN (implicit)
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE users.name = ?
('jack',)

>>> jack
<User(name='jack', fullname='Jack Bean', password='gjffdd')>
```

Let's look at the `addresses` collection. Watch the SQL:

```
>>> jack.addresses
SELECT addresses.id AS addresses_id,
        addresses.email_address AS
        addresses_email_address,
        addresses.user_id AS addresses_user_id
FROM addresses
WHERE ? = addresses.user_id ORDER BY addresses.id
(5,)
[<Address(email_address='jack@google.com')>, <Address(email_address='j25@yahoo.com')>]
```

When we accessed the `addresses` collection, SQL was suddenly issued. This is an example of a lazy loading relationship. The `addresses` collection is now loaded and behaves just like an ordinary list. We'll cover ways to optimize the loading of this collection in a bit.

2.1.12 Querying with Joins

Now that we have two tables, we can show some more features of *`Query`*, specifically how to create queries that deal with both tables at the same time. The Wikipedia page on SQL JOIN offers a good introduction to join techniques, several of which we'll illustrate here.

To construct a simple implicit join between `User` and `Address`, we can use *Query.filter()* to equate their related columns together. Below we load the `User` and `Address` entities at once using this method:

```
>>> for u, a in session.query(User, Address).\
...                      filter(User.id==Address.user_id).\
...                      filter(Address.email_address=='jack@google.com').\
...                      all():
...     print(u)
...     print(a)
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password,
        addresses.id AS addresses_id,
        addresses.email_address AS addresses_email_address,
        addresses.user_id AS addresses_user_id
FROM users, addresses
WHERE users.id = addresses.user_id
        AND addresses.email_address = ?
('jack@google.com',)
<User(name='jack', fullname='Jack Bean', password='gjffdd')>
<Address(email_address='jack@google.com')>
```

The actual SQL JOIN syntax, on the other hand, is most easily achieved using the *Query.join()* method:

```
>>> session.query(User).join(Address).\
...         filter(Address.email_address=='jack@google.com').\
...         all()
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users JOIN addresses ON users.id = addresses.user_id
WHERE addresses.email_address = ?
('jack@google.com',)
[<User(name='jack', fullname='Jack Bean', password='gjffdd')>]
```

Query.join() knows how to join between `User` and `Address` because there's only one foreign key between them. If there were no foreign keys, or several, *Query.join()* works better when one of the following forms are used:

```
query.join(Address, User.id==Address.user_id)     # explicit condition
query.join(User.addresses)                        # specify relationship from left to right
query.join(Address, User.addresses)               # same, with explicit target
query.join('addresses')                           # same, using a string
```

As you would expect, the same idea is used for "outer" joins, using the *outerjoin()* function:

```
query.outerjoin(User.addresses)    # LEFT OUTER JOIN
```

The reference documentation for *join()* contains detailed information and examples of the calling styles accepted by this method; *join()* is an important method at the center of usage for any SQL-fluent application.

What does `Query` select from if there's multiple entities?

The *Query.join()* method will **typically join from the leftmost item** in the list of entities, when the ON clause is omitted, or if the ON clause is a plain SQL expression. To control the first entity in the list of JOINs, use the *Query.select_from()* method:

```
query = Session.query(User, Address).select_from(Address).join(User)
```

Using Aliases

When querying across multiple tables, if the same table needs to be referenced more than once, SQL typically requires that the table be *aliased* with another name, so that it can be distinguished against other occurrences of that table. The *Query* supports this most explicitly using the *aliased* construct. Below we join to the `Address` entity twice, to locate a user who has two distinct email addresses at the same time:

```
>>> from sqlalchemy.orm import aliased
>>> adalias1 = aliased(Address)
>>> adalias2 = aliased(Address)
>>> for username, email1, email2 in \
...     session.query(User.name, adalias1.email_address, adalias2.email_address).\
...     join(adalias1, User.addresses).\
...     join(adalias2, User.addresses).\
...     filter(adalias1.email_address=='jack@google.com').\
...     filter(adalias2.email_address=='j25@yahoo.com'):
...     print(username, email1, email2)
SELECT users.name AS users_name,
        addresses_1.email_address AS addresses_1_email_address,
        addresses_2.email_address AS addresses_2_email_address
FROM users JOIN addresses AS addresses_1
        ON users.id = addresses_1.user_id
JOIN addresses AS addresses_2
        ON users.id = addresses_2.user_id
WHERE addresses_1.email_address = ?
        AND addresses_2.email_address = ?
('jack@google.com', 'j25@yahoo.com')
jack jack@google.com j25@yahoo.com
```

Using Subqueries

The *Query* is suitable for generating statements which can be used as subqueries. Suppose we wanted to load `User` objects along with a count of how many `Address` records each user has. The best way to generate SQL like this is to get the count of addresses grouped by user ids, and JOIN to the parent. In this case we use a LEFT OUTER JOIN so that we get rows back for those users who don't have any addresses, e.g.:

```
SELECT users.*, adr_count.address_count FROM users LEFT OUTER JOIN
    (SELECT user_id, count(*) AS address_count
        FROM addresses GROUP BY user_id) AS adr_count
    ON users.id=adr_count.user_id
```

Using the *Query*, we build a statement like this from the inside out. The `statement` accessor returns a SQL expression representing the statement generated by a particular *Query* - this is an instance of a *select()* construct, which are described in *SQL Expression Language Tutorial*:

```
>>> from sqlalchemy.sql import func
>>> stmt = session.query(Address.user_id, func.count('*').\
```

```
...         label('address_count')).\
...         group_by(Address.user_id).subquery()
```

The `func` keyword generates SQL functions, and the `subquery()` method on *Query* produces a SQL expression construct representing a SELECT statement embedded within an alias (it's actually shorthand for `query.statement.alias()`).

Once we have our statement, it behaves like a *Table* construct, such as the one we created for `users` at the start of this tutorial. The columns on the statement are accessible through an attribute called `c`:

```
>>> for u, count in session.query(User, stmt.c.address_count).\
...     outerjoin(stmt, User.id==stmt.c.user_id).order_by(User.id):
...     print(u, count)
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password,
        anon_1.address_count AS anon_1_address_count
FROM users LEFT OUTER JOIN
    (SELECT addresses.user_id AS user_id, count(?) AS address_count
    FROM addresses GROUP BY addresses.user_id) AS anon_1
    ON users.id = anon_1.user_id
ORDER BY users.id
('*',)
<User(name='ed', fullname='Ed Jones', password='f8s7ccs')> None
<User(name='wendy', fullname='Wendy Williams', password='foobar')> None
<User(name='mary', fullname='Mary Contrary', password='xxg527')> None
<User(name='fred', fullname='Fred Flinstone', password='blah')> None
<User(name='jack', fullname='Jack Bean', password='gjffdd')> 2
```

Selecting Entities from Subqueries

Above, we just selected a result that included a column from a subquery. What if we wanted our subquery to map to an entity ? For this we use `aliased()` to associate an "alias" of a mapped class to a subquery:

```
>>> stmt = session.query(Address).\
...                 filter(Address.email_address != 'j25@yahoo.com').\
...                 subquery()
>>> adalias = aliased(Address, stmt)
>>> for user, address in session.query(User, adalias).\
...         join(adalias, User.addresses):
...     print(user)
...     print(address)
SELECT users.id AS users_id,
            users.name AS users_name,
            users.fullname AS users_fullname,
            users.password AS users_password,
            anon_1.id AS anon_1_id,
            anon_1.email_address AS anon_1_email_address,
            anon_1.user_id AS anon_1_user_id
FROM users JOIN
    (SELECT addresses.id AS id,
            addresses.email_address AS email_address,
            addresses.user_id AS user_id
    FROM addresses
    WHERE addresses.email_address != ?) AS anon_1
    ON users.id = anon_1.user_id
```

```
('j25@yahoo.com',)
<User(name='jack', fullname='Jack Bean', password='gjffdd')>
<Address(email_address='jack@google.com')>
```

Using EXISTS

The EXISTS keyword in SQL is a boolean operator which returns True if the given expression contains any rows. It may be used in many scenarios in place of joins, and is also useful for locating rows which do not have a corresponding row in a related table.

There is an explicit EXISTS construct, which looks like this:

```
>>> from sqlalchemy.sql import exists
>>> stmt = exists().where(Address.user_id==User.id)
>>> for name, in session.query(User.name).filter(stmt):
...     print(name)
SELECT users.name AS users_name
FROM users
WHERE EXISTS (SELECT *
FROM addresses
WHERE addresses.user_id = users.id)
()
jack
```

The *Query* features several operators which make usage of EXISTS automatically. Above, the statement can be expressed along the `User.addresses` relationship using *any()*:

```
>>> for name, in session.query(User.name).\
...         filter(User.addresses.any()):
...     print(name)
SELECT users.name AS users_name
FROM users
WHERE EXISTS (SELECT 1
FROM addresses
WHERE users.id = addresses.user_id)
()
jack
```

any() takes criterion as well, to limit the rows matched:

```
>>> for name, in session.query(User.name).\
...     filter(User.addresses.any(Address.email_address.like('%google%'))):
...     print(name)
SELECT users.name AS users_name
FROM users
WHERE EXISTS (SELECT 1
FROM addresses
WHERE users.id = addresses.user_id AND addresses.email_address LIKE ?)
('%google%',)
jack
```

has() is the same operator as *any()* for many-to-one relationships (note the ~ operator here too, which means "NOT"):

```
>>> session.query(Address).\
...         filter(~Address.user.has(User.name=='jack')).all()
SELECT addresses.id AS addresses_id,
        addresses.email_address AS addresses_email_address,
```

```
        addresses.user_id AS addresses_user_id
FROM addresses
WHERE NOT (EXISTS (SELECT 1
FROM users
WHERE users.id = addresses.user_id AND users.name = ?))
('jack',)
[]
```

Common Relationship Operators

Here's all the operators which build on relationships - each one is linked to its API documentation which includes full details on usage and behavior:

- *__eq__()* (many-to-one "equals" comparison):

```
query.filter(Address.user == someuser)
```

- *__ne__()* (many-to-one "not equals" comparison):

```
query.filter(Address.user != someuser)
```

- IS NULL (many-to-one comparison, also uses *__eq__()*):

```
query.filter(Address.user == None)
```

- *contains()* (used for one-to-many collections):

```
query.filter(User.addresses.contains(someaddress))
```

- *any()* (used for collections):

```
query.filter(User.addresses.any(Address.email_address == 'bar'))

# also takes keyword arguments:
query.filter(User.addresses.any(email_address='bar'))
```

- *has()* (used for scalar references):

```
query.filter(Address.user.has(name='ed'))
```

- *Query.with_parent()* (used for any relationship):

```
session.query(Address).with_parent(someuser, 'addresses')
```

2.1.13 Eager Loading

Recall earlier that we illustrated a lazy loading operation, when we accessed the `User.addresses` collection of a `User` and SQL was emitted. If you want to reduce the number of queries (dramatically, in many cases), we can apply an eager load to the query operation. SQLAlchemy offers three types of eager loading, two of which are automatic, and a third which involves custom criterion. All three are usually invoked via functions known as query options which give additional instructions to the *Query* on how we would like various attributes to be loaded, via the *Query.options()* method.

Subquery Load

In this case we'd like to indicate that `User.addresses` should load eagerly. A good choice for loading a set of objects as well as their related collections is the *orm.subqueryload()* option, which emits a second SELECT

statement that fully loads the collections associated with the results just loaded. The name "subquery" originates from the fact that the SELECT statement constructed directly via the *Query* is re-used, embedded as a subquery into a SELECT against the related table. This is a little elaborate but very easy to use:

```
>>> from sqlalchemy.orm import subqueryload
>>> jack = session.query(User).\
...                 options(subqueryload(User.addresses)).\
...                 filter_by(name='jack').one()
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE users.name = ?
('jack',)
SELECT addresses.id AS addresses_id,
        addresses.email_address AS addresses_email_address,
        addresses.user_id AS addresses_user_id,
        anon_1.users_id AS anon_1_users_id
FROM (SELECT users.id AS users_id
    FROM users WHERE users.name = ?) AS anon_1
JOIN addresses ON anon_1.users_id = addresses.user_id
ORDER BY anon_1.users_id, addresses.id
('jack',)
>>> jack
<User(name='jack', fullname='Jack Bean', password='gjffdd')>

>>> jack.addresses
[<Address(email_address='jack@google.com')>, <Address(email_address='j25@yahoo.com')>]
```

Note: *subqueryload()* when used in conjunction with limiting such as *Query.first()*, *Query.limit()* or *Query.offset()* should also include *Query.order_by()* on a unique column in order to ensure correct results. See *The Importance of Ordering*.

Joined Load

The other automatic eager loading function is more well known and is called *orm.joinedload()*. This style of loading emits a JOIN, by default a LEFT OUTER JOIN, so that the lead object as well as the related object or collection is loaded in one step. We illustrate loading the same `addresses` collection in this way - note that even though the `User.addresses` collection on `jack` is actually populated right now, the query will emit the extra join regardless:

```
>>> from sqlalchemy.orm import joinedload

>>> jack = session.query(User).\
...                         options(joinedload(User.addresses)).\
...                         filter_by(name='jack').one()
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password,
        addresses_1.id AS addresses_1_id,
        addresses_1.email_address AS addresses_1_email_address,
        addresses_1.user_id AS addresses_1_user_id
FROM users
    LEFT OUTER JOIN addresses AS addresses_1 ON users.id = addresses_1.user_id
```

```
WHERE users.name = ? ORDER BY addresses_1.id
('jack',)

>>> jack
<User(name='jack', fullname='Jack Bean', password='gjffdd')>

>>> jack.addresses
[<Address(email_address='jack@google.com')>, <Address(email_address='j25@yahoo.com')>]
```

Note that even though the OUTER JOIN resulted in two rows, we still only got one instance of `User` back. This is because *`Query`* applies a "uniquing" strategy, based on object identity, to the returned entities. This is specifically so that joined eager loading can be applied without affecting the query results.

While *`joinedload()`* has been around for a long time, *`subqueryload()`* is a newer form of eager loading. *`subqueryload()`* tends to be more appropriate for loading related collections while *`joinedload()`* tends to be better suited for many-to-one relationships, due to the fact that only one row is loaded for both the lead and the related object.

`joinedload()` is not a replacement for `join()`

The join created by *`joinedload()`* is anonymously aliased such that it **does not affect the query results**. An *`Query.order_by()`* or *`Query.filter()`* call **cannot** reference these aliased tables - so-called "user space" joins are constructed using *`Query.join()`*. The rationale for this is that *`joinedload()`* is only applied in order to affect how related objects or collections are loaded as an optimizing detail - it can be added or removed with no impact on actual results. See the section *The Zen of Eager Loading* for a detailed description of how this is used.

Explicit Join + Eagerload

A third style of eager loading is when we are constructing a JOIN explicitly in order to locate the primary rows, and would like to additionally apply the extra table to a related object or collection on the primary object. This feature is supplied via the *`orm.contains_eager()`* function, and is most typically useful for pre-loading the many-to-one object on a query that needs to filter on that same object. Below we illustrate loading an `Address` row as well as the related `User` object, filtering on the `User` named "jack" and using *`orm.contains_eager()`* to apply the "user" columns to the `Address.user` attribute:

```
>>> from sqlalchemy.orm import contains_eager
>>> jacks_addresses = session.query(Address).\
...                             join(Address.user).\
...                             filter(User.name=='jack').\
...                             options(contains_eager(Address.user)).\
...                             all()
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password,
        addresses.id AS addresses_id,
        addresses.email_address AS addresses_email_address,
        addresses.user_id AS addresses_user_id
FROM addresses JOIN users ON users.id = addresses.user_id
WHERE users.name = ?
('jack',)

>>> jacks_addresses
[<Address(email_address='jack@google.com')>, <Address(email_address='j25@yahoo.com')>]
```

```
>>> jacks_addresses[0].user
<User(name='jack', fullname='Jack Bean', password='gjffdd')>
```

For more information on eager loading, including how to configure various forms of loading by default, see the section Relationship Loading Techniques.

2.1.14 Deleting

Let's try to delete `jack` and see how that goes. We'll mark as deleted in the session, then we'll issue a `count` query to see that no rows remain:

```
>>> session.delete(jack)
>>> session.query(User).filter_by(name='jack').count()
UPDATE addresses SET user_id=? WHERE addresses.id = ?
((None, 1), (None, 2))
DELETE FROM users WHERE users.id = ?
(5,)
SELECT count(*) AS count_1
FROM (SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE users.name = ?) AS anon_1
('jack',)
0
```

So far, so good. How about Jack's `Address` objects ?

```
>>> session.query(Address).filter(
...     Address.email_address.in_(['jack@google.com', 'j25@yahoo.com'])
...  ).count()
SELECT count(*) AS count_1
FROM (SELECT addresses.id AS addresses_id,
                addresses.email_address AS addresses_email_address,
                addresses.user_id AS addresses_user_id
FROM addresses
WHERE addresses.email_address IN (?, ?)) AS anon_1
('jack@google.com', 'j25@yahoo.com')
2
```

Uh oh, they're still there ! Analyzing the flush SQL, we can see that the `user_id` column of each address was set to NULL, but the rows weren't deleted. SQLAlchemy doesn't assume that deletes cascade, you have to tell it to do so.

Configuring delete/delete-orphan Cascade

We will configure **cascade** options on the `User.addresses` relationship to change the behavior. While SQLAlchemy allows you to add new attributes and relationships to mappings at any point in time, in this case the existing relationship needs to be removed, so we need to tear down the mappings completely and start again - we'll close the *`Session`*:

```
>>> session.close()
ROLLBACK
```

and use a new *`declarative_base()`*:

```
>>> Base = declarative_base()
```

Next we'll declare the `User` class, adding in the `addresses` relationship including the cascade configuration (we'll leave the constructor out too):

```
>>> class User(Base):
...     __tablename__ = 'users'
...
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     fullname = Column(String)
...     password = Column(String)
...
...     addresses = relationship("Address", back_populates='user',
...                      cascade="all, delete, delete-orphan")
...
...     def __repr__(self):
...        return "<User(name='%s', fullname='%s', password='%s')>" % (
...                                 self.name, self.fullname, self.password)
```

Then we recreate `Address`, noting that in this case we've created the `Address.user` relationship via the `User` class already:

```
>>> class Address(Base):
...     __tablename__ = 'addresses'
...     id = Column(Integer, primary_key=True)
...     email_address = Column(String, nullable=False)
...     user_id = Column(Integer, ForeignKey('users.id'))
...     user = relationship("User", back_populates="addresses")
...
...     def __repr__(self):
...         return "<Address(email_address='%s')>" % self.email_address
```

Now when we load the user `jack` (below using *`get()`*, which loads by primary key), removing an address from the corresponding `addresses` collection will result in that `Address` being deleted:

```
# load Jack by primary key
>>> jack = session.query(User).get(5)
BEGIN (implicit)
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE users.id = ?
(5,)


# remove one Address (lazy load fires off)
>>> del jack.addresses[1]
SELECT addresses.id AS addresses_id,
        addresses.email_address AS addresses_email_address,
        addresses.user_id AS addresses_user_id
FROM addresses
WHERE ? = addresses.user_id
(5,)


# only one address remains
```

```
>>> session.query(Address).filter(
...     Address.email_address.in_(['jack@google.com', 'j25@yahoo.com'])
... ).count()
DELETE FROM addresses WHERE addresses.id = ?
(2,)
SELECT count(*) AS count_1
FROM (SELECT addresses.id AS addresses_id,
                addresses.email_address AS addresses_email_address,
                addresses.user_id AS addresses_user_id
FROM addresses
WHERE addresses.email_address IN (?, ?)) AS anon_1
('jack@google.com', 'j25@yahoo.com')
1
```

Deleting Jack will delete both Jack and the remaining `Address` associated with the user:

```
>>> session.delete(jack)

>>> session.query(User).filter_by(name='jack').count()
DELETE FROM addresses WHERE addresses.id = ?
(1,)
DELETE FROM users WHERE users.id = ?
(5,)
SELECT count(*) AS count_1
FROM (SELECT users.id AS users_id,
                users.name AS users_name,
                users.fullname AS users_fullname,
                users.password AS users_password
FROM users
WHERE users.name = ?) AS anon_1
('jack',)
0

>>> session.query(Address).filter(
...    Address.email_address.in_(['jack@google.com', 'j25@yahoo.com'])
... ).count()
SELECT count(*) AS count_1
FROM (SELECT addresses.id AS addresses_id,
                addresses.email_address AS addresses_email_address,
                addresses.user_id AS addresses_user_id
FROM addresses
WHERE addresses.email_address IN (?, ?)) AS anon_1
('jack@google.com', 'j25@yahoo.com')
0
```

More on Cascades

Further detail on configuration of cascades is at *Cascades*. The cascade functionality can also integrate smoothly with the `ON DELETE CASCADE` functionality of the relational database. See *Using Passive Deletes* for details.

2.1.15 Building a Many To Many Relationship

We're moving into the bonus round here, but lets show off a many-to-many relationship. We'll sneak in some other features too, just to take a tour. We'll make our application a blog application, where users can write `BlogPost` items, which have `Keyword` items associated with them.

For a plain many-to-many, we need to create an un-mapped *Table* construct to serve as the association table. This looks like the following:

```
>>> from sqlalchemy import Table, Text
>>> # association table
>>> post_keywords = Table('post_keywords', Base.metadata,
...     Column('post_id', ForeignKey('posts.id'), primary_key=True),
...     Column('keyword_id', ForeignKey('keywords.id'), primary_key=True)
... )
```

Above, we can see declaring a *Table* directly is a little different than declaring a mapped class. *Table* is a constructor function, so each individual *Column* argument is separated by a comma. The *Column* object is also given its name explicitly, rather than it being taken from an assigned attribute name.

Next we define `BlogPost` and `Keyword`, using complementary *relationship()* constructs, each referring to the `post_keywords` table as an association table:

```
>>> class BlogPost(Base):
...     __tablename__ = 'posts'
...
...     id = Column(Integer, primary_key=True)
...     user_id = Column(Integer, ForeignKey('users.id'))
...     headline = Column(String(255), nullable=False)
...     body = Column(Text)
...
...     # many to many BlogPost<->Keyword
...     keywords = relationship('Keyword',
...                             secondary=post_keywords,
...                             back_populates='posts')
...
...     def __init__(self, headline, body, author):
...         self.author = author
...         self.headline = headline
...         self.body = body
...
...     def __repr__(self):
...         return "BlogPost(%r, %r, %r)" % (self.headline, self.body, self.author)


>>> class Keyword(Base):
...     __tablename__ = 'keywords'
...
...     id = Column(Integer, primary_key=True)
...     keyword = Column(String(50), nullable=False, unique=True)
...     posts = relationship('BlogPost',
...                          secondary=post_keywords,
...                          back_populates='keywords')
...
...     def __init__(self, keyword):
...         self.keyword = keyword
```

Note: The above class declarations illustrate explicit `__init__()` methods. Remember, when using Declarative, it's optional!

Above, the many-to-many relationship is `BlogPost.keywords`. The defining feature of a many-to-many relationship is the `secondary` keyword argument which references a *Table* object representing the association table. This table only contains columns which reference the two sides of the relationship; if it has *any* other columns, such as its own primary key, or foreign keys to other tables, SQLAlchemy requires a different usage pattern called the

"association object", described at *Association Object*.

We would also like our `BlogPost` class to have an `author` field. We will add this as another bidirectional relationship, except one issue we'll have is that a single user might have lots of blog posts. When we access `User.posts`, we'd like to be able to filter results further so as not to load the entire collection. For this we use a setting accepted by `relationship()` called `lazy='dynamic'`, which configures an alternate **loader strategy** on the attribute:

```
.. sourcecode:: python+sql
```

```
>>> BlogPost.author = relationship(User, back_populates="posts")
>>> User.posts = relationship(BlogPost, back_populates="author", lazy="dynamic")
```

Create new tables:

```
>>> Base.metadata.create_all(engine)
PRAGMA...
CREATE TABLE keywords (
    id INTEGER NOT NULL,
    keyword VARCHAR(50) NOT NULL,
    PRIMARY KEY (id),
    UNIQUE (keyword)
)
()
COMMIT
CREATE TABLE posts (
    id INTEGER NOT NULL,
    user_id INTEGER,
    headline VARCHAR(255) NOT NULL,
    body TEXT,
    PRIMARY KEY (id),
    FOREIGN KEY(user_id) REFERENCES users (id)
)
()
COMMIT
CREATE TABLE post_keywords (
    post_id INTEGER NOT NULL,
    keyword_id INTEGER NOT NULL,
    PRIMARY KEY (post_id, keyword_id),
    FOREIGN KEY(post_id) REFERENCES posts (id),
    FOREIGN KEY(keyword_id) REFERENCES keywords (id)
)
()
COMMIT
```

Usage is not too different from what we've been doing. Let's give Wendy some blog posts:

```
>>> wendy = session.query(User).\
...                 filter_by(name='wendy').\
...                 one()
SELECT users.id AS users_id,
        users.name AS users_name,
        users.fullname AS users_fullname,
        users.password AS users_password
FROM users
WHERE users.name = ?
('wendy',)

>>> post = BlogPost("Wendy's Blog Post", "This is a test", wendy)
>>> session.add(post)
```

We're storing keywords uniquely in the database, but we know that we don't have any yet, so we can just create them:

```
>>> post.keywords.append(Keyword('wendy'))
>>> post.keywords.append(Keyword('firstpost'))
```

We can now look up all blog posts with the keyword 'firstpost'. We'll use the `any` operator to locate "blog posts where any of its keywords has the keyword string 'firstpost'":

```
>>> session.query(BlogPost).\
...             filter(BlogPost.keywords.any(keyword='firstpost')).\
...             all()
INSERT INTO keywords (keyword) VALUES (?)
('wendy',)
INSERT INTO keywords (keyword) VALUES (?)
('firstpost',)
INSERT INTO posts (user_id, headline, body) VALUES (?, ?, ?)
(2, "Wendy's Blog Post", 'This is a test')
INSERT INTO post_keywords (post_id, keyword_id) VALUES (?, ?)
(...)
SELECT posts.id AS posts_id,
        posts.user_id AS posts_user_id,
        posts.headline AS posts_headline,
        posts.body AS posts_body
FROM posts
WHERE EXISTS (SELECT 1
    FROM post_keywords, keywords
    WHERE posts.id = post_keywords.post_id
        AND keywords.id = post_keywords.keyword_id
        AND keywords.keyword = ?)
('firstpost',)
[BlogPost("Wendy's Blog Post", 'This is a test', <User(name='wendy', fullname='Wendy Williams', passw
```

If we want to look up posts owned by the user `wendy`, we can tell the query to narrow down to that `User` object as a parent:

```
>>> session.query(BlogPost).\
...             filter(BlogPost.author==wendy).\
...             filter(BlogPost.keywords.any(keyword='firstpost')).\
...             all()
SELECT posts.id AS posts_id,
        posts.user_id AS posts_user_id,
        posts.headline AS posts_headline,
        posts.body AS posts_body
FROM posts
WHERE ? = posts.user_id AND (EXISTS (SELECT 1
    FROM post_keywords, keywords
    WHERE posts.id = post_keywords.post_id
        AND keywords.id = post_keywords.keyword_id
        AND keywords.keyword = ?))
(2, 'firstpost')
[BlogPost("Wendy's Blog Post", 'This is a test', <User(name='wendy', fullname='Wendy Williams', passw
```

Or we can use Wendy's own `posts` relationship, which is a "dynamic" relationship, to query straight from there:

```
>>> wendy.posts.\
...         filter(BlogPost.keywords.any(keyword='firstpost')).\
...         all()
SELECT posts.id AS posts_id,
        posts.user_id AS posts_user_id,
        posts.headline AS posts_headline,
```

```
        posts.body AS posts_body
FROM posts
WHERE ? = posts.user_id AND (EXISTS (SELECT 1
    FROM post_keywords, keywords
    WHERE posts.id = post_keywords.post_id
        AND keywords.id = post_keywords.keyword_id
        AND keywords.keyword = ?))
(2, 'firstpost')
[BlogPost("Wendy's Blog Post", 'This is a test', <User(name='wendy', fullname='Wendy Williams', passw
```

2.1.16 Further Reference

Query Reference: query_api_toplevel

Mapper Reference: *Mapper Configuration*

Relationship Reference: *Relationship Configuration*

Session Reference: Using the Session

2.2 Mapper Configuration

This section describes a variety of configurational patterns that are usable with mappers. It assumes you've worked through *Object Relational Tutorial* and know how to construct and use rudimentary mappers and relationships.

2.2.1 Types of Mappings

Modern SQLAlchemy features two distinct styles of mapper configuration. The "Classical" style is SQLAlchemy's original mapping API, whereas "Declarative" is the richer and more succinct system that builds on top of "Classical". Both styles may be used interchangeably, as the end result of each is exactly the same - a user-defined class mapped by the `mapper()` function onto a selectable unit, typically a `Table`.

Declarative Mapping

The *Declarative Mapping* is the typical way that mappings are constructed in modern SQLAlchemy. Making use of the *Declarative* system, the components of the user-defined class as well as the `Table` metadata to which the class is mapped are defined at once:

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, ForeignKey

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column(String)
```

Above, a basic single-table mapping with four columns. Additional attributes, such as relationships to other mapped classes, are also declared inline within the class definition:

```
class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column(String)

    addresses = relationship("Address", backref="user", order_by="Address.id")

class Address(Base):
    __tablename__ = 'address'

    id = Column(Integer, primary_key=True)
    user_id = Column(ForeignKey('user.id'))
    email_address = Column(String)
```

The declarative mapping system is introduced in the *Object Relational Tutorial*. For additional details on how this system works, see *Declarative*.

Classical Mappings

A *Classical Mapping* refers to the configuration of a mapped class using the `mapper()` function, without using the Declarative system. This is SQLAlchemy's original class mapping API, and is still the base mapping system provided by the ORM.

In "classical" form, the table metadata is created separately with the `Table` construct, then associated with the `User` class via the `mapper()` function:

```
from sqlalchemy import Table, MetaData, Column, Integer, String, ForeignKey
from sqlalchemy.orm import mapper

metadata = MetaData()

user = Table('user', metadata,
            Column('id', Integer, primary_key=True),
            Column('name', String(50)),
            Column('fullname', String(50)),
            Column('password', String(12))
        )

class User(object):
    def __init__(self, name, fullname, password):
        self.name = name
        self.fullname = fullname
        self.password = password

mapper(User, user)
```

Information about mapped attributes, such as relationships to other classes, are provided via the `properties` dictionary. The example below illustrates a second `Table` object, mapped to a class called `Address`, then linked to `User` via `relationship()`:

```
address = Table('address', metadata,
            Column('id', Integer, primary_key=True),
            Column('user_id', Integer, ForeignKey('user.id')),
            Column('email_address', String(50))
```

```
            )

mapper(User, user, properties={
    'addresses' : relationship(Address, backref='user', order_by=address.c.id)
})

mapper(Address, address)
```

When using classical mappings, classes must be provided directly without the benefit of the "string lookup" system provided by Declarative. SQL expressions are typically specified in terms of the *Table* objects, i.e. `address.c.id` above for the `Address` relationship, and not `Address.id`, as `Address` may not yet be linked to table metadata, nor can we specify a string here.

Some examples in the documentation still use the classical approach, but note that the classical as well as Declarative approaches are **fully interchangeable**. Both systems ultimately create the same configuration, consisting of a *Table*, user-defined class, linked together with a *mapper()*. When we talk about "the behavior of *mapper()*", this includes when using the Declarative system as well - it's still used, just behind the scenes.

Runtime Introspection of Mappings, Objects

The *Mapper* object is available from any mapped class, regardless of method, using the *Runtime Inspection API* system. Using the *inspect()* function, one can acquire the *Mapper* from a mapped class:

```
>>> from sqlalchemy import inspect
>>> insp = inspect(User)
```

Detailed information is available including *Mapper.columns*:

```
>>> insp.columns
<sqlalchemy.util._collections.OrderedProperties object at 0x102f407f8>
```

This is a namespace that can be viewed in a list format or via individual names:

```
>>> list(insp.columns)
[Column('id', Integer(), table=<user>, primary_key=True, nullable=False), Column('name', String(lengt
>>> insp.columns.name
Column('name', String(length=50), table=<user>)
```

Other namespaces include *Mapper.all_orm_descriptors*, which includes all mapped attributes as well as hybrids, association proxies:

```
>>> insp.all_orm_descriptors
<sqlalchemy.util._collections.ImmutableProperties object at 0x1040e2c68>
>>> insp.all_orm_descriptors.keys()
['fullname', 'password', 'name', 'id']
```

As well as *Mapper.column_attrs*:

```
>>> list(insp.column_attrs)
[<ColumnProperty at 0x10403fde0; id>, <ColumnProperty at 0x10403fce8; name>, <ColumnProperty at 0x104
>>> insp.column_attrs.name
<ColumnProperty at 0x10403fce8; name>
>>> insp.column_attrs.name.expression
Column('name', String(length=50), table=<user>)
```

See also:

Runtime Inspection API

Mapper

InstanceState

2.2.2 Mapping Columns and Expressions

The following sections discuss how table columns and SQL expressions are mapped to individual object attributes.

Mapping Table Columns

The default behavior of *mapper()* is to assemble all the columns in the mapped *Table* into mapped object attributes, each of which are named according to the name of the column itself (specifically, the `key` attribute of *Column*). This behavior can be modified in several ways.

Naming Columns Distinctly from Attribute Names

A mapping by default shares the same name for a *Column* as that of the mapped attribute - specifically it matches the `Column.key` attribute on *Column*, which by default is the same as the `Column.name`.

The name assigned to the Python attribute which maps to *Column* can be different from either `Column.name` or `Column.key` just by assigning it that way, as we illustrate here in a Declarative mapping:

```
class User(Base):
    __tablename__ = 'user'
    id = Column('user_id', Integer, primary_key=True)
    name = Column('user_name', String(50))
```

Where above `User.id` resolves to a column named `user_id` and `User.name` resolves to a column named `user_name`.

When mapping to an existing table, the *Column* object can be referenced directly:

```
class User(Base):
    __table__ = user_table
    id = user_table.c.user_id
    name = user_table.c.user_name
```

Or in a classical mapping, placed in the `properties` dictionary with the desired key:

```
mapper(User, user_table, properties={
   'id': user_table.c.user_id,
   'name': user_table.c.user_name,
})
```

In the next section we'll examine the usage of `.key` more closely.

Automating Column Naming Schemes from Reflected Tables

In the previous section *Naming Columns Distinctly from Attribute Names*, we showed how a *Column* explicitly mapped to a class can have a different attribute name than the column. But what if we aren't listing out *Column* objects explicitly, and instead are automating the production of *Table* objects using reflection (e.g. as described in *Reflecting Database Objects*)? In this case we can make use of the *DDLEvents.column_reflect()* event to intercept the production of *Column* objects and provide them with the `Column.key` of our choice:

```
@event.listens_for(Table, "column_reflect")
def column_reflect(inspector, table, column_info):
    # set column.key = "attr_<lower_case_name>"
    column_info['key'] = "attr_%s" % column_info['name'].lower()
```

With the above event, the reflection of *Column* objects will be intercepted with our event that adds a new ".key" element, such as in a mapping as below:

```
class MyClass(Base):
    __table__ = Table("some_table", Base.metadata,
                      autoload=True, autoload_with=some_engine)
```

If we want to qualify our event to only react for the specific *MetaData* object above, we can check for it in our event:

```
@event.listens_for(Table, "column_reflect")
def column_reflect(inspector, table, column_info):
    if table.metadata is Base.metadata:
        # set column.key = "attr_<lower_case_name>"
        column_info['key'] = "attr_%s" % column_info['name'].lower()
```

Naming All Columns with a Prefix

A quick approach to prefix column names, typically when mapping to an existing *Table* object, is to use `column_prefix`:

```
class User(Base):
    __table__ = user_table
    __mapper_args__ = {'column_prefix':'_'}
```

The above will place attribute names such as `_user_id`, `_user_name`, `_password` etc. on the mapped `User` class.

This approach is uncommon in modern usage. For dealing with reflected tables, a more flexible approach is to use that described in *Automating Column Naming Schemes from Reflected Tables*.

Using column_property for column level options

Options can be specified when mapping a *Column* using the *column_property()* function. This function explicitly creates the *ColumnProperty* used by the *mapper()* to keep track of the *Column*; normally, the *mapper()* creates this automatically. Using *column_property()*, we can pass additional arguments about how we'd like the *Column* to be mapped. Below, we pass an option `active_history`, which specifies that a change to this column's value should result in the former value being loaded first:

```
from sqlalchemy.orm import column_property

class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    name = column_property(Column(String(50)), active_history=True)
```

column_property() is also used to map a single attribute to multiple columns. This use case arises when mapping to a *join()* which has attributes which are equated to each other:

```
class User(Base):
    __table__ = user.join(address)
```

```
    # assign "user.id", "address.user_id" to the
    # "id" attribute
    id = column_property(user_table.c.id, address_table.c.user_id)
```

For more examples featuring this usage, see *Mapping a Class against Multiple Tables*.

Another place where `column_property()` is needed is to specify SQL expressions as mapped attributes, such as below where we create an attribute `fullname` that is the string concatenation of the `firstname` and `lastname` columns:

```
class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    firstname = Column(String(50))
    lastname = Column(String(50))
    fullname = column_property(firstname + " " + lastname)
```

See examples of this usage at *SQL Expressions as Mapped Attributes*.

`sqlalchemy.orm.`**`column_property`**(**columns*, ***kwargs*)

Provide a column-level property for use with a Mapper.

Column-based properties can normally be applied to the mapper's `properties` dictionary using the `Column` element directly. Use this function when the given column is not directly present within the mapper's selectable; examples include SQL expressions, functions, and scalar SELECT queries.

Columns that aren't present in the mapper's selectable won't be persisted by the mapper and are effectively "read-only" attributes.

Parameters

- **`*cols`** – list of Column objects to be mapped.
- **`active_history=False`** – When `True`, indicates that the "previous" value for a scalar attribute should be loaded when replaced, if not already loaded. Normally, history tracking logic for simple non-primary-key scalar values only needs to be aware of the "new" value in order to perform a flush. This flag is available for applications that make use of `attributes.get_history()` or `Session.is_modified()` which also need to know the "previous" value of the attribute.

  New in version 0.6.6.
- **`comparator_factory`** – a class which extends `ColumnProperty.Comparator` which provides custom SQL clause generation for comparison operations.
- **`group`** – a group name for this property when marked as deferred.
- **`deferred`** – when True, the column property is "deferred", meaning that it does not load immediately, and is instead loaded when the attribute is first accessed on an instance. See also `deferred()`.
- **`doc`** – optional string that will be applied as the doc on the class-bound descriptor.
- **`expire_on_flush=True`** – Disable expiry on flush. A column_property() which refers to a SQL expression (and not a single table-bound column) is considered to be a "read only" property; populating it has no effect on the state of data, and it can only return database state. For this reason a column_property()'s value is expired whenever the parent object is involved in a flush, that is, has any kind of "dirty" state within a flush. Setting this parameter to `False` will have the effect of leaving any existing value present after the flush proceeds. Note however that the `Session` with default expiration settings still expires all attributes after a `Session.commit()` call, however.

New in version 0.7.3.

- **info** – Optional data dictionary which will be populated into the *MapperProperty.info* attribute of this object.

  New in version 0.8.

- **extension** – an *AttributeExtension* instance, or list of extensions, which will be prepended to the list of attribute listeners for the resulting descriptor placed on the class. **Deprecated.** Please see *AttributeEvents*.

Mapping a Subset of Table Columns

Sometimes, a *Table* object was made available using the reflection process described at *Reflecting Database Objects* to load the table's structure from the database. For such a table that has lots of columns that don't need to be referenced in the application, the `include_properties` or `exclude_properties` arguments can specify that only a subset of columns should be mapped. For example:

```
class User(Base):
    __table__ = user_table
    __mapper_args__ = {
        'include_properties' :['user_id', 'user_name']
    }
```

...will map the `User` class to the `user_table` table, only including the `user_id` and `user_name` columns - the rest are not referenced. Similarly:

```
class Address(Base):
    __table__ = address_table
    __mapper_args__ = {
        'exclude_properties' : ['street', 'city', 'state', 'zip']
    }
```

...will map the `Address` class to the `address_table` table, including all columns present except `street`, `city`, `state`, and `zip`.

When this mapping is used, the columns that are not included will not be referenced in any SELECT statements emitted by *Query*, nor will there be any mapped attribute on the mapped class which represents the column; assigning an attribute of that name will have no effect beyond that of a normal Python attribute assignment.

In some cases, multiple columns may have the same name, such as when mapping to a join of two or more tables that share some column name. `include_properties` and `exclude_properties` can also accommodate *Column* objects to more accurately describe which columns should be included or excluded:

```
class UserAddress(Base):
    __table__ = user_table.join(addresses_table)
    __mapper_args__ = {
        'exclude_properties' :[address_table.c.id],
        'primary_key' : [user_table.c.id]
    }
```

Note: insert and update defaults configured on individual *Column* objects, i.e. those described at *Column Insert/Update Defaults* including those configured by the `default`, `update`, `server_default` and `server_onupdate` arguments, will continue to function normally even if those *Column* objects are not mapped. This is because in the case of `default` and `update`, the *Column* object is still present on the underlying *Table*, thus allowing the default functions to take place when the ORM emits an INSERT or UPDATE, and in the case of `server_default` and `server_onupdate`, the relational database itself maintains these functions.

SQL Expressions as Mapped Attributes

Attributes on a mapped class can be linked to SQL expressions, which can be used in queries.

Using a Hybrid

The easiest and most flexible way to link relatively simple SQL expressions to a class is to use a so-called "hybrid attribute", described in the section *Hybrid Attributes*. The hybrid provides for an expression that works at both the Python level as well as at the SQL expression level. For example, below we map a class `User`, containing attributes `firstname` and `lastname`, and include a hybrid that will provide for us the `fullname`, which is the string concatenation of the two:

```
from sqlalchemy.ext.hybrid import hybrid_property

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    firstname = Column(String(50))
    lastname = Column(String(50))

    @hybrid_property
    def fullname(self):
        return self.firstname + " " + self.lastname
```

Above, the `fullname` attribute is interpreted at both the instance and class level, so that it is available from an instance:

```
some_user = session.query(User).first()
print some_user.fullname
```

as well as usable within queries:

```
some_user = session.query(User).filter(User.fullname == "John Smith").first()
```

The string concatenation example is a simple one, where the Python expression can be dual purposed at the instance and class level. Often, the SQL expression must be distinguished from the Python expression, which can be achieved using *`hybrid_property.expression()`*. Below we illustrate the case where a conditional needs to be present inside the hybrid, using the `if` statement in Python and the *`sql.expression.case()`* construct for SQL expressions:

```
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy.sql import case

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    firstname = Column(String(50))
    lastname = Column(String(50))

    @hybrid_property
    def fullname(self):
        if self.firstname is not None:
            return self.firstname + " " + self.lastname
        else:
            return self.lastname

    @fullname.expression
```

```
    def fullname(cls):
        return case([
            (cls.firstname != None, cls.firstname + " " + cls.lastname),
        ], else_ = cls.lastname)
```

Using column_property

The *orm.column_property()* function can be used to map a SQL expression in a manner similar to a regularly mapped *Column*. With this technique, the attribute is loaded along with all other column-mapped attributes at load time. This is in some cases an advantage over the usage of hybrids, as the value can be loaded up front at the same time as the parent row of the object, particularly if the expression is one which links to other tables (typically as a correlated subquery) to access data that wouldn't normally be available on an already loaded object.

Disadvantages to using *orm.column_property()* for SQL expressions include that the expression must be compatible with the SELECT statement emitted for the class as a whole, and there are also some configurational quirks which can occur when using *orm.column_property()* from declarative mixins.

Our "fullname" example can be expressed using *orm.column_property()* as follows:

```
from sqlalchemy.orm import column_property

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    firstname = Column(String(50))
    lastname = Column(String(50))
    fullname = column_property(firstname + " " + lastname)
```

Correlated subqueries may be used as well. Below we use the *select()* construct to create a SELECT that links together the count of `Address` objects available for a particular `User`:

```
from sqlalchemy.orm import column_property
from sqlalchemy import select, func
from sqlalchemy import Column, Integer, String, ForeignKey

from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Address(Base):
    __tablename__ = 'address'
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('user.id'))

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    address_count = column_property(
        select([func.count(Address.id)]).\
            where(Address.user_id==id).\
            correlate_except(Address)
    )
```

In the above example, we define a *select()* construct like the following:

```
select([func.count(Address.id)]).\
    where(Address.user_id==id).\
    correlate_except(Address)
```

The meaning of the above statement is, select the count of `Address.id` rows where the `Address.user_id` column is equated to `id`, which in the context of the `User` class is the *Column* named `id` (note that `id` is also the name of a Python built in function, which is not what we want to use here - if we were outside of the `User` class definition, we'd use `User.id`).

The `select.correlate_except()` directive indicates that each element in the FROM clause of this *select()* may be omitted from the FROM list (that is, correlated to the enclosing SELECT statement against `User`) except for the one corresponding to `Address`. This isn't strictly necessary, but prevents `Address` from being inadvertently omitted from the FROM list in the case of a long string of joins between `User` and `Address` tables where SELECT statements against `Address` are nested.

If import issues prevent the *column_property()* from being defined inline with the class, it can be assigned to the class after both are configured. In Declarative this has the effect of calling *Mapper.add_property()* to add an additional property after the fact:

```
User.address_count = column_property(
        select([func.count(Address.id)]).\
            where(Address.user_id==User.id)
    )
```

For many-to-many relationships, use *and_()* to join the fields of the association table to both tables in a relation, illustrated here with a classical mapping:

```
from sqlalchemy import and_

mapper(Author, authors, properties={
    'book_count': column_property(
                        select([func.count(books.c.id)],
                            and_(
                                book_authors.c.author_id==authors.c.id,
                                book_authors.c.book_id==books.c.id
                            )))
    })
```

Using a plain descriptor

In cases where a SQL query more elaborate than what *orm.column_property()* or *hybrid_property* can provide must be emitted, a regular Python function accessed as an attribute can be used, assuming the expression only needs to be available on an already-loaded instance. The function is decorated with Python's own `@property` decorator to mark it as a read-only attribute. Within the function, *object_session()* is used to locate the *Session* corresponding to the current object, which is then used to emit a query:

```
from sqlalchemy.orm import object_session
from sqlalchemy import select, func

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    firstname = Column(String(50))
    lastname = Column(String(50))

    @property
    def address_count(self):
        return object_session(self).\
            scalar(
```

```
                select([func.count(Address.id)]).\
                    where(Address.user_id==self.id)
            )
```

The plain descriptor approach is useful as a last resort, but is less performant in the usual case than both the hybrid and column property approaches, in that it needs to emit a SQL query upon each access.

Changing Attribute Behavior

Simple Validators

A quick way to add a "validation" routine to an attribute is to use the *validates()* decorator. An attribute validator can raise an exception, halting the process of mutating the attribute's value, or can change the given value into something different. Validators, like all attribute extensions, are only called by normal userland code; they are not issued when the ORM is populating the object:

```
from sqlalchemy.orm import validates

class EmailAddress(Base):
    __tablename__ = 'address'

    id = Column(Integer, primary_key=True)
    email = Column(String)

    @validates('email')
    def validate_email(self, key, address):
        assert '@' in address
        return address
```

Changed in version 1.0.0: - validators are no longer triggered within the flush process when the newly fetched values for primary key columns as well as some python- or server-side defaults are fetched. Prior to 1.0, validators may be triggered in those cases as well.

Validators also receive collection append events, when items are added to a collection:

```
from sqlalchemy.orm import validates

class User(Base):
    # ...

    addresses = relationship("Address")

    @validates('addresses')
    def validate_address(self, key, address):
        assert '@' in address.email
        return address
```

The validation function by default does not get emitted for collection remove events, as the typical expectation is that a value being discarded doesn't require validation. However, *validates()* supports reception of these events by specifying `include_removes=True` to the decorator. When this flag is set, the validation function must receive an additional boolean argument which if `True` indicates that the operation is a removal:

```
from sqlalchemy.orm import validates

class User(Base):
    # ...
```

```
    addresses = relationship("Address")

    @validates('addresses', include_removes=True)
    def validate_address(self, key, address, is_remove):
        if is_remove:
            raise ValueError(
                    "not allowed to remove items from the collection")
        else:
            assert '@' in address.email
            return address
```

The case where mutually dependent validators are linked via a backref can also be tailored, using the `include_backrefs=False` option; this option, when set to `False`, prevents a validation function from emitting if the event occurs as a result of a backref:

```
from sqlalchemy.orm import validates

class User(Base):
    # ...

    addresses = relationship("Address", backref='user')

    @validates('addresses', include_backrefs=False)
    def validate_address(self, key, address):
        assert '@' in address.email
        return address
```

Above, if we were to assign to `Address.user` as in `some_address.user = some_user`, the `validate_address()` function would *not* be emitted, even though an append occurs to `some_user.addresses` - the event is caused by a backref.

Note that the *`validates()`* decorator is a convenience function built on top of attribute events. An application that requires more control over configuration of attribute change behavior can make use of this system, described at *`AttributeEvents`*.

`sqlalchemy.orm.`**`validates`**(**names*, ***kw*)

Decorate a method as a 'validator' for one or more named properties.

Designates a method as a validator, a method which receives the name of the attribute as well as a value to be assigned, or in the case of a collection, the value to be added to the collection. The function can then raise validation exceptions to halt the process from continuing (where Python's built-in `ValueError` and `AssertionError` exceptions are reasonable choices), or can modify or replace the value before proceeding. The function should otherwise return the given value.

Note that a validator for a collection **cannot** issue a load of that collection within the validation routine - this usage raises an assertion to avoid recursion overflows. This is a reentrant condition which is not supported.

Parameters

- **`*names`** – list of attribute names to be validated.
- **`include_removes`** – if True, "remove" events will be sent as well - the validation function must accept an additional argument "is_remove" which will be a boolean.

  New in version 0.7.7.
- **`include_backrefs`** – defaults to `True`; if `False`, the validation function will not emit if the originator is an attribute event related via a backref. This can be used for bi-directional *`validates()`* usage where only one validator should emit per attribute operation.

  New in version 0.9.0.

See also:

Simple Validators - usage examples for `validates()`

Using Descriptors and Hybrids

A more comprehensive way to produce modified behavior for an attribute is to use descriptors. These are commonly used in Python using the `property()` function. The standard SQLAlchemy technique for descriptors is to create a plain descriptor, and to have it read/write from a mapped attribute with a different name. Below we illustrate this using Python 2.6-style properties:

```
class EmailAddress(Base):
    __tablename__ = 'email_address'

    id = Column(Integer, primary_key=True)

    # name the attribute with an underscore,
    # different from the column name
    _email = Column("email", String)

    # then create an ".email" attribute
    # to get/set "._email"
    @property
    def email(self):
        return self._email

    @email.setter
    def email(self, email):
        self._email = email
```

The approach above will work, but there's more we can add. While our `EmailAddress` object will shuttle the value through the `email` descriptor and into the `_email` mapped attribute, the class level `EmailAddress.email` attribute does not have the usual expression semantics usable with *Query*. To provide these, we instead use the *hybrid* extension as follows:

```
from sqlalchemy.ext.hybrid import hybrid_property

class EmailAddress(Base):
    __tablename__ = 'email_address'

    id = Column(Integer, primary_key=True)

    _email = Column("email", String)

    @hybrid_property
    def email(self):
        return self._email

    @email.setter
    def email(self, email):
        self._email = email
```

The `.email` attribute, in addition to providing getter/setter behavior when we have an instance of `EmailAddress`, also provides a SQL expression when used at the class level, that is, from the `EmailAddress` class directly:

```
from sqlalchemy.orm import Session
session = Session()
```

```
address = session.query(EmailAddress).\
                 filter(EmailAddress.email == 'address@example.com').\
                 one()
SELECT address.email AS address_email, address.id AS address_id
FROM address
WHERE address.email = ?
('address@example.com',)


address.email = 'otheraddress@example.com'
session.commit()
UPDATE address SET email=? WHERE address.id = ?
('otheraddress@example.com', 1)
COMMIT
```

The *hybrid_property* also allows us to change the behavior of the attribute, including defining separate behaviors when the attribute is accessed at the instance level versus at the class/expression level, using the *hybrid_property.expression()* modifier. Such as, if we wanted to add a host name automatically, we might define two sets of string manipulation logic:

```
class EmailAddress(Base):
    __tablename__ = 'email_address'

    id = Column(Integer, primary_key=True)

    _email = Column("email", String)

    @hybrid_property
    def email(self):
        """Return the value of _email up until the last twelve
        characters."""

        return self._email[:-12]

    @email.setter
    def email(self, email):
        """Set the value of _email, tacking on the twelve character
        value @example.com."""

        self._email = email + "@example.com"

    @email.expression
    def email(cls):
        """Produce a SQL expression that represents the value
        of the _email column, minus the last twelve characters."""

        return func.substr(cls._email, 0, func.length(cls._email) - 12)
```

Above, accessing the `email` property of an instance of `EmailAddress` will return the value of the `_email` attribute, removing or adding the hostname `@example.com` from the value. When we query against the `email` attribute, a SQL function is rendered which produces the same effect:

```
address = session.query(EmailAddress).filter(EmailAddress.email == 'address').one()
SELECT address.email AS address_email, address.id AS address_id
FROM address
WHERE substr(address.email, ?, length(address.email) - ?) = ?
(0, 12, 'address')
```

Read more about Hybrids at *Hybrid Attributes*.

Synonyms

Synonyms are a mapper-level construct that allow any attribute on a class to "mirror" another attribute that is mapped.

In the most basic sense, the synonym is an easy way to make a certain attribute available by an additional name:

```
class MyClass(Base):
    __tablename__ = 'my_table'

    id = Column(Integer, primary_key=True)
    job_status = Column(String(50))

    status = synonym("job_status")
```

The above class `MyClass` has two attributes, `.job_status` and `.status` that will behave as one attribute, both at the expression level:

```
>>> print MyClass.job_status == 'some_status'
my_table.job_status = :job_status_1

>>> print MyClass.status == 'some_status'
my_table.job_status = :job_status_1
```

and at the instance level:

```
>>> m1 = MyClass(status='x')
>>> m1.status, m1.job_status
('x', 'x')

>>> m1.job_status = 'y'
>>> m1.status, m1.job_status
('y', 'y')
```

The *synonym()* can be used for any kind of mapped attribute that subclasses *MapperProperty*, including mapped columns and relationships, as well as synonyms themselves.

Beyond a simple mirror, *synonym()* can also be made to reference a user-defined descriptor. We can supply our `status` synonym with a `@property`:

```
class MyClass(Base):
    __tablename__ = 'my_table'

    id = Column(Integer, primary_key=True)
    status = Column(String(50))

    @property
    def job_status(self):
        return "Status: " + self.status

    job_status = synonym("status", descriptor=job_status)
```

When using Declarative, the above pattern can be expressed more succinctly using the *synonym_for()* decorator:

```
from sqlalchemy.ext.declarative import synonym_for

class MyClass(Base):
    __tablename__ = 'my_table'

    id = Column(Integer, primary_key=True)
    status = Column(String(50))
```

```
    @synonym_for("status")
    @property
    def job_status(self):
        return "Status: " + self.status
```

While the *synonym()* is useful for simple mirroring, the use case of augmenting attribute behavior with descriptors is better handled in modern usage using the *hybrid attribute* feature, which is more oriented towards Python descriptors. Technically, a *synonym()* can do everything that a *hybrid_property* can do, as it also supports injection of custom SQL capabilities, but the hybrid is more straightforward to use in more complex situations.

`sqlalchemy.orm.`**`synonym`**(*name*, *map_column=None*, *descriptor=None*, *comparator_factory=None*, *doc=None*, *info=None*)

Denote an attribute name as a synonym to a mapped property, in that the attribute will mirror the value and expression behavior of another attribute.

Parameters

- **name** – the name of the existing mapped property. This can refer to the string name of any *MapperProperty* configured on the class, including column-bound attributes and relationships.
- **descriptor** – a Python descriptor that will be used as a getter (and potentially a setter) when this attribute is accessed at the instance level.
- **map_column** – if `True`, the *synonym()* construct will locate the existing named *MapperProperty* based on the attribute name of this *synonym()*, and assign it to a new attribute linked to the name of this *synonym()*. That is, given a mapping like:

```
class MyClass(Base):
    __tablename__ = 'my_table'

    id = Column(Integer, primary_key=True)
    job_status = Column(String(50))

    job_status = synonym("_job_status", map_column=True)
```

  The above class `MyClass` will now have the `job_status` *Column* object mapped to the attribute named `_job_status`, and the attribute named `job_status` will refer to the synonym itself. This feature is typically used in conjunction with the `descriptor` argument in order to link a user-defined descriptor as a "wrapper" for an existing column.
- **info** – Optional data dictionary which will be populated into the `InspectionAttr.info` attribute of this object.

  New in version 1.0.0.
- **comparator_factory** – A subclass of *PropComparator* that will provide custom comparison behavior at the SQL expression level.

  Note: For the use case of providing an attribute which redefines both Python-level and SQL-expression level behavior of an attribute, please refer to the Hybrid attribute introduced at *Using Descriptors and Hybrids* for a more effective technique.

See also:

Synonyms - examples of functionality.

Using Descriptors and Hybrids - Hybrids provide a better approach for more complicated attribute-wrapping schemes than synonyms.

Operator Customization

The "operators" used by the SQLAlchemy ORM and Core expression language are fully customizable. For example, the comparison expression `User.name == 'ed'` makes usage of an operator built into Python itself called `operator.eq` - the actual SQL construct which SQLAlchemy associates with such an operator can be modified. New operations can be associated with column expressions as well. The operators which take place for column expressions are most directly redefined at the type level - see the section *Redefining and Creating New Operators* for a description.

ORM level functions like `column_property()`, `relationship()`, and `composite()` also provide for operator redefinition at the ORM level, by passing a `PropComparator` subclass to the `comparator_factory` argument of each function. Customization of operators at this level is a rare use case. See the documentation at `PropComparator` for an overview.

Composite Column Types

Sets of columns can be associated with a single user-defined datatype. The ORM provides a single attribute which represents the group of columns using the class you provide.

Changed in version 0.7: Composites have been simplified such that they no longer "conceal" the underlying column based attributes. Additionally, in-place mutation is no longer automatic; see the section below on enabling mutability to support tracking of in-place changes.

Changed in version 0.9: Composites will return their object-form, rather than as individual columns, when used in a column-oriented `Query` construct. See *Composite attributes are now returned as their object form when queried on a per-attribute basis*.

A simple example represents pairs of columns as a `Point` object. `Point` represents such a pair as `.x` and `.y`:

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __composite_values__(self):
        return self.x, self.y

    def __repr__(self):
        return "Point(x=%r, y=%r)" % (self.x, self.y)

    def __eq__(self, other):
        return isinstance(other, Point) and \
            other.x == self.x and \
            other.y == self.y

    def __ne__(self, other):
        return not self.__eq__(other)
```

The requirements for the custom datatype class are that it have a constructor which accepts positional arguments corresponding to its column format, and also provides a method `__composite_values__()` which returns the state of the object as a list or tuple, in order of its column-based attributes. It also should supply adequate `__eq__()` and `__ne__()` methods which test the equality of two instances.

We will create a mapping to a table `vertice`, which represents two points as `x1/y1` and `x2/y2`. These are created normally as `Column` objects. Then, the `composite()` function is used to assign new attributes that will represent sets of columns via the `Point` class:

```
from sqlalchemy import Column, Integer
from sqlalchemy.orm import composite
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Vertex(Base):
    __tablename__ = 'vertice'

    id = Column(Integer, primary_key=True)
    x1 = Column(Integer)
    y1 = Column(Integer)
    x2 = Column(Integer)
    y2 = Column(Integer)

    start = composite(Point, x1, y1)
    end = composite(Point, x2, y2)
```

A classical mapping above would define each *composite()* against the existing table:

```
mapper(Vertex, vertice_table, properties={
    'start':composite(Point, vertice_table.c.x1, vertice_table.c.y1),
    'end':composite(Point, vertice_table.c.x2, vertice_table.c.y2),
})
```

We can now persist and use `Vertex` instances, as well as query for them, using the `.start` and `.end` attributes against ad-hoc `Point` instances:

```
>>> v = Vertex(start=Point(3, 4), end=Point(5, 6))
>>> session.add(v)
>>> q = session.query(Vertex).filter(Vertex.start == Point(3, 4))
>>> print q.first().start
BEGIN (implicit)
INSERT INTO vertice (x1, y1, x2, y2) VALUES (?, ?, ?, ?)
(3, 4, 5, 6)
SELECT vertice.id AS vertice_id,
        vertice.x1 AS vertice_x1,
        vertice.y1 AS vertice_y1,
        vertice.x2 AS vertice_x2,
        vertice.y2 AS vertice_y2
FROM vertice
WHERE vertice.x1 = ? AND vertice.y1 = ?
 LIMIT ? OFFSET ?
(3, 4, 1, 0)
Point(x=3, y=4)
```

`sqlalchemy.orm.`**`composite`**(*class_*, **attrs*, ***kwargs*)

Return a composite column-based property for use with a Mapper.

See the mapping documentation section *Composite Column Types* for a full usage example.

The *MapperProperty* returned by *composite()* is the *CompositeProperty*.

Parameters

- **class_** – The "composite type" class.
- ***cols** – List of Column objects to be mapped.
- **active_history=False** – When `True`, indicates that the "previous" value for a scalar attribute should be loaded when replaced, if not already loaded. See the same flag on

column_property().

Changed in version 0.7: This flag specifically becomes meaningful - previously it was a placeholder.

- **group** – A group name for this property when marked as deferred.
- **deferred** – When True, the column property is "deferred", meaning that it does not load immediately, and is instead loaded when the attribute is first accessed on an instance. See also *deferred()*.
- **comparator_factory** – a class which extends *CompositeProperty.Comparator* which provides custom SQL clause generation for comparison operations.
- **doc** – optional string that will be applied as the doc on the class-bound descriptor.
- **info** – Optional data dictionary which will be populated into the *MapperProperty.info* attribute of this object.

  New in version 0.8.
- **extension** – an *AttributeExtension* instance, or list of extensions, which will be prepended to the list of attribute listeners for the resulting descriptor placed on the class. **Deprecated.** Please see *AttributeEvents*.

Tracking In-Place Mutations on Composites

In-place changes to an existing composite value are not tracked automatically. Instead, the composite class needs to provide events to its parent object explicitly. This task is largely automated via the usage of the *MutableComposite* mixin, which uses events to associate each user-defined composite object with all parent associations. Please see the example in *Establishing Mutability on Composites*.

Changed in version 0.7: In-place changes to an existing composite value are no longer tracked automatically; the functionality is superseded by the *MutableComposite* class.

Redefining Comparison Operations for Composites

The "equals" comparison operation by default produces an AND of all corresponding columns equated to one another. This can be changed using the `comparator_factory` argument to *composite()*, where we specify a custom *CompositeProperty.Comparator* class to define existing or new operations. Below we illustrate the "greater than" operator, implementing the same expression that the base "greater than" does:

```
from sqlalchemy.orm.properties import CompositeProperty
from sqlalchemy import sql

class PointComparator(CompositeProperty.Comparator):
    def __gt__(self, other):
        """redefine the 'greater than' operation"""

        return sql.and_(*[a>b for a, b in
                          zip(self.__clause_element__().clauses,
                              other.__composite_values__())])

class Vertex(Base):
    ___tablename__ = 'vertice'

    id = Column(Integer, primary_key=True)
```

```
    x1 = Column(Integer)
    y1 = Column(Integer)
    x2 = Column(Integer)
    y2 = Column(Integer)

    start = composite(Point, x1, y1,
                      comparator_factory=PointComparator)
    end = composite(Point, x2, y2,
                      comparator_factory=PointComparator)
```

2.2.3 Mapping Class Inheritance Hierarchies

SQLAlchemy supports three forms of inheritance: **single table inheritance**, where several types of classes are represented by a single table, **concrete table inheritance**, where each type of class is represented by independent tables, and **joined table inheritance**, where the class hierarchy is broken up among dependent tables, each class represented by its own table that only includes those attributes local to that class.

The most common forms of inheritance are single and joined table, while concrete inheritance presents more configurational challenges.

When mappers are configured in an inheritance relationship, SQLAlchemy has the ability to load elements polymorphically, meaning that a single query can return objects of multiple types.

Joined Table Inheritance

In joined table inheritance, each class along a particular classes' list of parents is represented by a unique table. The total set of attributes for a particular instance is represented as a join along all tables in its inheritance path. Here, we first define the `Employee` class. This table will contain a primary key column (or columns), and a column for each attribute that's represented by `Employee`. In this case it's just `name`:

```
class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    type = Column(String(50))

    __mapper_args__ = {
        'polymorphic_identity':'employee',
        'polymorphic_on':type
    }
```

The mapped table also has a column called `type`. The purpose of this column is to act as the **discriminator**, and stores a value which indicates the type of object represented within the row. The column may be of any datatype, though string and integer are the most common.

> **Warning:** Currently, **only one discriminator column may be set**, typically on the base-most class in the hierarchy. "Cascading" polymorphic columns are not yet supported.

The discriminator column is only needed if polymorphic loading is desired, as is usually the case. It is not strictly necessary that it be present directly on the base mapped table, and can instead be defined on a derived select statement that's used when the class is queried; however, this is a much more sophisticated configuration scenario.

The mapping receives additional arguments via the `__mapper_args__` dictionary. Here the `type` column is explicitly stated as the discriminator column, and the **polymorphic identity** of `employee` is also given; this is the value that will be stored in the polymorphic discriminator column for instances of this class.

We next define `Engineer` and `Manager` subclasses of `Employee`. Each contains columns that represent the attributes unique to the subclass they represent. Each table also must contain a primary key column (or columns), and in most cases a foreign key reference to the parent table:

```
class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    engineer_name = Column(String(30))

    __mapper_args__ = {
        'polymorphic_identity':'engineer',
    }

class Manager(Employee):
    __tablename__ = 'manager'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    manager_name = Column(String(30))

    __mapper_args__ = {
        'polymorphic_identity':'manager',
    }
```

It is standard practice that the same column is used for both the role of primary key as well as foreign key to the parent table, and that the column is also named the same as that of the parent table. However, both of these practices are optional. Separate columns may be used for primary key and parent-relationship, the column may be named differently than that of the parent, and even a custom join condition can be specified between parent and child tables instead of using a foreign key.

Joined inheritance primary keys

One natural effect of the joined table inheritance configuration is that the identity of any mapped object can be determined entirely from the base table. This has obvious advantages, so SQLAlchemy always considers the primary key columns of a joined inheritance class to be those of the base table only. In other words, the `id` columns of both the `engineer` and `manager` tables are not used to locate `Engineer` or `Manager` objects - only the value in `employee.id` is considered. `engineer.id` and `manager.id` are still of course critical to the proper operation of the pattern overall as they are used to locate the joined row, once the parent row has been determined within a statement.

With the joined inheritance mapping complete, querying against `Employee` will return a combination of `Employee`, `Engineer` and `Manager` objects. Newly saved `Engineer`, `Manager`, and `Employee` objects will automatically populate the `employee.type` column with `engineer`, `manager`, or `employee`, as appropriate.

Basic Control of Which Tables are Queried

The *orm.with_polymorphic()* function and the *with_polymorphic()* method of *Query* affects the specific tables which the *Query* selects from. Normally, a query such as this:

```
session.query(Employee).all()
```

...selects only from the `employee` table. When loading fresh from the database, our joined-table setup will query from the parent table only, using SQL such as this:

```
SELECT employee.id AS employee_id,
    employee.name AS employee_name, employee.type AS employee_type
```

```
FROM employee
[]
```

As attributes are requested from those `Employee` objects which are represented in either the `engineer` or `manager` child tables, a second load is issued for the columns in that related row, if the data was not already loaded. So above, after accessing the objects you'd see further SQL issued along the lines of:

```
SELECT manager.id AS manager_id,
    manager.manager_data AS manager_manager_data
FROM manager
WHERE ? = manager.id
[5]
SELECT engineer.id AS engineer_id,
    engineer.engineer_info AS engineer_engineer_info
FROM engineer
WHERE ? = engineer.id
[2]
```

This behavior works well when issuing searches for small numbers of items, such as when using *Query.get()*, since the full range of joined tables are not pulled in to the SQL statement unnecessarily. But when querying a larger span of rows which are known to be of many types, you may want to actively join to some or all of the joined tables. The `with_polymorphic` feature provides this.

Telling our query to polymorphically load `Engineer` and `Manager` objects, we can use the *orm.with_polymorphic()* function to create a new aliased class which represents a select of the base table combined with outer joins to each of the inheriting tables:

```
from sqlalchemy.orm import with_polymorphic

eng_plus_manager = with_polymorphic(Employee, [Engineer, Manager])

query = session.query(eng_plus_manager)
```

The above produces a query which joins the `employee` table to both the `engineer` and `manager` tables like the following:

```
query.all()

SELECT employee.id AS employee_id,
    engineer.id AS engineer_id,
    manager.id AS manager_id,
    employee.name AS employee_name,
    employee.type AS employee_type,
    engineer.engineer_info AS engineer_engineer_info,
    manager.manager_data AS manager_manager_data
FROM employee
    LEFT OUTER JOIN engineer
    ON employee.id = engineer.id
    LEFT OUTER JOIN manager
    ON employee.id = manager.id
[]
```

The entity returned by *orm.with_polymorphic()* is an *AliasedClass* object, which can be used in a *Query* like any other alias, including named attributes for those attributes on the `Employee` class. In our example, `eng_plus_manager` becomes the entity that we use to refer to the three-way outer join above. It also includes namespaces for each class named in the list of classes, so that attributes specific to those subclasses can be called upon as well. The following example illustrates calling upon attributes specific to `Engineer` as well as `Manager` in terms of `eng_plus_manager`:

```
eng_plus_manager = with_polymorphic(Employee, [Engineer, Manager])
query = session.query(eng_plus_manager).filter(
                or_(
                    eng_plus_manager.Engineer.engineer_info=='x',
                    eng_plus_manager.Manager.manager_data=='y'
                )
            )
```

orm.with_polymorphic() accepts a single class or mapper, a list of classes/mappers, or the string `'*'` to indicate all subclasses:

```
# join to the engineer table
entity = with_polymorphic(Employee, Engineer)

# join to the engineer and manager tables
entity = with_polymorphic(Employee, [Engineer, Manager])

# join to all subclass tables
entity = with_polymorphic(Employee, '*')

# use the 'entity' with a Query object
session.query(entity).all()
```

It also accepts a third argument `selectable` which replaces the automatic join creation and instead selects directly from the selectable given. This feature is normally used with "concrete" inheritance, described later, but can be used with any kind of inheritance setup in the case that specialized SQL should be used to load polymorphically:

```
# custom selectable
employee = Employee.__table__
manager = Manager.__table__
engineer = Engineer.__table__
entity = with_polymorphic(
            Employee,
            [Engineer, Manager],
            employee.outerjoin(manager).outerjoin(engineer)
        )

# use the 'entity' with a Query object
session.query(entity).all()
```

Note that if you only need to load a single subtype, such as just the `Engineer` objects, *orm.with_polymorphic()* is not needed since you would query against the `Engineer` class directly.

Query.with_polymorphic() has the same purpose as *orm.with_polymorphic()*, except is not as flexible in its usage patterns in that it only applies to the first full mapping, which then impacts all occurrences of that class or the target subclasses within the *Query*. For simple cases it might be considered to be more succinct:

```
session.query(Employee).with_polymorphic([Engineer, Manager]).\
    filter(or_(Engineer.engineer_info=='w', Manager.manager_data=='q'))
```

New in version 0.8: *orm.with_polymorphic()*, an improved version of *Query.with_polymorphic()* method.

The mapper also accepts `with_polymorphic` as a configurational argument so that the joined-style load will be issued automatically. This argument may be the string `'*'`, a list of classes, or a tuple consisting of either, followed by a selectable:

```
class Employee(Base):
    __tablename__ = 'employee'
```

```
    id = Column(Integer, primary_key=True)
    type = Column(String(20))

    __mapper_args__ = {
        'polymorphic_on':type,
        'polymorphic_identity':'employee',
        'with_polymorphic':'*'
    }

class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    __mapper_args__ = {'polymorphic_identity':'engineer'}

class Manager(Employee):
    __tablename__ = 'manager'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    __mapper_args__ = {'polymorphic_identity':'manager'}
```

The above mapping will produce a query similar to that of `with_polymorphic('*')` for every query of `Employee` objects.

Using *orm.with_polymorphic()* or *Query.with_polymorphic()* will override the mapper-level `with_polymorphic` setting.

`sqlalchemy.orm.`**`with_polymorphic`**(*base*, *classes*, *selectable=False*, *flat=False*, *polymorphic_on=None*, *aliased=False*, *innerjoin=False*, *_use_mapper_path=False*, *_existing_alias=None*)

Produce an *AliasedClass* construct which specifies columns for descendant mappers of the given base.

New in version 0.8: *orm.with_polymorphic()* is in addition to the existing *Query* method *Query.with_polymorphic()*, which has the same purpose but is not as flexible in its usage.

Using this method will ensure that each descendant mapper's tables are included in the FROM clause, and will allow filter() criterion to be used against those tables. The resulting instances will also have those columns already loaded so that no "post fetch" of those columns will be required.

See the examples at *Basic Control of Which Tables are Queried*.

Parameters

- **base** – Base class to be aliased.
- **classes** – a single class or mapper, or list of class/mappers, which inherit from the base class. Alternatively, it may also be the string `'*'`, in which case all descending mapped classes will be added to the FROM clause.
- **aliased** – when True, the selectable will be wrapped in an alias, that is `(SELECT * FROM <fromclauses>) AS anon_1`. This can be important when using the with_polymorphic() to create the target of a JOIN on a backend that does not support parenthesized joins, such as SQLite and older versions of MySQL.
- **flat** –

  Boolean, will be passed through to the *FromClause.alias()* call so that aliases of *Join* objects don't include an enclosing SELECT. This can lead to more efficient queries in many circumstances. A JOIN against a nested JOIN will be rewritten as a JOIN against an aliased SELECT subquery on backends that don't support this syntax.

  Setting `flat` to `True` implies the `aliased` flag is also `True`.

  New in version 0.9.0.

See also:

Join.alias()

- **selectable** – a table or select() statement that will be used in place of the generated FROM clause. This argument is required if any of the desired classes use concrete table inheritance, since SQLAlchemy currently cannot generate UNIONs among tables automatically. If used, the `selectable` argument must represent the full set of tables and columns mapped by every mapped class. Otherwise, the unaccounted mapped columns will result in their table being appended directly to the FROM clause which will usually lead to incorrect results.
- **polymorphic_on** – a column to be used as the "discriminator" column for the given selectable. If not given, the polymorphic_on attribute of the base classes' mapper will be used, if any. This is useful for mappings that don't have polymorphic loading behavior by default.
- **innerjoin** – if True, an INNER JOIN will be used. This should only be specified if querying for one specific subtype only

Advanced Control of Which Tables are Queried

The `with_polymorphic` functions work fine for simplistic scenarios. However, direct control of table rendering is called for, such as the case when one wants to render to only the subclass table and not the parent table.

This use case can be achieved by using the mapped *Table* objects directly. For example, to query the name of employees with particular criterion:

```
engineer = Engineer.__table__
manager = Manager.__table__

session.query(Employee.name).\
    outerjoin((engineer, engineer.c.employee_id==Employee.employee_id)).\
    outerjoin((manager, manager.c.employee_id==Employee.employee_id)).\
    filter(or_(Engineer.engineer_info=='w', Manager.manager_data=='q'))
```

The base table, in this case the "employees" table, isn't always necessary. A SQL query is always more efficient with fewer joins. Here, if we wanted to just load information specific to manager or engineer, we can instruct *Query* to use only those tables. The `FROM` clause is determined by what's specified in the *Session.query()*, *Query.filter()*, or *Query.select_from()* methods:

```
session.query(Manager.manager_data).select_from(manager)

session.query(engineer.c.id).\
        filter(engineer.c.engineer_info==manager.c.manager_data)
```

Creating Joins to Specific Subtypes

The *of_type()* method is a helper which allows the construction of joins along *relationship()* paths while narrowing the criterion to specific subclasses. Suppose the `employees` table represents a collection of employees which are associated with a `Company` object. We'll add a `company_id` column to the `employees` table and a new table `companies`:

```
class Company(Base):
    __tablename__ = 'company'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
```

```
    employees = relationship("Employee",
                     backref='company',
                     cascade='all, delete-orphan')

class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    type = Column(String(20))
    company_id = Column(Integer, ForeignKey('company.id'))
    __mapper_args__ = {
        'polymorphic_on':type,
        'polymorphic_identity':'employee',
        'with_polymorphic':'*'
    }

class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    engineer_info = Column(String(50))
    __mapper_args__ = {'polymorphic_identity':'engineer'}

class Manager(Employee):
    __tablename__ = 'manager'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    manager_data = Column(String(50))
    __mapper_args__ = {'polymorphic_identity':'manager'}
```

When querying from `Company` onto the `Employee` relationship, the `join()` method as well as the `any()` and `has()` operators will create a join from `company` to `employee`, without including `engineer` or `manager` in the mix. If we wish to have criterion which is specifically against the `Engineer` class, we can tell those methods to join or subquery against the joined table representing the subclass using the *`of_type()`* operator:

```
session.query(Company).\
    join(Company.employees.of_type(Engineer)).\
    filter(Engineer.engineer_info=='someinfo')
```

A longhand version of this would involve spelling out the full target selectable within a 2-tuple:

```
employee = Employee.__table__
engineer = Engineer.__table__

session.query(Company).\
    join((employee.join(engineer), Company.employees)).\
    filter(Engineer.engineer_info=='someinfo')
```

`of_type()` accepts a single class argument. More flexibility can be achieved either by joining to an explicit join as above, or by using the *`orm.with_polymorphic()`* function to create a polymorphic selectable:

```
manager_and_engineer = with_polymorphic(
                            Employee, [Manager, Engineer],
                            aliased=True)

session.query(Company).\
    join(manager_and_engineer, Company.employees).\
    filter(
        or_(manager_and_engineer.Engineer.engineer_info=='someinfo',
            manager_and_engineer.Manager.manager_data=='somedata')
    )
```

Above, we use the `aliased=True` argument with `orm.with_polymorhpic()` so that the right hand side of the join between `Company` and `manager_and_engineer` is converted into an aliased subquery. Some backends, such as SQLite and older versions of MySQL can't handle a FROM clause of the following form:

```
FROM x JOIN (y JOIN z ON <onclause>) ON <onclause>
```

Using `aliased=True` instead renders it more like:

```
FROM x JOIN (SELECT * FROM y JOIN z ON <onclause>) AS anon_1 ON <onclause>
```

The above join can also be expressed more succinctly by combining `of_type()` with the polymorphic construct:

```
manager_and_engineer = with_polymorphic(
                            Employee, [Manager, Engineer],
                            aliased=True)

session.query(Company).\
    join(Company.employees.of_type(manager_and_engineer)).\
    filter(
        or_(manager_and_engineer.Engineer.engineer_info=='someinfo',
            manager_and_engineer.Manager.manager_data=='somedata')
    )
```

The `any()` and `has()` operators also can be used with *of_type()* when the embedded criterion is in terms of a subclass:

```
session.query(Company).\
        filter(
            Company.employees.of_type(Engineer).
                any(Engineer.engineer_info=='someinfo')
            ).all()
```

Note that the `any()` and `has()` are both shorthand for a correlated EXISTS query. To build one by hand looks like:

```
session.query(Company).filter(
    exists([1],
        and_(Engineer.engineer_info=='someinfo',
            employees.c.company_id==companies.c.company_id),
        from_obj=employees.join(engineers)
    )
).all()
```

The EXISTS subquery above selects from the join of `employees` to `engineers`, and also specifies criterion which correlates the EXISTS subselect back to the parent `companies` table.

New in version 0.8: *of_type()* accepts *orm.aliased()* and *orm.with_polymorphic()* constructs in conjunction with *Query.join()*, `any()` and `has()`.

Eager Loading of Specific or Polymorphic Subtypes

The *joinedload()*, *subqueryload()*, *contains_eager()* and other loading-related options also support paths which make use of *of_type()*. Below we load `Company` rows while eagerly loading related `Engineer` objects, querying the `employee` and `engineer` tables simultaneously:

```
session.query(Company).\
    options(
        subqueryload(Company.employees.of_type(Engineer)).
        subqueryload("machines")
```

```
        )
    )
```

As is the case with *Query.join()*, `of_type()` also can be used with eager loading and *orm.with_polymorphic()* at the same time, so that all sub-attributes of all referenced subtypes can be loaded:

```
manager_and_engineer = with_polymorphic(
                            Employee, [Manager, Engineer],
                            aliased=True)

session.query(Company).\
    options(
        joinedload(Company.employees.of_type(manager_and_engineer))
        )
    )
```

New in version 0.8: *joinedload()*, *subqueryload()*, *contains_eager()* and related loader options support paths that are qualified with *of_type()*, supporting single target types as well as *orm.with_polymorphic()* targets.

Another option for the above query is to state the two subtypes separately; the *joinedload()* directive should detect this and create the above `with_polymorphic` construct automatically:

```
session.query(Company).\
    options(
        joinedload(Company.employees.of_type(Manager)),
        joinedload(Company.employees.of_type(Engineer)),
        )
    )
```

New in version 1.0: Eager loaders such as *joinedload()* will create a polymorphic entity when multiple overlapping `of_type()` directives are encountered.

Single Table Inheritance

Single table inheritance is where the attributes of the base class as well as all subclasses are represented within a single table. A column is present in the table for every attribute mapped to the base class and all subclasses; the columns which correspond to a single subclass are nullable. This configuration looks much like joined-table inheritance except there's only one table. In this case, a `type` column is required, as there would be no other way to discriminate between classes. The table is specified in the base mapper only; for the inheriting classes, leave their `table` parameter blank:

```
class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    manager_data = Column(String(50))
    engineer_info = Column(String(50))
    type = Column(String(20))

    __mapper_args__ = {
        'polymorphic_on':type,
        'polymorphic_identity':'employee'
    }

class Manager(Employee):
    __mapper_args__ = {
        'polymorphic_identity':'manager'
```

```
    }

class Engineer(Employee):
    __mapper_args__ = {
        'polymorphic_identity':'engineer'
    }
```

Note that the mappers for the derived classes Manager and Engineer omit the `__tablename__`, indicating they do not have a mapped table of their own.

Concrete Table Inheritance

Note: this section is currently using classical mappings. The Declarative system fully supports concrete inheritance however. See the links below for more information on using declarative with concrete table inheritance.

This form of inheritance maps each class to a distinct table, as below:

```
employees_table = Table('employees', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('name', String(50)),
)

managers_table = Table('managers', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('name', String(50)),
    Column('manager_data', String(50)),
)

engineers_table = Table('engineers', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('name', String(50)),
    Column('engineer_info', String(50)),
)
```

Notice in this case there is no `type` column. If polymorphic loading is not required, there's no advantage to using `inherits` here; you just define a separate mapper for each class.

```
mapper(Employee, employees_table)
mapper(Manager, managers_table)
mapper(Engineer, engineers_table)
```

To load polymorphically, the `with_polymorphic` argument is required, along with a selectable indicating how rows should be loaded. In this case we must construct a UNION of all three tables. SQLAlchemy includes a helper function to create these called *`polymorphic_union()`*, which will map all the different columns into a structure of selects with the same numbers and names of columns, and also generate a virtual `type` column for each subselect:

```
pjoin = polymorphic_union({
    'employee': employees_table,
    'manager': managers_table,
    'engineer': engineers_table
}, 'type', 'pjoin')

employee_mapper = mapper(Employee, employees_table,
                                   with_polymorphic=('*', pjoin),
                                   polymorphic_on=pjoin.c.type,
                                   polymorphic_identity='employee')
manager_mapper = mapper(Manager, managers_table,
```

```
                                    inherits=employee_mapper,
                                    concrete=True,
                                    polymorphic_identity='manager')
engineer_mapper = mapper(Engineer, engineers_table,
                                    inherits=employee_mapper,
                                    concrete=True,
                                    polymorphic_identity='engineer')
```

Upon select, the polymorphic union produces a query like this:

```
session.query(Employee).all()

SELECT pjoin.type AS pjoin_type,
        pjoin.manager_data AS pjoin_manager_data,
        pjoin.employee_id AS pjoin_employee_id,
pjoin.name AS pjoin_name, pjoin.engineer_info AS pjoin_engineer_info
FROM (
    SELECT employees.employee_id AS employee_id,
        CAST(NULL AS VARCHAR(50)) AS manager_data, employees.name AS name,
        CAST(NULL AS VARCHAR(50)) AS engineer_info, 'employee' AS type
    FROM employees
UNION ALL
    SELECT managers.employee_id AS employee_id,
        managers.manager_data AS manager_data, managers.name AS name,
        CAST(NULL AS VARCHAR(50)) AS engineer_info, 'manager' AS type
    FROM managers
UNION ALL
    SELECT engineers.employee_id AS employee_id,
        CAST(NULL AS VARCHAR(50)) AS manager_data, engineers.name AS name,
    engineers.engineer_info AS engineer_info, 'engineer' AS type
    FROM engineers
) AS pjoin
[]
```

Concrete Inheritance with Declarative

New in version 0.7.3: The *Declarative* module includes helpers for concrete inheritance. See *Using the Concrete Helpers* for more information.

Using Relationships with Inheritance

Both joined-table and single table inheritance scenarios produce mappings which are usable in `relationship()` functions; that is, it's possible to map a parent object to a child object which is polymorphic. Similarly, inheriting mappers can have `relationship()` objects of their own at any level, which are inherited to each child class. The only requirement for relationships is that there is a table relationship between parent and child. An example is the following modification to the joined table inheritance example, which sets a bi-directional relationship between `Employee` and `Company`:

```
employees_table = Table('employees', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('name', String(50)),
    Column('company_id', Integer, ForeignKey('companies.company_id'))
)

companies = Table('companies', metadata,
```

```
   Column('company_id', Integer, primary_key=True),
   Column('name', String(50)))

class Company(object):
    pass

mapper(Company, companies, properties={
   'employees': relationship(Employee, backref='company')
})
```

Relationships with Concrete Inheritance

In a concrete inheritance scenario, mapping relationships is more challenging since the distinct classes do not share a table. In this case, you *can* establish a relationship from parent to child if a join condition can be constructed from parent to child, if each child table contains a foreign key to the parent:

```
companies = Table('companies', metadata,
   Column('id', Integer, primary_key=True),
   Column('name', String(50)))

employees_table = Table('employees', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('name', String(50)),
    Column('company_id', Integer, ForeignKey('companies.id'))
)

managers_table = Table('managers', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('name', String(50)),
    Column('manager_data', String(50)),
    Column('company_id', Integer, ForeignKey('companies.id'))
)

engineers_table = Table('engineers', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('name', String(50)),
    Column('engineer_info', String(50)),
    Column('company_id', Integer, ForeignKey('companies.id'))
)

mapper(Employee, employees_table,
                with_polymorphic=('*', pjoin),
                polymorphic_on=pjoin.c.type,
                polymorphic_identity='employee')

mapper(Manager, managers_table,
                inherits=employee_mapper,
                concrete=True,
                polymorphic_identity='manager')

mapper(Engineer, engineers_table,
                inherits=employee_mapper,
                concrete=True,
                polymorphic_identity='engineer')

mapper(Company, companies, properties={
    'employees': relationship(Employee)
```

```
})
```

The big limitation with concrete table inheritance is that *relationship()* objects placed on each concrete mapper do **not** propagate to child mappers. If you want to have the same *relationship()* objects set up on all concrete mappers, they must be configured manually on each. To configure back references in such a configuration the `back_populates` keyword may be used instead of `backref`, such as below where both `A(object)` and `B(A)` bidirectionally reference `C`:

```
ajoin = polymorphic_union({
        'a':a_table,
        'b':b_table
    }, 'type', 'ajoin')

mapper(A, a_table, with_polymorphic=('*', ajoin),
    polymorphic_on=ajoin.c.type, polymorphic_identity='a',
    properties={
        'some_c':relationship(C, back_populates='many_a')
})
mapper(B, b_table,inherits=A, concrete=True,
    polymorphic_identity='b',
    properties={
        'some_c':relationship(C, back_populates='many_a')
})
mapper(C, c_table, properties={
    'many_a':relationship(A, collection_class=set,
                                back_populates='some_c'),
})
```

Using Inheritance with Declarative

Declarative makes inheritance configuration more intuitive. See the docs at *Inheritance Configuration*.

2.2.4 Non-Traditional Mappings

Mapping a Class against Multiple Tables

Mappers can be constructed against arbitrary relational units (called *selectables*) in addition to plain tables. For example, the *join()* function creates a selectable unit comprised of multiple tables, complete with its own composite primary key, which can be mapped in the same way as a *Table*:

```
from sqlalchemy import Table, Column, Integer, \
        String, MetaData, join, ForeignKey
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import column_property

metadata = MetaData()

# define two Table objects
user_table = Table('user', metadata,
            Column('id', Integer, primary_key=True),
            Column('name', String),
        )

address_table = Table('address', metadata,
            Column('id', Integer, primary_key=True),
```

```
            Column('user_id', Integer, ForeignKey('user.id')),
            Column('email_address', String)
            )

# define a join between them.  This
# takes place across the user.id and address.user_id
# columns.
user_address_join = join(user_table, address_table)

Base = declarative_base()

# map to it
class AddressUser(Base):
    __table__ = user_address_join

    id = column_property(user_table.c.id, address_table.c.user_id)
    address_id = address_table.c.id
```

In the example above, the join expresses columns for both the `user` and the `address` table. The `user.id` and `address.user_id` columns are equated by foreign key, so in the mapping they are defined as one attribute, `AddressUser.id`, using *`column_property()`* to indicate a specialized column mapping. Based on this part of the configuration, the mapping will copy new primary key values from `user.id` into the `address.user_id` column when a flush occurs.

Additionally, the `address.id` column is mapped explicitly to an attribute named `address_id`. This is to **disambiguate** the mapping of the `address.id` column from the same-named `AddressUser.id` attribute, which here has been assigned to refer to the `user` table combined with the `address.user_id` foreign key.

The natural primary key of the above mapping is the composite of `(user.id, address.id)`, as these are the primary key columns of the `user` and `address` table combined together. The identity of an `AddressUser` object will be in terms of these two values, and is represented from an `AddressUser` object as `(AddressUser.id, AddressUser.address_id)`.

Mapping a Class against Arbitrary Selects

Similar to mapping against a join, a plain *`select()`* object can be used with a mapper as well. The example fragment below illustrates mapping a class called `Customer` to a *`select()`* which includes a join to a subquery:

```
from sqlalchemy import select, func

subq = select([
            func.count(orders.c.id).label('order_count'),
            func.max(orders.c.price).label('highest_order'),
            orders.c.customer_id
            ]).group_by(orders.c.customer_id).alias()

customer_select = select([customers, subq]).\
            select_from(
                join(customers, subq,
                         customers.c.id == subq.c.customer_id)
            ).alias()

class Customer(Base):
    __table__ = customer_select
```

Above, the full row represented by `customer_select` will be all the columns of the `customers` table, in addition to those columns exposed by the `subq` subquery, which are `order_count`, `highest_order`, and

customer_id. Mapping the Customer class to this selectable then creates a class which will contain those attributes.

When the ORM persists new instances of Customer, only the customers table will actually receive an INSERT. This is because the primary key of the orders table is not represented in the mapping; the ORM will only emit an INSERT into a table for which it has mapped the primary key.

Note: The practice of mapping to arbitrary SELECT statements, especially complex ones as above, is almost never needed; it necessarily tends to produce complex queries which are often less efficient than that which would be produced by direct query construction. The practice is to some degree based on the very early history of SQLAlchemy where the *mapper()* construct was meant to represent the primary querying interface; in modern usage, the *Query* object can be used to construct virtually any SELECT statement, including complex composites, and should be favored over the "map-to-selectable" approach.

Multiple Mappers for One Class

In modern SQLAlchemy, a particular class is mapped by only one so-called **primary** mapper at a time. This mapper is involved in three main areas of functionality: querying, persistence, and instrumentation of the mapped class. The rationale of the primary mapper relates to the fact that the *mapper()* modifies the class itself, not only persisting it towards a particular *Table*, but also instrumenting attributes upon the class which are structured specifically according to the table metadata. It's not possible for more than one mapper to be associated with a class in equal measure, since only one mapper can actually instrument the class.

However, there is a class of mapper known as the **non primary** mapper with allows additional mappers to be associated with a class, but with a limited scope of use. This scope typically applies to being able to load rows from an alternate table or selectable unit, but still producing classes which are ultimately persisted using the primary mapping. The non-primary mapper is created using the classical style of mapping against a class that is already mapped with a primary mapper, and involves the use of the *non_primary* flag.

The non primary mapper is of very limited use in modern SQLAlchemy, as the task of being able to load classes from subqueries or other compound statements can be now accomplished using the *Query* object directly.

There is really only one use case for the non-primary mapper, which is that we wish to build a *relationship()* to such a mapper; this is useful in the rare and advanced case that our relationship is attempting to join two classes together using many tables and/or joins in between. An example of this pattern is at *Relationship to Non Primary Mapper*.

As far as the use case of a class that can actually be fully persisted to different tables under different scenarios, very early versions of SQLAlchemy offered a feature for this adapted from Hibernate, known as the "entity name" feature. However, this use case became infeasable within SQLAlchemy once the mapped class itself became the source of SQL expression construction; that is, the class' attributes themselves link directly to mapped table columns. The feature was removed and replaced with a simple recipe-oriented approach to accomplishing this task without any ambiguity of instrumentation - to create new subclasses, each mapped individually. This pattern is now available as a recipe at Entity Name.

2.2.5 Configuring a Version Counter

The *Mapper* supports management of a version id column, which is a single table column that increments or otherwise updates its value each time an UPDATE to the mapped table occurs. This value is checked each time the ORM emits an UPDATE or DELETE against the row to ensure that the value held in memory matches the database value.

Warning: Because the versioning feature relies upon comparison of the **in memory** record of an object, the feature only applies to the *Session.flush()* process, where the ORM flushes individual in-memory rows to the database. It does **not** take effect when performing a multirow UPDATE or DELETE using *Query.update()* or *Query.delete()* methods, as these methods only emit an UPDATE or DELETE statement but otherwise do not have direct access to the contents of those rows being affected.

The purpose of this feature is to detect when two concurrent transactions are modifying the same row at roughly the same time, or alternatively to provide a guard against the usage of a "stale" row in a system that might be re-using data from a previous transaction without refreshing (e.g. if one sets `expire_on_commit=False` with a *Session*, it is possible to re-use the data from a previous transaction).

Concurrent transaction updates

When detecting concurrent updates within transactions, it is typically the case that the database's transaction isolation level is below the level of repeatable read; otherwise, the transaction will not be exposed to a new row value created by a concurrent update which conflicts with the locally updated value. In this case, the SQLAlchemy versioning feature will typically not be useful for in-transaction conflict detection, though it still can be used for cross-transaction staleness detection.
The database that enforces repeatable reads will typically either have locked the target row against a concurrent update, or is employing some form of multi version concurrency control such that it will emit an error when the transaction is committed. SQLAlchemy's version_id_col is an alternative which allows version tracking to occur for specific tables within a transaction that otherwise might not have this isolation level set.
See also:
Repeatable Read Isolation Level - Postgresql's implementation of repeatable read, including a description of the error condition.

Simple Version Counting

The most straightforward way to track versions is to add an integer column to the mapped table, then establish it as the `version_id_col` within the mapper options:

```
class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    version_id = Column(Integer, nullable=False)
    name = Column(String(50), nullable=False)

    __mapper_args__ = {
        "version_id_col": version_id
    }
```

Above, the `User` mapping tracks integer versions using the column `version_id`. When an object of type `User` is first flushed, the `version_id` column will be given a value of "1". Then, an UPDATE of the table later on will always be emitted in a manner similar to the following:

```
UPDATE user SET version_id=:version_id, name=:name
WHERE user.id = :user_id AND user.version_id = :user_version_id
{"name": "new name", "version_id": 2, "user_id": 1, "user_version_id": 1}
```

The above UPDATE statement is updating the row that not only matches `user.id = 1`, it also is requiring that `user.version_id = 1`, where "1" is the last version identifier we've been known to use on this object. If a transaction elsewhere has modified the row independently, this version id will no longer match, and the UPDATE statement will report that no rows matched; this is the condition that SQLAlchemy tests, that exactly one row matched

our UPDATE (or DELETE) statement. If zero rows match, that indicates our version of the data is stale, and a *StaleDataError* is raised.

Custom Version Counters / Types

Other kinds of values or counters can be used for versioning. Common types include dates and GUIDs. When using an alternate type or counter scheme, SQLAlchemy provides a hook for this scheme using the `version_id_generator` argument, which accepts a version generation callable. This callable is passed the value of the current known version, and is expected to return the subsequent version.

For example, if we wanted to track the versioning of our `User` class using a randomly generated GUID, we could do this (note that some backends support a native GUID type, but we illustrate here using a simple string):

```
import uuid

class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    version_uuid = Column(String(32))
    name = Column(String(50), nullable=False)

    __mapper_args__ = {
        'version_id_col':version_uuid,
        'version_id_generator':lambda version: uuid.uuid4().hex
    }
```

The persistence engine will call upon `uuid.uuid4()` each time a `User` object is subject to an INSERT or an UPDATE. In this case, our version generation function can disregard the incoming value of `version`, as the `uuid4()` function generates identifiers without any prerequisite value. If we were using a sequential versioning scheme such as numeric or a special character system, we could make use of the given `version` in order to help determine the subsequent value.

See also:

Backend-agnostic GUID Type

Server Side Version Counters

The `version_id_generator` can also be configured to rely upon a value that is generated by the database. In this case, the database would need some means of generating new identifiers when a row is subject to an INSERT as well as with an UPDATE. For the UPDATE case, typically an update trigger is needed, unless the database in question supports some other native version identifier. The Postgresql database in particular supports a system column called xmin which provides UPDATE versioning. We can make use of the Postgresql `xmin` column to version our `User` class as follows:

```
class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    name = Column(String(50), nullable=False)
    xmin = Column("xmin", Integer, system=True)

    __mapper_args__ = {
        'version_id_col': xmin,
        'version_id_generator': False
    }
```

With the above mapping, the ORM will rely upon the `xmin` column for automatically providing the new value of the version id counter.

creating tables that refer to system columns

In the above scenario, as `xmin` is a system column provided by Postgresql, we use the `system=True` argument to mark it as a system-provided column, omitted from the `CREATE TABLE` statement.

The ORM typically does not actively fetch the values of database-generated values when it emits an INSERT or UPDATE, instead leaving these columns as "expired" and to be fetched when they are next accessed, unless the `eager_defaults` *`mapper()`* flag is set. However, when a server side version column is used, the ORM needs to actively fetch the newly generated value. This is so that the version counter is set up *before* any concurrent transaction may update it again. This fetching is also best done simultaneously within the INSERT or UPDATE statement using RETURNING, otherwise if emitting a SELECT statement afterwards, there is still a potential race condition where the version counter may change before it can be fetched.

When the target database supports RETURNING, an INSERT statement for our `User` class will look like this:

```
INSERT INTO "user" (name) VALUES (%(name)s) RETURNING "user".id, "user".xmin
{'name': 'ed'}
```

Where above, the ORM can acquire any newly generated primary key values along with server-generated version identifiers in one statement. When the backend does not support RETURNING, an additional SELECT must be emitted for **every** INSERT and UPDATE, which is much less efficient, and also introduces the possibility of missed version counters:

```
INSERT INTO "user" (name) VALUES (%(name)s)
{'name': 'ed'}

SELECT "user".version_id AS user_version_id FROM "user" where
"user".id = :param_1
{"param_1": 1}
```

It is *strongly recommended* that server side version counters only be used when absolutely necessary and only on backends that support RETURNING, e.g. Postgresql, Oracle, SQL Server (though SQL Server has major caveats when triggers are used), Firebird.

New in version 0.9.0: Support for server side version identifier tracking.

Programmatic or Conditional Version Counters

When `version_id_generator` is set to False, we can also programmatically (and conditionally) set the version identifier on our object in the same way we assign any other mapped attribute. Such as if we used our UUID example, but set `version_id_generator` to `False`, we can set the version identifier at our choosing:

```
import uuid

class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    version_uuid = Column(String(32))
    name = Column(String(50), nullable=False)

    __mapper_args__ = {
        'version_id_col':version_uuid,
        'version_id_generator': False
```

```
    }

u1 = User(name='u1', version_uuid=uuid.uuid4())

session.add(u1)

session.commit()

u1.name = 'u2'
u1.version_uuid = uuid.uuid4()

session.commit()
```

We can update our `User` object without incrementing the version counter as well; the value of the counter will remain unchanged, and the UPDATE statement will still check against the previous value. This may be useful for schemes where only certain classes of UPDATE are sensitive to concurrency issues:

```
# will leave version_uuid unchanged
u1.name = 'u3'
session.commit()
```

New in version 0.9.0: Support for programmatic and conditional version identifier tracking.

2.2.6 Class Mapping API

`sqlalchemy.orm.`**`mapper`**(*class_*, *local_table=None*, *properties=None*, *primary_key=None*, *non_primary=False*, *inherits=None*, *inherit_condition=None*, *inherit_foreign_keys=None*, *extension=None*, *order_by=False*, *always_refresh=False*, *version_id_col=None*, *version_id_generator=None*, *polymorphic_on=None*, *_polymorphic_map=None*, *polymorphic_identity=None*, *concrete=False*, *with_polymorphic=None*, *allow_partial_pks=True*, *batch=True*, *column_prefix=None*, *include_properties=None*, *exclude_properties=None*, *passive_updates=True*, *confirm_deleted_rows=True*, *eager_defaults=False*, *legacy_is_orphan=False*, *_compiled_cache_size=100*)

Return a new *`Mapper`* object.

This function is typically used behind the scenes via the Declarative extension. When using Declarative, many of the usual *`mapper()`* arguments are handled by the Declarative extension itself, including `class_`, `local_table`, `properties`, and `inherits`. Other options are passed to *`mapper()`* using the `__mapper_args__` class variable:

```
class MyClass(Base):
    __tablename__ = 'my_table'
    id = Column(Integer, primary_key=True)
    type = Column(String(50))
    alt = Column("some_alt", Integer)

    __mapper_args__ = {
        'polymorphic_on' : type
    }
```

Explicit use of *`mapper()`* is often referred to as *classical mapping*. The above declarative example is equivalent in classical form to:

```
my_table = Table("my_table", metadata,
    Column('id', Integer, primary_key=True),
```

```
        Column('type', String(50)),
        Column("some_alt", Integer)
    )

    class MyClass(object):
        pass

    mapper(MyClass, my_table,
        polymorphic_on=my_table.c.type,
        properties={
            'alt':my_table.c.some_alt
        })
```

See also:

Classical Mappings - discussion of direct usage of `mapper()`

Parameters

- **class_** – The class to be mapped. When using Declarative, this argument is automatically passed as the declared class itself.
- **local_table** – The `Table` or other selectable to which the class is mapped. May be `None` if this mapper inherits from another mapper using single-table inheritance. When using Declarative, this argument is automatically passed by the extension, based on what is configured via the `__table__` argument or via the `Table` produced as a result of the `__tablename__` and `Column` arguments present.
- **always_refresh** – If True, all query operations for this mapped class will overwrite all data within object instances that already exist within the session, erasing any in-memory changes with whatever information was loaded from the database. Usage of this flag is highly discouraged; as an alternative, see the method `Query.populate_existing()`.
- **allow_partial_pks** – Defaults to True. Indicates that a composite primary key with some NULL values should be considered as possibly existing within the database. This affects whether a mapper will assign an incoming row to an existing identity, as well as if `Session.merge()` will check the database first for a particular primary key value. A "partial primary key" can occur if one has mapped to an OUTER JOIN, for example.
- **batch** – Defaults to `True`, indicating that save operations of multiple entities can be batched together for efficiency. Setting to False indicates that an instance will be fully saved before saving the next instance. This is used in the extremely rare case that a `MapperEvents` listener requires being called in between individual row persistence operations.
- **column_prefix** – A string which will be prepended to the mapped attribute name when `Column` objects are automatically assigned as attributes to the mapped class. Does not affect explicitly specified column-based properties.

  See the section *Naming All Columns with a Prefix* for an example.
- **concrete** – If True, indicates this mapper should use concrete table inheritance with its parent mapper.

  See the section *Concrete Table Inheritance* for an example.
- **confirm_deleted_rows** – defaults to True; when a DELETE occurs of one more rows based on specific primary keys, a warning is emitted when the number of rows matched does not equal the number of rows expected. This parameter may be set to False to handle the

case where database ON DELETE CASCADE rules may be deleting some of those rows automatically. The warning may be changed to an exception in a future release.

New in version 0.9.4: - added `mapper.confirm_deleted_rows` as well as conditional matched row checking on delete.

- **eager_defaults** – if True, the ORM will immediately fetch the value of server-generated default values after an INSERT or UPDATE, rather than leaving them as expired to be fetched on next access. This can be used for event schemes where the server-generated values are needed immediately before the flush completes. By default, this scheme will emit an individual `SELECT` statement per row inserted or updated, which note can add significant performance overhead. However, if the target database supports RETURNING, the default values will be returned inline with the INSERT or UPDATE statement, which can greatly enhance performance for an application that needs frequent access to just-generated server defaults.

  Changed in version 0.9.0: The `eager_defaults` option can now make use of RETURNING for backends which support it.

- **exclude_properties** – A list or set of string column names to be excluded from mapping.

  See *Mapping a Subset of Table Columns* for an example.

- **extension** – A `MapperExtension` instance or list of `MapperExtension` instances which will be applied to all operations by this `Mapper`. **Deprecated.** Please see `MapperEvents`.

- **include_properties** – An inclusive list or set of string column names to map.

  See *Mapping a Subset of Table Columns* for an example.

- **inherits** – A mapped class or the corresponding `Mapper` of one indicating a superclass to which this `Mapper` should *inherit* from. The mapped class here must be a subclass of the other mapper's class. When using Declarative, this argument is passed automatically as a result of the natural class hierarchy of the declared classes.

  See also:

  Mapping Class Inheritance Hierarchies

- **inherit_condition** – For joined table inheritance, a SQL expression which will define how the two tables are joined; defaults to a natural join between the two tables.

- **inherit_foreign_keys** – When `inherit_condition` is used and the columns present are missing a `ForeignKey` configuration, this parameter can be used to specify which columns are "foreign". In most cases can be left as `None`.

- **legacy_is_orphan** – Boolean, defaults to `False`. When `True`, specifies that "legacy" orphan consideration is to be applied to objects mapped by this mapper, which means that a pending (that is, not persistent) object is auto-expunged from an owning `Session` only when it is de-associated from *all* parents that specify a `delete-orphan` cascade towards this mapper. The new default behavior is that the object is auto-expunged when it is de-associated with *any* of its parents that specify `delete-orphan` cascade. This behavior is more consistent with that of a persistent object, and allows behavior to be consistent in more scenarios independently of whether or not an orphanable object has been flushed yet or not.

  See the change note and example at *The consideration of a "pending" object as an "orphan" has been made more aggressive* for more detail on this change.

  New in version 0.8: - the consideration of a pending object as an "orphan" has been modified to more closely match the behavior as that of persistent objects, which is that the

object is expunged from the *Session* as soon as it is de-associated from any of its orphan-enabled parents. Previously, the pending object would be expunged only if de-associated from all of its orphan-enabled parents. The new flag `legacy_is_orphan` is added to *orm.mapper()* which re-establishes the legacy behavior.

- **non_primary** – Specify that this *Mapper* is in addition to the "primary" mapper, that is, the one used for persistence. The *Mapper* created here may be used for ad-hoc mapping of the class to an alternate selectable, for loading only.

  Mapper.non_primary is not an often used option, but is useful in some specific *relationship()* cases.

  See also:

  Relationship to Non Primary Mapper

- **order_by** – A single *Column* or list of *Column* objects for which selection operations should use as the default ordering for entities. By default mappers have no pre-defined ordering.

- **passive_updates** – Indicates UPDATE behavior of foreign key columns when a primary key column changes on a joined-table inheritance mapping. Defaults to `True`.

  When True, it is assumed that ON UPDATE CASCADE is configured on the foreign key in the database, and that the database will handle propagation of an UPDATE from a source column to dependent columns on joined-table rows.

  When False, it is assumed that the database does not enforce referential integrity and will not be issuing its own CASCADE operation for an update. The unit of work process will emit an UPDATE statement for the dependent columns during a primary key change.

  See also:

  Mutable Primary Keys / Update Cascades - description of a similar feature as used with *relationship()*

- **polymorphic_on** – Specifies the column, attribute, or SQL expression used to determine the target class for an incoming row, when inheriting classes are present.

  This value is commonly a *Column* object that's present in the mapped *Table*:

```
class Employee(Base):
    __tablename__ = 'employee'

    id = Column(Integer, primary_key=True)
    discriminator = Column(String(50))

    __mapper_args__ = {
        "polymorphic_on":discriminator,
        "polymorphic_identity":"employee"
    }
```

  It may also be specified as a SQL expression, as in this example where we use the *case()* construct to provide a conditional approach:

```
class Employee(Base):
    __tablename__ = 'employee'

    id = Column(Integer, primary_key=True)
    discriminator = Column(String(50))
```

```
    __mapper_args__ = {
        "polymorphic_on":case([
            (discriminator == "EN", "engineer"),
            (discriminator == "MA", "manager"),
        ], else_="employee"),
        "polymorphic_identity":"employee"
    }
```

It may also refer to any attribute configured with *column_property()*, or to the string name of one:

```
class Employee(Base):
    __tablename__ = 'employee'

    id = Column(Integer, primary_key=True)
    discriminator = Column(String(50))
    employee_type = column_property(
        case([
            (discriminator == "EN", "engineer"),
            (discriminator == "MA", "manager"),
        ], else_="employee")
    )

    __mapper_args__ = {
        "polymorphic_on":employee_type,
        "polymorphic_identity":"employee"
    }
```

Changed in version 0.7.4: `polymorphic_on` may be specified as a SQL expression, or refer to any attribute configured with *column_property()*, or to the string name of one.

When setting `polymorphic_on` to reference an attribute or expression that's not present in the locally mapped *Table*, yet the value of the discriminator should be persisted to the database, the value of the discriminator is not automatically set on new instances; this must be handled by the user, either through manual means or via event listeners. A typical approach to establishing such a listener looks like:

```
from sqlalchemy import event
from sqlalchemy.orm import object_mapper

@event.listens_for(Employee, "init", propagate=True)
def set_identity(instance, *arg, **kw):
    mapper = object_mapper(instance)
    instance.discriminator = mapper.polymorphic_identity
```

Where above, we assign the value of `polymorphic_identity` for the mapped class to the `discriminator` attribute, thus persisting the value to the `discriminator` column in the database.

> **Warning:** Currently, **only one discriminator column may be set**, typically on the base-most class in the hierarchy. "Cascading" polymorphic columns are not yet supported.

See also:

Mapping Class Inheritance Hierarchies

- **polymorphic_identity** – Specifies the value which identifies this particular class as returned by the column expression referred to by the `polymorphic_on` setting. As rows are received, the value corresponding to the `polymorphic_on` column expression is compared to this value, indicating which subclass should be used for the newly reconstructed object.
- **properties** – A dictionary mapping the string names of object attributes to *MapperProperty* instances, which define the persistence behavior of that attribute. Note that *Column* objects present in the mapped *Table* are automatically placed into `ColumnProperty` instances upon mapping, unless overridden. When using Declarative, this argument is passed automatically, based on all those *MapperProperty* instances declared in the declared class body.
- **primary_key** – A list of *Column* objects which define the primary key to be used against this mapper's selectable unit. This is normally simply the primary key of the `local_table`, but can be overridden here.
- **version_id_col** – A *Column* that will be used to keep a running version id of rows in the table. This is used to detect concurrent updates or the presence of stale data in a flush. The methodology is to detect if an UPDATE statement does not match the last known version id, a *StaleDataError* exception is thrown. By default, the column must be of *Integer* type, unless `version_id_generator` specifies an alternative version generator.

  See also:

  Configuring a Version Counter - discussion of version counting and rationale.
- **version_id_generator** – Define how new version ids should be generated. Defaults to `None`, which indicates that a simple integer counting scheme be employed. To provide a custom versioning scheme, provide a callable function of the form:

```
def generate_version(version):
    return next_version
```

  Alternatively, server-side versioning functions such as triggers, or programmatic versioning schemes outside of the version id generator may be used, by specifying the value `False`. Please see *Server Side Version Counters* for a discussion of important points when using this option.

  New in version 0.9.0: `version_id_generator` supports server-side version number generation.

  See also:

  Custom Version Counters / Types

  Server Side Version Counters
- **with_polymorphic** – A tuple in the form `(<classes>, <selectable>)` indicating the default style of "polymorphic" loading, that is, which tables are queried at once. <classes> is any single or list of mappers and/or classes indicating the inherited classes that should be loaded at once. The special value `'*'` may be used to indicate all descending classes should be loaded immediately. The second tuple argument <selectable> indicates a selectable that will be used to query for multiple classes.

  See also:

  Basic Control of Which Tables are Queried - discussion of polymorphic querying techniques.

sqlalchemy.orm.**object_mapper**(*instance*)

Given an object, return the primary Mapper associated with the object instance.

Raises `sqlalchemy.orm.exc.UnmappedInstanceError` if no mapping is configured.

This function is available via the inspection system as:

```
inspect(instance).mapper
```

Using the inspection system will raise `sqlalchemy.exc.NoInspectionAvailable` if the instance is not part of a mapping.

sqlalchemy.orm.**class_mapper**(*class_*, *configure=True*)

Given a class, return the primary `Mapper` associated with the key.

Raises `UnmappedClassError` if no mapping is configured on the given class, or `ArgumentError` if a non-class object is passed.

Equivalent functionality is available via the `inspect()` function as:

```
inspect(some_mapped_class)
```

Using the inspection system will raise `sqlalchemy.exc.NoInspectionAvailable` if the class is not mapped.

sqlalchemy.orm.**configure_mappers**()

Initialize the inter-mapper relationships of all mappers that have been constructed thus far.

This function can be called any number of times, but in most cases is invoked automatically, the first time mappings are used, as well as whenever mappings are used and additional not-yet-configured mappers have been constructed.

Points at which this occur include when a mapped class is instantiated into an instance, as well as when the `Session.query()` method is used.

The `configure_mappers()` function provides several event hooks that can be used to augment its functionality. These methods include:

- `MapperEvents.before_configured()` - called once before `configure_mappers()` does any work; this can be used to establish additional options, properties, or related mappings before the operation proceeds.
- `MapperEvents.mapper_configured()` - called as each indivudal `Mapper` is configured within the process; will include all mapper state except for backrefs set up by other mappers that are still to be configured.
- `MapperEvents.after_configured()` - called once after `configure_mappers()` is complete; at this stage, all `Mapper` objects that are known to SQLAlchemy will be fully configured. Note that the calling application may still have other mappings that haven't been produced yet, such as if they are in modules as yet unimported.

sqlalchemy.orm.**clear_mappers**()

Remove all mappers from all classes.

This function removes all instrumentation from classes and disposes of their associated mappers. Once called, the classes are unmapped and can be later re-mapped with new mappers.

`clear_mappers()` is *not* for normal use, as there is literally no valid usage for it outside of very specific testing scenarios. Normally, mappers are permanent structural components of user-defined classes, and are never discarded independently of their class. If a mapped class itself is garbage collected, its mapper is automatically disposed of as well. As such, `clear_mappers()` is only for usage in test suites that re-use the same classes with different mappings, which is itself an extremely rare use case - the only such use case is in fact

SQLAlchemy's own test suite, and possibly the test suites of other ORM extension libraries which intend to test various combinations of mapper construction upon a fixed set of classes.

`sqlalchemy.orm.util.identity_key`(**args*, ***kwargs*)

Generate "identity key" tuples, as are used as keys in the `Session.identity_map` dictionary.

This function has several call styles:

- `identity_key(class, ident)`

  This form receives a mapped class and a primary key scalar or tuple as an argument.

  E.g.:

```
>>> identity_key(MyClass, (1, 2))
(<class '__main__.MyClass'>, (1, 2))
```

  param class mapped class (must be a positional argument)

  param ident primary key, may be a scalar or tuple argument.

- `identity_key(instance=instance)`

  This form will produce the identity key for a given instance. The instance need not be persistent, only that its primary key attributes are populated (else the key will contain `None` for those missing values).

  E.g.:

```
>>> instance = MyClass(1, 2)
>>> identity_key(instance=instance)
(<class '__main__.MyClass'>, (1, 2))
```

  In this form, the given instance is ultimately run though `Mapper.identity_key_from_instance()`, which will have the effect of performing a database check for the corresponding row if the object is expired.

  param instance object instance (must be given as a keyword arg)

- `identity_key(class, row=row)`

  This form is similar to the class/tuple form, except is passed a database result row as a `RowProxy` object.

  E.g.:

```
>>> row = engine.execute("select * from table where a=1 and b=2").first()
>>> identity_key(MyClass, row=row)
(<class '__main__.MyClass'>, (1, 2))
```

  param class mapped class (must be a positional argument)

  param row `RowProxy` row returned by a `ResultProxy` (must be given as a keyword arg)

`sqlalchemy.orm.util.polymorphic_union`(*table_map*, *typecolname*, *aliasname='p_union'*, *cast_nulls=True*)

Create a `UNION` statement used by a polymorphic mapper.

See *Concrete Table Inheritance* for an example of how this is used.

Parameters

- **`table_map`** – mapping of polymorphic identities to `Table` objects.

- **typecolname** – string name of a "discriminator" column, which will be derived from the query, producing the polymorphic identity for each row. If `None`, no polymorphic discriminator is generated.
- **aliasname** – name of the `alias()` construct generated.
- **cast_nulls** – if True, non-existent columns, which are represented as labeled NULLs, will be passed into CAST. This is a legacy behavior that is problematic on some backends such as Oracle - in which case it can be set to False.

class `sqlalchemy.orm.mapper.`**`Mapper`**(*class_, local_table=None, properties=None, primary_key=None, non_primary=False, inherits=None, inherit_condition=None, inherit_foreign_keys=None, extension=None, order_by=False, always_refresh=False, version_id_col=None, version_id_generator=None, polymorphic_on=None, _polymorphic_map=None, polymorphic_identity=None, concrete=False, with_polymorphic=None, allow_partial_pks=True, batch=True, column_prefix=None, include_properties=None, exclude_properties=None, passive_updates=True, confirm_deleted_rows=True, eager_defaults=False, legacy_is_orphan=False, _compiled_cache_size=100*)

Bases: `sqlalchemy.orm.base.InspectionAttr`

Define the correlation of class attributes to database table columns.

The `Mapper` object is instantiated using the `mapper()` function. For information about instantiating new `Mapper` objects, see that function's documentation.

When `mapper()` is used explicitly to link a user defined class with table metadata, this is referred to as *classical mapping*. Modern SQLAlchemy usage tends to favor the `sqlalchemy.ext.declarative` extension for class configuration, which makes usage of `mapper()` behind the scenes.

Given a particular class known to be mapped by the ORM, the `Mapper` which maintains it can be acquired using the `inspect()` function:

```
from sqlalchemy import inspect

mapper = inspect(MyClass)
```

A class which was mapped by the `sqlalchemy.ext.declarative` extension will also have its mapper available via the `__mapper__` attribute.

`__init__`(*class_, local_table=None, properties=None, primary_key=None, non_primary=False, inherits=None, inherit_condition=None, inherit_foreign_keys=None, extension=None, order_by=False, always_refresh=False, version_id_col=None, version_id_generator=None, polymorphic_on=None, _polymorphic_map=None, polymorphic_identity=None, concrete=False, with_polymorphic=None, allow_partial_pks=True, batch=True, column_prefix=None, include_properties=None, exclude_properties=None, passive_updates=True, confirm_deleted_rows=True, eager_defaults=False, legacy_is_orphan=False, _compiled_cache_size=100*)

Construct a new `Mapper` object.

This constructor is mirrored as a public API function; see `mapper()` for a full usage and argument description.

`add_properties`(*dict_of_properties*)

Add the given dictionary of properties to this mapper, using *add_property*.

`add_property`(*key, prop*)

Add an individual MapperProperty to this mapper.

If the mapper has not been configured yet, just adds the property to the initial properties dictionary sent to the constructor. If this Mapper has already been configured, then the given MapperProperty is configured immediately.

all_orm_descriptors

A namespace of all *InspectionAttr* attributes associated with the mapped class.

These attributes are in all cases Python descriptors associated with the mapped class or its superclasses.

This namespace includes attributes that are mapped to the class as well as attributes declared by extension modules. It includes any Python descriptor type that inherits from *InspectionAttr*. This includes *QueryableAttribute*, as well as extension types such as *hybrid_property*, *hybrid_method* and *AssociationProxy*.

To distinguish between mapped attributes and extension attributes, the attribute *InspectionAttr.extension_type* will refer to a constant that distinguishes between different extension types.

When dealing with a *QueryableAttribute*, the *QueryableAttribute.property* attribute refers to the *MapperProperty* property, which is what you get when referring to the collection of mapped properties via *Mapper.attrs*.

> **Warning:** the *Mapper.relationships* accessor namespace is an instance of `OrderedProperties`. This is a dictionary-like object which includes a small number of named methods such as `OrderedProperties.items()` and `OrderedProperties.values()`. When accessing attributes dynamically, favor using the dict-access scheme, e.g. `mapper.attrs[somename]` over `getattr(mapper.attrs, somename)` to avoid name collisions.

New in version 0.8.0.

See also:

Mapper.attrs

attrs

A namespace of all *MapperProperty* objects associated this mapper.

This is an object that provides each property based on its key name. For instance, the mapper for a `User` class which has `User.name` attribute would provide `mapper.attrs.name`, which would be the *ColumnProperty* representing the `name` column. The namespace object can also be iterated, which would yield each *MapperProperty*.

Mapper has several pre-filtered views of this attribute which limit the types of properties returned, incldding *synonyms*, *column_attrs*, *relationships*, and *composites*.

> **Warning:** the *Mapper.relationships* accessor namespace is an instance of `OrderedProperties`. This is a dictionary-like object which includes a small number of named methods such as `OrderedProperties.items()` and `OrderedProperties.values()`. When accessing attributes dynamically, favor using the dict-access scheme, e.g. `mapper.attrs[somename]` over `getattr(mapper.attrs, somename)` to avoid name collisions.

See also:

Mapper.all_orm_descriptors

base_mapper = None

The base-most *Mapper* in an inheritance chain.

In a non-inheriting scenario, this attribute will always be this `Mapper`. In an inheritance scenario, it references the `Mapper` which is parent to all other `Mapper` objects in the inheritance chain.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

c = **None**

A synonym for `columns`.

cascade_iterator(*type_*, *state*, *halt_on=None*)

Iterate each element and its mapper in an object graph, for all relationships that meet the given cascade rule.

Parameters

- **type_** – The name of the cascade rule (i.e. `"save-update"`, `"delete"`, etc.).

  Note: the `"all"` cascade is not accepted here. For a generic object traversal function, see *How do I walk all objects that are related to a given object?*.

- **state** – The lead InstanceState. child items will be processed per the relationships defined for this object's mapper.

Returns the method yields individual object instances.

See also:

Cascades

How do I walk all objects that are related to a given object? - illustrates a generic function to traverse all objects without relying on cascades.

class_ = **None**

The Python class which this `Mapper` maps.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

class_manager = **None**

The `ClassManager` which maintains event listeners and class-bound descriptors for this `Mapper`.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

column_attrs

Return a namespace of all `ColumnProperty` properties maintained by this `Mapper`.

See also:

`Mapper.attrs` - namespace of all `MapperProperty` objects.

columns = **None**

A collection of `Column` or other scalar expression objects maintained by this `Mapper`.

The collection behaves the same as that of the `c` attribute on any `Table` object, except that only those columns included in this mapping are present, and are keyed based on the attribute name defined in the mapping, not necessarily the `key` attribute of the `Column` itself. Additionally, scalar expressions mapped by `column_property()` are also present here.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

common_parent(*other*)

Return true if the given mapper shares a common inherited parent as this mapper.

composites

Return a namespace of all `CompositeProperty` properties maintained by this `Mapper`.

See also:

`Mapper.attrs` - namespace of all `MapperProperty` objects.

concrete = None

Represent `True` if this `Mapper` is a concrete inheritance mapper.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

configured = None

Represent `True` if this `Mapper` has been configured.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

See also:

`configure_mappers()`.

entity

Part of the inspection API.

Returns self.class_.

get_property(*key*, *_configure_mappers=True*)

return a MapperProperty associated with the given key.

get_property_by_column(*column*)

Given a `Column` object, return the `MapperProperty` which maps this column.

identity_key_from_instance(*instance*)

Return the identity key for the given instance, based on its primary key attributes.

If the instance's state is expired, calling this method will result in a database check to see if the object has been deleted. If the row no longer exists, `ObjectDeletedError` is raised.

This value is typically also found on the instance state under the attribute name *key*.

identity_key_from_primary_key(*primary_key*)

Return an identity-map key for use in storing/retrieving an item from an identity map.

Parameters **primary_key** – A list of values indicating the identifier.

identity_key_from_row(*row*, *adapter=None*)

Return an identity-map key for use in storing/retrieving an item from the identity map.

Parameters **row** – A `RowProxy` instance. The columns which are mapped by this `Mapper` should be locatable in the row, preferably via the `Column` object directly (as is the case when a `select()` construct is executed), or via string names of the form `<tablename>_<colname>`.

inherits = None

References the `Mapper` which this `Mapper` inherits from, if any.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

is_mapper = True

Part of the inspection API.

isa(*other*)
Return True if the this mapper inherits from the given mapper.

iterate_properties
return an iterator of all MapperProperty objects.

local_table = None
The *Selectable* which this *Mapper* manages.

Typically is an instance of *Table* or *Alias*. May also be `None`.

The "local" table is the selectable that the *Mapper* is directly responsible for managing from an attribute access and flush perspective. For non-inheriting mappers, the local table is the same as the "mapped" table. For joined-table inheritance mappers, local_table will be the particular sub-table of the overall "join" which this *Mapper* represents. If this mapper is a single-table inheriting mapper, local_table will be `None`.

See also:

mapped_table.

mapped_table = None
The *Selectable* to which this *Mapper* is mapped.

Typically an instance of *Table*, *Join*, or *Alias*.

The "mapped" table is the selectable that the mapper selects from during queries. For non-inheriting mappers, the mapped table is the same as the "local" table. For joined-table inheritance mappers, mapped_table references the full *Join* representing full rows for this particular subclass. For single-table inheritance mappers, mapped_table references the base table.

See also:

local_table.

mapper
Part of the inspection API.

Returns self.

non_primary = None
Represent `True` if this *Mapper* is a "non-primary" mapper, e.g. a mapper that is used only to selet rows but not for persistence management.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

polymorphic_identity = None
Represent an identifier which is matched against the *polymorphic_on* column during result row loading.

Used only with inheritance, this object can be of any type which is comparable to the type of column represented by *polymorphic_on*.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

polymorphic_iterator()
Iterate through the collection including this mapper and all descendant mappers.

This includes not just the immediately inheriting mappers but all their inheriting mappers as well.

To iterate through an entire hierarchy, use `mapper.base_mapper.polymorphic_iterator()`.

polymorphic_map = None

A mapping of "polymorphic identity" identifiers mapped to *Mapper* instances, within an inheritance scenario.

The identifiers can be of any type which is comparable to the type of column represented by *polymorphic_on*.

An inheritance chain of mappers will all reference the same polymorphic map object. The object is used to correlate incoming result rows to target mappers.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

polymorphic_on = None

The *Column* or SQL expression specified as the `polymorphic_on` argument for this *Mapper*, within an inheritance scenario.

This attribute is normally a *Column* instance but may also be an expression, such as one derived from *cast()*.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

primary_key = None

An iterable containing the collection of *Column* objects which comprise the 'primary key' of the mapped table, from the perspective of this *Mapper*.

This list is against the selectable in *mapped_table*. In the case of inheriting mappers, some columns may be managed by a superclass mapper. For example, in the case of a *Join*, the primary key is determined by all of the primary key columns across all tables referenced by the *Join*.

The list is also not necessarily the same as the primary key column collection associated with the underlying tables; the *Mapper* features a `primary_key` argument that can override what the *Mapper* considers as primary key columns.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

primary_key_from_instance(*instance*)

Return the list of primary key values for the given instance.

If the instance's state is expired, calling this method will result in a database check to see if the object has been deleted. If the row no longer exists, *ObjectDeletedError* is raised.

primary_mapper()

Return the primary mapper corresponding to this mapper's class key (class).

relationships

Return a namespace of all *RelationshipProperty* properties maintained by this *Mapper*.

> **Warning:** the *Mapper.relationships* accessor namespace is an instance of `OrderedProperties`. This is a dictionary-like object which includes a small number of named methods such as `OrderedProperties.items()` and `OrderedProperties.values()`. When accessing attributes dynamically, favor using the dict-access scheme, e.g. `mapper.attrs[somename]` over `getattr(mapper.attrs, somename)` to avoid name collisions.

See also:

Mapper.attrs - namespace of all *MapperProperty* objects.

selectable
The *select()* construct this *Mapper* selects from by default.

Normally, this is equivalent to *mapped_table*, unless the `with_polymorphic` feature is in use, in which case the full "polymorphic" selectable is returned.

self_and_descendants
The collection including this mapper and all descendant mappers.

This includes not just the immediately inheriting mappers but all their inheriting mappers as well.

single = None
Represent `True` if this *Mapper* is a single table inheritance mapper.

local_table will be `None` if this flag is set.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

synonyms
Return a namespace of all *SynonymProperty* properties maintained by this *Mapper*.

See also:

Mapper.attrs - namespace of all *MapperProperty* objects.

tables = None
An iterable containing the collection of *Table* objects which this *Mapper* is aware of.

If the mapper is mapped to a *Join*, or an *Alias* representing a *Select*, the individual *Table* objects that comprise the full construct will be represented here.

This is a *read only* attribute determined during mapper construction. Behavior is undefined if directly modified.

validators = None
An immutable dictionary of attributes which have been decorated using the *validates()* decorator.

The dictionary contains string attribute names as keys mapped to the actual validation method.

with_polymorphic_mappers
The list of *Mapper* objects included in the default "polymorphic" query.

2.3 Relationship Configuration

This section describes the *relationship()* function and in depth discussion of its usage. For an introduction to relationships, start with the *Object Relational Tutorial* and head into *Building a Relationship*.

2.3.1 Basic Relationship Patterns

A quick walkthrough of the basic relational patterns.

The imports used for each of the following sections is as follows:

```
from sqlalchemy import Table, Column, Integer, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
```

One To Many

A one to many relationship places a foreign key on the child table referencing the parent. *relationship()* is then specified on the parent, as referencing a collection of items represented by the child:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
```

To establish a bidirectional relationship in one-to-many, where the "reverse" side is a many to one, specify an additional *relationship()* and connect the two using the *relationship.back_populates* parameter:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child", back_populates="parent")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
    parent = relationship("Parent", back_populates="children")
```

`Child` will get a `parent` attribute with many-to-one semantics.

Alternatively, the *backref* option may be used on a single *relationship()* instead of using *back_populates*:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child", backref="parent")
```

Many To One

Many to one places a foreign key in the parent table referencing the child. *relationship()* is declared on the parent, where a new scalar-holding attribute will be created:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
```

Bidirectional behavior is achieved by adding a second *relationship()* and applying the *relationship.back_populates* parameter in both directions:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", back_populates="parents")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parents = relationship("Parent", back_populates="child")
```

Alternatively, the *backref* parameter may be applied to a single *relationship()*, such as `Parent.child`:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", backref="parents")
```

One To One

One To One is essentially a bidirectional relationship with a scalar attribute on both sides. To achieve this, the *uselist* flag indicates the placement of a scalar attribute instead of a collection on the "many" side of the relationship. To convert one-to-many into one-to-one:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child = relationship("Child", uselist=False, back_populates="parent")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))
    parent = relationship("Child", back_populates="child")
```

Or for many-to-one:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", back_populates="parent")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent = relationship("Child", back_populates="child", uselist=False)
```

As always, the *relationship.backref* and *backref()* functions may be used in lieu of the *relationship.back_populates* approach; to specify `uselist` on a backref, use the *backref()* function:

```
from sqlalchemy.orm import backref

class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
```

```
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", backref=backref("parent", uselist=False))
```

Many To Many

Many to Many adds an association table between two classes. The association table is indicated by the *secondary* argument to *relationship()*. Usually, the *Table* uses the *MetaData* object associated with the declarative base class, so that the *ForeignKey* directives can locate the remote tables with which to link:

```
association_table = Table('association', Base.metadata,
    Column('left_id', Integer, ForeignKey('left.id')),
    Column('right_id', Integer, ForeignKey('right.id'))
)

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Child",
                    secondary=association_table)

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

For a bidirectional relationship, both sides of the relationship contain a collection. Specify using *relationship.back_populates*, and for each *relationship()* specify the common association table:

```
association_table = Table('association', Base.metadata,
    Column('left_id', Integer, ForeignKey('left.id')),
    Column('right_id', Integer, ForeignKey('right.id'))
)

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship(
        "Child",
        secondary=association_table,
        back_populates="parents")

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
    parents = relationship(
        "Parent",
        secondary=association_table,
        back_populates="children")
```

When using the *backref* parameter instead of *relationship.back_populates*, the backref will automatically use the same *secondary* argument for the reverse relationship:

```
association_table = Table('association', Base.metadata,
    Column('left_id', Integer, ForeignKey('left.id')),
    Column('right_id', Integer, ForeignKey('right.id'))
)

class Parent(Base):
```

```
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Child",
                    secondary=association_table,
                    backref="parents")

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

The *secondary* argument of *relationship()* also accepts a callable that returns the ultimate argument, which is evaluated only when mappers are first used. Using this, we can define the `association_table` at a later point, as long as it's available to the callable after all module initialization is complete:

```
class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Child",
                    secondary=lambda: association_table,
                    backref="parents")
```

With the declarative extension in use, the traditional "string name of the table" is accepted as well, matching the name of the table as stored in `Base.metadata.tables`:

```
class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Child",
                    secondary="association",
                    backref="parents")
```

Deleting Rows from the Many to Many Table

A behavior which is unique to the *secondary* argument to *relationship()* is that the *Table* which is specified here is automatically subject to INSERT and DELETE statements, as objects are added or removed from the collection. There is **no need to delete from this table manually**. The act of removing a record from the collection will have the effect of the row being deleted on flush:

```
# row will be deleted from the "secondary" table
# automatically
myparent.children.remove(somechild)
```

A question which often arises is how the row in the "secondary" table can be deleted when the child object is handed directly to *Session.delete()*:

```
session.delete(somechild)
```

There are several possibilities here:

- If there is a *relationship()* from `Parent` to `Child`, but there is **not** a reverse-relationship that links a particular `Child` to each `Parent`, SQLAlchemy will not have any awareness that when deleting this particular `Child` object, it needs to maintain the "secondary" table that links it to the `Parent`. No delete of the "secondary" table will occur.
- If there is a relationship that links a particular `Child` to each `Parent`, suppose it's called `Child.parents`, SQLAlchemy by default will load in the `Child.parents` collection to locate all `Parent` objects, and remove each row from the "secondary" table which establishes this link. Note that this relationship does not need to be

bidirectional; SQLAlchemy is strictly looking at every *relationship()* associated with the `Child` object being deleted.

- A higher performing option here is to use ON DELETE CASCADE directives with the foreign keys used by the database. Assuming the database supports this feature, the database itself can be made to automatically delete rows in the "secondary" table as referencing rows in "child" are deleted. SQLAlchemy can be instructed to forego actively loading in the `Child.parents` collection in this case using the *passive_deletes* directive on *relationship()*; see *Using Passive Deletes* for more details on this.

Note again, these behaviors are *only* relevant to the *secondary* option used with *relationship()*. If dealing with association tables that are mapped explicitly and are *not* present in the *secondary* option of a relevant *relationship()*, cascade rules can be used instead to automatically delete entities in reaction to a related entity being deleted - see *Cascades* for information on this feature.

Association Object

The association object pattern is a variant on many-to-many: it's used when your association table contains additional columns beyond those which are foreign keys to the left and right tables. Instead of using the *secondary* argument, you map a new class directly to the association table. The left side of the relationship references the association object via one-to-many, and the association class references the right side via many-to-one. Below we illustrate an association table mapped to the `Association` class which includes a column called `extra_data`, which is a string value that is stored along with each association between `Parent` and `Child`:

```
class Association(Base):
    __tablename__ = 'association'
    left_id = Column(Integer, ForeignKey('left.id'), primary_key=True)
    right_id = Column(Integer, ForeignKey('right.id'), primary_key=True)
    extra_data = Column(String(50))
    child = relationship("Child")

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Association")

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

As always, the bidirectional version make use of *relationship.back_populates* or *relationship.backref*:

```
class Association(Base):
    __tablename__ = 'association'
    left_id = Column(Integer, ForeignKey('left.id'), primary_key=True)
    right_id = Column(Integer, ForeignKey('right.id'), primary_key=True)
    extra_data = Column(String(50))
    child = relationship("Child", back_populates="parents")
    parent = relationship("Parent", back_populates="children")

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Association", back_populates="parent")

class Child(Base):
    __tablename__ = 'right'
```

```
    id = Column(Integer, primary_key=True)
    parents = relationship("Association", back_populates="child")
```

Working with the association pattern in its direct form requires that child objects are associated with an association instance before being appended to the parent; similarly, access from parent to child goes through the association object:

```
# create parent, append a child via association
p = Parent()
a = Association(extra_data="some data")
a.child = Child()
p.children.append(a)

# iterate through child objects via association, including association
# attributes
for assoc in p.children:
    print assoc.extra_data
    print assoc.child
```

To enhance the association object pattern such that direct access to the `Association` object is optional, SQLAlchemy provides the *Association Proxy* extension. This extension allows the configuration of attributes which will access two "hops" with a single access, one "hop" to the associated object, and a second to a target attribute.

Note: When using the association object pattern, it is advisable that the association-mapped table not be used as the `secondary` argument on a `relationship()` elsewhere, unless that `relationship()` contains the option `viewonly` set to `True`. SQLAlchemy otherwise may attempt to emit redundant INSERT and DELETE statements on the same table, if similar state is detected on the related attribute as well as the associated object.

2.3.2 Adjacency List Relationships

The **adjacency list** pattern is a common relational pattern whereby a table contains a foreign key reference to itself. This is the most common way to represent hierarchical data in flat tables. Other methods include **nested sets**, sometimes called "modified preorder", as well as **materialized path**. Despite the appeal that modified preorder has when evaluated for its fluency within SQL queries, the adjacency list model is probably the most appropriate pattern for the large majority of hierarchical storage needs, for reasons of concurrency, reduced complexity, and that modified preorder has little advantage over an application which can fully load subtrees into the application space.

In this example, we'll work with a single mapped class called `Node`, representing a tree structure:

```
class Node(Base):
    __tablename__ = 'node'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('node.id'))
    data = Column(String(50))
    children = relationship("Node")
```

With this structure, a graph such as the following:

```
root --+---> child1
       +---> child2 --+--> subchild1
       |              +--> subchild2
       +---> child3
```

Would be represented with data such as:

```
id       parent_id     data
---      -------       ----
1        NULL          root
```

```
2         1              child1
3         1              child2
4         3              subchild1
5         3              subchild2
6         1              child3
```

The *relationship()* configuration here works in the same way as a "normal" one-to-many relationship, with the exception that the "direction", i.e. whether the relationship is one-to-many or many-to-one, is assumed by default to be one-to-many. To establish the relationship as many-to-one, an extra directive is added known as *remote_side*, which is a *Column* or collection of *Column* objects that indicate those which should be considered to be "remote":

```
class Node(Base):
    __tablename__ = 'node'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('node.id'))
    data = Column(String(50))
    parent = relationship("Node", remote_side=[id])
```

Where above, the `id` column is applied as the *remote_side* of the `parent` *relationship()*, thus establishing `parent_id` as the "local" side, and the relationship then behaves as a many-to-one.

As always, both directions can be combined into a bidirectional relationship using the *backref()* function:

```
class Node(Base):
    __tablename__ = 'node'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('node.id'))
    data = Column(String(50))
    children = relationship("Node",
                backref=backref('parent', remote_side=[id])
            )
```

There are several examples included with SQLAlchemy illustrating self-referential strategies; these include *Adjacency List* and *XML Persistence*.

Composite Adjacency Lists

A sub-category of the adjacency list relationship is the rare case where a particular column is present on both the "local" and "remote" side of the join condition. An example is the `Folder` class below; using a composite primary key, the `account_id` column refers to itself, to indicate sub folders which are within the same account as that of the parent; while `folder_id` refers to a specific folder within that account:

```
class Folder(Base):
    __tablename__ = 'folder'
    __table_args__ = (
      ForeignKeyConstraint(
          ['account_id', 'parent_id'],
          ['folder.account_id', 'folder.folder_id']),
    )

    account_id = Column(Integer, primary_key=True)
    folder_id = Column(Integer, primary_key=True)
    parent_id = Column(Integer)
    name = Column(String)

    parent_folder = relationship("Folder",
                        backref="child_folders",
```

```
                        remote_side=[account_id, folder_id]
                    )
```

Above, we pass `account_id` into the *remote_side* list. *relationship()* recognizes that the `account_id` column here is on both sides, and aligns the "remote" column along with the `folder_id` column, which it recognizes as uniquely present on the "remote" side.

New in version 0.8: Support for self-referential composite keys in *relationship()* where a column points to itself.

Self-Referential Query Strategies

Querying of self-referential structures works like any other query:

```
# get all nodes named 'child2'
session.query(Node).filter(Node.data=='child2')
```

However extra care is needed when attempting to join along the foreign key from one level of the tree to the next. In SQL, a join from a table to itself requires that at least one side of the expression be "aliased" so that it can be unambiguously referred to.

Recall from *Using Aliases* in the ORM tutorial that the *orm.aliased()* construct is normally used to provide an "alias" of an ORM entity. Joining from `Node` to itself using this technique looks like:

```
from sqlalchemy.orm import aliased

nodealias = aliased(Node)
session.query(Node).filter(Node.data=='subchild1').\
                join(nodealias, Node.parent).\
                filter(nodealias.data=="child2").\
                all()
SELECT node.id AS node_id,
        node.parent_id AS node_parent_id,
        node.data AS node_data
FROM node JOIN node AS node_1
    ON node.parent_id = node_1.id
WHERE node.data = ?
    AND node_1.data = ?
['subchild1', 'child2']
```

Query.join() also includes a feature known as *Query.join.aliased* that can shorten the verbosity self-referential joins, at the expense of query flexibility. This feature performs a similar "aliasing" step to that above, without the need for an explicit entity. Calls to *Query.filter()* and similar subsequent to the aliased join will **adapt** the `Node` entity to be that of the alias:

```
session.query(Node).filter(Node.data=='subchild1').\
        join(Node.parent, aliased=True).\
        filter(Node.data=='child2').\
        all()
SELECT node.id AS node_id,
        node.parent_id AS node_parent_id,
        node.data AS node_data
FROM node
    JOIN node AS node_1 ON node_1.id = node.parent_id
WHERE node.data = ? AND node_1.data = ?
['subchild1', 'child2']
```

To add criterion to multiple points along a longer join, add `Query.join.from_joinpoint` to the additional `join()` calls:

```
# get all nodes named 'subchild1' with a
# parent named 'child2' and a grandparent 'root'
session.query(Node).\
        filter(Node.data=='subchild1').\
        join(Node.parent, aliased=True).\
        filter(Node.data=='child2').\
        join(Node.parent, aliased=True, from_joinpoint=True).\
        filter(Node.data=='root').\
        all()
SELECT node.id AS node_id,
        node.parent_id AS node_parent_id,
        node.data AS node_data
FROM node
    JOIN node AS node_1 ON node_1.id = node.parent_id
    JOIN node AS node_2 ON node_2.id = node_1.parent_id
WHERE node.data = ?
    AND node_1.data = ?
    AND node_2.data = ?
['subchild1', 'child2', 'root']
```

`Query.reset_joinpoint()` will also remove the "aliasing" from filtering calls:

```
session.query(Node).\
        join(Node.children, aliased=True).\
        filter(Node.data == 'foo').\
        reset_joinpoint().\
        filter(Node.data == 'bar')
```

For an example of using `Query.join.aliased` to arbitrarily join along a chain of self-referential nodes, see *XML Persistence*.

Configuring Self-Referential Eager Loading

Eager loading of relationships occurs using joins or outerjoins from parent to child table during a normal query operation, such that the parent and its immediate child collection or reference can be populated from a single SQL statement, or a second statement for all immediate child collections. SQLAlchemy's joined and subquery eager loading use aliased tables in all cases when joining to related items, so are compatible with self-referential joining. However, to use eager loading with a self-referential relationship, SQLAlchemy needs to be told how many levels deep it should join and/or query; otherwise the eager load will not take place at all. This depth setting is configured via `join_depth`:

```
class Node(Base):
    __tablename__ = 'node'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('node.id'))
    data = Column(String(50))
    children = relationship("Node",
                    lazy="joined",
                    join_depth=2)

session.query(Node).all()
SELECT node_1.id AS node_1_id,
        node_1.parent_id AS node_1_parent_id,
        node_1.data AS node_1_data,
        node_2.id AS node_2_id,
        node_2.parent_id AS node_2_parent_id,
```

```
        node_2.data AS node_2_data,
        node.id AS node_id,
        node.parent_id AS node_parent_id,
        node.data AS node_data
FROM node
    LEFT OUTER JOIN node AS node_2
        ON node.id = node_2.parent_id
    LEFT OUTER JOIN node AS node_1
        ON node_2.id = node_1.parent_id
[]
```

2.3.3 Linking Relationships with Backref

The `backref` keyword argument was first introduced in *Object Relational Tutorial*, and has been mentioned throughout many of the examples here. What does it actually do ? Let's start with the canonical `User` and `Address` scenario:

```
from sqlalchemy import Integer, ForeignKey, String, Column
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String)

    addresses = relationship("Address", backref="user")

class Address(Base):
    __tablename__ = 'address'
    id = Column(Integer, primary_key=True)
    email = Column(String)
    user_id = Column(Integer, ForeignKey('user.id'))
```

The above configuration establishes a collection of `Address` objects on `User` called `User.addresses`. It also establishes a `.user` attribute on `Address` which will refer to the parent `User` object.

In fact, the `backref` keyword is only a common shortcut for placing a second `relationship()` onto the `Address` mapping, including the establishment of an event listener on both sides which will mirror attribute operations in both directions. The above configuration is equivalent to:

```
from sqlalchemy import Integer, ForeignKey, String, Column
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String)

    addresses = relationship("Address", back_populates="user")

class Address(Base):
    __tablename__ = 'address'
```

```
    id = Column(Integer, primary_key=True)
    email = Column(String)
    user_id = Column(Integer, ForeignKey('user.id'))

    user = relationship("User", back_populates="addresses")
```

Above, we add a `.user` relationship to `Address` explicitly. On both relationships, the *back_populates* directive tells each relationship about the other one, indicating that they should establish "bidirectional" behavior between each other. The primary effect of this configuration is that the relationship adds event handlers to both attributes which have the behavior of "when an append or set event occurs here, set ourselves onto the incoming attribute using this particular attribute name". The behavior is illustrated as follows. Start with a `User` and an `Address` instance. The `.addresses` collection is empty, and the `.user` attribute is `None`:

```
>>> u1 = User()
>>> a1 = Address()
>>> u1.addresses
[]
>>> print a1.user
None
```

However, once the `Address` is appended to the `u1.addresses` collection, both the collection and the scalar attribute have been populated:

```
>>> u1.addresses.append(a1)
>>> u1.addresses
[<__main__.Address object at 0x12a6ed0>]
>>> a1.user
<__main__.User object at 0x12a6590>
```

This behavior of course works in reverse for removal operations as well, as well as for equivalent operations on both sides. Such as when `.user` is set again to `None`, the `Address` object is removed from the reverse collection:

```
>>> a1.user = None
>>> u1.addresses
[]
```

The manipulation of the `.addresses` collection and the `.user` attribute occurs entirely in Python without any interaction with the SQL database. Without this behavior, the proper state would be apparent on both sides once the data has been flushed to the database, and later reloaded after a commit or expiration operation occurs. The *backref*/*back_populates* behavior has the advantage that common bidirectional operations can reflect the correct state without requiring a database round trip.

Remember, when the *backref* keyword is used on a single relationship, it's exactly the same as if the above two relationships were created individually using *back_populates* on each.

Backref Arguments

We've established that the *backref* keyword is merely a shortcut for building two individual *relationship()* constructs that refer to each other. Part of the behavior of this shortcut is that certain configurational arguments applied to the *relationship()* will also be applied to the other direction - namely those arguments that describe the relationship at a schema level, and are unlikely to be different in the reverse direction. The usual case here is a many-to-many *relationship()* that has a *secondary* argument, or a one-to-many or many-to-one which has a *primaryjoin* argument (the *primaryjoin* argument is discussed in *Specifying Alternate Join Conditions*). Such as if we limited the list of `Address` objects to those which start with "tony":

```
from sqlalchemy import Integer, ForeignKey, String, Column
from sqlalchemy.ext.declarative import declarative_base
```

```
from sqlalchemy.orm import relationship

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String)

    addresses = relationship("Address",
                    primaryjoin="and_(User.id==Address.user_id, "
                        "Address.email.startswith('tony'))",
                    backref="user")

class Address(Base):
    __tablename__ = 'address'
    id = Column(Integer, primary_key=True)
    email = Column(String)
    user_id = Column(Integer, ForeignKey('user.id'))
```

We can observe, by inspecting the resulting property, that both sides of the relationship have this join condition applied:

```
>>> print User.addresses.property.primaryjoin
"user".id = address.user_id AND address.email LIKE :email_1 || '%%'
>>>
>>> print Address.user.property.primaryjoin
"user".id = address.user_id AND address.email LIKE :email_1 || '%%'
>>>
```

This reuse of arguments should pretty much do the "right thing" - it uses only arguments that are applicable, and in the case of a many-to- many relationship, will reverse the usage of `primaryjoin` and `secondaryjoin` to correspond to the other direction (see the example in *Self-Referential Many-to-Many Relationship* for this).

It's very often the case however that we'd like to specify arguments that are specific to just the side where we happened to place the "backref". This includes `relationship()` arguments like `lazy`, `remote_side`, `cascade` and `cascade_backrefs`. For this case we use the `backref()` function in place of a string:

```
# <other imports>
from sqlalchemy.orm import backref

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String)

    addresses = relationship("Address",
                    backref=backref("user", lazy="joined"))
```

Where above, we placed a `lazy="joined"` directive only on the `Address.user` side, indicating that when a query against `Address` is made, a join to the `User` entity should be made automatically which will populate the `.user` attribute of each returned `Address`. The `backref()` function formatted the arguments we gave it into a form that is interpreted by the receiving `relationship()` as additional arguments to be applied to the new relationship it creates.

One Way Backrefs

An unusual case is that of the "one way backref". This is where the "back-populating" behavior of the backref is only desirable in one direction. An example of this is a collection which contains a filtering `primaryjoin` condition.

We'd like to append items to this collection as needed, and have them populate the "parent" object on the incoming object. However, we'd also like to have items that are not part of the collection, but still have the same "parent" association - these items should never be in the collection.

Taking our previous example, where we established a *primaryjoin* that limited the collection only to `Address` objects whose email address started with the word `tony`, the usual backref behavior is that all items populate in both directions. We wouldn't want this behavior for a case like the following:

```
>>> u1 = User()
>>> a1 = Address(email='mary')
>>> a1.user = u1
>>> u1.addresses
[<__main__.Address object at 0x1411910>]
```

Above, the `Address` object that doesn't match the criterion of "starts with 'tony'" is present in the `addresses` collection of `u1`. After these objects are flushed, the transaction committed and their attributes expired for a re-load, the `addresses` collection will hit the database on next access and no longer have this `Address` object present, due to the filtering condition. But we can do away with this unwanted side of the "backref" behavior on the Python side by using two separate *relationship()* constructs, placing *back_populates* only on one side:

```
from sqlalchemy import Integer, ForeignKey, String, Column
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    addresses = relationship("Address",
                    primaryjoin="and_(User.id==Address.user_id, "
                        "Address.email.startswith('tony'))",
                    back_populates="user")

class Address(Base):
    __tablename__ = 'address'
    id = Column(Integer, primary_key=True)
    email = Column(String)
    user_id = Column(Integer, ForeignKey('user.id'))
    user = relationship("User")
```

With the above scenario, appending an `Address` object to the `.addresses` collection of a `User` will always establish the `.user` attribute on that `Address`:

```
>>> u1 = User()
>>> a1 = Address(email='tony')
>>> u1.addresses.append(a1)
>>> a1.user
<__main__.User object at 0x1411850>
```

However, applying a `User` to the `.user` attribute of an `Address`, will not append the `Address` object to the collection:

```
>>> a2 = Address(email='mary')
>>> a2.user = u1
>>> a2 in u1.addresses
False
```

Of course, we've disabled some of the usefulness of *backref* here, in that when we do append an `Address` that

corresponds to the criteria of `email.startswith('tony')`, it won't show up in the `User.addresses` collection until the session is flushed, and the attributes reloaded after a commit or expire operation. While we could consider an attribute event that checks this criterion in Python, this starts to cross the line of duplicating too much SQL behavior in Python. The backref behavior itself is only a slight transgression of this philosophy - SQLAlchemy tries to keep these to a minimum overall.

2.3.4 Configuring how Relationship Joins

relationship() will normally create a join between two tables by examining the foreign key relationship between the two tables to determine which columns should be compared. There are a variety of situations where this behavior needs to be customized.

Handling Multiple Join Paths

One of the most common situations to deal with is when there are more than one foreign key path between two tables.

Consider a `Customer` class that contains two foreign keys to an `Address` class:

```
from sqlalchemy import Integer, ForeignKey, String, Column
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class Customer(Base):
    __tablename__ = 'customer'
    id = Column(Integer, primary_key=True)
    name = Column(String)

    billing_address_id = Column(Integer, ForeignKey("address.id"))
    shipping_address_id = Column(Integer, ForeignKey("address.id"))

    billing_address = relationship("Address")
    shipping_address = relationship("Address")

class Address(Base):
    __tablename__ = 'address'
    id = Column(Integer, primary_key=True)
    street = Column(String)
    city = Column(String)
    state = Column(String)
    zip = Column(String)
```

The above mapping, when we attempt to use it, will produce the error:

```
sqlalchemy.exc.AmbiguousForeignKeysError: Could not determine join
condition between parent/child tables on relationship
Customer.billing_address - there are multiple foreign key
paths linking the tables.  Specify the 'foreign_keys' argument,
providing a list of those columns which should be
counted as containing a foreign key reference to the parent table.
```

The above message is pretty long. There are many potential messages that *relationship()* can return, which have been carefully tailored to detect a variety of common configurational issues; most will suggest the additional configuration that's needed to resolve the ambiguity or other missing information.

In this case, the message wants us to qualify each *relationship()* by instructing for each one which foreign key column should be considered, and the appropriate form is as follows:

```
class Customer(Base):
    __tablename__ = 'customer'
    id = Column(Integer, primary_key=True)
    name = Column(String)

    billing_address_id = Column(Integer, ForeignKey("address.id"))
    shipping_address_id = Column(Integer, ForeignKey("address.id"))

    billing_address = relationship("Address", foreign_keys=[billing_address_id])
    shipping_address = relationship("Address", foreign_keys=[shipping_address_id])
```

Above, we specify the `foreign_keys` argument, which is a *Column* or list of *Column* objects which indicate those columns to be considered "foreign", or in other words, the columns that contain a value referring to a parent table. Loading the `Customer.billing_address` relationship from a `Customer` object will use the value present in `billing_address_id` in order to identify the row in `Address` to be loaded; similarly, `shipping_address_id` is used for the `shipping_address` relationship. The linkage of the two columns also plays a role during persistence; the newly generated primary key of a just-inserted `Address` object will be copied into the appropriate foreign key column of an associated `Customer` object during a flush.

When specifying `foreign_keys` with Declarative, we can also use string names to specify, however it is important that if using a list, the **list is part of the string**:

```
billing_address = relationship("Address", foreign_keys="[Customer.billing_address_id]")
```

In this specific example, the list is not necessary in any case as there's only one *Column* we need:

```
billing_address = relationship("Address", foreign_keys="Customer.billing_address_id")
```

Changed in version 0.8: *relationship()* can resolve ambiguity between foreign key targets on the basis of the `foreign_keys` argument alone; the *primaryjoin* argument is no longer needed in this situation.

Specifying Alternate Join Conditions

The default behavior of *relationship()* when constructing a join is that it equates the value of primary key columns on one side to that of foreign-key-referring columns on the other. We can change this criterion to be anything we'd like using the *primaryjoin* argument, as well as the *secondaryjoin* argument in the case when a "secondary" table is used.

In the example below, using the `User` class as well as an `Address` class which stores a street address, we create a relationship `boston_addresses` which will only load those `Address` objects which specify a city of "Boston":

```
from sqlalchemy import Integer, ForeignKey, String, Column
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    boston_addresses = relationship("Address",
                    primaryjoin="and_(User.id==Address.user_id, "
                        "Address.city=='Boston')")
```

```
class Address(Base):
    __tablename__ = 'address'
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('user.id'))

    street = Column(String)
    city = Column(String)
    state = Column(String)
    zip = Column(String)
```

Within this string SQL expression, we made use of the *and_()* conjunction construct to establish two distinct predicates for the join condition - joining both the `User.id` and `Address.user_id` columns to each other, as well as limiting rows in `Address` to just `city='Boston'`. When using Declarative, rudimentary SQL functions like *and_()* are automatically available in the evaluated namespace of a string *relationship()* argument.

The custom criteria we use in a *primaryjoin* is generally only significant when SQLAlchemy is rendering SQL in order to load or represent this relationship. That is, it's used in the SQL statement that's emitted in order to perform a per-attribute lazy load, or when a join is constructed at query time, such as via *Query.join()*, or via the eager "joined" or "subquery" styles of loading. When in-memory objects are being manipulated, we can place any `Address` object we'd like into the `boston_addresses` collection, regardless of what the value of the `.city` attribute is. The objects will remain present in the collection until the attribute is expired and re-loaded from the database where the criterion is applied. When a flush occurs, the objects inside of `boston_addresses` will be flushed unconditionally, assigning value of the primary key `user.id` column onto the foreign-key-holding `address.user_id` column for each row. The `city` criteria has no effect here, as the flush process only cares about synchronizing primary key values into referencing foreign key values.

Creating Custom Foreign Conditions

Another element of the primary join condition is how those columns considered "foreign" are determined. Usually, some subset of *Column* objects will specify *ForeignKey*, or otherwise be part of a *ForeignKeyConstraint* that's relevant to the join condition. *relationship()* looks to this foreign key status as it decides how it should load and persist data for this relationship. However, the *primaryjoin* argument can be used to create a join condition that doesn't involve any "schema" level foreign keys. We can combine *primaryjoin* along with *foreign_keys* and *remote_side* explicitly in order to establish such a join.

Below, a class `HostEntry` joins to itself, equating the string `content` column to the `ip_address` column, which is a Postgresql type called `INET`. We need to use *cast()* in order to cast one side of the join to the type of the other:

```
from sqlalchemy import cast, String, Column, Integer
from sqlalchemy.orm import relationship
from sqlalchemy.dialects.postgresql import INET

from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class HostEntry(Base):
    __tablename__ = 'host_entry'

    id = Column(Integer, primary_key=True)
    ip_address = Column(INET)
    content = Column(String(50))

    # relationship() using explicit foreign_keys, remote_side
    parent_host = relationship("HostEntry",
                        primaryjoin=ip_address == cast(content, INET),
```

```
                        foreign_keys=content,
                        remote_side=ip_address
                    )
```

The above relationship will produce a join like:

```
SELECT host_entry.id, host_entry.ip_address, host_entry.content
FROM host_entry JOIN host_entry AS host_entry_1
ON host_entry_1.ip_address = CAST(host_entry.content AS INET)
```

An alternative syntax to the above is to use the *foreign()* and *remote()* annotations, inline within the *primaryjoin* expression. This syntax represents the annotations that *relationship()* normally applies by itself to the join condition given the *foreign_keys* and *remote_side* arguments. These functions may be more succinct when an explicit join condition is present, and additionally serve to mark exactly the column that is "foreign" or "remote" independent of whether that column is stated multiple times or within complex SQL expressions:

```
from sqlalchemy.orm import foreign, remote

class HostEntry(Base):
    __tablename__ = 'host_entry'

    id = Column(Integer, primary_key=True)
    ip_address = Column(INET)
    content = Column(String(50))

    # relationship() using explicit foreign() and remote() annotations
    # in lieu of separate arguments
    parent_host = relationship("HostEntry",
                        primaryjoin=remote(ip_address) == \
                                cast(foreign(content), INET),
                    )
```

Using custom operators in join conditions

Another use case for relationships is the use of custom operators, such as Postgresql's "is contained within" << operator when joining with types such as *postgresql.INET* and *postgresql.CIDR*. For custom operators we use the *Operators.op()* function:

```
inet_column.op("<<")(cidr_column)
```

However, if we construct a *primaryjoin* using this operator, *relationship()* will still need more information. This is because when it examines our primaryjoin condition, it specifically looks for operators used for **comparisons**, and this is typically a fixed list containing known comparison operators such as ==, <, etc. So for our custom operator to participate in this system, we need it to register as a comparison operator using the *is_comparison* parameter:

```
inet_column.op("<<", is_comparison=True)(cidr_column)
```

A complete example:

```
class IPA(Base):
    __tablename__ = 'ip_address'

    id = Column(Integer, primary_key=True)
    v4address = Column(INET)

    network = relationship("Network",
                        primaryjoin="IPA.v4address.op('<<', is_comparison=True)"
```

```
                            "(foreign(Network.v4representation))",
                        viewonly=True
                    )
class Network(Base):
    __tablename__ = 'network'

    id = Column(Integer, primary_key=True)
    v4representation = Column(CIDR)
```

Above, a query such as:

```
session.query(IPA).join(IPA.network)
```

Will render as:

```
SELECT ip_address.id AS ip_address_id, ip_address.v4address AS ip_address_v4address
FROM ip_address JOIN network ON ip_address.v4address << network.v4representation
```

New in version 0.9.2: - Added the *Operators.op.is_comparison* flag to assist in the creation of *relationship()* constructs using custom operators.

Overlapping Foreign Keys

A rare scenario can arise when composite foreign keys are used, such that a single column may be the subject of more than one column referred to via foreign key constraint.

Consider an (admittedly complex) mapping such as the `Magazine` object, referred to both by the `Writer` object and the `Article` object using a composite primary key scheme that includes `magazine_id` for both; then to make `Article` refer to `Writer` as well, `Article.magazine_id` is involved in two separate relationships; `Article.magazine` and `Article.writer`:

```
class Magazine(Base):
    __tablename__ = 'magazine'

    id = Column(Integer, primary_key=True)


class Article(Base):
    __tablename__ = 'article'

    article_id = Column(Integer)
    magazine_id = Column(ForeignKey('magazine.id'))
    writer_id = Column()

    magazine = relationship("Magazine")
    writer = relationship("Writer")

    __table_args__ = (
        PrimaryKeyConstraint('article_id', 'magazine_id'),
        ForeignKeyConstraint(
            ['writer_id', 'magazine_id'],
            ['writer.id', 'writer.magazine_id']
        ),
    )


class Writer(Base):
    __tablename__ = 'writer'
```

```
    id = Column(Integer, primary_key=True)
    magazine_id = Column(ForeignKey('magazine.id'), primary_key=True)
    magazine = relationship("Magazine")
```

When the above mapping is configured, we will see this warning emitted:

```
SAWarning: relationship 'Article.writer' will copy column
writer.magazine_id to column article.magazine_id,
which conflicts with relationship(s): 'Article.magazine'
(copies magazine.id to article.magazine_id). Consider applying
viewonly=True to read-only relationships, or provide a primaryjoin
condition marking writable columns with the foreign() annotation.
```

What this refers to originates from the fact that `Article.magazine_id` is the subject of two different foreign key constraints; it refers to `Magazine.id` directly as a source column, but also refers to `Writer.magazine_id` as a source column in the context of the composite key to `Writer`. If we associate an `Article` with a particular `Magazine`, but then associate the `Article` with a `Writer` that's associated with a *different* `Magazine`, the ORM will overwrite `Article.magazine_id` non-deterministically, silently changing which magazine we refer towards; it may also attempt to place NULL into this columnn if we de-associate a `Writer` from an `Article`. The warning lets us know this is the case.

To solve this, we need to break out the behavior of `Article` to include all three of the following features:

1. `Article` first and foremost writes to `Article.magazine_id` based on data persisted in the `Article.magazine` relationship only, that is a value copied from `Magazine.id`.
2. `Article` can write to `Article.writer_id` on behalf of data persisted in the `Article.writer` relationship, but only the `Writer.id` column; the `Writer.magazine_id` column should not be written into `Article.magazine_id` as it ultimately is sourced from `Magazine.id`.
3. `Article` takes `Article.magazine_id` into account when loading `Article.writer`, even though it *doesn't* write to it on behalf of this relationship.

To get just #1 and #2, we could specify only `Article.writer_id` as the "foreign keys" for `Article.writer`:

```
class Article(Base):
    # ...

    writer = relationship("Writer", foreign_keys='Article.writer_id')
```

However, this has the effect of `Article.writer` not taking `Article.magazine_id` into account when querying against `Writer`:

```
SELECT article.article_id AS article_article_id,
    article.magazine_id AS article_magazine_id,
    article.writer_id AS article_writer_id
FROM article
JOIN writer ON writer.id = article.writer_id
```

Therefore, to get at all of #1, #2, and #3, we express the join condition as well as which columns to be written by combining *primaryjoin* fully, along with either the *foreign_keys* argument, or more succinctly by annotating with *foreign()*:

```
class Article(Base):
    # ...

    writer = relationship(
        "Writer",
        primaryjoin="and_(Writer.id == foreign(Article.writer_id), "
                    "Writer.magazine_id == Article.magazine_id)")
```

Changed in version 1.0.0: the ORM will attempt to warn when a column is used as the synchronization target from more than one relationship simultaneously.

Non-relational Comparisons / Materialized Path

Warning: this section details an experimental feature.

Using custom expressions means we can produce unorthodox join conditions that don't obey the usual primary/foreign key model. One such example is the materialized path pattern, where we compare strings for overlapping path tokens in order to produce a tree structure.

Through careful use of `foreign()` and `remote()`, we can build a relationship that effectively produces a rudimentary materialized path system. Essentially, when `foreign()` and `remote()` are on the *same* side of the comparison expression, the relationship is considered to be "one to many"; when they are on *different* sides, the relationship is considered to be "many to one". For the comparison we'll use here, we'll be dealing with collections so we keep things configured as "one to many":

```
class Element(Base):
    __tablename__ = 'element'

    path = Column(String, primary_key=True)

    descendants = relationship('Element',
                               primaryjoin=
                                    remote(foreign(path)).like(
                                            path.concat('/%')),
                               viewonly=True,
                               order_by=path)
```

Above, if given an `Element` object with a path attribute of `"/foo/bar2"`, we seek for a load of `Element.descendants` to look like:

```
SELECT element.path AS element_path
FROM element
WHERE element.path LIKE ('/foo/bar2' || '/%') ORDER BY element.path
```

New in version 0.9.5: Support has been added to allow a single-column comparison to itself within a primaryjoin condition, as well as for primaryjoin conditions that use `ColumnOperators.like()` as the comparison operator.

Self-Referential Many-to-Many Relationship

Many to many relationships can be customized by one or both of `primaryjoin` and `secondaryjoin` - the latter is significant for a relationship that specifies a many-to-many reference using the `secondary` argument. A common situation which involves the usage of `primaryjoin` and `secondaryjoin` is when establishing a many-to-many relationship from a class to itself, as shown below:

```
from sqlalchemy import Integer, ForeignKey, String, Column, Table
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

node_to_node = Table("node_to_node", Base.metadata,
    Column("left_node_id", Integer, ForeignKey("node.id"), primary_key=True),
    Column("right_node_id", Integer, ForeignKey("node.id"), primary_key=True)
```

```
)

class Node(Base):
    __tablename__ = 'node'
    id = Column(Integer, primary_key=True)
    label = Column(String)
    right_nodes = relationship("Node",
                        secondary=node_to_node,
                        primaryjoin=id==node_to_node.c.left_node_id,
                        secondaryjoin=id==node_to_node.c.right_node_id,
                        backref="left_nodes"
    )
```

Where above, SQLAlchemy can't know automatically which columns should connect to which for the `right_nodes` and `left_nodes` relationships. The *primaryjoin* and *secondaryjoin* arguments establish how we'd like to join to the association table. In the Declarative form above, as we are declaring these conditions within the Python block that corresponds to the `Node` class, the `id` variable is available directly as the *Column* object we wish to join with.

Alternatively, we can define the *primaryjoin* and *secondaryjoin* arguments using strings, which is suitable in the case that our configuration does not have either the `Node.id` column object available yet or the `node_to_node` table perhaps isn't yet available. When referring to a plain *Table* object in a declarative string, we use the string name of the table as it is present in the *MetaData*:

```
class Node(Base):
    __tablename__ = 'node'
    id = Column(Integer, primary_key=True)
    label = Column(String)
    right_nodes = relationship("Node",
                        secondary="node_to_node",
                        primaryjoin="Node.id==node_to_node.c.left_node_id",
                        secondaryjoin="Node.id==node_to_node.c.right_node_id",
                        backref="left_nodes"
    )
```

A classical mapping situation here is similar, where `node_to_node` can be joined to `node.c.id`:

```
from sqlalchemy import Integer, ForeignKey, String, Column, Table, MetaData
from sqlalchemy.orm import relationship, mapper

metadata = MetaData()

node_to_node = Table("node_to_node", metadata,
    Column("left_node_id", Integer, ForeignKey("node.id"), primary_key=True),
    Column("right_node_id", Integer, ForeignKey("node.id"), primary_key=True)
)

node = Table("node", metadata,
    Column('id', Integer, primary_key=True),
    Column('label', String)
)
class Node(object):
    pass

mapper(Node, node, properties={
    'right_nodes':relationship(Node,
                        secondary=node_to_node,
                        primaryjoin=node.c.id==node_to_node.c.left_node_id,
                        secondaryjoin=node.c.id==node_to_node.c.right_node_id,
```

```
                        backref="left_nodes"
                    )})
```

Note that in both examples, the *backref* keyword specifies a `left_nodes` backref - when *relationship()* creates the second relationship in the reverse direction, it's smart enough to reverse the *primaryjoin* and *secondaryjoin* arguments.

Composite "Secondary" Joins

Note: This section features some new and experimental features of SQLAlchemy.

Sometimes, when one seeks to build a *relationship()* between two tables there is a need for more than just two or three tables to be involved in order to join them. This is an area of *relationship()* where one seeks to push the boundaries of what's possible, and often the ultimate solution to many of these exotic use cases needs to be hammered out on the SQLAlchemy mailing list.

In more recent versions of SQLAlchemy, the *secondary* parameter can be used in some of these cases in order to provide a composite target consisting of multiple tables. Below is an example of such a join condition (requires version 0.9.2 at least to function as is):

```
class A(Base):
    __tablename__ = 'a'

    id = Column(Integer, primary_key=True)
    b_id = Column(ForeignKey('b.id'))

    d = relationship("D",
                secondary="join(B, D, B.d_id == D.id)."
                            "join(C, C.d_id == D.id)",
                primaryjoin="and_(A.b_id == B.id, A.id == C.a_id)",
                secondaryjoin="D.id == B.d_id",
                uselist=False
                )

class B(Base):
    __tablename__ = 'b'

    id = Column(Integer, primary_key=True)
    d_id = Column(ForeignKey('d.id'))

class C(Base):
    __tablename__ = 'c'

    id = Column(Integer, primary_key=True)
    a_id = Column(ForeignKey('a.id'))
    d_id = Column(ForeignKey('d.id'))

class D(Base):
    __tablename__ = 'd'

    id = Column(Integer, primary_key=True)
```

In the above example, we provide all three of *secondary*, *primaryjoin*, and *secondaryjoin*, in the declarative style referring to the named tables `a`, `b`, `c`, `d` directly. A query from `A` to `D` looks like:

```
sess.query(A).join(A.d).all()

SELECT a.id AS a_id, a.b_id AS a_b_id
FROM a JOIN (
    b AS b_1 JOIN d AS d_1 ON b_1.d_id = d_1.id
        JOIN c AS c_1 ON c_1.d_id = d_1.id)
    ON a.b_id = b_1.id AND a.id = c_1.a_id JOIN d ON d.id = b_1.d_id
```

In the above example, we take advantage of being able to stuff multiple tables into a "secondary" container, so that we can join across many tables while still keeping things "simple" for *relationship()*, in that there's just "one" table on both the "left" and the "right" side; the complexity is kept within the middle.

New in version 0.9.2: Support is improved for allowing a *join()* construct to be used directly as the target of the *secondary* argument, including support for joins, eager joins and lazy loading, as well as support within declarative to specify complex conditions such as joins involving class names as targets.

Relationship to Non Primary Mapper

In the previous section, we illustrated a technique where we used *secondary* in order to place additional tables within a join condition. There is one complex join case where even this technique is not sufficient; when we seek to join from `A` to `B`, making use of any number of `C`, `D`, etc. in between, however there are also join conditions between `A` and `B` *directly*. In this case, the join from `A` to `B` may be difficult to express with just a complex *primaryjoin* condition, as the intermediary tables may need special handling, and it is also not expressable with a *secondary* object, since the `A->secondary->B` pattern does not support any references between `A` and `B` directly. When this **extremely advanced** case arises, we can resort to creating a second mapping as a target for the relationship. This is where we use *mapper()* in order to make a mapping to a class that includes all the additional tables we need for this join. In order to produce this mapper as an "alternative" mapping for our class, we use the *non_primary* flag.

Below illustrates a *relationship()* with a simple join from `A` to `B`, however the primaryjoin condition is augmented with two additional entities `C` and `D`, which also must have rows that line up with the rows in both `A` and `B` simultaneously:

```
class A(Base):
    __tablename__ = 'a'

    id = Column(Integer, primary_key=True)
    b_id = Column(ForeignKey('b.id'))

class B(Base):
    __tablename__ = 'b'

    id = Column(Integer, primary_key=True)

class C(Base):
    __tablename__ = 'c'

    id = Column(Integer, primary_key=True)
    a_id = Column(ForeignKey('a.id'))

class D(Base):
    __tablename__ = 'd'

    id = Column(Integer, primary_key=True)
    c_id = Column(ForeignKey('c.id'))
    b_id = Column(ForeignKey('b.id'))

# 1. set up the join() as a variable, so we can refer
```

```
# to it in the mapping multiple times.
j = join(B, D, D.b_id == B.id).join(C, C.id == D.c_id)

# 2. Create a new mapper() to B, with non_primary=True.
# Columns in the join with the same name must be
# disambiguated within the mapping, using named properties.
B_viacd = mapper(B, j, non_primary=True, properties={
    "b_id": [j.c.b_id, j.c.d_b_id],
    "d_id": j.c.d_id
    })

A.b = relationship(B_viacd, primaryjoin=A.b_id == B_viacd.c.b_id)
```

In the above case, our non-primary mapper for `B` will emit for additional columns when we query; these can be ignored:

```
sess.query(A).join(A.b).all()

SELECT a.id AS a_id, a.b_id AS a_b_id
FROM a JOIN (b JOIN d ON d.b_id = b.id JOIN c ON c.id = d.c_id) ON a.b_id = b.id
```

Building Query-Enabled Properties

Very ambitious custom join conditions may fail to be directly persistable, and in some cases may not even load correctly. To remove the persistence part of the equation, use the flag *viewonly* on the *relationship()*, which establishes it as a read-only attribute (data written to the collection will be ignored on flush()). However, in extreme cases, consider using a regular Python property in conjunction with *Query* as follows:

```
class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)

    def _get_addresses(self):
        return object_session(self).query(Address).with_parent(self).filter(...).all()
    addresses = property(_get_addresses)
```

2.3.5 Collection Configuration and Techniques

The *relationship()* function defines a linkage between two classes. When the linkage defines a one-to-many or many-to-many relationship, it's represented as a Python collection when objects are loaded and manipulated. This section presents additional information about collection configuration and techniques.

Working with Large Collections

The default behavior of *relationship()* is to fully load the collection of items in, as according to the loading strategy of the relationship. Additionally, the *Session* by default only knows how to delete objects which are actually present within the session. When a parent instance is marked for deletion and flushed, the *Session* loads its full list of child items in so that they may either be deleted as well, or have their foreign key value set to null; this is to avoid constraint violations. For large collections of child items, there are several strategies to bypass full loading of child items both at load time as well as deletion time.

Dynamic Relationship Loaders

A key feature to enable management of a large collection is the so-called "dynamic" relationship. This is an optional form of *relationship()* which returns a *Query* object in place of a collection when accessed. *filter()* criterion may be applied as well as limits and offsets, either explicitly or via array slices:

```
class User(Base):
    __tablename__ = 'user'

    posts = relationship(Post, lazy="dynamic")

jack = session.query(User).get(id)

# filter Jack's blog posts
posts = jack.posts.filter(Post.headline=='this is a post')

# apply array slices
posts = jack.posts[5:20]
```

The dynamic relationship supports limited write operations, via the `append()` and `remove()` methods:

```
oldpost = jack.posts.filter(Post.headline=='old post').one()
jack.posts.remove(oldpost)

jack.posts.append(Post('new post'))
```

Since the read side of the dynamic relationship always queries the database, changes to the underlying collection will not be visible until the data has been flushed. However, as long as "autoflush" is enabled on the *Session* in use, this will occur automatically each time the collection is about to emit a query.

To place a dynamic relationship on a backref, use the *backref()* function in conjunction with `lazy='dynamic'`:

```
class Post(Base):
    __table__ = posts_table

    user = relationship(User,
                backref=backref('posts', lazy='dynamic')
            )
```

Note that eager/lazy loading options cannot be used in conjunction dynamic relationships at this time.

Note: The *dynamic_loader()* function is essentially the same as *relationship()* with the `lazy='dynamic'` argument specified.

Warning: The "dynamic" loader applies to **collections only**. It is not valid to use "dynamic" loaders with many-to-one, one-to-one, or uselist=False relationships. Newer versions of SQLAlchemy emit warnings or exceptions in these cases.

Setting Noload

A "noload" relationship never loads from the database, even when accessed. It is configured using `lazy='noload'`:

```
class MyClass(Base):
    __tablename__ = 'some_table'

    children = relationship(MyOtherClass, lazy='noload')
```

Above, the `children` collection is fully writeable, and changes to it will be persisted to the database as well as locally available for reading at the time they are added. However when instances of `MyClass` are freshly loaded from the database, the `children` collection stays empty.

Using Passive Deletes

Use *`passive_deletes`* to disable child object loading on a DELETE operation, in conjunction with "ON DELETE (CASCADE|SET NULL)" on your database to automatically cascade deletes to child objects:

```
class MyClass(Base):
    __tablename__ = 'mytable'
    id = Column(Integer, primary_key=True)
    children = relationship("MyOtherClass",
                    cascade="all, delete-orphan",
                    passive_deletes=True)

class MyOtherClass(Base):
    __tablename__ = 'myothertable'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer,
                ForeignKey('mytable.id', ondelete='CASCADE')
                    )
```

Note: To use "ON DELETE CASCADE", the underlying database engine must support foreign keys.

- When using MySQL, an appropriate storage engine must be selected. See *CREATE TABLE arguments including Storage Engines* for details.
- When using SQLite, foreign key support must be enabled explicitly. See *Foreign Key Support* for details.

When *`passive_deletes`* is applied, the `children` relationship will not be loaded into memory when an instance of `MyClass` is marked for deletion. The `cascade="all, delete-orphan"` *will* take effect for instances of `MyOtherClass` which are currently present in the session; however for instances of `MyOtherClass` which are not loaded, SQLAlchemy assumes that "ON DELETE CASCADE" rules will ensure that those rows are deleted by the database.

Customizing Collection Access

Mapping a one-to-many or many-to-many relationship results in a collection of values accessible through an attribute on the parent instance. By default, this collection is a `list`:

```
class Parent(Base):
    __tablename__ = 'parent'
    parent_id = Column(Integer, primary_key=True)

    children = relationship(Child)

parent = Parent()
parent.children.append(Child())
print parent.children[0]
```

Collections are not limited to lists. Sets, mutable sequences and almost any other Python object that can act as a container can be used in place of the default list, by specifying the *`collection_class`* option on *`relationship()`*:

```
class Parent(Base):
    __tablename__ = 'parent'
```

```
    parent_id = Column(Integer, primary_key=True)

    # use a set
    children = relationship(Child, collection_class=set)

parent = Parent()
child = Child()
parent.children.add(child)
assert child in parent.children
```

Dictionary Collections

A little extra detail is needed when using a dictionary as a collection. This because objects are always loaded from the database as lists, and a key-generation strategy must be available to populate the dictionary correctly. The *attribute_mapped_collection()* function is by far the most common way to achieve a simple dictionary collection. It produces a dictionary class that will apply a particular attribute of the mapped class as a key. Below we map an `Item` class containing a dictionary of `Note` items keyed to the `Note.keyword` attribute:

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.orm.collections import attribute_mapped_collection
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Item(Base):
    __tablename__ = 'item'
    id = Column(Integer, primary_key=True)
    notes = relationship("Note",
                collection_class=attribute_mapped_collection('keyword'),
                cascade="all, delete-orphan")

class Note(Base):
    __tablename__ = 'note'
    id = Column(Integer, primary_key=True)
    item_id = Column(Integer, ForeignKey('item.id'), nullable=False)
    keyword = Column(String)
    text = Column(String)

    def __init__(self, keyword, text):
        self.keyword = keyword
        self.text = text
```

`Item.notes` is then a dictionary:

```
>>> item = Item()
>>> item.notes['a'] = Note('a', 'atext')
>>> item.notes.items()
{'a': <__main__.Note object at 0x2eaaf0>}
```

attribute_mapped_collection() will ensure that the `.keyword` attribute of each `Note` complies with the key in the dictionary. Such as, when assigning to `Item.notes`, the dictionary key we supply must match that of the actual `Note` object:

```
item = Item()
item.notes = {
            'a': Note('a', 'atext'),
```

```
            'b': Note('b', 'btext')
        }
```

The attribute which *attribute_mapped_collection()* uses as a key does not need to be mapped at all! Using a regular Python `@property` allows virtually any detail or combination of details about the object to be used as the key, as below when we establish it as a tuple of `Note.keyword` and the first ten letters of the `Note.text` field:

```
class Item(Base):
    __tablename__ = 'item'
    id = Column(Integer, primary_key=True)
    notes = relationship("Note",
                collection_class=attribute_mapped_collection('note_key'),
                backref="item",
                cascade="all, delete-orphan")

class Note(Base):
    __tablename__ = 'note'
    id = Column(Integer, primary_key=True)
    item_id = Column(Integer, ForeignKey('item.id'), nullable=False)
    keyword = Column(String)
    text = Column(String)

    @property
    def note_key(self):
        return (self.keyword, self.text[0:10])

    def __init__(self, keyword, text):
        self.keyword = keyword
        self.text = text
```

Above we added a `Note.item` backref. Assigning to this reverse relationship, the `Note` is added to the `Item.notes` dictionary and the key is generated for us automatically:

```
>>> item = Item()
>>> n1 = Note("a", "atext")
>>> n1.item = item
>>> item.notes
{('a', 'atext'): <__main__.Note object at 0x2eaaf0>}
```

Other built-in dictionary types include *column_mapped_collection()*, which is almost like *attribute_mapped_collection()* except given the *Column* object directly:

```
from sqlalchemy.orm.collections import column_mapped_collection

class Item(Base):
    __tablename__ = 'item'
    id = Column(Integer, primary_key=True)
    notes = relationship("Note",
                collection_class=column_mapped_collection(Note.__table__.c.keyword),
                cascade="all, delete-orphan")
```

as well as *mapped_collection()* which is passed any callable function. Note that it's usually easier to use *attribute_mapped_collection()* along with a `@property` as mentioned earlier:

```
from sqlalchemy.orm.collections import mapped_collection

class Item(Base):
    __tablename__ = 'item'
    id = Column(Integer, primary_key=True)
```

```
    notes = relationship("Note",
                collection_class=mapped_collection(lambda note: note.text[0:10]),
                cascade="all, delete-orphan")
```

Dictionary mappings are often combined with the "Association Proxy" extension to produce streamlined dictionary views. See *Proxying to Dictionary Based Collections* and *Composite Association Proxies* for examples.

`sqlalchemy.orm.collections.`**`attribute_mapped_collection`**(*attr_name*)

A dictionary-based collection type with attribute-based keying.

Returns a `MappedCollection` factory with a keying based on the 'attr_name' attribute of entities in the collection, where `attr_name` is the string name of the attribute.

The key value must be immutable for the lifetime of the object. You can not, for example, map on foreign key values if those key values will change during the session, i.e. from None to a database-assigned integer after a session flush.

`sqlalchemy.orm.collections.`**`column_mapped_collection`**(*mapping_spec*)

A dictionary-based collection type with column-based keying.

Returns a `MappedCollection` factory with a keying function generated from mapping_spec, which may be a Column or a sequence of Columns.

The key value must be immutable for the lifetime of the object. You can not, for example, map on foreign key values if those key values will change during the session, i.e. from None to a database-assigned integer after a session flush.

`sqlalchemy.orm.collections.`**`mapped_collection`**(*keyfunc*)

A dictionary-based collection type with arbitrary keying.

Returns a `MappedCollection` factory with a keying function generated from keyfunc, a callable that takes an entity and returns a key value.

The key value must be immutable for the lifetime of the object. You can not, for example, map on foreign key values if those key values will change during the session, i.e. from None to a database-assigned integer after a session flush.

Custom Collection Implementations

You can use your own types for collections as well. In simple cases, inherting from `list` or `set`, adding custom behavior, is all that's needed. In other cases, special decorators are needed to tell SQLAlchemy more detail about how the collection operates.

Do I need a custom collection implementation?

In most cases not at all! The most common use cases for a "custom" collection is one that validates or marshals incoming values into a new form, such as a string that becomes a class instance, or one which goes a step beyond and represents the data internally in some fashion, presenting a "view" of that data on the outside of a different form.
For the first use case, the `orm.validates()` decorator is by far the simplest way to intercept incoming values in all cases for the purposes of validation and simple marshaling. See *Simple Validators* for an example of this.
For the second use case, the *Association Proxy* extension is a well-tested, widely used system that provides a read/write "view" of a collection in terms of some attribute present on the target object. As the target attribute can be a `@property` that returns virtually anything, a wide array of "alternative" views of a collection can be constructed with just a few functions. This approach leaves the underlying mapped collection unaffected and avoids the need to carefully tailor collection behavior on a method-by-method basis.
Customized collections are useful when the collection needs to have special behaviors upon access or mutation operations that can't otherwise be modeled externally to the collection. They can of course be combined with the above two approaches.

Collections in SQLAlchemy are transparently *instrumented*. Instrumentation means that normal operations on the collection are tracked and result in changes being written to the database at flush time. Additionally, collection operations can fire *events* which indicate some secondary operation must take place. Examples of a secondary operation include saving the child item in the parent's `Session` (i.e. the `save-update` cascade), as well as synchronizing the state of a bi-directional relationship (i.e. a `backref()`).

The collections package understands the basic interface of lists, sets and dicts and will automatically apply instrumentation to those built-in types and their subclasses. Object-derived types that implement a basic collection interface are detected and instrumented via duck-typing:

```
class ListLike(object):
    def __init__(self):
        self.data = []
    def append(self, item):
        self.data.append(item)
    def remove(self, item):
        self.data.remove(item)
    def extend(self, items):
        self.data.extend(items)
    def __iter__(self):
        return iter(self.data)
    def foo(self):
        return 'foo'
```

`append`, `remove`, and `extend` are known list-like methods, and will be instrumented automatically. `__iter__` is not a mutator method and won't be instrumented, and `foo` won't be either.

Duck-typing (i.e. guesswork) isn't rock-solid, of course, so you can be explicit about the interface you are implementing by providing an `__emulates__` class attribute:

```
class SetLike(object):
    __emulates__ = set

    def __init__(self):
        self.data = set()
    def append(self, item):
        self.data.add(item)
    def remove(self, item):
        self.data.remove(item)
    def __iter__(self):
        return iter(self.data)
```

This class looks list-like because of `append`, but `__emulates__` forces it to set-like. `remove` is known to be part of the set interface and will be instrumented.

But this class won't work quite yet: a little glue is needed to adapt it for use by SQLAlchemy. The ORM needs to know which methods to use to append, remove and iterate over members of the collection. When using a type like `list` or `set`, the appropriate methods are well-known and used automatically when present. This set-like class does not provide the expected `add` method, so we must supply an explicit mapping for the ORM via a decorator.

Annotating Custom Collections via Decorators

Decorators can be used to tag the individual methods the ORM needs to manage collections. Use them when your class doesn't quite meet the regular interface for its container type, or when you otherwise would like to use a different method to get the job done.

```
from sqlalchemy.orm.collections import collection

class SetLike(object):
    __emulates__ = set

    def __init__(self):
        self.data = set()

    @collection.appender
    def append(self, item):
        self.data.add(item)

    def remove(self, item):
        self.data.remove(item)

    def __iter__(self):
        return iter(self.data)
```

And that's all that's needed to complete the example. SQLAlchemy will add instances via the `append` method. `remove` and `__iter__` are the default methods for sets and will be used for removing and iteration. Default methods can be changed as well:

```
from sqlalchemy.orm.collections import collection

class MyList(list):
    @collection.remover
    def zark(self, item):
        # do something special...

    @collection.iterator
    def hey_use_this_instead_for_iteration(self):
        # ...
```

There is no requirement to be list-, or set-like at all. Collection classes can be any shape, so long as they have the append, remove and iterate interface marked for SQLAlchemy's use. Append and remove methods will be called with a mapped entity as the single argument, and iterator methods are called with no arguments and must return an iterator.

class `sqlalchemy.orm.collections.`**`collection`**

Decorators for entity collection classes.

The decorators fall into two groups: annotations and interception recipes.

The annotating decorators (appender, remover, iterator, linker, converter, internally_instrumented) indicate the method's purpose and take no arguments. They are not written with parens:

```
@collection.appender
def append(self, append): ...
```

The recipe decorators all require parens, even those that take no arguments:

```
@collection.adds('entity')
def insert(self, position, entity): ...

@collection.removes_return()
def popitem(self): ...
```

static adds (*arg*)

Mark the method as adding an entity to the collection.

Adds "add to collection" handling to the method. The decorator argument indicates which method argument holds the SQLAlchemy-relevant value. Arguments can be specified positionally (i.e. integer) or by name:

```
@collection.adds(1)
def push(self, item): ...

@collection.adds('entity')
def do_stuff(self, thing, entity=None): ...
```

static appender (*fn*)

Tag the method as the collection appender.

The appender method is called with one positional argument: the value to append. The method will be automatically decorated with 'adds(1)' if not already decorated:

```
@collection.appender
def add(self, append): ...

# or, equivalently
@collection.appender
@collection.adds(1)
def add(self, append): ...

# for mapping type, an 'append' may kick out a previous value
# that occupies that slot.  consider d['a'] = 'foo'- any previous
# value in d['a'] is discarded.
@collection.appender
@collection.replaces(1)
def add(self, entity):
    key = some_key_func(entity)
    previous = None
    if key in self:
        previous = self[key]
    self[key] = entity
    return previous
```

If the value to append is not allowed in the collection, you may raise an exception. Something to remember is that the appender will be called for each object mapped by a database query. If the database contains rows that violate your collection semantics, you will need to get creative to fix the problem, as access via the collection will not work.

If the appender method is internally instrumented, you must also receive the keyword argument

'_sa_initiator' and ensure its promulgation to collection events.

static converter (*fn*)

Tag the method as the collection converter.

This optional method will be called when a collection is being replaced entirely, as in:

```
myobj.acollection = [newvalue1, newvalue2]
```

The converter method will receive the object being assigned and should return an iterable of values suitable for use by the `appender` method. A converter must not assign values or mutate the collection, its sole job is to adapt the value the user provides into an iterable of values for the ORM's use.

The default converter implementation will use duck-typing to do the conversion. A dict-like collection will be convert into an iterable of dictionary values, and other types will simply be iterated:

```
@collection.converter
def convert(self, other): ...
```

If the duck-typing of the object does not match the type of this collection, a TypeError is raised.

Supply an implementation of this method if you want to expand the range of possible types that can be assigned in bulk or perform validation on the values about to be assigned.

static internally_instrumented (*fn*)

Tag the method as instrumented.

This tag will prevent any decoration from being applied to the method. Use this if you are orchestrating your own calls to `collection_adapter()` in one of the basic SQLAlchemy interface methods, or to prevent an automatic ABC method decoration from wrapping your implementation:

```
# normally an 'extend' method on a list-like class would be
# automatically intercepted and re-implemented in terms of
# SQLAlchemy events and append().  your implementation will
# never be called, unless:
@collection.internally_instrumented
def extend(self, items): ...
```

static iterator (*fn*)

Tag the method as the collection remover.

The iterator method is called with no arguments. It is expected to return an iterator over all collection members:

```
@collection.iterator
def __iter__(self): ...
```

static link (*fn*)

deprecated; synonym for *collection.linker()*.

static linker (*fn*)

Tag the method as a "linked to attribute" event handler.

This optional event handler will be called when the collection class is linked to or unlinked from the InstrumentedAttribute. It is invoked immediately after the '_sa_adapter' property is set on the instance. A single argument is passed: the collection adapter that has been linked, or None if unlinking.

Deprecated since version 1.0.0: - the *collection.linker()* handler is superseded by the *AttributeEvents.init_collection()* and *AttributeEvents.dispose_collection()* handlers.

static remover (*fn*)

Tag the method as the collection remover.

The remover method is called with one positional argument: the value to remove. The method will be automatically decorated with *removes_return()* if not already decorated:

```
@collection.remover
def zap(self, entity): ...

# or, equivalently
@collection.remover
@collection.removes_return()
def zap(self, ): ...
```

If the value to remove is not present in the collection, you may raise an exception or return None to ignore the error.

If the remove method is internally instrumented, you must also receive the keyword argument '_sa_initiator' and ensure its promulgation to collection events.

static removes (*arg*)

Mark the method as removing an entity in the collection.

Adds "remove from collection" handling to the method. The decorator argument indicates which method argument holds the SQLAlchemy-relevant value to be removed. Arguments can be specified positionally (i.e. integer) or by name:

```
@collection.removes(1)
def zap(self, item): ...
```

For methods where the value to remove is not known at call-time, use collection.removes_return.

static removes_return ()

Mark the method as removing an entity in the collection.

Adds "remove from collection" handling to the method. The return value of the method, if any, is considered the value to remove. The method arguments are not inspected:

```
@collection.removes_return()
def pop(self): ...
```

For methods where the value to remove is known at call-time, use collection.remove.

static replaces (*arg*)

Mark the method as replacing an entity in the collection.

Adds "add to collection" and "remove from collection" handling to the method. The decorator argument indicates which method argument holds the SQLAlchemy-relevant value to be added, and return value, if any will be considered the value to remove.

Arguments can be specified positionally (i.e. integer) or by name:

```
@collection.replaces(2)
def __setitem__(self, index, item): ...
```

Custom Dictionary-Based Collections

The *MappedCollection* class can be used as a base class for your custom types or as a mix-in to quickly add `dict` collection support to other classes. It uses a keying function to delegate to `__setitem__` and `__delitem__`:

```
from sqlalchemy.util import OrderedDict
from sqlalchemy.orm.collections import MappedCollection

class NodeMap(OrderedDict, MappedCollection):
    """Holds 'Node' objects, keyed by the 'name' attribute with insert order maintained."""

    def __init__(self, *args, **kw):
        MappedCollection.__init__(self, keyfunc=lambda node: node.name)
        OrderedDict.__init__(self, *args, **kw)
```

When subclassing *MappedCollection*, user-defined versions of `__setitem__()` or `__delitem__()` should be decorated with *collection.internally_instrumented()*, **if** they call down to those same methods on *MappedCollection*. This because the methods on *MappedCollection* are already instrumented - calling them from within an already instrumented call can cause events to be fired off repeatedly, or inappropriately, leading to internal state corruption in rare cases:

```
from sqlalchemy.orm.collections import MappedCollection,\
                                       collection

class MyMappedCollection(MappedCollection):
    """Use @internally_instrumented when your methods
    call down to already-instrumented methods.

    """

    @collection.internally_instrumented
    def __setitem__(self, key, value, _sa_initiator=None):
        # do something with key, value
        super(MyMappedCollection, self).__setitem__(key, value, _sa_initiator)

    @collection.internally_instrumented
    def __delitem__(self, key, _sa_initiator=None):
        # do something with key
        super(MyMappedCollection, self).__delitem__(key, _sa_initiator)
```

The ORM understands the `dict` interface just like lists and sets, and will automatically instrument all dict-like methods if you choose to subclass `dict` or provide dict-like collection behavior in a duck-typed class. You must decorate appender and remover methods, however- there are no compatible methods in the basic dictionary interface for SQLAlchemy to use by default. Iteration will go through `itervalues()` unless otherwise decorated.

Note: Due to a bug in MappedCollection prior to version 0.7.6, this workaround usually needs to be called before a custom subclass of *MappedCollection* which uses *collection.internally_instrumented()* can be used:

```
from sqlalchemy.orm.collections import _instrument_class, MappedCollection
_instrument_class(MappedCollection)
```

This will ensure that the *MappedCollection* has been properly initialized with custom `__setitem__()` and `__delitem__()` methods before used in a custom subclass.

---

class `sqlalchemy.orm.collections.`**`MappedCollection`**(*keyfunc*)

Bases: `__builtin__.dict`

A basic dictionary-based collection class.

Extends dict with the minimal bag semantics that collection classes require. `set` and `remove` are implemented in terms of a keying function: any callable that takes an object and returns an object for use as a dictionary key.

__init__(*keyfunc*)
Create a new collection with keying provided by keyfunc.

keyfunc may be any callable that takes an object and returns an object for use as a dictionary key.

The keyfunc will be called every time the ORM needs to add a member by value-only (such as when loading instances from the database) or remove a member. The usual cautions about dictionary keying apply- `keyfunc(object)` should return the same output for the life of the collection. Keying based on mutable properties can result in unreachable instances "lost" in the collection.

clear() → None. Remove all items from D.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

remove(*value*, *_sa_initiator=None*)
Remove an item by value, consulting the keyfunc for the key.

set(*value*, *_sa_initiator=None*)
Add an item by value, consulting the keyfunc for the key.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], **F) → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

Instrumentation and Custom Types

Many custom types and existing library classes can be used as a entity collection type as-is without further ado. However, it is important to note that the instrumentation process will modify the type, adding decorators around methods automatically.

The decorations are lightweight and no-op outside of relationships, but they do add unneeded overhead when triggered elsewhere. When using a library class as a collection, it can be good practice to use the "trivial subclass" trick to restrict the decorations to just your usage in relationships. For example:

```
class MyAwesomeList(some.great.library.AwesomeList):
    pass

# ... relationship(..., collection_class=MyAwesomeList)
```

The ORM uses this approach for built-ins, quietly substituting a trivial subclass when a `list`, `set` or `dict` is used directly.

Collection Internals

Various internal methods.

`sqlalchemy.orm.collections.`**bulk_replace**(*values*, *existing_adapter*, *new_adapter*)
Load a new collection, firing events based on prior like membership.

Appends instances in `values` onto the `new_adapter`. Events will be fired for any instance not present in the `existing_adapter`. Any instances in `existing_adapter` not present in `values` will have remove events fired upon them.

Parameters

- **values** – An iterable of collection member instances
- **existing_adapter** – A *CollectionAdapter* of instances to be replaced
- **new_adapter** – An empty *CollectionAdapter* to load with values

class sqlalchemy.orm.collections.**collection**

Decorators for entity collection classes.

The decorators fall into two groups: annotations and interception recipes.

The annotating decorators (appender, remover, iterator, linker, converter, internally_instrumented) indicate the method's purpose and take no arguments. They are not written with parens:

```
@collection.appender
def append(self, append): ...
```

The recipe decorators all require parens, even those that take no arguments:

```
@collection.adds('entity')
def insert(self, position, entity): ...

@collection.removes_return()
def popitem(self): ...
```

sqlalchemy.orm.collections.**collection_adapter** = **<operator.attrgetter object>**

Fetch the *CollectionAdapter* for a collection.

class sqlalchemy.orm.collections.**CollectionAdapter**(*attr*, *owner_state*, *data*)

Bridges between the ORM and arbitrary Python collections.

Proxies base-level collection operations (append, remove, iterate) to the underlying Python collection, and emits add/remove events for entities entering or leaving the collection.

The ORM uses *CollectionAdapter* exclusively for interaction with entity collections.

class sqlalchemy.orm.collections.**InstrumentedDict**

Bases: __builtin__.dict

An instrumented version of the built-in dict.

class sqlalchemy.orm.collections.**InstrumentedList**

Bases: __builtin__.list

An instrumented version of the built-in list.

class sqlalchemy.orm.collections.**InstrumentedSet**

Bases: __builtin__.set

An instrumented version of the built-in set.

sqlalchemy.orm.collections.**prepare_instrumentation**(*factory*)

Prepare a callable for future use as a collection class factory.

Given a collection class factory (either a type or no-arg callable), return another factory that will produce compatible instances when called.

This function is responsible for converting collection_class=list into the run-time behavior of collection_class=InstrumentedList.

2.3.6 Special Relationship Persistence Patterns

Rows that point to themselves / Mutually Dependent Rows

This is a very specific case where relationship() must perform an INSERT and a second UPDATE in order to properly populate a row (and vice versa an UPDATE and DELETE in order to delete without violating foreign key constraints). The two use cases are:

- A table contains a foreign key to itself, and a single row will have a foreign key value pointing to its own primary key.
- Two tables each contain a foreign key referencing the other table, with a row in each table referencing the other.

For example:

```
          user
---------------------------------
user_id    name   related_user_id
   1       'ed'          1
```

Or:

```
              widget                                                  entry
-----------------------------------------             ---------------------------------
widget_id     name         favorite_entry_id            entry_id      name      widget_id
   1       'somewidget'          5                         5       'someentry'     1
```

In the first case, a row points to itself. Technically, a database that uses sequences such as PostgreSQL or Oracle can INSERT the row at once using a previously generated value, but databases which rely upon autoincrement-style primary key identifiers cannot. The *relationship()* always assumes a "parent/child" model of row population during flush, so unless you are populating the primary key/foreign key columns directly, *relationship()* needs to use two statements.

In the second case, the "widget" row must be inserted before any referring "entry" rows, but then the "favorite_entry_id" column of that "widget" row cannot be set until the "entry" rows have been generated. In this case, it's typically impossible to insert the "widget" and "entry" rows using just two INSERT statements; an UPDATE must be performed in order to keep foreign key constraints fulfilled. The exception is if the foreign keys are configured as "deferred until commit" (a feature some databases support) and if the identifiers were populated manually (again essentially bypassing *relationship()*).

To enable the usage of a supplementary UPDATE statement, we use the *post_update* option of *relationship()*. This specifies that the linkage between the two rows should be created using an UPDATE statement after both rows have been INSERTED; it also causes the rows to be de-associated with each other via UPDATE before a DELETE is emitted. The flag should be placed on just *one* of the relationships, preferably the many-to-one side. Below we illustrate a complete example, including two *ForeignKey* constructs:

```
from sqlalchemy import Integer, ForeignKey, Column
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class Entry(Base):
    __tablename__ = 'entry'
    entry_id = Column(Integer, primary_key=True)
    widget_id = Column(Integer, ForeignKey('widget.widget_id'))
    name = Column(String(50))

class Widget(Base):
```

```
    __tablename__ = 'widget'

    widget_id = Column(Integer, primary_key=True)
    favorite_entry_id = Column(Integer,
                               ForeignKey('entry.entry_id',
                               name="fk_favorite_entry"))
    name = Column(String(50))

    entries = relationship(Entry, primaryjoin=
                                     widget_id==Entry.widget_id)
    favorite_entry = relationship(Entry,
                                  primaryjoin=
                                     favorite_entry_id==Entry.entry_id,
                                  post_update=True)
```

When a structure against the above configuration is flushed, the "widget" row will be INSERTed minus the "favorite_entry_id" value, then all the "entry" rows will be INSERTed referencing the parent "widget" row, and then an UPDATE statement will populate the "favorite_entry_id" column of the "widget" table (it's one row at a time for the time being):

```
>>> w1 = Widget(name='somewidget')
>>> e1 = Entry(name='someentry')
>>> w1.favorite_entry = e1
>>> w1.entries = [e1]
>>> session.add_all([w1, e1])
>>> session.commit()
BEGIN (implicit)
INSERT INTO widget (favorite_entry_id, name) VALUES (?, ?)
(None, 'somewidget')
INSERT INTO entry (widget_id, name) VALUES (?, ?)
(1, 'someentry')
UPDATE widget SET favorite_entry_id=? WHERE widget.widget_id = ?
(1, 1)
COMMIT
```

An additional configuration we can specify is to supply a more comprehensive foreign key constraint on `Widget`, such that it's guaranteed that `favorite_entry_id` refers to an `Entry` that also refers to this `Widget`. We can use a composite foreign key, as illustrated below:

```
from sqlalchemy import Integer, ForeignKey, String, \
        Column, UniqueConstraint, ForeignKeyConstraint
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class Entry(Base):
    __tablename__ = 'entry'
    entry_id = Column(Integer, primary_key=True)
    widget_id = Column(Integer, ForeignKey('widget.widget_id'))
    name = Column(String(50))
    __table_args__ = (
        UniqueConstraint("entry_id", "widget_id"),
    )

class Widget(Base):
    __tablename__ = 'widget'
```

```
    widget_id = Column(Integer, autoincrement='ignore_fk', primary_key=True)
    favorite_entry_id = Column(Integer)

    name = Column(String(50))

    __table_args__ = (
        ForeignKeyConstraint(
            ["widget_id", "favorite_entry_id"],
            ["entry.widget_id", "entry.entry_id"],
            name="fk_favorite_entry"
        ),
    )

    entries = relationship(Entry, primaryjoin=
                                widget_id==Entry.widget_id,
                                foreign_keys=Entry.widget_id)
    favorite_entry = relationship(Entry,
                              primaryjoin=
                                  favorite_entry_id==Entry.entry_id,
                              foreign_keys=favorite_entry_id,
                              post_update=True)
```

The above mapping features a composite *ForeignKeyConstraint* bridging the `widget_id` and `favorite_entry_id` columns. To ensure that `Widget.widget_id` remains an "autoincrementing" column we specify *autoincrement* to the value `"ignore_fk"` on *Column*, and additionally on each *relationship()* we must limit those columns considered as part of the foreign key for the purposes of joining and cross-population.

Mutable Primary Keys / Update Cascades

When the primary key of an entity changes, related items which reference the primary key must also be updated as well. For databases which enforce referential integrity, the best strategy is to use the database's ON UPDATE CASCADE functionality in order to propagate primary key changes to referenced foreign keys - the values cannot be out of sync for any moment unless the constraints are marked as "deferrable", that is, not enforced until the transaction completes.

It is **highly recommended** that an application which seeks to employ natural primary keys with mutable values to use the `ON UPDATE CASCADE` capabilities of the database. An example mapping which illustrates this is:

```
class User(Base):
    __tablename__ = 'user'
    __table_args__ = {'mysql_engine': 'InnoDB'}

    username = Column(String(50), primary_key=True)
    fullname = Column(String(100))

    addresses = relationship("Address")


class Address(Base):
    __tablename__ = 'address'
    __table_args__ = {'mysql_engine': 'InnoDB'}

    email = Column(String(50), primary_key=True)
    username = Column(String(50),
                ForeignKey('user.username', onupdate="cascade")
            )
```

Above, we illustrate `onupdate="cascade"` on the *ForeignKey* object, and we also illustrate the

`mysql_engine='InnoDB'` setting which, on a MySQL backend, ensures that the `InnoDB` engine supporting referential integrity is used. When using SQLite, referential integrity should be enabled, using the configuration described at *Foreign Key Support*.

Simulating limited ON UPDATE CASCADE without foreign key support

In those cases when a database that does not support referential integrity is used, and natural primary keys with mutable values are in play, SQLAlchemy offers a feature in order to allow propagation of primary key values to already-referenced foreign keys to a **limited** extent, by emitting an UPDATE statement against foreign key columns that immediately reference a primary key column whose value has changed. The primary platforms without referential integrity features are MySQL when the `MyISAM` storage engine is used, and SQLite when the `PRAGMA foreign_keys=ON` pragma is not used. The Oracle database also has no support for `ON UPDATE CASCADE`, but because it still enforces referential integrity, needs constraints to be marked as deferrable so that SQLAlchemy can emit UPDATE statements.

The feature is enabled by setting the *`passive_updates`* flag to `False`, most preferably on a one-to-many or many-to-many *`relationship()`*. When "updates" are no longer "passive" this indicates that SQLAlchemy will issue UPDATE statements individually for objects referenced in the collection referred to by the parent object with a changing primary key value. This also implies that collections will be fully loaded into memory if not already locally present.

Our previous mapping using `passive_updates=False` looks like:

```
class User(Base):
    __tablename__ = 'user'

    username = Column(String(50), primary_key=True)
    fullname = Column(String(100))

    # passive_updates=False *only* needed if the database
    # does not implement ON UPDATE CASCADE
    addresses = relationship("Address", passive_updates=False)

class Address(Base):
    __tablename__ = 'address'

    email = Column(String(50), primary_key=True)
    username = Column(String(50), ForeignKey('user.username'))
```

Key limitations of `passive_updates=False` include:

- it performs much more poorly than direct database ON UPDATE CASCADE, because it needs to fully pre-load affected collections using SELECT and also must emit UPDATE statements against those values, which it will attempt to run in "batches" but still runs on a per-row basis at the DBAPI level.
- the feature cannot "cascade" more than one level. That is, if mapping X has a foreign key which refers to the primary key of mapping Y, but then mapping Y's primary key is itself a foreign key to mapping Z, `passive_updates=False` cannot cascade a change in primary key value from `Z` to `X`.
- Configuring `passive_updates=False` only on the many-to-one side of a relationship will not have a full effect, as the unit of work searches only through the current identity map for objects that may be referencing the one with a mutating primary key, not throughout the database.

As virtually all databases other than Oracle now support `ON UPDATE CASCADE`, it is highly recommended that traditional `ON UPDATE CASCADE` support be used in the case that natural and mutable primary key values are in use. Functional constructs for ORM configuration.

See the SQLAlchemy object relational tutorial and mapper configuration documentation for an overview of how this module is used.

2.3.7 Relationships API

`sqlalchemy.orm.relationship`(*argument*, *secondary=None*, *primaryjoin=None*, *secondaryjoin=None*, *foreign_keys=None*, *uselist=None*, *order_by=False*, *backref=None*, *back_populates=None*, *post_update=False*, *cascade=False*, *extension=None*, *viewonly=False*, *lazy=True*, *collection_class=None*, *passive_deletes=False*, *passive_updates=True*, *remote_side=None*, *enable_typechecks=True*, *join_depth=None*, *comparator_factory=None*, *single_parent=False*, *innerjoin=False*, *distinct_target_key=None*, *doc=None*, *active_history=False*, *cascade_backrefs=True*, *load_on_pending=False*, *bake_queries=True*, *strategy_class=None*, *_local_remote_pairs=None*, *query_class=None*, *info=None*)

Provide a relationship between two mapped classes.

This corresponds to a parent-child or associative table relationship. The constructed class is an instance of `RelationshipProperty`.

A typical `relationship()`, used in a classical mapping:

```
mapper(Parent, properties={
  'children': relationship(Child)
})
```

Some arguments accepted by `relationship()` optionally accept a callable function, which when called produces the desired value. The callable is invoked by the parent `Mapper` at "mapper initialization" time, which happens only when mappers are first used, and is assumed to be after all mappings have been constructed. This can be used to resolve order-of-declaration and other dependency issues, such as if `Child` is declared below `Parent` in the same file:

```
mapper(Parent, properties={
    "children":relationship(lambda: Child,
                            order_by=lambda: Child.id)
})
```

When using the *Declarative* extension, the Declarative initializer allows string arguments to be passed to `relationship()`. These string arguments are converted into callables that evaluate the string as Python code, using the Declarative class-registry as a namespace. This allows the lookup of related classes to be automatic via their string name, and removes the need to import related classes at all into the local module space:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child", order_by="Child.id")
```

See also:

Relationship Configuration - Full introductory and reference documentation for `relationship()`.

Building a Relationship - ORM tutorial introduction.

Parameters

- **argument** – a mapped class, or actual `Mapper` instance, representing the target of the relationship.

`argument` may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

See also:

Configuring Relationships - further detail on relationship configuration when using Declarative.

- **`secondary`** – for a many-to-many relationship, specifies the intermediary table, and is typically an instance of `Table`. In less common circumstances, the argument may also be specified as an `Alias` construct, or even a `Join` construct.

  `secondary` may also be passed as a callable function which is evaluated at mapper initialization time. When using Declarative, it may also be a string argument noting the name of a `Table` that is present in the `MetaData` collection associated with the parent-mapped `Table`.

  The `secondary` keyword argument is typically applied in the case where the intermediary `Table` is not otherwise expressed in any direct class mapping. If the "secondary" table is also explicitly mapped elsewhere (e.g. as in *Association Object*), one should consider applying the `viewonly` flag so that this `relationship()` is not used for persistence operations which may conflict with those of the association object pattern.

  See also:

  Many To Many - Reference example of "many to many".

  Building a Many To Many Relationship - ORM tutorial introduction to many-to-many relationships.

  Self-Referential Many-to-Many Relationship - Specifics on using many-to-many in a self-referential case.

  Configuring Many-to-Many Relationships - Additional options when using Declarative.

  Association Object - an alternative to `secondary` when composing association table relationships, allowing additional attributes to be specified on the association table.

  Composite "Secondary" Joins - a lesser-used pattern which in some cases can enable complex `relationship()` SQL conditions to be used.

  New in version 0.9.2: `secondary` works more effectively when referring to a `Join` instance.

- **`active_history=False`** – When `True`, indicates that the "previous" value for a many-to-one reference should be loaded when replaced, if not already loaded. Normally, history tracking logic for simple many-to-ones only needs to be aware of the "new" value in order to perform a flush. This flag is available for applications that make use of `attributes.get_history()` which also need to know the "previous" value of the attribute.

- **`backref`** – indicates the string name of a property to be placed on the related mapper's class that will handle this relationship in the other direction. The other property will be created automatically when the mappers are configured. Can also be passed as a `backref()` object to control the configuration of the new relationship.

  See also:

  Linking Relationships with Backref - Introductory documentation and examples.

  `back_populates` - alternative form of backref specification.

backref() - allows control over *relationship()* configuration when using *backref*.

- **back_populates** – Takes a string name and has the same meaning as *backref*, except the complementing property is **not** created automatically, and instead must be configured explicitly on the other mapper. The complementing property should also indicate *back_populates* to this relationship to ensure proper functioning.

  See also:

  Linking Relationships with Backref - Introductory documentation and examples.

  backref - alternative form of backref specification.

- **bake_queries=True** – Use the *BakedQuery* cache to cache the construction of SQL used in lazy loads, when the *bake_lazy_loaders()* function has first been called. Defaults to True and is intended to provide an "opt out" flag per-relationship when the baked query cache system is in use.

  > **Warning:** This flag **only** has an effect when the application-wide *bake_lazy_loaders()* function has been called. It defaults to True so is an "opt out" flag.

  Setting this flag to False when baked queries are otherwise in use might be to reduce ORM memory use for this *relationship()*, or to work around unresolved stability issues observed within the baked query cache system.

  New in version 1.0.0.

  See also:

  Baked Queries

- **cascade** – a comma-separated list of cascade rules which determines how Session operations should be "cascaded" from parent to child. This defaults to `False`, which means the default cascade should be used - this default cascade is `"save-update, merge"`.

  The available cascades are `save-update`, `merge`, `expunge`, `delete`, `delete-orphan`, and `refresh-expire`. An additional option, `all` indicates shorthand for `"save-update, merge, refresh-expire, expunge, delete"`, and is often used as in `"all, delete-orphan"` to indicate that related objects should follow along with the parent object in all cases, and be deleted when de-associated.

  See also:

  Cascades - Full detail on each of the available cascade options.

  Configuring delete/delete-orphan Cascade - Tutorial example describing a delete cascade.

- **cascade_backrefs=True** – a boolean value indicating if the `save-update` cascade should operate along an assignment event intercepted by a backref. When set to `False`, the attribute managed by this relationship will not cascade an incoming transient object into the session of a persistent parent, if the event is received via backref.

  See also:

  Controlling Cascade on Backrefs - Full discussion and examples on how the *cascade_backrefs* option is used.

- **collection_class** – a class or callable that returns a new list-holding object. will be used in place of a plain list for storing elements.

  See also:

Customizing Collection Access - Introductory documentation and examples.

- **comparator_factory** – a class which extends `RelationshipProperty.Comparator` which provides custom SQL clause generation for comparison operations.

  See also:

  `PropComparator` - some detail on redefining comparators at this level.

  Operator Customization - Brief intro to this feature.

- **distinct_target_key=None** – Indicate if a "subquery" eager load should apply the DISTINCT keyword to the innermost SELECT statement. When left as `None`, the DISTINCT keyword will be applied in those cases when the target columns do not comprise the full primary key of the target table. When set to `True`, the DISTINCT keyword is applied to the innermost SELECT unconditionally.

  It may be desirable to set this flag to False when the DISTINCT is reducing performance of the innermost subquery beyond that of what duplicate innermost rows may be causing.

  New in version 0.8.3: - `distinct_target_key` allows the subquery eager loader to apply a DISTINCT modifier to the innermost SELECT.

  Changed in version 0.9.0: - `distinct_target_key` now defaults to `None`, so that the feature enables itself automatically for those cases where the innermost query targets a non-unique key.

  See also:

  Relationship Loading Techniques - includes an introduction to subquery eager loading.

- **doc** – docstring which will be applied to the resulting descriptor.

- **extension** – an `AttributeExtension` instance, or list of extensions, which will be prepended to the list of attribute listeners for the resulting descriptor placed on the class.

  Deprecated since version 0.7: Please see `AttributeEvents`.

- **foreign_keys** – a list of columns which are to be used as "foreign key" columns, or columns which refer to the value in a remote column, within the context of this `relationship()` object's `primaryjoin` condition. That is, if the `primaryjoin` condition of this `relationship()` is `a.id == b.a_id`, and the values in `b.a_id` are required to be present in `a.id`, then the "foreign key" column of this `relationship()` is `b.a_id`.

  In normal cases, the `foreign_keys` parameter is **not required.** `relationship()` will automatically determine which columns in the `primaryjoin` condition are to be considered "foreign key" columns based on those `Column` objects that specify `ForeignKey`, or are otherwise listed as referencing columns in a `ForeignKeyConstraint` construct. `foreign_keys` is only needed when:

  1. There is more than one way to construct a join from the local table to the remote table, as there are multiple foreign key references present. Setting `foreign_keys` will limit the `relationship()` to consider just those columns specified here as "foreign".

     Changed in version 0.8: A multiple-foreign key join ambiguity can be resolved by setting the `foreign_keys` parameter alone, without the need to explicitly set `primaryjoin` as well.

  2. The `Table` being mapped does not actually have `ForeignKey` or `ForeignKeyConstraint` constructs present, often because the table was reflected from a database that does not support foreign key reflection (MySQL MyISAM).

3. The `primaryjoin` argument is used to construct a non-standard join condition, which makes use of columns or expressions that do not normally refer to their "parent" column, such as a join condition expressed by a complex comparison using a SQL function.

The `relationship()` construct will raise informative error messages that suggest the use of the `foreign_keys` parameter when presented with an ambiguous condition. In typical cases, if `relationship()` doesn't raise any exceptions, the `foreign_keys` parameter is usually not needed.

`foreign_keys` may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

See also:

Handling Multiple Join Paths

Creating Custom Foreign Conditions

`foreign()` - allows direct annotation of the "foreign" columns within a `primaryjoin` condition.

New in version 0.8: The `foreign()` annotation can also be applied directly to the `primaryjoin` expression, which is an alternate, more specific system of describing which columns in a particular `primaryjoin` should be considered "foreign".

- **`info`** – Optional data dictionary which will be populated into the `MapperProperty.info` attribute of this object.

  New in version 0.8.

- **`innerjoin=False`** – when `True`, joined eager loads will use an inner join to join against related tables instead of an outer join. The purpose of this option is generally one of performance, as inner joins generally perform better than outer joins.

  This flag can be set to `True` when the relationship references an object via many-to-one using local foreign keys that are not nullable, or when the reference is one-to-one or a collection that is guaranteed to have one or at least one entry.

  The option supports the same "nested" and "unnested" options as that of `joinedload.innerjoin`. See that flag for details on nested / unnested behaviors.

  See also:

  `joinedload.innerjoin` - the option as specified by loader option, including detail on nesting behavior.

  What Kind of Loading to Use ? - Discussion of some details of various loader options.

- **`join_depth`** – when non-`None`, an integer value indicating how many levels deep "eager" loaders should join on a self-referring or cyclical relationship. The number counts how many times the same Mapper shall be present in the loading condition along a particular join branch. When left at its default of `None`, eager loaders will stop chaining when they encounter a the same target mapper which is already higher up in the chain. This option applies both to joined- and subquery- eager loaders.

  See also:

  Configuring Self-Referential Eager Loading - Introductory documentation and examples.

- **`lazy='select'`** – specifies how the related items should be loaded. Default value is `select`. Values include:

- `select` - items should be loaded lazily when the property is first accessed, using a separate SELECT statement, or identity map fetch for simple many-to-one references.
- `immediate` - items should be loaded as the parents are loaded, using a separate SELECT statement, or identity map fetch for simple many-to-one references.
- `joined` - items should be loaded "eagerly" in the same query as that of the parent, using a JOIN or LEFT OUTER JOIN. Whether the join is "outer" or not is determined by the `innerjoin` parameter.
- `subquery` - items should be loaded "eagerly" as the parents are loaded, using one additional SQL statement, which issues a JOIN to a subquery of the original statement, for each collection requested.
- `noload` - no loading should occur at any time. This is to support "write-only" attributes, or attributes which are populated in some manner specific to the application.
- `dynamic` - the attribute will return a pre-configured `Query` object for all read operations, onto which further filtering operations can be applied before iterating the results. See the section *Dynamic Relationship Loaders* for more details.
- True - a synonym for 'select'
- False - a synonym for 'joined'
- None - a synonym for 'noload'

See also:

Relationship Loading Techniques - Full documentation on relationship loader configuration.

Dynamic Relationship Loaders - detail on the `dynamic` option.

- **`load_on_pending=False`** – Indicates loading behavior for transient or pending parent objects.

  When set to `True`, causes the lazy-loader to issue a query for a parent object that is not persistent, meaning it has never been flushed. This may take effect for a pending object when autoflush is disabled, or for a transient object that has been "attached" to a `Session` but is not part of its pending collection.

  The `load_on_pending` flag does not improve behavior when the ORM is used normally - object references should be constructed at the object level, not at the foreign key level, so that they are present in an ordinary way before a flush proceeds. This flag is not not intended for general use.

  See also:

  `Session.enable_relationship_loading()` - this method establishes "load on pending" behavior for the whole object, and also allows loading on objects that remain transient or detached.

- **`order_by`** – indicates the ordering that should be applied when loading these items. `order_by` is expected to refer to one of the `Column` objects to which the target class is mapped, or the attribute itself bound to the target class which refers to the column.

  `order_by` may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

- **`passive_deletes=False`** – Indicates loading behavior during delete operations.

  A value of True indicates that unloaded child items should not be loaded during a delete operation on the parent. Normally, when a parent item is deleted, all child items are loaded so that they can either be marked as deleted, or have their foreign key to the parent set

to NULL. Marking this flag as True usually implies an ON DELETE <CASCADE|SET NULL> rule is in place which will handle updating/deleting child rows on the database side.

Additionally, setting the flag to the string value 'all' will disable the "nulling out" of the child foreign keys, when there is no delete or delete-orphan cascade enabled. This is typically used when a triggering or error raise scenario is in place on the database side. Note that the foreign key attributes on in-session child objects will not be changed after a flush occurs so this is a very special use-case setting.

See also:

Using Passive Deletes - Introductory documentation and examples.

- **`passive_updates=True`** – Indicates the persistence behavior to take when a referenced primary key value changes in place, indicating that the referencing foreign key columns will also need their value changed.

  When True, it is assumed that `ON UPDATE CASCADE` is configured on the foreign key in the database, and that the database will handle propagation of an UPDATE from a source column to dependent rows. When False, the SQLAlchemy *`relationship()`* construct will attempt to emit its own UPDATE statements to modify related targets. However note that SQLAlchemy **cannot** emit an UPDATE for more than one level of cascade. Also, setting this flag to False is not compatible in the case where the database is in fact enforcing referential integrity, unless those constraints are explicitly "deferred", if the target backend supports it.

  It is highly advised that an application which is employing mutable primary keys keeps `passive_updates` set to True, and instead uses the referential integrity features of the database itself in order to handle the change efficiently and fully.

  See also:

  Mutable Primary Keys / Update Cascades - Introductory documentation and examples.

  `mapper.passive_updates` - a similar flag which takes effect for joined-table inheritance mappings.

- **`post_update`** – this indicates that the relationship should be handled by a second UPDATE statement after an INSERT or before a DELETE. Currently, it also will issue an UPDATE after the instance was UPDATEd as well, although this technically should be improved. This flag is used to handle saving bi-directional dependencies between two individual rows (i.e. each row references the other), where it would otherwise be impossible to INSERT or DELETE both rows fully since one row exists before the other. Use this flag when a particular mapping arrangement will incur two rows that are dependent on each other, such as a table that has a one-to-many relationship to a set of child rows, and also has a column that references a single child row within that list (i.e. both tables contain a foreign key to each other). If a flush operation returns an error that a "cyclical dependency" was detected, this is a cue that you might want to use *`post_update`* to "break" the cycle.

  See also:

  Rows that point to themselves / Mutually Dependent Rows - Introductory documentation and examples.

- **`primaryjoin`** – a SQL expression that will be used as the primary join of this child object against the parent object, or in a many-to-many relationship the join of the primary object to the association table. By default, this value is computed based on the foreign key relationships of the parent and child tables (or association table).

primaryjoin may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

See also:

Specifying Alternate Join Conditions

- **remote_side** – used for self-referential relationships, indicates the column or list of columns that form the "remote side" of the relationship.

  relationship.remote_side may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  Changed in version 0.8: The *remote()* annotation can also be applied directly to the `primaryjoin` expression, which is an alternate, more specific system of describing which columns in a particular `primaryjoin` should be considered "remote".

  See also:

  Adjacency List Relationships - in-depth explanation of how *remote_side* is used to configure self-referential relationships.

  remote() - an annotation function that accomplishes the same purpose as *remote_side*, typically when a custom *primaryjoin* condition is used.

- **query_class** – a *Query* subclass that will be used as the base of the "appender query" returned by a "dynamic" relationship, that is, a relationship that specifies `lazy="dynamic"` or was otherwise constructed using the *orm.dynamic_loader()* function.

  See also:

  Dynamic Relationship Loaders - Introduction to "dynamic" relationship loaders.

- **secondaryjoin** – a SQL expression that will be used as the join of an association table to the child object. By default, this value is computed based on the foreign key relationships of the association and child tables.

  secondaryjoin may also be passed as a callable function which is evaluated at mapper initialization time, and may be passed as a Python-evaluable string when using Declarative.

  See also:

  Specifying Alternate Join Conditions

- **single_parent** – when True, installs a validator which will prevent objects from being associated with more than one parent at a time. This is used for many-to-one or many-to-many relationships that should be treated either as one-to-one or one-to-many. Its usage is optional, except for *relationship()* constructs which are many-to-one or many-to-many and also specify the `delete-orphan` cascade option. The *relationship()* construct itself will raise an error instructing when this option is required.

  See also:

  Cascades - includes detail on when the *single_parent* flag may be appropriate.

- **uselist** – a boolean that indicates if this property should be loaded as a list or a scalar. In most cases, this value is determined automatically by *relationship()* at mapper configuration time, based on the type and direction of the relationship - one to many forms a list, many to one forms a scalar, many to many is a list. If a scalar is desired where normally a list would be present, such as a bi-directional one-to-one relationship, set *uselist* to False.

The *uselist* flag is also available on an existing *relationship()* construct as a read-only attribute, which can be used to determine if this *relationship()* deals with collections or scalar attributes:

```
>>> User.addresses.property.uselist
True
```

See also:

One To One - Introduction to the "one to one" relationship pattern, which is typically when the *uselist* flag is needed.

- **viewonly=False** – when set to True, the relationship is used only for loading objects, and not for any persistence operation. A *relationship()* which specifies *viewonly* can work with a wider range of SQL operations within the *primaryjoin* condition, including operations that feature the use of a variety of comparison operators as well as SQL functions such as *cast()*. The *viewonly* flag is also of general use when defining any kind of *relationship()* that doesn't represent the full set of related objects, to prevent modifications of the collection from resulting in persistence operations.

sqlalchemy.orm.**backref**(*name*, ***kwargs*)

Create a back reference with explicit keyword arguments, which are the same arguments one can send to *relationship()*.

Used with the backref keyword argument to *relationship()* in place of a string argument, e.g.:

```
'items':relationship(
    SomeItem, backref=backref('parent', lazy='subquery'))
```

See also:

Linking Relationships with Backref

sqlalchemy.orm.**relation**(**arg*, ***kw*)

A synonym for *relationship()*.

sqlalchemy.orm.**dynamic_loader**(*argument*, ***kw*)

Construct a dynamically-loading mapper property.

This is essentially the same as using the lazy='dynamic' argument with *relationship()*:

```
dynamic_loader(SomeClass)

# is the same as

relationship(SomeClass, lazy="dynamic")
```

See the section *Dynamic Relationship Loaders* for more details on dynamic loading.

sqlalchemy.orm.**foreign**(*expr*)

Annotate a portion of a primaryjoin expression with a 'foreign' annotation.

See the section *Creating Custom Foreign Conditions* for a description of use.

New in version 0.8.

See also:

Creating Custom Foreign Conditions

remote()

`sqlalchemy.orm.remote(expr)`

Annotate a portion of a primaryjoin expression with a 'remote' annotation.

See the section *Creating Custom Foreign Conditions* for a description of use.

New in version 0.8.

See also:

Creating Custom Foreign Conditions

`foreign()`

2.4 Loading Objects

Notes and features regarding the general loading of mapped objects.

For an in-depth introduction to querying with the SQLAlchemy ORM, please see the *Object Relational Tutorial*.

2.4.1 Loading Columns

This section presents additional options regarding the loading of columns.

Deferred Column Loading

This feature allows particular columns of a table be loaded only upon direct access, instead of when the entity is queried using `Query`. This feature is useful when one wants to avoid loading a large text or binary field into memory when it's not needed. Individual columns can be lazy loaded by themselves or placed into groups that lazy-load together, using the `orm.deferred()` function to mark them as "deferred". In the example below, we define a mapping that will load each of `.excerpt` and `.photo` in separate, individual-row SELECT statements when each attribute is first referenced on the individual object instance:

```
from sqlalchemy.orm import deferred
from sqlalchemy import Integer, String, Text, Binary, Column

class Book(Base):
    __tablename__ = 'book'

    book_id = Column(Integer, primary_key=True)
    title = Column(String(200), nullable=False)
    summary = Column(String(2000))
    excerpt = deferred(Column(Text))
    photo = deferred(Column(Binary))
```

Classical mappings as always place the usage of `orm.deferred()` in the `properties` dictionary against the table-bound `Column`:

```
mapper(Book, book_table, properties={
    'photo':deferred(book_table.c.photo)
})
```

Deferred columns can be associated with a "group" name, so that they load together when any of them are first accessed. The example below defines a mapping with a `photos` deferred group. When one `.photo` is accessed, all three photos will be loaded in one SELECT statement. The `.excerpt` will be loaded separately when it is accessed:

```
class Book(Base):
    __tablename__ = 'book'

    book_id = Column(Integer, primary_key=True)
    title = Column(String(200), nullable=False)
    summary = Column(String(2000))
    excerpt = deferred(Column(Text))
    photo1 = deferred(Column(Binary), group='photos')
    photo2 = deferred(Column(Binary), group='photos')
    photo3 = deferred(Column(Binary), group='photos')
```

You can defer or undefer columns at the *Query* level using options, including *orm.defer()* and *orm.undefer()*:

```
from sqlalchemy.orm import defer, undefer

query = session.query(Book)
query = query.options(defer('summary'))
query = query.options(undefer('excerpt'))
query.all()
```

orm.deferred() attributes which are marked with a "group" can be undeferred using *orm.undefer_group()*, sending in the group name:

```
from sqlalchemy.orm import undefer_group

query = session.query(Book)
query.options(undefer_group('photos')).all()
```

Load Only Cols

An arbitrary set of columns can be selected as "load only" columns, which will be loaded while deferring all other columns on a given entity, using *orm.load_only()*:

```
from sqlalchemy.orm import load_only

session.query(Book).options(load_only("summary", "excerpt"))
```

New in version 0.9.0.

Deferred Loading with Multiple Entities

To specify column deferral options within a *Query* that loads multiple types of entity, the *Load* object can specify which parent entity to start with:

```
from sqlalchemy.orm import Load

query = session.query(Book, Author).join(Book.author)
query = query.options(
            Load(Book).load_only("summary", "excerpt"),
            Load(Author).defer("bio")
        )
```

To specify column deferral options along the path of various relationships, the options support chaining, where the loading style of each relationship is specified first, then is chained to the deferral options. Such as, to load `Book` instances, then joined-eager-load the `Author`, then apply deferral options to the `Author` entity:

```
from sqlalchemy.orm import joinedload

query = session.query(Book)
query = query.options(
            joinedload(Book.author).load_only("summary", "excerpt"),
        )
```

In the case where the loading style of parent relationships should be left unchanged, use `orm.defaultload()`:

```
from sqlalchemy.orm import defaultload

query = session.query(Book)
query = query.options(
            defaultload(Book.author).load_only("summary", "excerpt"),
        )
```

New in version 0.9.0: support for `Load` and other options which allow for better targeting of deferral options.

Column Deferral API

`sqlalchemy.orm.`**`deferred`**(**columns*, ***kw*)

Indicate a column-based mapped attribute that by default will not load unless accessed.

Parameters

- ***columns** – columns to be mapped. This is typically a single `Column` object, however a collection is supported in order to support multiple columns mapped under the same attribute.
- ****kw** – additional keyword arguments passed to `ColumnProperty`.

See also:

Deferred Column Loading

`sqlalchemy.orm.`**`defer`**(*key*, **addl_attrs*)

Indicate that the given column-oriented attribute should be deferred, e.g. not loaded until accessed.

This function is part of the `Load` interface and supports both method-chained and standalone operation.

e.g.:

```
from sqlalchemy.orm import defer

session.query(MyClass).options(
                    defer("attribute_one"),
                    defer("attribute_two"))

session.query(MyClass).options(
                    defer(MyClass.attribute_one),
                    defer(MyClass.attribute_two))
```

To specify a deferred load of an attribute on a related class, the path can be specified one token at a time, specifying the loading style for each link along the chain. To leave the loading style for a link unchanged, use `orm.defaultload()`:

```
session.query(MyClass).options(defaultload("someattr").defer("some_column"))
```

A `Load` object that is present on a certain path can have `Load.defer()` called multiple times, each will operate on the same parent entity:

```
session.query(MyClass).options(
                defaultload("someattr").
                    defer("some_column").
                    defer("some_other_column").
                    defer("another_column")
    )
```

Parameters

- **key** – Attribute to be deferred.
- ***addl_attrs** – Deprecated; this option supports the old 0.8 style of specifying a path as a series of attributes, which is now superseded by the method-chained style.

See also:

Deferred Column Loading

orm.undefer()

sqlalchemy.orm.**load_only**(**attrs*)

Indicate that for a particular entity, only the given list of column-based attribute names should be loaded; all others will be deferred.

This function is part of the *Load* interface and supports both method-chained and standalone operation.

Example - given a class `User`, load only the `name` and `fullname` attributes:

```
session.query(User).options(load_only("name", "fullname"))
```

Example - given a relationship `User.addresses -> Address`, specify subquery loading for the `User.addresses` collection, but on each `Address` object load only the `email_address` attribute:

```
session.query(User).options(
        subqueryload("addreses").load_only("email_address")
)
```

For a *Query* that has multiple entities, the lead entity can be specifically referred to using the *Load* constructor:

```
session.query(User, Address).join(User.addresses).options(
            Load(User).load_only("name", "fullname"),
            Load(Address).load_only("email_addres")
        )
```

New in version 0.9.0.

sqlalchemy.orm.**undefer**(*key*, **addl_attrs*)

Indicate that the given column-oriented attribute should be undeferred, e.g. specified within the SELECT statement of the entity as a whole.

The column being undeferred is typically set up on the mapping as a *deferred()* attribute.

This function is part of the *Load* interface and supports both method-chained and standalone operation.

Examples:

```
# undefer two columns
session.query(MyClass).options(undefer("col1"), undefer("col2"))

# undefer all columns specific to a single class using Load + *
session.query(MyClass, MyOtherClass).options(
    Load(MyClass).undefer("*"))
```

Parameters

- **key** – Attribute to be undeferred.
- ***addl_attrs** – Deprecated; this option supports the old 0.8 style of specifying a path as a series of attributes, which is now superseded by the method-chained style.

See also:

Deferred Column Loading

orm.defer()

orm.undefer_group()

sqlalchemy.orm.**undefer_group**(*name*)

Indicate that columns within the given deferred group name should be undeferred.

The columns being undeferred are set up on the mapping as *deferred()* attributes and include a "group" name.

E.g:

```
session.query(MyClass).options(undefer_group("large_attrs"))
```

To undefer a group of attributes on a related entity, the path can be spelled out using relationship loader options, such as *orm.defaultload()*:

```
session.query(MyClass).options(
    defaultload("someattr").undefer_group("large_attrs"))
```

Changed in version 0.9.0: *orm.undefer_group()* is now specific to a particiular entity load path.

See also:

Deferred Column Loading

orm.defer()

orm.undefer()

Column Bundles

The *Bundle* may be used to query for groups of columns under one namespace.

New in version 0.9.0.

The bundle allows columns to be grouped together:

```
from sqlalchemy.orm import Bundle

bn = Bundle('mybundle', MyClass.data1, MyClass.data2)
for row in session.query(bn).filter(bn.c.data1 == 'd1'):
    print row.mybundle.data1, row.mybundle.data2
```

The bundle can be subclassed to provide custom behaviors when results are fetched. The method *Bundle.create_row_processor()* is given the *Query* and a set of "row processor" functions at query execution time; these processor functions when given a result row will return the individual attribute value, which can then be adapted into any kind of return data structure. Below illustrates replacing the usual *KeyedTuple* return structure with a straight Python dictionary:

```
from sqlalchemy.orm import Bundle

class DictBundle(Bundle):
    def create_row_processor(self, query, procs, labels):
        """Override create_row_processor to return values as dictionaries"""
        def proc(row):
            return dict(
                        zip(labels, (proc(row) for proc in procs))
                    )
        return proc
```

Changed in version 1.0: The `proc()` callable passed to the `create_row_processor()` method of custom *Bundle* classes now accepts only a single "row" argument.

A result from the above bundle will return dictionary values:

```
bn = DictBundle('mybundle', MyClass.data1, MyClass.data2)
for row in session.query(bn).filter(bn.c.data1 == 'd1'):
    print row.mybundle['data1'], row.mybundle['data2']
```

The *Bundle* construct is also integrated into the behavior of *composite()*, where it is used to return composite attributes as objects when queried as individual attributes.

2.4.2 Relationship Loading Techniques

A big part of SQLAlchemy is providing a wide range of control over how related objects get loaded when querying. This behavior can be configured at mapper construction time using the `lazy` parameter to the *relationship()* function, as well as by using options with the *Query* object.

Using Loader Strategies: Lazy Loading, Eager Loading

By default, all inter-object relationships are **lazy loading**. The scalar or collection attribute associated with a *relationship()* contains a trigger which fires the first time the attribute is accessed. This trigger, in all but one case, issues a SQL call at the point of access in order to load the related object or objects:

```
>>> jack.addresses
SELECT addresses.id AS addresses_id, addresses.email_address AS addresses_email_address,
addresses.user_id AS addresses_user_id
FROM addresses
WHERE ? = addresses.user_id
[5]
[<Address(u'jack@google.com')>, <Address(u'j25@yahoo.com')>]
```

The one case where SQL is not emitted is for a simple many-to-one relationship, when the related object can be identified by its primary key alone and that object is already present in the current *Session*.

This default behavior of "load upon attribute access" is known as "lazy" or "select" loading - the name "select" because a "SELECT" statement is typically emitted when the attribute is first accessed.

In the *Object Relational Tutorial*, we introduced the concept of **Eager Loading**. We used an `option` in conjunction with the *Query* object in order to indicate that a relationship should be loaded at the same time as the parent, within a single SQL query. This option, known as *joinedload()*, connects a JOIN (by default a LEFT OUTER join) to the statement and populates the scalar/collection from the same result set as that of the parent:

```
>>> jack = session.query(User).\
... options(joinedload('addresses')).\
... filter_by(name='jack').all()
```

```
SELECT addresses_1.id AS addresses_1_id, addresses_1.email_address AS addresses_1_email_address,
addresses_1.user_id AS addresses_1_user_id, users.id AS users_id, users.name AS users_name,
users.fullname AS users_fullname, users.password AS users_password
FROM users LEFT OUTER JOIN addresses AS addresses_1 ON users.id = addresses_1.user_id
WHERE users.name = ?
['jack']
```

In addition to "joined eager loading", a second option for eager loading exists, called "subquery eager loading". This kind of eager loading emits an additional SQL statement for each collection requested, aggregated across all parent objects:

```
>>> jack = session.query(User).\
... options(subqueryload('addresses')).\
... filter_by(name='jack').all()
SELECT users.id AS users_id, users.name AS users_name, users.fullname AS users_fullname,
users.password AS users_password
FROM users
WHERE users.name = ?
('jack',)
SELECT addresses.id AS addresses_id, addresses.email_address AS addresses_email_address,
addresses.user_id AS addresses_user_id, anon_1.users_id AS anon_1_users_id
FROM (SELECT users.id AS users_id
FROM users
WHERE users.name = ?) AS anon_1 JOIN addresses ON anon_1.users_id = addresses.user_id
ORDER BY anon_1.users_id, addresses.id
('jack',)
```

The default **loader strategy** for any *relationship()* is configured by the `lazy` keyword argument, which defaults to `select` - this indicates a "select" statement . Below we set it as `joined` so that the `children` relationship is eager loaded using a JOIN:

```
# load the 'children' collection using LEFT OUTER JOIN
class Parent(Base):
    __tablename__ = 'parent'

    id = Column(Integer, primary_key=True)
    children = relationship("Child", lazy='joined')
```

We can also set it to eagerly load using a second query for all collections, using `subquery`:

```
# load the 'children' collection using a second query which
# JOINS to a subquery of the original
class Parent(Base):
    __tablename__ = 'parent'

    id = Column(Integer, primary_key=True)
    children = relationship("Child", lazy='subquery')
```

When querying, all three choices of loader strategy are available on a per-query basis, using the *joinedload()*, *subqueryload()* and *lazyload()* query options:

```
# set children to load lazily
session.query(Parent).options(lazyload('children')).all()

# set children to load eagerly with a join
session.query(Parent).options(joinedload('children')).all()

# set children to load eagerly with a second statement
session.query(Parent).options(subqueryload('children')).all()
```

The Importance of Ordering

A query which makes use of *subqueryload()* in conjunction with a limiting modifier such as *Query.first()*, *Query.limit()*, or *Query.offset()* should **always** include *Query.order_by()* against unique column(s) such as the primary key, so that the additional queries emitted by *subqueryload()* include the same ordering as used by the parent query. Without it, there is a chance that the inner query could return the wrong rows:

```
# incorrect, no ORDER BY
session.query(User).options(subqueryload(User.addresses)).first()

# incorrect if User.name is not unique
session.query(User).options(subqueryload(User.addresses)).order_by(User.name).first()

# correct
session.query(User).options(subqueryload(User.addresses)).order_by(User.name, User.id).first()
```

See also:

Why is ORDER BY required with LIMIT (especially with subqueryload())? - detailed example

Loading Along Paths

To reference a relationship that is deeper than one level, method chaining may be used. The object returned by all loader options is an instance of the *Load* class, which provides a so-called "generative" interface:

```
session.query(Parent).options(
                              joinedload('foo').
                                  joinedload('bar').
                                  joinedload('bat')
                              ).all()
```

Using method chaining, the loader style of each link in the path is explicitly stated. To navigate along a path without changing the existing loader style of a particular attribute, the *defaultload()* method/function may be used:

```
session.query(A).options(
                     defaultload("atob").joinedload("btoc")
                 ).all()
```

Changed in version 0.9.0: The previous approach of specifying dot-separated paths within loader options has been superseded by the less ambiguous approach of the *Load* object and related methods. With this system, the user specifies the style of loading for each link along the chain explicitly, rather than guessing between options like `joinedload()` vs. `joinedload_all()`. The *orm.defaultload()* is provided to allow path navigation without modification of existing loader options. The dot-separated path system as well as the `_all()` functions will remain available for backwards- compatibility indefinitely.

Default Loading Strategies

New in version 0.7.5: Default loader strategies as a new feature.

Each of *joinedload()*, *subqueryload()*, *lazyload()*, and *noload()* can be used to set the default style of *relationship()* loading for a particular query, affecting all *relationship()* -mapped attributes not otherwise specified in the *Query*. This feature is available by passing the string `'*'` as the argument to any of these options:

```
session.query(MyClass).options(lazyload('*'))
```

Above, the `lazyload('*')` option will supersede the `lazy` setting of all *relationship()* constructs in use for that query, except for those which use the `'dynamic'` style of loading. If some relationships specify `lazy='joined'` or `lazy='subquery'`, for example, using `lazyload('*')` will unilaterally cause all those relationships to use `'select'` loading, e.g. emit a SELECT statement when each attribute is accessed.

The option does not supersede loader options stated in the query, such as *eagerload()*, *subqueryload()*, etc. The query below will still use joined loading for the `widget` relationship:

```
session.query(MyClass).options(
                            lazyload('*'),
                            joinedload(MyClass.widget)
                        )
```

If multiple `'*'` options are passed, the last one overrides those previously passed.

Per-Entity Default Loading Strategies

New in version 0.9.0: Per-entity default loader strategies.

A variant of the default loader strategy is the ability to set the strategy on a per-entity basis. For example, if querying for `User` and `Address`, we can instruct all relationships on `Address` only to use lazy loading by first applying the *Load* object, then specifying the `*` as a chained option:

```
session.query(User, Address).options(Load(Address).lazyload('*'))
```

Above, all relationships on `Address` will be set to a lazy load.

The Zen of Eager Loading

The philosophy behind loader strategies is that any set of loading schemes can be applied to a particular query, and *the results don't change* - only the number of SQL statements required to fully load related objects and collections changes. A particular query might start out using all lazy loads. After using it in context, it might be revealed that particular attributes or collections are always accessed, and that it would be more efficient to change the loader strategy for these. The strategy can be changed with no other modifications to the query, the results will remain identical, but fewer SQL statements would be emitted. In theory (and pretty much in practice), nothing you can do to the *Query* would make it load a different set of primary or related objects based on a change in loader strategy.

How *joinedload()* in particular achieves this result of not impacting entity rows returned in any way is that it creates an anonymous alias of the joins it adds to your query, so that they can't be referenced by other parts of the query. For example, the query below uses *joinedload()* to create a LEFT OUTER JOIN from `users` to `addresses`, however the `ORDER BY` added against `Address.email_address` is not valid - the `Address` entity is not named in the query:

```
>>> jack = session.query(User).\
... options(joinedload(User.addresses)).\
... filter(User.name=='jack').\
... order_by(Address.email_address).all()
SELECT addresses_1.id AS addresses_1_id, addresses_1.email_address AS addresses_1_email_address,
addresses_1.user_id AS addresses_1_user_id, users.id AS users_id, users.name AS users_name,
users.fullname AS users_fullname, users.password AS users_password
FROM users LEFT OUTER JOIN addresses AS addresses_1 ON users.id = addresses_1.user_id
WHERE users.name = ? ORDER BY addresses.email_address   <-- this part is wrong !
['jack']
```

Above, `ORDER BY addresses.email_address` is not valid since `addresses` is not in the FROM list. The correct way to load the `User` records and order by email address is to use *Query.join()*:

```
>>> jack = session.query(User).\
... join(User.addresses).\
... filter(User.name=='jack').\
... order_by(Address.email_address).all()

SELECT users.id AS users_id, users.name AS users_name,
users.fullname AS users_fullname, users.password AS users_password
FROM users JOIN addresses ON users.id = addresses.user_id
WHERE users.name = ? ORDER BY addresses.email_address
['jack']
```

The statement above is of course not the same as the previous one, in that the columns from `addresses` are not included in the result at all. We can add *joinedload()* back in, so that there are two joins - one is that which we are ordering on, the other is used anonymously to load the contents of the `User.addresses` collection:

```
>>> jack = session.query(User).\
... join(User.addresses).\
... options(joinedload(User.addresses)).\
... filter(User.name=='jack').\
... order_by(Address.email_address).all()
SELECT addresses_1.id AS addresses_1_id, addresses_1.email_address AS addresses_1_email_address,
addresses_1.user_id AS addresses_1_user_id, users.id AS users_id, users.name AS users_name,
users.fullname AS users_fullname, users.password AS users_password
FROM users JOIN addresses ON users.id = addresses.user_id
LEFT OUTER JOIN addresses AS addresses_1 ON users.id = addresses_1.user_id
WHERE users.name = ? ORDER BY addresses.email_address
['jack']
```

What we see above is that our usage of *Query.join()* is to supply JOIN clauses we'd like to use in subsequent query criterion, whereas our usage of *joinedload()* only concerns itself with the loading of the `User.addresses` collection, for each `User` in the result. In this case, the two joins most probably appear redundant - which they are. If we wanted to use just one JOIN for collection loading as well as ordering, we use the *contains_eager()* option, described in *Routing Explicit Joins/Statements into Eagerly Loaded Collections* below. But to see why *joinedload()* does what it does, consider if we were **filtering** on a particular `Address`:

```
>>> jack = session.query(User).\
... join(User.addresses).\
... options(joinedload(User.addresses)).\
... filter(User.name=='jack').\
... filter(Address.email_address=='someaddress@foo.com').\
... all()
SELECT addresses_1.id AS addresses_1_id, addresses_1.email_address AS addresses_1_email_address,
addresses_1.user_id AS addresses_1_user_id, users.id AS users_id, users.name AS users_name,
users.fullname AS users_fullname, users.password AS users_password
FROM users JOIN addresses ON users.id = addresses.user_id
LEFT OUTER JOIN addresses AS addresses_1 ON users.id = addresses_1.user_id
WHERE users.name = ? AND addresses.email_address = ?
['jack', 'someaddress@foo.com']
```

Above, we can see that the two JOINs have very different roles. One will match exactly one row, that of the join of `User` and `Address` where `Address.email_address=='someaddress@foo.com'`. The other LEFT OUTER JOIN will match *all* `Address` rows related to `User`, and is only used to populate the `User.addresses` collection, for those `User` objects that are returned.

By changing the usage of *joinedload()* to another style of loading, we can change how the collection is loaded completely independently of SQL used to retrieve the actual `User` rows we want. Below we change *joinedload()* into *subqueryload()*:

```
>>> jack = session.query(User).\
... join(User.addresses).\
... options(subqueryload(User.addresses)).\
... filter(User.name=='jack').\
... filter(Address.email_address=='someaddress@foo.com').\
... all()
SELECT users.id AS users_id, users.name AS users_name,
users.fullname AS users_fullname, users.password AS users_password
FROM users JOIN addresses ON users.id = addresses.user_id
WHERE users.name = ? AND addresses.email_address = ?
['jack', 'someaddress@foo.com']

# ... subqueryload() emits a SELECT in order
# to load all address records ...
```

When using joined eager loading, if the query contains a modifier that impacts the rows returned externally to the joins, such as when using DISTINCT, LIMIT, OFFSET or equivalent, the completed statement is first wrapped inside a subquery, and the joins used specifically for joined eager loading are applied to the subquery. SQLAlchemy's joined eager loading goes the extra mile, and then ten miles further, to absolutely ensure that it does not affect the end result of the query, only the way collections and related objects are loaded, no matter what the format of the query is.

What Kind of Loading to Use ?

Which type of loading to use typically comes down to optimizing the tradeoff between number of SQL executions, complexity of SQL emitted, and amount of data fetched. Lets take two examples, a `relationship()` which references a collection, and a `relationship()` that references a scalar many-to-one reference.

- One to Many Collection
- When using the default lazy loading, if you load 100 objects, and then access a collection on each of them, a total of 101 SQL statements will be emitted, although each statement will typically be a simple SELECT without any joins.
- When using joined loading, the load of 100 objects and their collections will emit only one SQL statement. However, the total number of rows fetched will be equal to the sum of the size of all the collections, plus one extra row for each parent object that has an empty collection. Each row will also contain the full set of columns represented by the parents, repeated for each collection item - SQLAlchemy does not re-fetch these columns other than those of the primary key, however most DBAPIs (with some exceptions) will transmit the full data of each parent over the wire to the client connection in any case. Therefore joined eager loading only makes sense when the size of the collections are relatively small. The LEFT OUTER JOIN can also be performance intensive compared to an INNER join.
- When using subquery loading, the load of 100 objects will emit two SQL statements. The second statement will fetch a total number of rows equal to the sum of the size of all collections. An INNER JOIN is used, and a minimum of parent columns are requested, only the primary keys. So a subquery load makes sense when the collections are larger.
- When multiple levels of depth are used with joined or subquery loading, loading collections-within- collections will multiply the total number of rows fetched in a cartesian fashion. Both forms of eager loading always join from the original parent class.
- Many to One Reference
- When using the default lazy loading, a load of 100 objects will like in the case of the collection emit as many as 101 SQL statements. However - there is a significant exception to this, in that if the many-to-one reference is a simple foreign key reference to the target's primary key, each reference will be checked first in the current identity map using `Query.get()`. So here, if the collection of objects references a relatively small set of

target objects, or the full set of possible target objects have already been loaded into the session and are strongly referenced, using the default of *lazy='select'* is by far the most efficient way to go.

- When using joined loading, the load of 100 objects will emit only one SQL statement. The join will be a LEFT OUTER JOIN, and the total number of rows will be equal to 100 in all cases. If you know that each parent definitely has a child (i.e. the foreign key reference is NOT NULL), the joined load can be configured with `innerjoin` set to `True`, which is usually specified within the `relationship()`. For a load of objects where there are many possible target references which may have not been loaded already, joined loading with an INNER JOIN is extremely efficient.
- Subquery loading will issue a second load for all the child objects, so for a load of 100 objects there would be two SQL statements emitted. There's probably not much advantage here over joined loading, however, except perhaps that subquery loading can use an INNER JOIN in all cases whereas joined loading requires that the foreign key is NOT NULL.

Routing Explicit Joins/Statements into Eagerly Loaded Collections

The behavior of `joinedload()` is such that joins are created automatically, using anonymous aliases as targets, the results of which are routed into collections and scalar references on loaded objects. It is often the case that a query already includes the necessary joins which represent a particular collection or scalar reference, and the joins added by the joinedload feature are redundant - yet you'd still like the collections/references to be populated.

For this SQLAlchemy supplies the `contains_eager()` option. This option is used in the same manner as the `joinedload()` option except it is assumed that the `Query` will specify the appropriate joins explicitly. Below, we specify a join between `User` and `Address` and addtionally establish this as the basis for eager loading of `User.addresses`:

```
class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    addresses = relationship("Address")

class Address(Base):
    __tablename__ = 'address'

    # ...

q = session.query(User).join(User.addresses).\
            options(contains_eager(User.addresses))
```

If the "eager" portion of the statement is "aliased", the `alias` keyword argument to `contains_eager()` may be used to indicate it. This is sent as a reference to an `aliased()` or `Alias` construct:

```
# use an alias of the Address entity
adalias = aliased(Address)

# construct a Query object which expects the "addresses" results
query = session.query(User).\
    outerjoin(adalias, User.addresses).\
    options(contains_eager(User.addresses, alias=adalias))

# get results normally
r = query.all()
SELECT users.user_id AS users_user_id, users.user_name AS users_user_name, adalias.address_id AS adal
adalias.user_id AS adalias_user_id, adalias.email_address AS adalias_email_address, (...other columns
FROM users LEFT OUTER JOIN email_addresses AS email_addresses_1 ON users.user_id = email_addresses_1.
```

The path given as the argument to *contains_eager()* needs to be a full path from the starting entity. For example if we were loading `Users->orders->Order->items->Item`, the string version would look like:

```
query(User).options(contains_eager('orders').contains_eager('items'))
```

Or using the class-bound descriptor:

```
query(User).options(contains_eager(User.orders).contains_eager(Order.items))
```

Using contains_eager() to load a custom-filtered collection result

When we use *contains_eager()*, *we* are constructing ourselves the SQL that will be used to populate collections. From this, it naturally follows that we can opt to **modify** what values the collection is intended to store, by writing our SQL to load a subset of elements for collections or scalar attributes.

As an example, we can load a `User` object and eagerly load only particular addresses into its `.addresses` collection just by filtering:

```
q = session.query(User).join(User.addresses).\
            filter(Address.email.like('%ed%')).\
            options(contains_eager(User.addresses))
```

The above query will load only `User` objects which contain at least `Address` object that contains the substring `'ed'` in its `email` field; the `User.addresses` collection will contain **only** these `Address` entries, and *not* any other `Address` entries that are in fact associated with the collection.

Warning: Keep in mind that when we load only a subset of objects into a collection, that collection no longer represents what's actually in the database. If we attempted to add entries to this collection, we might find ourselves conflicting with entries that are already in the database but not locally loaded.
In addition, the **collection will fully reload normally** once the object or attribute is expired. This expiration occurs whenever the *Session.commit()*, *Session.rollback()* methods are used assuming default session settings, or the *Session.expire_all()* or *Session.expire()* methods are used.
For these reasons, prefer returning separate fields in a tuple rather than artificially altering a collection, when an object plus a custom set of related objects is desired:

```
q = session.query(User, Address).join(User.addresses).\
            filter(Address.email.like('%ed%'))
```

Advanced Usage with Arbitrary Statements

The `alias` argument can be more creatively used, in that it can be made to represent any set of arbitrary names to match up into a statement. Below it is linked to a *select()* which links a set of column objects to a string SQL statement:

```
# label the columns of the addresses table
eager_columns = select([
                    addresses.c.address_id.label('a1'),
                    addresses.c.email_address.label('a2'),
                    addresses.c.user_id.label('a3')])

# select from a raw SQL statement which uses those label names for the
# addresses table.  contains_eager() matches them up.
query = session.query(User).\
    from_statement("select users.*, addresses.address_id as a1, "
```

```
            "addresses.email_address as a2, addresses.user_id as a3 "
            "from users left outer join addresses on users.user_id=addresses.user_id").\
    options(contains_eager(User.addresses, alias=eager_columns))
```

Creating Custom Load Rules

Warning: This is an advanced technique! Great care and testing should be applied.

The ORM has various edge cases where the value of an attribute is locally available, however the ORM itself doesn't have awareness of this. There are also cases when a user-defined system of loading attributes is desirable. To support the use case of user-defined loading systems, a key function *attributes.set_committed_value()* is provided. This function is basically equivalent to Python's own `setattr()` function, except that when applied to a target object, SQLAlchemy's "attribute history" system which is used to determine flush-time changes is bypassed; the attribute is assigned in the same way as if the ORM loaded it that way from the database.

The use of *attributes.set_committed_value()* can be combined with another key event known as *InstanceEvents.load()* to produce attribute-population behaviors when an object is loaded. One such example is the bi-directional "one-to-one" case, where loading the "many-to-one" side of a one-to-one should also imply the value of the "one-to-many" side. The SQLAlchemy ORM does not consider backrefs when loading related objects, and it views a "one-to-one" as just another "one-to-many", that just happens to be one row.

Given the following mapping:

```
from sqlalchemy import Integer, ForeignKey, Column
from sqlalchemy.orm import relationship, backref
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()


class A(Base):
    __tablename__ = 'a'
    id = Column(Integer, primary_key=True)
    b_id = Column(ForeignKey('b.id'))
    b = relationship("B", backref=backref("a", uselist=False), lazy='joined')


class B(Base):
    __tablename__ = 'b'
    id = Column(Integer, primary_key=True)
```

If we query for an `A` row, and then ask it for `a.b.a`, we will get an extra SELECT:

```
>>> a1.b.a
SELECT a.id AS a_id, a.b_id AS a_b_id
FROM a
WHERE ? = a.b_id
```

This SELECT is redundant becasue `b.a` is the same value as `a1`. We can create an on-load rule to populate this for us:

```
from sqlalchemy import event
from sqlalchemy.orm import attributes

@event.listens_for(A, "load")
def load_b(target, context):
```

```
    if 'b' in target.__dict__:
        attributes.set_committed_value(target.b, 'a', target)
```

Now when we query for `A`, we will get `A.b` from the joined eager load, and `A.b.a` from our event:

```
a1 = s.query(A).first()
SELECT a.id AS a_id, a.b_id AS a_b_id, b_1.id AS b_1_id
FROM a LEFT OUTER JOIN b AS b_1 ON b_1.id = a.b_id
 LIMIT ? OFFSET ?
(1, 0)
assert a1.b.a is a1
```

Relationship Loader API

`sqlalchemy.orm.`**`contains_alias`**(*alias*)

Return a `MapperOption` that will indicate to the *Query* that the main table has been aliased.

This is a seldom-used option to suit the very rare case that *contains_eager()* is being used in conjunction with a user-defined SELECT statement that aliases the parent table. E.g.:

```
    # define an aliased UNION called 'ulist'
    ulist = users.select(users.c.user_id==7).\
                    union(users.select(users.c.user_id>7)).\
                    alias('ulist')

    # add on an eager load of "addresses"
    statement = ulist.outerjoin(addresses).\
                    select().apply_labels()

    # create query, indicating "ulist" will be an
    # alias for the main table, "addresses"
    # property should be eager loaded
    query = session.query(User).options(
                            contains_alias(ulist),
                            contains_eager(User.addresses))

    # then get results via the statement
    results = query.from_statement(statement).all()
```

Parameters **`alias`** – is the string name of an alias, or a *Alias* object representing the alias.

`sqlalchemy.orm.`**`contains_eager`**(**keys*, ***kw*)

Indicate that the given attribute should be eagerly loaded from columns stated manually in the query.

This function is part of the *Load* interface and supports both method-chained and standalone operation.

The option is used in conjunction with an explicit join that loads the desired rows, i.e.:

```
    sess.query(Order).\
            join(Order.user).\
            options(contains_eager(Order.user))
```

The above query would join from the `Order` entity to its related `User` entity, and the returned `Order` objects would have the `Order.user` attribute pre-populated.

contains_eager() also accepts an *alias* argument, which is the string name of an alias, an *alias()* construct, or an *aliased()* construct. Use this when the eagerly-loaded rows are to come from an aliased table:

```
user_alias = aliased(User)
sess.query(Order).\
        join((user_alias, Order.user)).\
        options(contains_eager(Order.user, alias=user_alias))
```

See also:

Routing Explicit Joins/Statements into Eagerly Loaded Collections

sqlalchemy.orm.**defaultload**(**keys*)

Indicate an attribute should load using its default loader style.

This method is used to link to other loader options, such as to set the `orm.defer()` option on a class that is linked to a relationship of the parent class being loaded, `orm.defaultload()` can be used to navigate this path without changing the loading style of the relationship:

```
session.query(MyClass).options(defaultload("someattr").defer("some_column"))
```

See also:

`orm.defer()`

`orm.undefer()`

sqlalchemy.orm.**eagerload**(**args*, ***kwargs*)

A synonym for `joinedload()`.

sqlalchemy.orm.**eagerload_all**(**args*, ***kwargs*)

A synonym for `joinedload_all()`

sqlalchemy.orm.**immediateload**(**keys*)

Indicate that the given attribute should be loaded using an immediate load with a per-attribute SELECT statement.

This function is part of the `Load` interface and supports both method-chained and standalone operation.

See also:

Relationship Loading Techniques

`orm.joinedload()`

`orm.lazyload()`

`relationship.lazy`

sqlalchemy.orm.**joinedload**(**keys*, ***kw*)

Indicate that the given attribute should be loaded using joined eager loading.

This function is part of the `Load` interface and supports both method-chained and standalone operation.

examples:

```
# joined-load the "orders" collection on "User"
query(User).options(joinedload(User.orders))

# joined-load Order.items and then Item.keywords
query(Order).options(joinedload(Order.items).joinedload(Item.keywords))

# lazily load Order.items, but when Items are loaded,
# joined-load the keywords collection
query(Order).options(lazyload(Order.items).joinedload(Item.keywords))
```

Parameters **innerjoin** – if `True`, indicates that the joined eager load should use an inner join instead of the default of left outer join:

```
query(Order).options(joinedload(Order.user, innerjoin=True))
```

In order to chain multiple eager joins together where some may be OUTER and others INNER, right-nested joins are used to link them:

```
query(A).options(
    joinedload(A.bs, innerjoin=False).
        joinedload(B.cs, innerjoin=True)
)
```

The above query, linking A.bs via "outer" join and B.cs via "inner" join would render the joins as "a LEFT OUTER JOIN (b JOIN c)". When using SQLite, this form of JOIN is translated to use full subqueries as this syntax is otherwise not directly supported.

The `innerjoin` flag can also be stated with the term `"unnested"`. This will prevent joins from being right-nested, and will instead link an "innerjoin" eagerload to an "outerjoin" eagerload by bypassing the "inner" join. Using this form as follows:

```
query(A).options(
    joinedload(A.bs, innerjoin=False).
        joinedload(B.cs, innerjoin="unnested")
)
```

Joins will be rendered as "a LEFT OUTER JOIN b LEFT OUTER JOIN c", so that all of "a" is matched rather than being incorrectly limited by a "b" that does not contain a "c".

Note: The "unnested" flag does **not** affect the JOIN rendered from a many-to-many association table, e.g. a table configured as *`relationship.secondary`*, to the target table; for correctness of results, these joins are always INNER and are therefore right-nested if linked to an OUTER join.

New in version 0.9.4: Added support for "nesting" of eager "inner" joins. See *Right-nested inner joins available in joined eager loads*.

Changed in version 1.0.0: `innerjoin=True` now implies `innerjoin="nested"`, whereas in 0.9 it implied `innerjoin="unnested"`. In order to achieve the pre-1.0 "unnested" inner join behavior, use the value `innerjoin="unnested"`. See *Right inner join nesting now the default for joinedload with innerjoin=True*.

Note: The joins produced by *`orm.joinedload()`* are **anonymously aliased**. The criteria by which the join proceeds cannot be modified, nor can the *`Query`* refer to these joins in any way, including ordering.

To produce a specific SQL JOIN which is explicitly available, use *`Query.join()`*. To combine explicit JOINs with eager loading of collections, use *`orm.contains_eager()`*; see *Routing Explicit Joins/Statements into Eagerly Loaded Collections*.

See also:

Relationship Loading Techniques

Routing Explicit Joins/Statements into Eagerly Loaded Collections

`orm.subqueryload()`

`orm.lazyload()`

relationship.lazy

relationship.innerjoin - *relationship()*-level version of the *joinedload.innerjoin* option.

sqlalchemy.orm.**joinedload_all**(**keys*, ***kw*)

Produce a standalone "all" option for *orm.joinedload()*.

Deprecated since version 0.9.0: The "_all()" style is replaced by method chaining, e.g.:

```
session.query(MyClass).options(
    joinedload("someattribute").joinedload("anotherattribute")
)
```

sqlalchemy.orm.**lazyload**(**keys*)

Indicate that the given attribute should be loaded using "lazy" loading.

This function is part of the *Load* interface and supports both method-chained and standalone operation.

See also:

relationship.lazy

sqlalchemy.orm.**noload**(**keys*)

Indicate that the given relationship attribute should remain unloaded.

This function is part of the *Load* interface and supports both method-chained and standalone operation.

orm.noload() applies to *relationship()* attributes; for column-based attributes, see *orm.defer()*.

sqlalchemy.orm.**subqueryload**(**keys*)

Indicate that the given attribute should be loaded using subquery eager loading.

This function is part of the *Load* interface and supports both method-chained and standalone operation.

examples:

```
# subquery-load the "orders" collection on "User"
query(User).options(subqueryload(User.orders))

# subquery-load Order.items and then Item.keywords
query(Order).options(subqueryload(Order.items).subqueryload(Item.keywords))

# lazily load Order.items, but when Items are loaded,
# subquery-load the keywords collection
query(Order).options(lazyload(Order.items).subqueryload(Item.keywords))
```

See also:

Relationship Loading Techniques

orm.joinedload()

orm.lazyload()

relationship.lazy

sqlalchemy.orm.**subqueryload_all**(**keys*)

Produce a standalone "all" option for *orm.subqueryload()*.

Deprecated since version 0.9.0: The "_all()" style is replaced by method chaining, e.g.:

```
session.query(MyClass).options(
    subqueryload("someattribute").subqueryload("anotherattribute")
)
```

2.4.3 Constructors and Object Initialization

Mapping imposes no restrictions or requirements on the constructor (`__init__`) method for the class. You are free to require any arguments for the function that you wish, assign attributes to the instance that are unknown to the ORM, and generally do anything else you would normally do when writing a constructor for a Python class.

The SQLAlchemy ORM does not call `__init__` when recreating objects from database rows. The ORM's process is somewhat akin to the Python standard library's `pickle` module, invoking the low level `__new__` method and then quietly restoring attributes directly on the instance rather than calling `__init__`.

If you need to do some setup on database-loaded instances before they're ready to use, you can use the `@reconstructor` decorator to tag a method as the ORM counterpart to `__init__`. SQLAlchemy will call this method with no arguments every time it loads or reconstructs one of your instances. This is useful for recreating transient properties that are normally assigned in your `__init__`:

```
from sqlalchemy import orm

class MyMappedClass(object):
    def __init__(self, data):
        self.data = data
        # we need stuff on all instances, but not in the database.
        self.stuff = []

    @orm.reconstructor
    def init_on_load(self):
        self.stuff = []
```

When `obj = MyMappedClass()` is executed, Python calls the `__init__` method as normal and the `data` argument is required. When instances are loaded during a *Query* operation as in `query(MyMappedClass).one()`, `init_on_load` is called.

Any method may be tagged as the *reconstructor()*, even the `__init__` method. SQLAlchemy will call the reconstructor method with no arguments. Scalar (non-collection) database-mapped attributes of the instance will be available for use within the function. Eagerly-loaded collections are generally not yet available and will usually only contain the first element. ORM state changes made to objects at this stage will not be recorded for the next flush() operation, so the activity within a reconstructor should be conservative.

reconstructor() is a shortcut into a larger system of "instance level" events, which can be subscribed to using the event API - see *InstanceEvents* for the full API description of these events.

`sqlalchemy.orm.`**`reconstructor`**(*fn*)

Decorate a method as the 'reconstructor' hook.

Designates a method as the "reconstructor", an `__init__`-like method that will be called by the ORM after the instance has been loaded from the database or otherwise reconstituted.

The reconstructor will be invoked with no arguments. Scalar (non-collection) database-mapped attributes of the instance will be available for use within the function. Eagerly-loaded collections are generally not yet available and will usually only contain the first element. ORM state changes made to objects at this stage will not be recorded for the next flush() operation, so the activity within a reconstructor should be conservative.

2.4.4 Query API

The Query Object

Query is produced in terms of a given *Session*, using the *query()* method:

```
q = session.query(SomeMappedClass)
```

Following is the full interface for the *Query* object.

class `sqlalchemy.orm.query.`**`Query`**(*entities*, *session=None*)

ORM-level SQL construction object.

Query is the source of all SELECT statements generated by the ORM, both those formulated by end-user query operations as well as by high level internal operations such as related collection loading. It features a generative interface whereby successive calls return a new *Query* object, a copy of the former with additional criteria and options associated with it.

Query objects are normally initially generated using the *query()* method of *Session*. For a full walk-through of *Query* usage, see the *Object Relational Tutorial*.

`add_column`(*column*)

Add a column expression to the list of result columns to be returned.

Pending deprecation: *add_column()* will be superseded by *add_columns()*.

`add_columns`(**column*)

Add one or more column expressions to the list of result columns to be returned.

`add_entity`(*entity*, *alias=None*)

add a mapped entity to the list of result columns to be returned.

`all`()

Return the results represented by this `Query` as a list.

This results in an execution of the underlying query.

`as_scalar`()

Return the full SELECT statement represented by this *Query*, converted to a scalar subquery.

Analogous to *sqlalchemy.sql.expression.SelectBase.as_scalar()*.

New in version 0.6.5.

`autoflush`(*setting*)

Return a Query with a specific 'autoflush' setting.

Note that a Session with autoflush=False will not autoflush, even if this flag is set to True at the Query level. Therefore this flag is usually used only to disable autoflush for a specific Query.

`column_descriptions`

Return metadata about the columns which would be returned by this *Query*.

Format is a list of dictionaries:

```
user_alias = aliased(User, name='user2')
q = sess.query(User, User.id, user_alias)

# this expression:
q.column_descriptions

# would return:
[
    {
        'name':'User',
        'type':User,
        'aliased':False,
        'expr':User,
        'entity': User
```

```
        },
        {
            'name':'id',
            'type':Integer(),
            'aliased':False,
            'expr':User.id,
            'entity': User
        },
        {
            'name':'user2',
            'type':User,
            'aliased':True,
            'expr':user_alias,
            'entity': user_alias
        }
    ]
```

correlate(**args*)

Return a `Query` construct which will correlate the given FROM clauses to that of an enclosing `Query` or `select()`.

The method here accepts mapped classes, `aliased()` constructs, and `mapper()` constructs as arguments, which are resolved into expression constructs, in addition to appropriate expression constructs.

The correlation arguments are ultimately passed to `Select.correlate()` after coercion to expression constructs.

The correlation arguments take effect in such cases as when `Query.from_self()` is used, or when a subquery as returned by `Query.subquery()` is embedded in another `select()` construct.

count()

Return a count of rows this Query would return.

This generates the SQL for this Query as follows:

```
SELECT count(1) AS count_1 FROM (
    SELECT <rest of query follows...>
) AS anon_1
```

Changed in version 0.7: The above scheme is newly refined as of 0.7b3.

For fine grained control over specific columns to count, to skip the usage of a subquery or otherwise control of the FROM clause, or to use other aggregate functions, use `func` expressions in conjunction with `query()`, i.e.:

```
from sqlalchemy import func

# count User records, without
# using a subquery.
session.query(func.count(User.id))

# return count of user "id" grouped
# by "name"
session.query(func.count(User.id)).\
        group_by(User.name)

from sqlalchemy import distinct

# count distinct "name" values
session.query(func.count(distinct(User.name)))
```

cte(*name=None*, *recursive=False*)

Return the full SELECT statement represented by this `Query` represented as a common table expression (CTE).

New in version 0.7.6.

Parameters and usage are the same as those of the `SelectBase.cte()` method; see that method for further details.

Here is the Postgresql WITH RECURSIVE example. Note that, in this example, the `included_parts` cte and the `incl_alias` alias of it are Core selectables, which means the columns are accessed via the `.c.` attribute. The `parts_alias` object is an `orm.aliased()` instance of the `Part` entity, so column-mapped attributes are available directly:

```
from sqlalchemy.orm import aliased

class Part(Base):
    __tablename__ = 'part'
    part = Column(String, primary_key=True)
    sub_part = Column(String, primary_key=True)
    quantity = Column(Integer)

included_parts = session.query(
                Part.sub_part,
                Part.part,
                Part.quantity).\
                    filter(Part.part=="our part").\
                    cte(name="included_parts", recursive=True)

incl_alias = aliased(included_parts, name="pr")
parts_alias = aliased(Part, name="p")
included_parts = included_parts.union_all(
    session.query(
        parts_alias.sub_part,
        parts_alias.part,
        parts_alias.quantity).\
            filter(parts_alias.part==incl_alias.c.sub_part)
    )

q = session.query(
        included_parts.c.sub_part,
        func.sum(included_parts.c.quantity).
            label('total_quantity')
    ).\
    group_by(included_parts.c.sub_part)
```

See also:

`SelectBase.cte()`

delete(*synchronize_session='evaluate'*)

Perform a bulk delete query.

Deletes rows matched by this query from the database.

E.g.:

```
sess.query(User).filter(User.age == 25).\
    delete(synchronize_session=False)
```

```
sess.query(User).filter(User.age == 25).\
    delete(synchronize_session='evaluate')
```

Warning: The `Query.delete()` method is a "bulk" operation, which bypasses ORM unit-of-work automation in favor of greater performance. **Please read all caveats and warnings below.**

Parameters **synchronize_session** – chooses the strategy for the removal of matched objects from the session. Valid values are:

`False` - don't synchronize the session. This option is the most efficient and is reliable once the session is expired, which typically occurs after a commit(), or explicitly using expire_all(). Before the expiration, objects may still remain in the session which were in fact deleted which can lead to confusing results if they are accessed via get() or already loaded collections.

`'fetch'` - performs a select query before the delete to find objects that are matched by the delete query and need to be removed from the session. Matched objects are removed from the session.

`'evaluate'` - Evaluate the query's criteria in Python straight on the objects in the session. If evaluation of the criteria isn't implemented, an error is raised.

The expression evaluator currently doesn't account for differing string collations between the database and Python.

Returns the count of rows matched as returned by the database's "row count" feature.

Warning: Additional Caveats for bulk query deletes

- The method does **not** offer in-Python cascading of relationships - it is assumed that ON DELETE CASCADE/SET NULL/etc. is configured for any foreign key references which require it, otherwise the database may emit an integrity violation if foreign key references are being enforced. After the DELETE, dependent objects in the `Session` which were impacted by an ON DELETE may not contain the current state, or may have been deleted. This issue is resolved once the `Session` is expired, which normally occurs upon `Session.commit()` or can be forced by using `Session.expire_all()`. Accessing an expired object whose row has been deleted will invoke a SELECT to locate the row; when the row is not found, an `ObjectDeletedError` is raised.
- The `'fetch'` strategy results in an additional SELECT statement emitted and will significantly reduce performance.
- The `'evaluate'` strategy performs a scan of all matching objects within the `Session`; if the contents of the `Session` are expired, such as via a proceeding `Session.commit()` call, **this will result in SELECT queries emitted for every matching object**.
- The `MapperEvents.before_delete()` and `MapperEvents.after_delete()` events **are not invoked** from this method. Instead, the `SessionEvents.after_bulk_delete()` method is provided to act upon a mass DELETE of entity rows.

See also:

`Query.update()`

Inserts, Updates and Deletes - Core SQL tutorial

distinct(**criterion*)

Apply a `DISTINCT` to the query and return the newly resulting `Query`.

Note: The `distinct()` call includes logic that will automatically add columns from the ORDER BY of the query to the columns clause of the SELECT statement, to satisfy the common need of the database

backend that ORDER BY columns be part of the SELECT list when DISTINCT is used. These columns *are not* added to the list of columns actually fetched by the `Query`, however, so would not affect results. The columns are passed through when using the `Query.statement` accessor, however.

Parameters ***expr** – optional column expressions. When present, the Postgresql dialect will render a `DISTINCT ON (<expressions>>)` construct.

enable_assertions(*value*)

Control whether assertions are generated.

When set to False, the returned Query will not assert its state before certain operations, including that LIMIT/OFFSET has not been applied when filter() is called, no criterion exists when get() is called, and no "from_statement()" exists when filter()/order_by()/group_by() etc. is called. This more permissive mode is used by custom Query subclasses to specify criterion or other modifiers outside of the usual usage patterns.

Care should be taken to ensure that the usage pattern is even possible. A statement applied by from_statement() will override any criterion set by filter() or order_by(), for example.

enable_eagerloads(*value*)

Control whether or not eager joins and subqueries are rendered.

When set to False, the returned Query will not render eager joins regardless of `joinedload()`, `subqueryload()` options or mapper-level `lazy='joined'/lazy='subquery'` configurations.

This is used primarily when nesting the Query's statement into a subquery or other selectable, or when using `Query.yield_per()`.

except_(**q*)

Produce an EXCEPT of this Query against one or more queries.

Works the same way as `union()`. See that method for usage examples.

except_all(**q*)

Produce an EXCEPT ALL of this Query against one or more queries.

Works the same way as `union()`. See that method for usage examples.

execution_options(***kwargs*)

Set non-SQL options which take effect during execution.

The options are the same as those accepted by `Connection.execution_options()`.

Note that the `stream_results` execution option is enabled automatically if the `yield_per()` method is used.

exists()

A convenience method that turns a query into an EXISTS subquery of the form EXISTS (SELECT 1 FROM ... WHERE ...).

e.g.:

```
q = session.query(User).filter(User.name == 'fred')
session.query(q.exists())
```

Producing SQL similar to:

```
SELECT EXISTS (
    SELECT 1 FROM users WHERE users.name = :name_1
) AS anon_1
```

The EXISTS construct is usually used in the WHERE clause:

```
session.query(User.id).filter(q.exists()).scalar()
```

Note that some databases such as SQL Server don't allow an EXISTS expression to be present in the columns clause of a SELECT. To select a simple boolean value based on the exists as a WHERE, use *literal()*:

```
from sqlalchemy import literal

session.query(literal(True)).filter(q.exists()).scalar()
```

New in version 0.8.1.

filter(**criterion*)

apply the given filtering criterion to a copy of this *Query*, using SQL expressions.

e.g.:

```
session.query(MyClass).filter(MyClass.name == 'some name')
```

Multiple criteria may be specified as comma separated; the effect is that they will be joined together using the *and_()* function:

```
session.query(MyClass).\
    filter(MyClass.name == 'some name', MyClass.id > 5)
```

The criterion is any SQL expression object applicable to the WHERE clause of a select. String expressions are coerced into SQL expression constructs via the *text()* construct.

See also:

Query.filter_by() - filter on keyword expressions.

filter_by(***kwargs*)

apply the given filtering criterion to a copy of this *Query*, using keyword expressions.

e.g.:

```
session.query(MyClass).filter_by(name = 'some name')
```

Multiple criteria may be specified as comma separated; the effect is that they will be joined together using the *and_()* function:

```
session.query(MyClass).\
    filter_by(name = 'some name', id = 5)
```

The keyword expressions are extracted from the primary entity of the query, or the last entity that was the target of a call to *Query.join()*.

See also:

Query.filter() - filter on SQL expressions.

first()

Return the first result of this `Query` or None if the result doesn't contain any row.

first() applies a limit of one within the generated SQL, so that only one primary entity row is generated on the server side (note this may consist of multiple result rows if join-loaded collections are present).

Calling *Query.first()* results in an execution of the underlying query.

See also:

Query.one()

Query.one_or_none()

from_self(**entities*)

return a Query that selects from this Query's SELECT statement.

Query.from_self() essentially turns the SELECT statement into a SELECT of itself. Given a query such as:

```
q = session.query(User).filter(User.name.like('e%'))
```

Given the *Query.from_self()* version:

```
q = session.query(User).filter(User.name.like('e%')).from_self()
```

This query renders as:

```
SELECT anon_1.user_id AS anon_1_user_id,
       anon_1.user_name AS anon_1_user_name
FROM (SELECT "user".id AS user_id, "user".name AS user_name
FROM "user"
WHERE "user".name LIKE :name_1) AS anon_1
```

There are lots of cases where *Query.from_self()* may be useful. A simple one is where above, we may want to apply a row LIMIT to the set of user objects we query against, and then apply additional joins against that row-limited set:

```
q = session.query(User).filter(User.name.like('e%')).\
    limit(5).from_self().\
    join(User.addresses).filter(Address.email.like('q%'))
```

The above query joins to the `Address` entity but only against the first five results of the `User` query:

```
SELECT anon_1.user_id AS anon_1_user_id,
       anon_1.user_name AS anon_1_user_name
FROM (SELECT "user".id AS user_id, "user".name AS user_name
FROM "user"
WHERE "user".name LIKE :name_1
 LIMIT :param_1) AS anon_1
JOIN address ON anon_1.user_id = address.user_id
WHERE address.email LIKE :email_1
```

Automatic Aliasing

Another key behavior of *Query.from_self()* is that it applies **automatic aliasing** to the entities inside the subquery, when they are referenced on the outside. Above, if we continue to refer to the `User` entity without any additional aliasing applied to it, those references wil be in terms of the subquery:

```
q = session.query(User).filter(User.name.like('e%')).\
    limit(5).from_self().\
    join(User.addresses).filter(Address.email.like('q%')).\
    order_by(User.name)
```

The ORDER BY against `User.name` is aliased to be in terms of the inner subquery:

```
SELECT anon_1.user_id AS anon_1_user_id,
       anon_1.user_name AS anon_1_user_name
FROM (SELECT "user".id AS user_id, "user".name AS user_name
FROM "user"
WHERE "user".name LIKE :name_1
 LIMIT :param_1) AS anon_1
```

```
JOIN address ON anon_1.user_id = address.user_id
WHERE address.email LIKE :email_1 ORDER BY anon_1.user_name
```

The automatic aliasing feature only works in a **limited** way, for simple filters and orderings. More ambitious constructions such as referring to the entity in joins should prefer to use explicit subquery objects, typically making use of the *Query.subquery()* method to produce an explicit subquery object. Always test the structure of queries by viewing the SQL to ensure a particular structure does what's expected!

Changing the Entities

Query.from_self() also includes the ability to modify what columns are being queried. In our example, we want `User.id` to be queried by the inner query, so that we can join to the `Address` entity on the outside, but we only wanted the outer query to return the `Address.email` column:

```
q = session.query(User).filter(User.name.like('e%')).\
    limit(5).from_self(Address.email).\
    join(User.addresses).filter(Address.email.like('q%'))
```

yielding:

```
SELECT address.email AS address_email
FROM (SELECT "user".id AS user_id, "user".name AS user_name
FROM "user"
WHERE "user".name LIKE :name_1
 LIMIT :param_1) AS anon_1
JOIN address ON anon_1.user_id = address.user_id
WHERE address.email LIKE :email_1
```

Looking out for Inner / Outer Columns

Keep in mind that when referring to columns that originate from inside the subquery, we need to ensure they are present in the columns clause of the subquery itself; this is an ordinary aspect of SQL. For example, if we wanted to load from a joined entity inside the subquery using *contains_eager()*, we need to add those columns. Below illustrates a join of `Address` to `User`, then a subquery, and then we'd like *contains_eager()* to access the `User` columns:

```
q = session.query(Address).join(Address.user).\
    filter(User.name.like('e%'))

q = q.add_entity(User).from_self().\
    options(contains_eager(Address.user))
```

We use *Query.add_entity()* above **before** we call *Query.from_self()* so that the `User` columns are present in the inner subquery, so that they are available to the *contains_eager()* modifier we are using on the outside, producing:

```
SELECT anon_1.address_id AS anon_1_address_id,
       anon_1.address_email AS anon_1_address_email,
       anon_1.address_user_id AS anon_1_address_user_id,
       anon_1.user_id AS anon_1_user_id,
       anon_1.user_name AS anon_1_user_name
FROM (
    SELECT address.id AS address_id,
    address.email AS address_email,
    address.user_id AS address_user_id,
    "user".id AS user_id,
    "user".name AS user_name
FROM address JOIN "user" ON "user".id = address.user_id
WHERE "user".name LIKE :name_1) AS anon_1
```

If we didn't call `add_entity(User)`, but still asked *contains_eager()* to load the `User` entity, it would be forced to add the table on the outside without the correct join criteria - note the `anon1, "user"` phrase at the end:

```
-- incorrect query
SELECT anon_1.address_id AS anon_1_address_id,
       anon_1.address_email AS anon_1_address_email,
       anon_1.address_user_id AS anon_1_address_user_id,
       "user".id AS user_id,
       "user".name AS user_name
FROM (
    SELECT address.id AS address_id,
    address.email AS address_email,
    address.user_id AS address_user_id
FROM address JOIN "user" ON "user".id = address.user_id
WHERE "user".name LIKE :name_1) AS anon_1, "user"
```

Parameters **`*entities`** – optional list of entities which will replace those being selected.

`from_statement`(*statement*)

Execute the given SELECT statement and return results.

This method bypasses all internal statement compilation, and the statement is executed without modification.

The statement is typically either a *text()* or *select()* construct, and should return the set of columns appropriate to the entity class represented by this *Query*.

See also:

Using Textual SQL - usage examples in the ORM tutorial

`get`(*ident*)

Return an instance based on the given primary key identifier, or `None` if not found.

E.g.:

```
my_user = session.query(User).get(5)

some_object = session.query(VersionedFoo).get((5, 10))
```

get() is special in that it provides direct access to the identity map of the owning *Session*. If the given primary key identifier is present in the local identity map, the object is returned directly from this collection and no SQL is emitted, unless the object has been marked fully expired. If not present, a SELECT is performed in order to locate the object.

get() also will perform a check if the object is present in the identity map and marked as expired - a SELECT is emitted to refresh the object as well as to ensure that the row is still present. If not, *ObjectDeletedError* is raised.

get() is only used to return a single mapped instance, not multiple instances or individual column constructs, and strictly on a single primary key value. The originating *Query* must be constructed in this way, i.e. against a single mapped entity, with no additional filtering criterion. Loading options via *options()* may be applied however, and will be used if the object is not yet locally present.

A lazy-loading, many-to-one attribute configured by *relationship()*, using a simple foreign-key-to-primary-key criterion, will also use an operation equivalent to *get()* in order to retrieve the target value from the local identity map before querying the database. See Relationship Loading Techniques for further details on relationship loading.

Parameters **ident** – A scalar or tuple value representing the primary key. For a composite primary key, the order of identifiers corresponds in most cases to that of the mapped *Table* object's primary key columns. For a *mapper()* that was given the `primary key` argument during construction, the order of identifiers corresponds to the elements present in this collection.

Returns The object instance, or `None`.

group_by(**criterion*)

apply one or more GROUP BY criterion to the query and return the newly resulting *Query*

having(*criterion*)

apply a HAVING criterion to the query and return the newly resulting *Query*.

having() is used in conjunction with *group_by()*.

HAVING criterion makes it possible to use filters on aggregate functions like COUNT, SUM, AVG, MAX, and MIN, eg.:

```
q = session.query(User.id).\
            join(User.addresses).\
            group_by(User.id).\
            having(func.count(Address.id) > 2)
```

instances(*cursor*, *_Query__context=None*)

Given a ResultProxy cursor as returned by connection.execute(), return an ORM result as an iterator.

e.g.:

```
result = engine.execute("select * from users")
for u in session.query(User).instances(result):
    print u
```

intersect(**q*)

Produce an INTERSECT of this Query against one or more queries.

Works the same way as *union()*. See that method for usage examples.

intersect_all(**q*)

Produce an INTERSECT ALL of this Query against one or more queries.

Works the same way as *union()*. See that method for usage examples.

join(**props*, ***kwargs*)

Create a SQL JOIN against this *Query* object's criterion and apply generatively, returning the newly resulting *Query*.

Simple Relationship Joins

Consider a mapping between two classes `User` and `Address`, with a relationship `User.addresses` representing a collection of `Address` objects associated with each `User`. The most common usage of *join()* is to create a JOIN along this relationship, using the `User.addresses` attribute as an indicator for how this should occur:

```
q = session.query(User).join(User.addresses)
```

Where above, the call to *join()* along `User.addresses` will result in SQL equivalent to:

```
SELECT user.* FROM user JOIN address ON user.id = address.user_id
```

In the above example we refer to `User.addresses` as passed to *join()* as the *on clause*, that is, it indicates how the "ON" portion of the JOIN should be constructed. For a single-entity query such as

the one above (i.e. we start by selecting only from `User` and nothing else), the relationship can also be specified by its string name:

```
q = session.query(User).join("addresses")
```

join() can also accommodate multiple "on clause" arguments to produce a chain of joins, such as below where a join across four related entities is constructed:

```
q = session.query(User).join("orders", "items", "keywords")
```

The above would be shorthand for three separate calls to *join()*, each using an explicit attribute to indicate the source entity:

```
q = session.query(User).\
        join(User.orders).\
        join(Order.items).\
        join(Item.keywords)
```

Joins to a Target Entity or Selectable

A second form of *join()* allows any mapped entity or core selectable construct as a target. In this usage, *join()* will attempt to create a JOIN along the natural foreign key relationship between two entities:

```
q = session.query(User).join(Address)
```

The above calling form of *join()* will raise an error if either there are no foreign keys between the two entities, or if there are multiple foreign key linkages between them. In the above calling form, *join()* is called upon to create the "on clause" automatically for us. The target can be any mapped entity or selectable, such as a *Table*:

```
q = session.query(User).join(addresses_table)
```

Joins to a Target with an ON Clause

The third calling form allows both the target entity as well as the ON clause to be passed explicitly. Suppose for example we wanted to join to `Address` twice, using an alias the second time. We use *aliased()* to create a distinct alias of `Address`, and join to it using the `target, onclause` form, so that the alias can be specified explicitly as the target along with the relationship to instruct how the ON clause should proceed:

```
a_alias = aliased(Address)

q = session.query(User).\
        join(User.addresses).\
        join(a_alias, User.addresses).\
        filter(Address.email_address=='ed@foo.com').\
        filter(a_alias.email_address=='ed@bar.com')
```

Where above, the generated SQL would be similar to:

```
SELECT user.* FROM user
    JOIN address ON user.id = address.user_id
    JOIN address AS address_1 ON user.id=address_1.user_id
    WHERE address.email_address = :email_address_1
    AND address_1.email_address = :email_address_2
```

The two-argument calling form of *join()* also allows us to construct arbitrary joins with SQL-oriented "on clause" expressions, not relying upon configured relationships at all. Any SQL expression can be passed as the ON clause when using the two-argument form, which should refer to the target entity in some way as well as an applicable source entity:

```
q = session.query(User).join(Address, User.id==Address.user_id)
```

Changed in version 0.7: In SQLAlchemy 0.6 and earlier, the two argument form of *join()* requires the usage of a tuple: `query(User).join((Address, User.id==Address.user_id))`. This calling form is accepted in 0.7 and further, though is not necessary unless multiple join conditions are passed to a single *join()* call, which itself is also not generally necessary as it is now equivalent to multiple calls (this wasn't always the case).

Advanced Join Targeting and Adaption

There is a lot of flexibility in what the "target" can be when using *join()*. As noted previously, it also accepts *Table* constructs and other selectables such as *alias()* and *select()* constructs, with either the one or two-argument forms:

```
addresses_q = select([Address.user_id]).\
            where(Address.email_address.endswith("@bar.com")).\
            alias()

q = session.query(User).\
            join(addresses_q, addresses_q.c.user_id==User.id)
```

join() also features the ability to *adapt* a *relationship()* -driven ON clause to the target selectable. Below we construct a JOIN from `User` to a subquery against `Address`, allowing the relationship denoted by `User.addresses` to *adapt* itself to the altered target:

```
address_subq = session.query(Address).\
                filter(Address.email_address == 'ed@foo.com').\
                subquery()

q = session.query(User).join(address_subq, User.addresses)
```

Producing SQL similar to:

```
SELECT user.* FROM user
    JOIN (
        SELECT address.id AS id,
                address.user_id AS user_id,
                address.email_address AS email_address
        FROM address
        WHERE address.email_address = :email_address_1
    ) AS anon_1 ON user.id = anon_1.user_id
```

The above form allows one to fall back onto an explicit ON clause at any time:

```
q = session.query(User).\
        join(address_subq, User.id==address_subq.c.user_id)
```

Controlling what to Join From

While *join()* exclusively deals with the "right" side of the JOIN, we can also control the "left" side, in those cases where it's needed, using *select_from()*. Below we construct a query against `Address` but can still make usage of `User.addresses` as our ON clause by instructing the *Query* to select first from the `User` entity:

```
q = session.query(Address).select_from(User).\
                join(User.addresses).\
                filter(User.name == 'ed')
```

Which will produce SQL similar to:

```
SELECT address.* FROM user
    JOIN address ON user.id=address.user_id
    WHERE user.name = :name_1
```

Constructing Aliases Anonymously

join() can construct anonymous aliases using the `aliased=True` flag. This feature is useful when a query is being joined algorithmically, such as when querying self-referentially to an arbitrary depth:

```
q = session.query(Node).\
        join("children", "children", aliased=True)
```

When `aliased=True` is used, the actual "alias" construct is not explicitly available. To work with it, methods such as *Query.filter()* will adapt the incoming entity to the last join point:

```
q = session.query(Node).\
        join("children", "children", aliased=True).\
        filter(Node.name == 'grandchild 1')
```

When using automatic aliasing, the `from_joinpoint=True` argument can allow a multi-node join to be broken into multiple calls to *join()*, so that each path along the way can be further filtered:

```
q = session.query(Node).\
        join("children", aliased=True).\
        filter(Node.name='child 1').\
        join("children", aliased=True, from_joinpoint=True).\
        filter(Node.name == 'grandchild 1')
```

The filtering aliases above can then be reset back to the original `Node` entity using *reset_joinpoint()*:

```
q = session.query(Node).\
        join("children", "children", aliased=True).\
        filter(Node.name == 'grandchild 1').\
        reset_joinpoint().\
        filter(Node.name == 'parent 1)
```

For an example of `aliased=True`, see the distribution example *XML Persistence* which illustrates an XPath-like query system using algorithmic joins.

Parameters

- ***props** – A collection of one or more join conditions, each consisting of a relationship-bound attribute or string relationship name representing an "on clause", or a single target entity, or a tuple in the form of `(target, onclause)`. A special two-argument calling form of the form `target, onclause` is also accepted.
- **aliased=False** – If True, indicate that the JOIN target should be anonymously aliased. Subsequent calls to *filter()* and similar will adapt the incoming criterion to the target alias, until *reset_joinpoint()* is called.
- **isouter=False** – If True, the join used will be a left outer join, just as if the *Query.outerjoin()* method were called. This flag is here to maintain consistency with the same flag as accepted by *FromClause.join()* and other Core constructs.

  New in version 1.0.0.
- **from_joinpoint=False** – When using `aliased=True`, a setting of True here will cause the join to be from the most recent joined target, rather than starting back from the original FROM clauses of the query.

See also:

Querying with Joins in the ORM tutorial.

Mapping Class Inheritance Hierarchies for details on how `join()` is used for inheritance relationships.

`orm.join()` - a standalone ORM-level join function, used internally by `Query.join()`, which in previous SQLAlchemy versions was the primary ORM-level joining interface.

label(*name*)

Return the full SELECT statement represented by this `Query`, converted to a scalar subquery with a label of the given name.

Analogous to `sqlalchemy.sql.expression.SelectBase.label()`.

New in version 0.6.5.

limit(*limit*)

Apply a `LIMIT` to the query and return the newly resulting

`Query`.

merge_result(*iterator*, *load=True*)

Merge a result into this `Query` object's Session.

Given an iterator returned by a `Query` of the same structure as this one, return an identical iterator of results, with all mapped instances merged into the session using `Session.merge()`. This is an optimized method which will merge all mapped instances, preserving the structure of the result rows and unmapped columns with less method overhead than that of calling `Session.merge()` explicitly for each value.

The structure of the results is determined based on the column list of this `Query` - if these do not correspond, unchecked errors will occur.

The 'load' argument is the same as that of `Session.merge()`.

For an example of how `merge_result()` is used, see the source code for the example *Dogpile Caching*, where `merge_result()` is used to efficiently restore state from a cache back into a target `Session`.

offset(*offset*)

Apply an `OFFSET` to the query and return the newly resulting `Query`.

one()

Return exactly one result or raise an exception.

Raises `sqlalchemy.orm.exc.NoResultFound` if the query selects no rows. Raises `sqlalchemy.orm.exc.MultipleResultsFound` if multiple object identities are returned, or if multiple rows are returned for a query that returns only scalar values as opposed to full identity-mapped entities.

Calling `one()` results in an execution of the underlying query.

See also:

`Query.first()`

`Query.one_or_none()`

one_or_none()

Return at most one result or raise an exception.

Returns `None` if the query selects no rows. Raises `sqlalchemy.orm.exc.MultipleResultsFound` if multiple object identities are returned, or if multiple rows are returned for a query that returns only scalar values as opposed to full identity-mapped entities.

Calling `Query.one_or_none()` results in an execution of the underlying query.

New in version 1.0.9: Added `Query.one_or_none()`

See also:

`Query.first()`

`Query.one()`

options(**args*)

Return a new Query object, applying the given list of mapper options.

Most supplied options regard changing how column- and relationship-mapped attributes are loaded. See the sections *Deferred Column Loading* and Relationship Loading Techniques for reference documentation.

order_by(**criterion*)

apply one or more ORDER BY criterion to the query and return the newly resulting `Query`

All existing ORDER BY settings can be suppressed by passing `None` - this will suppress any ORDER BY configured on mappers as well.

Alternatively, an existing ORDER BY setting on the Query object can be entirely cancelled by passing `False` as the value - use this before calling methods where an ORDER BY is invalid.

outerjoin(**props*, ***kwargs*)

Create a left outer join against this `Query` object's criterion and apply generatively, returning the newly resulting `Query`.

Usage is the same as the `join()` method.

params(**args*, ***kwargs*)

add values for bind parameters which may have been specified in filter().

parameters may be specified using **kwargs, or optionally a single dictionary as the first positional argument. The reason for both is that **kwargs is convenient, however some parameter dictionaries contain unicode keys in which case **kwargs cannot be used.

populate_existing()

Return a `Query` that will expire and refresh all instances as they are loaded, or reused from the current `Session`.

`populate_existing()` does not improve behavior when the ORM is used normally - the `Session` object's usual behavior of maintaining a transaction and expiring all attributes after rollback or commit handles object state automatically. This method is not intended for general use.

prefix_with(**prefixes*)

Apply the prefixes to the query and return the newly resulting `Query`.

Parameters ***prefixes** – optional prefixes, typically strings, not using any commas. In particular is useful for MySQL keywords.

e.g.:

```
query = sess.query(User.name).\
    prefix_with('HIGH_PRIORITY').\
    prefix_with('SQL_SMALL_RESULT', 'ALL')
```

Would render:

```
SELECT HIGH_PRIORITY SQL_SMALL_RESULT ALL users.name AS users_name
FROM users
```

New in version 0.7.7.

See also:

HasPrefixes.prefix_with()

reset_joinpoint()

Return a new *Query*, where the "join point" has been reset back to the base FROM entities of the query.

This method is usually used in conjunction with the `aliased=True` feature of the *join()* method. See the example in *join()* for how this is used.

scalar()

Return the first element of the first result or None if no rows present. If multiple rows are returned, raises MultipleResultsFound.

```
>>> session.query(Item).scalar()
<Item>
>>> session.query(Item.id).scalar()
1
>>> session.query(Item.id).filter(Item.id < 0).scalar()
None
>>> session.query(Item.id, Item.name).scalar()
1
>>> session.query(func.count(Parent.id)).scalar()
20
```

This results in an execution of the underlying query.

select_entity_from(*from_obj*)

Set the FROM clause of this *Query* to a core selectable, applying it as a replacement FROM clause for corresponding mapped entities.

This method is similar to the *Query.select_from()* method, in that it sets the FROM clause of the query. However, where *Query.select_from()* only affects what is placed in the FROM, this method also applies the given selectable to replace the FROM which the selected entities would normally select from.

The given `from_obj` must be an instance of a *FromClause*, e.g. a *select()* or *Alias* construct.

An example would be a *Query* that selects `User` entities, but uses *Query.select_entity_from()* to have the entities selected from a *select()* construct instead of the base `user` table:

```
select_stmt = select([User]).where(User.id == 7)

q = session.query(User).\
        select_entity_from(select_stmt).\
        filter(User.name == 'ed')
```

The query generated will select `User` entities directly from the given *select()* construct, and will be:

```
SELECT anon_1.id AS anon_1_id, anon_1.name AS anon_1_name
FROM (SELECT "user".id AS id, "user".name AS name
FROM "user"
WHERE "user".id = :id_1) AS anon_1
WHERE anon_1.name = :name_1
```

Notice above that even the WHERE criterion was "adapted" such that the `anon_1` subquery effectively replaces all references to the `user` table, except for the one that it refers to internally.

Compare this to *Query.select_from()*, which as of version 0.9, does not affect existing entities. The statement below:

```
q = session.query(User).\
        select_from(select_stmt).\
        filter(User.name == 'ed')
```

Produces SQL where both the `user` table as well as the `select_stmt` construct are present as separate elements in the FROM clause. No "adaptation" of the `user` table is applied:

```
SELECT "user".id AS user_id, "user".name AS user_name
FROM "user", (SELECT "user".id AS id, "user".name AS name
FROM "user"
WHERE "user".id = :id_1) AS anon_1
WHERE "user".name = :name_1
```

Query.select_entity_from() maintains an older behavior of *Query.select_from()*. In modern usage, similar results can also be achieved using *aliased()*:

```
select_stmt = select([User]).where(User.id == 7)
user_from_select = aliased(User, select_stmt.alias())

q = session.query(user_from_select)
```

Parameters **from_obj** – a *FromClause* object that will replace the FROM clause of this *Query*.

See also:

Query.select_from()

New in version 0.8: *Query.select_entity_from()* was added to specify the specific behavior of entity replacement, however the *Query.select_from()* maintains this behavior as well until 0.9.

select_from(**from_obj*)

Set the FROM clause of this *Query* explicitly.

Query.select_from() is often used in conjunction with *Query.join()* in order to control which entity is selected from on the "left" side of the join.

The entity or selectable object here effectively replaces the "left edge" of any calls to *join()*, when no joinpoint is otherwise established - usually, the default "join point" is the leftmost entity in the *Query* object's list of entities to be selected.

A typical example:

```
q = session.query(Address).select_from(User).\
    join(User.addresses).\
    filter(User.name == 'ed')
```

Which produces SQL equivalent to:

```
SELECT address.* FROM user
JOIN address ON user.id=address.user_id
WHERE user.name = :name_1
```

Parameters ***from_obj** – collection of one or more entities to apply to the FROM clause. Entities can be mapped classes, *AliasedClass* objects, *Mapper* objects as well as core *FromClause* elements like subqueries.

Changed in version 0.9: This method no longer applies the given FROM object to be the selectable from which matching entities select from; the *select_entity_from()* method now accomplishes this. See that method for a description of this behavior.

See also:

join()

Query.select_entity_from()

selectable

Return the *Select* object emitted by this *Query*.

Used for *inspect()* compatibility, this is equivalent to:

```
query.enable_eagerloads(False).with_labels().statement
```

slice(*start*, *stop*)

apply LIMIT/OFFSET to the `Query` based on a " "range and return the newly resulting `Query`.

statement

The full SELECT statement represented by this Query.

The statement by default will not have disambiguating labels applied to the construct unless with_labels(True) is called first.

subquery(*name=None*, *with_labels=False*, *reduce_columns=False*)

return the full SELECT statement represented by this *Query*, embedded within an *Alias*.

Eager JOIN generation within the query is disabled.

Parameters

- **name** – string name to be assigned as the alias; this is passed through to *FromClause.alias()*. If `None`, a name will be deterministically generated at compile time.
- **with_labels** – if True, *with_labels()* will be called on the *Query* first to apply table-qualified labels to all columns.
- **reduce_columns** – if True, *Select.reduce_columns()* will be called on the resulting *select()* construct, to remove same-named columns where one also refers to the other via foreign key or WHERE clause equivalence.

  Changed in version 0.8: the `with_labels` and `reduce_columns` keyword arguments were added.

suffix_with(**suffixes*)

Apply the suffix to the query and return the newly resulting `Query`.

Parameters ***suffixes** – optional suffixes, typically strings, not using any commas.

New in version 1.0.0.

See also:

Query.prefix_with()

HasSuffixes.suffix_with()

union(**q*)

Produce a UNION of this Query against one or more queries.

e.g.:

```
q1 = sess.query(SomeClass).filter(SomeClass.foo=='bar')
q2 = sess.query(SomeClass).filter(SomeClass.bar=='foo')

q3 = q1.union(q2)
```

The method accepts multiple Query objects so as to control the level of nesting. A series of `union()` calls such as:

```
x.union(y).union(z).all()
```

will nest on each `union()`, and produces:

```
SELECT * FROM (SELECT * FROM (SELECT * FROM X UNION
                SELECT * FROM y) UNION SELECT * FROM Z)
```

Whereas:

```
x.union(y, z).all()
```

produces:

```
SELECT * FROM (SELECT * FROM X UNION SELECT * FROM y UNION
                SELECT * FROM Z)
```

Note that many database backends do not allow ORDER BY to be rendered on a query called within UNION, EXCEPT, etc. To disable all ORDER BY clauses including those configured on mappers, issue `query.order_by(None)` - the resulting *Query* object will not render ORDER BY within its SELECT statement.

union_all(**q*)

Produce a UNION ALL of this Query against one or more queries.

Works the same way as *union()*. See that method for usage examples.

update(*values*, *synchronize_session='evaluate'*, *update_args=None*)

Perform a bulk update query.

Updates rows matched by this query in the database.

E.g.:

```
sess.query(User).filter(User.age == 25).\
    update({User.age: User.age - 10}, synchronize_session=False)

sess.query(User).filter(User.age == 25).\
    update({"age": User.age - 10}, synchronize_session='evaluate')
```

Warning: The *Query.update()* method is a "bulk" operation, which bypasses ORM unit-of-work automation in favor of greater performance. **Please read all caveats and warnings below.**

Parameters

- **values** – a dictionary with attributes names, or alternatively mapped attributes or SQL expressions, as keys, and literal values or sql expressions as values. If *parameter-ordered mode* is desired, the values can be passed as a list of 2-tuples; this requires that the *preserve_parameter_order* flag is passed to the *Query.update.update_args* dictionary as well.

  Changed in version 1.0.0: - string names in the values dictionary are now resolved against the mapped entity; previously, these strings were passed as literal column names with no mapper-level translation.

- **synchronize_session** – chooses the strategy to update the attributes on objects in the session. Valid values are:

  `False` - don't synchronize the session. This option is the most efficient and is reliable once the session is expired, which typically occurs after a commit(), or explicitly using expire_all(). Before the expiration, updated objects may still remain in the session with stale values on their attributes, which can lead to confusing results.

`'fetch'` - performs a select query before the update to find objects that are matched by the update query. The updated attributes are expired on matched objects.

`'evaluate'` - Evaluate the Query's criteria in Python straight on the objects in the session. If evaluation of the criteria isn't implemented, an exception is raised.

The expression evaluator currently doesn't account for differing string collations between the database and Python.

- **update_args** – Optional dictionary, if present will be passed to the underlying *update()* construct as the `**kw` for the object. May be used to pass dialect-specific arguments such as `mysql_limit`, as well as other special arguments such as *preserve_parameter_order*.

  New in version 1.0.0.

Returns the count of rows matched as returned by the database's "row count" feature.

Warning: Additional Caveats for bulk query updates

- The method does **not** offer in-Python cascading of relationships - it is assumed that ON UPDATE CASCADE is configured for any foreign key references which require it, otherwise the database may emit an integrity violation if foreign key references are being enforced.
  After the UPDATE, dependent objects in the *Session* which were impacted by an ON UPDATE CASCADE may not contain the current state; this issue is resolved once the *Session* is expired, which normally occurs upon *Session.commit()* or can be forced by using *Session.expire_all()*.
- The `'fetch'` strategy results in an additional SELECT statement emitted and will significantly reduce performance.
- The `'evaluate'` strategy performs a scan of all matching objects within the *Session*; if the contents of the *Session* are expired, such as via a proceeding *Session.commit()* call, **this will result in SELECT queries emitted for every matching object**.
- The method supports multiple table updates, as detailed in *Multiple Table Updates*, and this behavior does extend to support updates of joined-inheritance and other multiple table mappings. However, the **join condition of an inheritance mapper is not automatically rendered**. Care must be taken in any multiple-table update to explicitly include the joining condition between those tables, even in mappings where this is normally automatic. E.g. if a class `Engineer` subclasses `Employee`, an UPDATE of the `Engineer` local table using criteria against the `Employee` local table might look like:

```
session.query(Engineer).\
    filter(Engineer.id == Employee.id).\
    filter(Employee.name == 'dilbert').\
    update({"engineer_type": "programmer"})
```

- The *MapperEvents.before_update()* and *MapperEvents.after_update()* events **are not invoked from this method**. Instead, the *SessionEvents.after_bulk_update()* method is provided to act upon a mass UPDATE of entity rows.

See also:

Query.delete()

Inserts, Updates and Deletes - Core SQL tutorial

value(*column*)

Return a scalar result corresponding to the given column expression.

values(**columns*)

Return an iterator yielding result tuples corresponding to the given list of columns

whereclause

A readonly attribute which returns the current WHERE criterion for this Query.

This returned value is a SQL expression construct, or `None` if no criterion has been established.

with_entities(**entities*)

Return a new *Query* replacing the SELECT list with the given entities.

e.g.:

```
# Users, filtered on some arbitrary criterion
# and then ordered by related email address
q = session.query(User).\
            join(User.address).\
            filter(User.name.like('%ed%')).\
            order_by(Address.email)

# given *only* User.id==5, Address.email, and 'q', what
# would the *next* User in the result be ?
subq = q.with_entities(Address.email).\
            order_by(None).\
            filter(User.id==5).\
            subquery()
q = q.join((subq, subq.c.email < Address.email)).\
            limit(1)
```

New in version 0.6.5.

with_for_update(*read=False*, *nowait=False*, *of=None*)

return a new *Query* with the specified options for the `FOR UPDATE` clause.

The behavior of this method is identical to that of `SelectBase.with_for_update()`. When called with no arguments, the resulting `SELECT` statement will have a `FOR UPDATE` clause appended. When additional arguments are specified, backend-specific options such as `FOR UPDATE NOWAIT` or `LOCK IN SHARE MODE` can take effect.

E.g.:

```
q = sess.query(User).with_for_update(nowait=True, of=User)
```

The above query on a Postgresql backend will render like:

```
SELECT users.id AS users_id FROM users FOR UPDATE OF users NOWAIT
```

New in version 0.9.0: *Query.with_for_update()* supersedes the *Query.with_lockmode()* method.

See also:

GenerativeSelect.with_for_update() - Core level method with full argument and behavioral description.

with_hint(*selectable*, *text*, *dialect_name='*'*)

Add an indexing or other executional context hint for the given entity or selectable to this *Query*.

Functionality is passed straight through to *with_hint()*, with the addition that `selectable` can be a *Table*, *Alias*, or ORM entity / mapped class /etc.

See also:

Query.with_statement_hint()

with_labels()

Apply column labels to the return value of Query.statement.

Indicates that this Query's *statement* accessor should return a SELECT statement that applies labels to all columns in the form <tablename>_<columnname>; this is commonly used to disambiguate columns from multiple tables which have the same name.

When the *Query* actually issues SQL to load rows, it always uses column labeling.

Note: The `Query.with_labels()` method *only* applies the output of `Query.statement`, and *not* to any of the result-row invoking systems of `Query` itself, e.g. `Query.first()`, `Query.all()`, etc. To execute a query using `Query.with_labels()`, invoke the `Query.statement` using `Session.execute()`:

```
result = session.execute(query.with_labels().statement)
```

with_lockmode(*mode*)

Return a new `Query` object with the specified "locking mode", which essentially refers to the `FOR UPDATE` clause.

Deprecated since version 0.9.0: superseded by `Query.with_for_update()`.

Parameters **mode** – a string representing the desired locking mode. Valid values are:

- `None` - translates to no lockmode
- `'update'` - translates to `FOR UPDATE` (standard SQL, supported by most dialects)
- `'update_nowait'` - translates to `FOR UPDATE NOWAIT` (supported by Oracle, PostgreSQL 8.1 upwards)
- `'read'` - translates to `LOCK IN SHARE MODE` (for MySQL), and `FOR SHARE` (for PostgreSQL)

See also:

`Query.with_for_update()` - improved API for specifying the `FOR UPDATE` clause.

with_parent(*instance*, *property=None*)

Add filtering criterion that relates the given instance to a child object or collection, using its attribute state as well as an established `relationship()` configuration.

The method uses the `with_parent()` function to generate the clause, the result of which is passed to `Query.filter()`.

Parameters are the same as `with_parent()`, with the exception that the given property can be None, in which case a search is performed against this `Query` object's target mapper.

with_polymorphic(*cls_or_mappers*, *selectable=None*, *polymorphic_on=None*)

Load columns for inheriting classes.

`Query.with_polymorphic()` applies transformations to the "main" mapped class represented by this `Query`. The "main" mapped class here means the `Query` object's first argument is a full class, i.e. `session.query(SomeClass)`. These transformations allow additional tables to be present in the FROM clause so that columns for a joined-inheritance subclass are available in the query, both for the purposes of load-time efficiency as well as the ability to use these columns at query time.

See the documentation section *Basic Control of Which Tables are Queried* for details on how this method is used.

Changed in version 0.8: A new and more flexible function `orm.with_polymorphic()` supersedes `Query.with_polymorphic()`, as it can apply the equivalent functionality to any set of columns or classes in the `Query`, not just the "zero mapper". See that function for a description of arguments.

with_session(*session*)

Return a `Query` that will use the given `Session`.

with_statement_hint(*text*, *dialect_name='*')

add a statement hint to this `Select`.

This method is similar to `Select.with_hint()` except that it does not require an individual table, and instead applies to the statement as a whole.

This feature calls down into `Select.with_statement_hint()`.

New in version 1.0.0.

See also:

`Query.with_hint()`

with_transformation(*fn*)

Return a new `Query` object transformed by the given function.

E.g.:

```
def filter_something(criterion):
    def transform(q):
        return q.filter(criterion)
    return transform

q = q.with_transformation(filter_something(x==5))
```

This allows ad-hoc recipes to be created for `Query` objects. See the example at *Building Transformers*.

New in version 0.7.4.

yield_per(*count*)

Yield only `count` rows at a time.

The purpose of this method is when fetching very large result sets (> 10K rows), to batch results in sub-collections and yield them out partially, so that the Python interpreter doesn't need to declare very large areas of memory which is both time consuming and leads to excessive memory use. The performance from fetching hundreds of thousands of rows can often double when a suitable yield-per setting (e.g. approximately 1000) is used, even with DBAPIs that buffer rows (which are most).

The `Query.yield_per()` method **is not compatible with most eager loading schemes, including subqueryload and joinedload with collections**. For this reason, it may be helpful to disable eager loads, either unconditionally with `Query.enable_eagerloads()`:

```
q = sess.query(Object).yield_per(100).enable_eagerloads(False)
```

Or more selectively using `lazyload()`; such as with an asterisk to specify the default loader scheme:

```
q = sess.query(Object).yield_per(100).\
    options(lazyload('*'), joinedload(Object.some_related))
```

> **Warning:** Use this method with caution; if the same instance is present in more than one batch of rows, end-user changes to attributes will be overwritten.
> In particular, it's usually impossible to use this setting with eagerly loaded collections (i.e. any lazy='joined' or 'subquery') since those collections will be cleared for a new load when encountered in a subsequent result batch. In the case of 'subquery' loading, the full result for all rows is fetched which generally defeats the purpose of `yield_per()`.
> Also note that while `yield_per()` will set the `stream_results` execution option to True, currently this is only understood by `psycopg2` dialect which will stream results using server side cursors instead of pre-buffer all rows for this query. Other DBAPIs **pre-buffer all rows** before making them available. The memory use of raw database rows is much less than that of an ORM-mapped object, but should still be taken into consideration when benchmarking.

See also:

`Query.enable_eagerloads()`

ORM-Specific Query Constructs

`sqlalchemy.orm.`**`aliased`**(*element*, *alias=None*, *name=None*, *flat=False*, *adapt_on_names=False*)

Produce an alias of the given element, usually an `AliasedClass` instance.

E.g.:

```
my_alias = aliased(MyClass)

session.query(MyClass, my_alias).filter(MyClass.id > my_alias.id)
```

The `aliased()` function is used to create an ad-hoc mapping of a mapped class to a new selectable. By default, a selectable is generated from the normally mapped selectable (typically a `Table`) using the `FromClause.alias()` method. However, `aliased()` can also be used to link the class to a new `select()` statement. Also, the `with_polymorphic()` function is a variant of `aliased()` that is intended to specify a so-called "polymorphic selectable", that corresponds to the union of several joined-inheritance subclasses at once.

For convenience, the `aliased()` function also accepts plain `FromClause` constructs, such as a `Table` or `select()` construct. In those cases, the `FromClause.alias()` method is called on the object and the new `Alias` object returned. The returned `Alias` is not ORM-mapped in this case.

Parameters

- **`element`** – element to be aliased. Is normally a mapped class, but for convenience can also be a `FromClause` element.
- **`alias`** – Optional selectable unit to map the element to. This should normally be a `Alias` object corresponding to the `Table` to which the class is mapped, or to a `select()` construct that is compatible with the mapping. By default, a simple anonymous alias of the mapped table is generated.
- **`name`** – optional string name to use for the alias, if not specified by the `alias` parameter. The name, among other things, forms the attribute name that will be accessible via tuples returned by a `Query` object.
- **`flat`** – Boolean, will be passed through to the `FromClause.alias()` call so that aliases of `Join` objects don't include an enclosing SELECT. This can lead to more efficient queries in many circumstances. A JOIN against a nested JOIN will be rewritten as a JOIN against an aliased SELECT subquery on backends that don't support this syntax.

  New in version 0.9.0.

See also:

Join.alias()

- **adapt_on_names** – if True, more liberal "matching" will be used when mapping the mapped columns of the ORM entity to those of the given selectable - a name-based match will be performed if the given selectable doesn't otherwise have a column that corresponds to one on the entity. The use case for this is when associating an entity with some derived selectable such as one that uses aggregate functions:

```
class UnitPrice(Base):
    __tablename__ = 'unit_price'
    ...
    unit_id = Column(Integer)
    price = Column(Numeric)

aggregated_unit_price = Session.query(
                            func.sum(UnitPrice.price).label('price')
                        ).group_by(UnitPrice.unit_id).subquery()

aggregated_unit_price = aliased(UnitPrice,
            alias=aggregated_unit_price, adapt_on_names=True)
```

Above, functions on `aggregated_unit_price` which refer to `.price` will return the `fund.sum(UnitPrice.price).label('price')` column, as it is matched on the name "price". Ordinarily, the "price" function wouldn't have any "column correspondence" to the actual `UnitPrice.price` column as it is not a proxy of the original.

New in version 0.7.3.

class `sqlalchemy.orm.util.`**`AliasedClass`**(*cls*, *alias=None*, *name=None*, *flat=False*, *adapt_on_names=False*, *with_polymorphic_mappers=()*, *with_polymorphic_discriminator=None*, *base_alias=None*, *use_mapper_path=False*)

Represents an "aliased" form of a mapped class for usage with Query.

The ORM equivalent of a *sqlalchemy.sql.expression.alias()* construct, this object mimics the mapped class using a __getattr__ scheme and maintains a reference to a real *Alias* object.

Usage is via the *orm.aliased()* function, or alternatively via the *orm.with_polymorphic()* function.

Usage example:

```
# find all pairs of users with the same name
user_alias = aliased(User)
session.query(User, user_alias).\
                join((user_alias, User.id > user_alias.id)).\
                filter(User.name==user_alias.name)
```

The resulting object is an instance of *AliasedClass*. This object implements an attribute scheme which produces the same attribute and method interface as the original mapped class, allowing *AliasedClass* to be compatible with any attribute technique which works on the original class, including hybrid attributes (see *Hybrid Attributes*).

The *AliasedClass* can be inspected for its underlying *Mapper*, aliased selectable, and other information using *inspect()*:

```
from sqlalchemy import inspect
my_alias = aliased(MyClass)
insp = inspect(my_alias)
```

The resulting inspection object is an instance of `AliasedInsp`.

See `aliased()` and `with_polymorphic()` for construction argument descriptions.

class `sqlalchemy.orm.util.`**`AliasedInsp`**(*entity*, *mapper*, *selectable*, *name*, *with_polymorphic_mappers*, *polymorphic_on*, *_base_alias*, *_use_mapper_path*, *adapt_on_names*)

Bases: `sqlalchemy.orm.base.InspectionAttr`

Provide an inspection interface for an `AliasedClass` object.

The `AliasedInsp` object is returned given an `AliasedClass` using the `inspect()` function:

```
from sqlalchemy import inspect
from sqlalchemy.orm import aliased

my_alias = aliased(MyMappedClass)
insp = inspect(my_alias)
```

Attributes on `AliasedInsp` include:

- `entity` - the `AliasedClass` represented.
- `mapper` - the `Mapper` mapping the underlying class.
- `selectable` - the `Alias` construct which ultimately represents an aliased `Table` or `Select` construct.
- `name` - the name of the alias. Also is used as the attribute name when returned in a result tuple from `Query`.
- `with_polymorphic_mappers` - collection of `Mapper` objects indicating all those mappers expressed in the select construct for the `AliasedClass`.
- `polymorphic_on` - an alternate column or SQL expression which will be used as the "discriminator" for a polymorphic load.

See also:

Runtime Inspection API

class `sqlalchemy.orm.query.`**`Bundle`**(*name*, **exprs*, ***kw*)

Bases: `sqlalchemy.orm.base.InspectionAttr`

A grouping of SQL expressions that are returned by a `Query` under one namespace.

The `Bundle` essentially allows nesting of the tuple-based results returned by a column-oriented `Query` object. It also is extensible via simple subclassing, where the primary capability to override is that of how the set of expressions should be returned, allowing post-processing as well as custom return types, without involving ORM identity-mapped classes.

New in version 0.9.0.

See also:

Column Bundles

`__init__`(*name*, **exprs*, ***kw*)

Construct a new `Bundle`.

e.g.:

```
bn = Bundle("mybundle", MyClass.x, MyClass.y)

for row in session.query(bn).filter(
        bn.c.x == 5).filter(bn.c.y == 4):
    print(row.mybundle.x, row.mybundle.y)
```

Parameters

- **name** – name of the bundle.
- ***exprs** – columns or SQL expressions comprising the bundle.
- **single_entity=False** – if True, rows for this `Bundle` can be returned as a "single entity" outside of any enclosing tuple in the same manner as a mapped entity.

c = None

An alias for `Bundle.columns`.

columns = None

A namespace of SQL expressions referred to by this `Bundle`.

e.g.:

```
bn = Bundle("mybundle", MyClass.x, MyClass.y)

q = sess.query(bn).filter(bn.c.x == 5)
```

Nesting of bundles is also supported:

```
b1 = Bundle("b1",
        Bundle('b2', MyClass.a, MyClass.b),
        Bundle('b3', MyClass.x, MyClass.y)
    )

q = sess.query(b1).filter(
    b1.c.b2.c.a == 5).filter(b1.c.b3.c.y == 9)
```

See also:

`Bundle.c`

create_row_processor(*query*, *procs*, *labels*)

Produce the "row processing" function for this `Bundle`.

May be overridden by subclasses.

See also:

Column Bundles - includes an example of subclassing.

label(*name*)

Provide a copy of this `Bundle` passing a new label.

single_entity = False

If True, queries for a single Bundle will be returned as a single entity, rather than an element within a keyed tuple.

class sqlalchemy.util.**KeyedTuple**

Bases: `sqlalchemy.util._collections.AbstractKeyedTuple`

`tuple` subclass that adds labeled names.

E.g.:

```
>>> k = KeyedTuple([1, 2, 3], labels=["one", "two", "three"])
>>> k.one
1
>>> k.two
2
```

Result rows returned by *Query* that contain multiple ORM entities and/or column expressions make use of this class to return rows.

The *KeyedTuple* exhibits similar behavior to the `collections.namedtuple()` construct provided in the Python standard library, however is architected very differently. Unlike `collections.namedtuple()`, *KeyedTuple* is does not rely on creation of custom subtypes in order to represent a new series of keys, instead each *KeyedTuple* instance receives its list of keys in place. The subtype approach of `collections.namedtuple()` introduces significant complexity and performance overhead, which is not necessary for the *Query* object's use case.

Changed in version 0.8: Compatibility methods with `collections.namedtuple()` have been added including *KeyedTuple._fields* and *KeyedTuple._asdict()*.

See also:

Querying

_asdict()

Return the contents of this *KeyedTuple* as a dictionary.

This method provides compatibility with `collections.namedtuple()`, with the exception that the dictionary returned is **not** ordered.

New in version 0.8.

_fields

Return a tuple of string key names for this *KeyedTuple*.

This method provides compatibility with `collections.namedtuple()`.

New in version 0.8.

See also:

KeyedTuple.keys()

keys()

Return a list of string key names for this *KeyedTuple*.

See also:

KeyedTuple._fields

class sqlalchemy.orm.strategy_options.**Load**(*entity*)

Bases: sqlalchemy.sql.expression.Generative, sqlalchemy.orm.interfaces.MapperOption

Represents loader options which modify the state of a *Query* in order to affect how various mapped attributes are loaded.

New in version 0.9.0: The `Load()` system is a new foundation for the existing system of loader options, including options such as *orm.joinedload()*, *orm.defer()*, and others. In particular, it introduces a new method-chained system that replaces the need for dot-separated paths as well as "_all()" options such as *orm.joinedload_all()*.

A *Load* object can be used directly or indirectly. To use one directly, instantiate given the parent class. This style of usage is useful when dealing with a *Query* that has multiple entities, or when producing a loader option that can be applied generically to any style of query:

```
myopt = Load(MyClass).joinedload("widgets")
```

The above `myopt` can now be used with *Query.options()*:

```
session.query(MyClass).options(myopt)
```

The *Load* construct is invoked indirectly whenever one makes use of the various loader options that are present in `sqlalchemy.orm`, including options such as *orm.joinedload()*, *orm.defer()*, *orm.subqueryload()*, and all the rest. These constructs produce an "anonymous" form of the *Load* object which tracks attributes and options, but is not linked to a parent class until it is associated with a parent *Query*:

```
# produce "unbound" Load object
myopt = joinedload("widgets")

# when applied using options(), the option is "bound" to the
# class observed in the given query, e.g. MyClass
session.query(MyClass).options(myopt)
```

Whether the direct or indirect style is used, the *Load* object returned now represents a specific "path" along the entities of a *Query*. This path can be traversed using a standard method-chaining approach. Supposing a class hierarchy such as `User`, `User.addresses -> Address`, `User.orders -> Order` and `Order.items -> Item`, we can specify a variety of loader options along each element in the "path":

```
session.query(User).options(
            joinedload("addresses"),
            subqueryload("orders").joinedload("items")
        )
```

Where above, the `addresses` collection will be joined-loaded, the `orders` collection will be subquery-loaded, and within that subquery load the `items` collection will be joined-loaded.

baked_lazyload(*loadopt*, *attr*)

Produce a new *Load* object with the `orm.baked_lazyload()` option applied.

See `orm.baked_lazyload()` for usage examples.

contains_eager(*loadopt*, *attr*, *alias=None*)

Produce a new *Load* object with the *orm.contains_eager()* option applied.

See *orm.contains_eager()* for usage examples.

defaultload(*loadopt*, *attr*)

Produce a new *Load* object with the *orm.defaultload()* option applied.

See *orm.defaultload()* for usage examples.

defer(*loadopt*, *key*)

Produce a new *Load* object with the *orm.defer()* option applied.

See *orm.defer()* for usage examples.

immediateload(*loadopt*, *attr*)

Produce a new *Load* object with the *orm.immediateload()* option applied.

See *orm.immediateload()* for usage examples.

joinedload(*loadopt*, *attr*, *innerjoin=None*)

Produce a new *Load* object with the *orm.joinedload()* option applied.

See `orm.joinedload()` for usage examples.

lazyload(*loadopt*, *attr*)
Produce a new `Load` object with the `orm.lazyload()` option applied.

See `orm.lazyload()` for usage examples.

load_only(*loadopt*, **attrs*)
Produce a new `Load` object with the `orm.load_only()` option applied.

See `orm.load_only()` for usage examples.

noload(*loadopt*, *attr*)
Produce a new `Load` object with the `orm.noload()` option applied.

See `orm.noload()` for usage examples.

subqueryload(*loadopt*, *attr*)
Produce a new `Load` object with the `orm.subqueryload()` option applied.

See `orm.subqueryload()` for usage examples.

undefer(*loadopt*, *key*)
Produce a new `Load` object with the `orm.undefer()` option applied.

See `orm.undefer()` for usage examples.

undefer_group(*loadopt*, *name*)
Produce a new `Load` object with the `orm.undefer_group()` option applied.

See `orm.undefer_group()` for usage examples.

sqlalchemy.orm.**join**(*left*, *right*, *onclause=None*, *isouter=False*, *join_to_left=None*)
Produce an inner join between left and right clauses.

`orm.join()` is an extension to the core join interface provided by `sql.expression.join()`, where the left and right selectables may be not only core selectable objects such as `Table`, but also mapped classes or `AliasedClass` instances. The "on" clause can be a SQL expression, or an attribute or string name referencing a configured `relationship()`.

`orm.join()` is not commonly needed in modern usage, as its functionality is encapsulated within that of the `Query.join()` method, which features a significant amount of automation beyond `orm.join()` by itself. Explicit usage of `orm.join()` with `Query` involves usage of the `Query.select_from()` method, as in:

```
from sqlalchemy.orm import join
session.query(User).\
    select_from(join(User, Address, User.addresses)).\
    filter(Address.email_address=='foo@bar.com')
```

In modern SQLAlchemy the above join can be written more succinctly as:

```
session.query(User).\
        join(User.addresses).\
        filter(Address.email_address=='foo@bar.com')
```

See `Query.join()` for information on modern usage of ORM level joins.

Changed in version 0.8.1: - the `join_to_left` parameter is no longer used, and is deprecated.

sqlalchemy.orm.**outerjoin**(*left*, *right*, *onclause=None*, *join_to_left=None*)
Produce a left outer join between left and right clauses.

This is the "outer join" version of the `orm.join()` function, featuring the same behavior except that an OUTER JOIN is generated. See that function's documentation for other usage details.

`sqlalchemy.orm.with_parent`(*instance*, *prop*)

Create filtering criterion that relates this query's primary entity to the given related instance, using established `relationship()` configuration.

The SQL rendered is the same as that rendered when a lazy loader would fire off from the given parent on that attribute, meaning that the appropriate state is taken from the parent object in Python without the need to render joins to the parent table in the rendered statement.

Changed in version 0.6.4: This method accepts parent instances in all persistence states, including transient, persistent, and detached. Only the requisite primary key/foreign key attributes need to be populated. Previous versions didn't work with transient instances.

Parameters

- **instance** – An instance which has some `relationship()`.
- **property** – String property name, or class-bound attribute, which indicates what relationship from the instance should be used to reconcile the parent/child relationship.

2.5 Using the Session

The `orm.mapper()` function and `declarative` extensions are the primary configurational interface for the ORM. Once mappings are configured, the primary usage interface for persistence operations is the `Session`.

2.5.1 Session Basics

What does the Session do ?

In the most general sense, the `Session` establishes all conversations with the database and represents a "holding zone" for all the objects which you've loaded or associated with it during its lifespan. It provides the entrypoint to acquire a `Query` object, which sends queries to the database using the `Session` object's current database connection, populating result rows into objects that are then stored in the `Session`, inside a structure called the Identity Map - a data structure that maintains unique copies of each object, where "unique" means "only one object with a particular primary key".

The `Session` begins in an essentially stateless form. Once queries are issued or other objects are persisted with it, it requests a connection resource from an `Engine` that is associated either with the `Session` itself or with the mapped `Table` objects being operated upon. This connection represents an ongoing transaction, which remains in effect until the `Session` is instructed to commit or roll back its pending state.

All changes to objects maintained by a `Session` are tracked - before the database is queried again or before the current transaction is committed, it **flushes** all pending changes to the database. This is known as the Unit of Work pattern.

When using a `Session`, it's important to note that the objects which are associated with it are **proxy objects** to the transaction being held by the `Session` - there are a variety of events that will cause objects to re-access the database in order to keep synchronized. It is possible to "detach" objects from a `Session`, and to continue using them, though this practice has its caveats. It's intended that usually, you'd re-associate detached objects with another `Session` when you want to work with them again, so that they can resume their normal task of representing database state.

Getting a Session

`Session` is a regular Python class which can be directly instantiated. However, to standardize how sessions are configured and acquired, the `sessionmaker` class is normally used to create a top level `Session` configuration which can then be used throughout an application without the need to repeat the configurational arguments.

The usage of *sessionmaker* is illustrated below:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

# an Engine, which the Session will use for connection
# resources
some_engine = create_engine('postgresql://scott:tiger@localhost/')

# create a configured "Session" class
Session = sessionmaker(bind=some_engine)

# create a Session
session = Session()

# work with sess
myobject = MyObject('foo', 'bar')
session.add(myobject)
session.commit()
```

Above, the *sessionmaker* call creates a factory for us, which we assign to the name `Session`. This factory, when called, will create a new *Session* object using the configurational arguments we've given the factory. In this case, as is typical, we've configured the factory to specify a particular *Engine* for connection resources.

A typical setup will associate the *sessionmaker* with an *Engine*, so that each *Session* generated will use this *Engine* to acquire connection resources. This association can be set up as in the example above, using the `bind` argument.

When you write your application, place the *sessionmaker* factory at the global level. This factory can then be used by the rest of the applcation as the source of new *Session* instances, keeping the configuration for how *Session* objects are constructed in one place.

The *sessionmaker* factory can also be used in conjunction with other helpers, which are passed a user-defined *sessionmaker* that is then maintained by the helper. Some of these helpers are discussed in the section *When do I construct a Session, when do I commit it, and when do I close it?*.

Adding Additional Configuration to an Existing sessionmaker()

A common scenario is where the *sessionmaker* is invoked at module import time, however the generation of one or more *Engine* instances to be associated with the *sessionmaker* has not yet proceeded. For this use case, the *sessionmaker* construct offers the *sessionmaker.configure()* method, which will place additional configuration directives into an existing *sessionmaker* that will take place when the construct is invoked:

```
from sqlalchemy.orm import sessionmaker
from sqlalchemy import create_engine

# configure Session class with desired options
Session = sessionmaker()

# later, we create the engine
engine = create_engine('postgresql://...')

# associate it with our custom Session class
Session.configure(bind=engine)

# work with the session
session = Session()
```

Creating Ad-Hoc Session Objects with Alternate Arguments

For the use case where an application needs to create a new *Session* with special arguments that deviate from what is normally used throughout the application, such as a *Session* that binds to an alternate source of connectivity, or a *Session* that should have other arguments such as `expire_on_commit` established differently from what most of the application wants, specific arguments can be passed to the *sessionmaker* factory's *sessionmaker.__call__()* method. These arguments will override whatever configurations have already been placed, such as below, where a new *Session* is constructed against a specific *Connection*:

```
# at the module level, the global sessionmaker,
# bound to a specific Engine
Session = sessionmaker(bind=engine)

# later, some unit of code wants to create a
# Session that is bound to a specific Connection
conn = engine.connect()
session = Session(bind=conn)
```

The typical rationale for the association of a *Session* with a specific *Connection* is that of a test fixture that maintains an external transaction - see *Joining a Session into an External Transaction (such as for test suites)* for an example of this.

Session Frequently Asked Questions

By this point, many users already have questions about sessions. This section presents a mini-FAQ (note that we have also a real FAQ) of the most basic issues one is presented with when using a *Session*.

When do I make a `sessionmaker`?

Just one time, somewhere in your application's global scope. It should be looked upon as part of your application's configuration. If your application has three .py files in a package, you could, for example, place the *sessionmaker* line in your `__init__.py` file; from that point on your other modules say "from mypackage import Session". That way, everyone else just uses *Session()*, and the configuration of that session is controlled by that central point.

If your application starts up, does imports, but does not know what database it's going to be connecting to, you can bind the *Session* at the "class" level to the engine later on, using *sessionmaker.configure()*.

In the examples in this section, we will frequently show the *sessionmaker* being created right above the line where we actually invoke *Session*. But that's just for example's sake! In reality, the *sessionmaker* would be somewhere at the module level. The calls to instantiate *Session* would then be placed at the point in the application where database conversations begin.

When do I construct a `Session`, when do I commit it, and when do I close it?

tl;dr;

1. As a general rule, keep the lifecycle of the session **separate and external** from functions and objects that access and/or manipulate database data. This will greatly help with achieving a predictable and consistent transactional scope.
2. Make sure you have a clear notion of where transactions begin and end, and keep transactions **short**, meaning, they end at the series of a sequence of operations, instead of being held open indefinitely.

A `Session` is typically constructed at the beginning of a logical operation where database access is potentially anticipated.

The `Session`, whenever it is used to talk to the database, begins a database transaction as soon as it starts communicating. Assuming the `autocommit` flag is left at its recommended default of `False`, this transaction remains in progress until the `Session` is rolled back, committed, or closed. The `Session` will begin a new transaction if it is used again, subsequent to the previous transaction ending; from this it follows that the `Session` is capable of having a lifespan across many transactions, though only one at a time. We refer to these two concepts as **transaction scope** and **session scope**.

The implication here is that the SQLAlchemy ORM is encouraging the developer to establish these two scopes in their application, including not only when the scopes begin and end, but also the expanse of those scopes, for example should a single `Session` instance be local to the execution flow within a function or method, should it be a global object used by the entire application, or somewhere in between these two.

The burden placed on the developer to determine this scope is one area where the SQLAlchemy ORM necessarily has a strong opinion about how the database should be used. The unit of work pattern is specifically one of accumulating changes over time and flushing them periodically, keeping in-memory state in sync with what's known to be present in a local transaction. This pattern is only effective when meaningful transaction scopes are in place.

It's usually not very hard to determine the best points at which to begin and end the scope of a `Session`, though the wide variety of application architectures possible can introduce challenging situations.

A common choice is to tear down the `Session` at the same time the transaction ends, meaning the transaction and session scopes are the same. This is a great choice to start out with as it removes the need to consider session scope as separate from transaction scope.

While there's no one-size-fits-all recommendation for how transaction scope should be determined, there are common patterns. Especially if one is writing a web application, the choice is pretty much established.

A web application is the easiest case because such an application is already constructed around a single, consistent scope - this is the **request**, which represents an incoming request from a browser, the processing of that request to formulate a response, and finally the delivery of that response back to the client. Integrating web applications with the `Session` is then the straightforward task of linking the scope of the `Session` to that of the request. The `Session` can be established as the request begins, or using a lazy initialization pattern which establishes one as soon as it is needed. The request then proceeds, with some system in place where application logic can access the current `Session` in a manner associated with how the actual request object is accessed. As the request ends, the `Session` is torn down as well, usually through the usage of event hooks provided by the web framework. The transaction used by the `Session` may also be committed at this point, or alternatively the application may opt for an explicit commit pattern, only committing for those requests where one is warranted, but still always tearing down the `Session` unconditionally at the end.

Some web frameworks include infrastructure to assist in the task of aligning the lifespan of a `Session` with that of a web request. This includes products such as Flask-SQLAlchemy, for usage in conjunction with the Flask web framework, and Zope-SQLAlchemy, typically used with the Pyramid framework. SQLAlchemy recommends that these products be used as available.

In those situations where the integration libraries are not provided or are insufficient, SQLAlchemy includes its own "helper" class known as `scoped_session`. A tutorial on the usage of this object is at *Contextual/Thread-local Sessions*. It provides both a quick way to associate a `Session` with the current thread, as well as patterns to associate `Session` objects with other kinds of scopes.

As mentioned before, for non-web applications there is no one clear pattern, as applications themselves don't have just one pattern of architecture. The best strategy is to attempt to demarcate "operations", points at which a particular thread begins to perform a series of operations for some period of time, which can be committed at the end. Some examples:

- A background daemon which spawns off child forks would want to create a `Session` local to each child process, work with that `Session` through the life of the "job" that the fork is handling, then tear it down when

the job is completed.

- For a command-line script, the application would create a single, global `Session` that is established when the program begins to do its work, and commits it right as the program is completing its task.
- For a GUI interface-driven application, the scope of the `Session` may best be within the scope of a user-generated event, such as a button push. Or, the scope may correspond to explicit user interaction, such as the user "opening" a series of records, then "saving" them.

As a general rule, the application should manage the lifecycle of the session *externally* to functions that deal with specific data. This is a fundamental separation of concerns which keeps data-specific operations agnostic of the context in which they access and manipulate that data.

E.g. **don't do this**:

```
### this is the **wrong way to do it** ###

class ThingOne(object):
    def go(self):
        session = Session()
        try:
            session.query(FooBar).update({"x": 5})
            session.commit()
        except:
            session.rollback()
            raise

class ThingTwo(object):
    def go(self):
        session = Session()
        try:
            session.query(Widget).update({"q": 18})
            session.commit()
        except:
            session.rollback()
            raise

def run_my_program():
    ThingOne().go()
    ThingTwo().go()
```

Keep the lifecycle of the session (and usually the transaction) **separate and external**:

```
### this is a **better** (but not the only) way to do it ###

class ThingOne(object):
    def go(self, session):
        session.query(FooBar).update({"x": 5})

class ThingTwo(object):
    def go(self, session):
        session.query(Widget).update({"q": 18})

def run_my_program():
    session = Session()
    try:
        ThingOne().go(session)
        ThingTwo().go(session)

        session.commit()
```

```
    except:
        session.rollback()
        raise
    finally:
        session.close()
```

The advanced developer will try to keep the details of session, transaction and exception management as far as possible from the details of the program doing its work. For example, we can further separate concerns using a context manager:

```
### another way (but again *not the only way*) to do it ###

from contextlib import contextmanager

@contextmanager
def session_scope():
    """Provide a transactional scope around a series of operations."""
    session = Session()
    try:
        yield session
        session.commit()
    except:
        session.rollback()
        raise
    finally:
        session.close()


def run_my_program():
    with session_scope() as session:
        ThingOne().go(session)
        ThingTwo().go(session)
```

Is the Session a cache?

Yeee...no. It's somewhat used as a cache, in that it implements the identity map pattern, and stores objects keyed to their primary key. However, it doesn't do any kind of query caching. This means, if you say `session.query(Foo).filter_by(name='bar')`, even if `Foo(name='bar')` is right there, in the identity map, the session has no idea about that. It has to issue SQL to the database, get the rows back, and then when it sees the primary key in the row, *then* it can look in the local identity map and see that the object is already there. It's only when you say `query.get({some primary key})` that the *`Session`* doesn't have to issue a query.

Additionally, the Session stores object instances using a weak reference by default. This also defeats the purpose of using the Session as a cache.

The *`Session`* is not designed to be a global object from which everyone consults as a "registry" of objects. That's more the job of a **second level cache**. SQLAlchemy provides a pattern for implementing second level caching using dogpile.cache, via the *Dogpile Caching* example.

How can I get the `Session` for a certain object?

Use the *`object_session()`* classmethod available on *`Session`*:

```
session = Session.object_session(someobject)
```

The newer *Runtime Inspection API* system can also be used:

```
from sqlalchemy import inspect
session = inspect(someobject).session
```

Is the session thread-safe?

The `Session` is very much intended to be used in a **non-concurrent** fashion, which usually means in only one thread at a time.

The `Session` should be used in such a way that one instance exists for a single series of operations within a single transaction. One expedient way to get this effect is by associating a `Session` with the current thread (see *Contextual/Thread-local Sessions* for background). Another is to use a pattern where the `Session` is passed between functions and is otherwise not shared with other threads.

The bigger point is that you should not *want* to use the session with multiple concurrent threads. That would be like having everyone at a restaurant all eat from the same plate. The session is a local "workspace" that you use for a specific set of tasks; you don't want to, or need to, share that session with other threads who are doing some other task.

Making sure the `Session` is only used in a single concurrent thread at a time is called a "share nothing" approach to concurrency. But actually, not sharing the `Session` implies a more significant pattern; it means not just the `Session` object itself, but also **all objects that are associated with that Session**, must be kept within the scope of a single concurrent thread. The set of mapped objects associated with a `Session` are essentially proxies for data within database rows accessed over a database connection, and so just like the `Session` itself, the whole set of objects is really just a large-scale proxy for a database connection (or connections). Ultimately, it's mostly the DBAPI connection itself that we're keeping away from concurrent access; but since the `Session` and all the objects associated with it are all proxies for that DBAPI connection, the entire graph is essentially not safe for concurrent access.

If there are in fact multiple threads participating in the same task, then you may consider sharing the session and its objects between those threads; however, in this extremely unusual scenario the application would need to ensure that a proper locking scheme is implemented so that there isn't *concurrent* access to the `Session` or its state. A more common approach to this situation is to maintain a single `Session` per concurrent thread, but to instead *copy* objects from one `Session` to another, often using the `Session.merge()` method to copy the state of an object into a new object local to a different `Session`.

Basics of Using a Session

The most basic `Session` use patterns are presented here.

Querying

The `query()` function takes one or more *entities* and returns a new `Query` object which will issue mapper queries within the context of this Session. An entity is defined as a mapped class, a `Mapper` object, an orm-enabled *descriptor*, or an `AliasedClass` object:

```
# query from a class
session.query(User).filter_by(name='ed').all()

# query with multiple classes, returns tuples
session.query(User, Address).join('addresses').filter_by(name='ed').all()

# query using orm-enabled descriptors
session.query(User.name, User.fullname).all()

# query from a mapper
```

```
user_mapper = class_mapper(User)
session.query(user_mapper)
```

When *Query* returns results, each object instantiated is stored within the identity map. When a row matches an object which is already present, the same object is returned. In the latter case, whether or not the row is populated onto an existing object depends upon whether the attributes of the instance have been *expired* or not. A default-configured *Session* automatically expires all instances along transaction boundaries, so that with a normally isolated transaction, there shouldn't be any issue of instances representing data which is stale with regards to the current transaction.

The *Query* object is introduced in great detail in *Object Relational Tutorial*, and further documented in query_api_toplevel.

Adding New or Existing Items

add() is used to place instances in the session. For *transient* (i.e. brand new) instances, this will have the effect of an INSERT taking place for those instances upon the next flush. For instances which are *persistent* (i.e. were loaded by this session), they are already present and do not need to be added. Instances which are *detached* (i.e. have been removed from a session) may be re-associated with a session using this method:

```
user1 = User(name='user1')
user2 = User(name='user2')
session.add(user1)
session.add(user2)

session.commit()      # write changes to the database
```

To add a list of items to the session at once, use *add_all()*:

```
session.add_all([item1, item2, item3])
```

The *add()* operation **cascades** along the `save-update` cascade. For more details see the section *Cascades*.

Deleting

The *delete()* method places an instance into the Session's list of objects to be marked as deleted:

```
# mark two objects to be deleted
session.delete(obj1)
session.delete(obj2)

# commit (or flush)
session.commit()
```

Deleting from Collections A common confusion that arises regarding *delete()* is when objects which are members of a collection are being deleted. While the collection member is marked for deletion from the database, this does not impact the collection itself in memory until the collection is expired. Below, we illustrate that even after an `Address` object is marked for deletion, it's still present in the collection associated with the parent `User`, even after a flush:

```
>>> address = user.addresses[1]
>>> session.delete(address)
>>> session.flush()
>>> address in user.addresses
True
```

When the above session is committed, all attributes are expired. The next access of `user.addresses` will re-load the collection, revealing the desired state:

```
>>> session.commit()
>>> address in user.addresses
False
```

The usual practice of deleting items within collections is to forego the usage of *delete()* directly, and instead use cascade behavior to automatically invoke the deletion as a result of removing the object from the parent collection. The `delete-orphan` cascade accomplishes this, as illustrated in the example below:

```
mapper(User, users_table, properties={
    'addresses':relationship(Address, cascade="all, delete, delete-orphan")
})
del user.addresses[1]
session.flush()
```

Where above, upon removing the `Address` object from the `User.addresses` collection, the `delete-orphan` cascade has the effect of marking the `Address` object for deletion in the same way as passing it to *delete()*.

See also *Cascades* for detail on cascades.

Deleting based on Filter Criterion The caveat with `Session.delete()` is that you need to have an object handy already in order to delete. The Query includes a *delete()* method which deletes based on filtering criteria:

```
session.query(User).filter(User.id==7).delete()
```

The `Query.delete()` method includes functionality to "expire" objects already in the session which match the criteria. However it does have some caveats, including that "delete" and "delete-orphan" cascades won't be fully expressed for collections which are already loaded. See the API docs for *delete()* for more details.

Flushing

When the *Session* is used with its default configuration, the flush step is nearly always done transparently. Specifically, the flush occurs before any individual *Query* is issued, as well as within the *commit()* call before the transaction is committed. It also occurs before a SAVEPOINT is issued when *begin_nested()* is used.

Regardless of the autoflush setting, a flush can always be forced by issuing *flush()*:

```
session.flush()
```

The "flush-on-Query" aspect of the behavior can be disabled by constructing *sessionmaker* with the flag `autoflush=False`:

```
Session = sessionmaker(autoflush=False)
```

Additionally, autoflush can be temporarily disabled by setting the `autoflush` flag at any time:

```
mysession = Session()
mysession.autoflush = False
```

Some autoflush-disable recipes are available at DisableAutoFlush.

The flush process *always* occurs within a transaction, even if the *Session* has been configured with `autocommit=True`, a setting that disables the session's persistent transactional state. If no transaction is present, *flush()* creates its own transaction and commits it. Any failures during flush will always result in a rollback of whatever transaction is present. If the Session is not in `autocommit=True` mode, an explicit call to *rollback()* is required after a flush fails, even though the underlying transaction will have been rolled back already - this is so that the overall nesting pattern of so-called "subtransactions" is consistently maintained.

Committing

`commit()` is used to commit the current transaction. It always issues `flush()` beforehand to flush any remaining state to the database; this is independent of the "autoflush" setting. If no transaction is present, it raises an error. Note that the default behavior of the `Session` is that a "transaction" is always present; this behavior can be disabled by setting `autocommit=True`. In autocommit mode, a transaction can be initiated by calling the `begin()` method.

Note: The term "transaction" here refers to a transactional construct within the `Session` itself which may be maintaining zero or more actual database (DBAPI) transactions. An individual DBAPI connection begins participation in the "transaction" as it is first used to execute a SQL statement, then remains present until the session-level "transaction" is completed. See *Managing Transactions* for further detail.

Another behavior of `commit()` is that by default it expires the state of all instances present after the commit is complete. This is so that when the instances are next accessed, either through attribute access or by them being present in a `Query` result set, they receive the most recent state. To disable this behavior, configure `sessionmaker` with `expire_on_commit=False`.

Normally, instances loaded into the `Session` are never changed by subsequent queries; the assumption is that the current transaction is isolated so the state most recently loaded is correct as long as the transaction continues. Setting `autocommit=True` works against this model to some degree since the `Session` behaves in exactly the same way with regard to attribute state, except no transaction is present.

Rolling Back

`rollback()` rolls back the current transaction. With a default configured session, the post-rollback state of the session is as follows:

- All transactions are rolled back and all connections returned to the connection pool, unless the Session was bound directly to a Connection, in which case the connection is still maintained (but still rolled back).
- Objects which were initially in the *pending* state when they were added to the `Session` within the lifespan of the transaction are expunged, corresponding to their INSERT statement being rolled back. The state of their attributes remains unchanged.
- Objects which were marked as *deleted* within the lifespan of the transaction are promoted back to the *persistent* state, corresponding to their DELETE statement being rolled back. Note that if those objects were first *pending* within the transaction, that operation takes precedence instead.
- All objects not expunged are fully expired.

With that state understood, the `Session` may safely continue usage after a rollback occurs.

When a `flush()` fails, typically for reasons like primary key, foreign key, or "not nullable" constraint violations, a `rollback()` is issued automatically (it's currently not possible for a flush to continue after a partial failure). However, the flush process always uses its own transactional demarcator called a *subtransaction*, which is described more fully in the docstrings for `Session`. What it means here is that even though the database transaction has been rolled back, the end user must still issue `rollback()` to fully reset the state of the `Session`.

Closing

The `close()` method issues a `expunge_all()`, and releases any transactional/connection resources. When connections are returned to the connection pool, transactional state is rolled back as well.

2.5.2 State Management

Quickie Intro to Object States

It's helpful to know the states which an instance can have within a session:

- **Transient** - an instance that's not in a session, and is not saved to the database; i.e. it has no database identity. The only relationship such an object has to the ORM is that its class has a `mapper()` associated with it.
- **Pending** - when you *add()* a transient instance, it becomes pending. It still wasn't actually flushed to the database yet, but it will be when the next flush occurs.
- **Persistent** - An instance which is present in the session and has a record in the database. You get persistent instances by either flushing so that the pending instances become persistent, or by querying the database for existing instances (or moving persistent instances from other sessions into your local session).
- **Deleted** - An instance which has been deleted within a flush, but the transaction has not yet completed. Objects in this state are essentially in the opposite of "pending" state; when the session's transaction is committed, the object will move to the detached state. Alternatively, when the session's transaction is rolled back, a deleted object moves *back* to the persistent state.

  Changed in version 1.1: The 'deleted' state is a newly added session object state distinct from the 'persistent' state.
- **Detached** - an instance which corresponds, or previously corresponded, to a record in the database, but is not currently in any session. The detached object will contain a database identity marker, however because it is not associated with a session, it is unknown whether or not this database identity actually exists in a target database. Detached objects are safe to use normally, except that they have no ability to load unloaded attributes or attributes that were previously marked as "expired".

For a deeper dive into all possible state transitions, see the section *Object Lifecycle Events* which describes each transition as well as how to programmatically track each one.

Getting the Current State of an Object

The actual state of any mapped object can be viewed at any time using the *inspect()* system:

```
>>> from sqlalchemy import inspect
>>> insp = inspect(my_object)
>>> insp.persistent
True
```

See also:

InstanceState.transient

InstanceState.pending

InstanceState.persistent

InstanceState.deleted

InstanceState.detached

Session Attributes

The *Session* itself acts somewhat like a set-like collection. All items present may be accessed using the iterator interface:

```
for obj in session:
    print obj
```

And presence may be tested for using regular "contains" semantics:

```
if obj in session:
    print "Object is present"
```

The session is also keeping track of all newly created (i.e. pending) objects, all objects which have had changes since they were last loaded or saved (i.e. "dirty"), and everything that's been marked as deleted:

```
# pending objects recently added to the Session
session.new

# persistent objects which currently have changes detected
# (this collection is now created on the fly each time the property is called)
session.dirty

# persistent objects that have been marked as deleted via session.delete(obj)
session.deleted

# dictionary of all persistent objects, keyed on their
# identity key
session.identity_map
```

(Documentation: `Session.new`, `Session.dirty`, `Session.deleted`, `Session.identity_map`).

Session Referencing Behavior

Objects within the session are *weakly referenced*. This means that when they are dereferenced in the outside application, they fall out of scope from within the `Session` as well and are subject to garbage collection by the Python interpreter. The exceptions to this include objects which are pending, objects which are marked as deleted, or persistent objects which have pending changes on them. After a full flush, these collections are all empty, and all objects are again weakly referenced.

To cause objects in the `Session` to remain strongly referenced, usually a simple approach is all that's needed. Examples of externally managed strong-referencing behavior include loading objects into a local dictionary keyed to their primary key, or into lists or sets for the span of time that they need to remain referenced. These collections can be associated with a `Session`, if desired, by placing them into the `Session.info` dictionary.

An event based approach is also feasable. A simple recipe that provides "strong referencing" behavior for all objects as they remain within the persistent state is as follows:

```
from sqlalchemy import event

def strong_reference_session(session):
    @event.listens_for(session, "pending_to_persistent")
    @event.listens_for(session, "deleted_to_persistent")
    @event.listens_for(session, "detached_to_persistent")
    @event.listens_for(session, "loaded_as_persistent")
    def strong_ref_object(sess, instance):
        if 'refs' not in sess.info:
            sess.info['refs'] = refs = set()
        else:
            refs = sess.info['refs']

        refs.add(instance)
```

```
    @event.listens_for(session, "persistent_to_detached")
    @event.listens_for(session, "persistent_to_deleted")
    @event.listens_for(session, "persistent_to_transient")
    def deref_object(sess, instance):
        sess.info['refs'].discard(instance)
```

Above, we intercept the *SessionEvents.pending_to_persistent()*, *SessionEvents.detached_to_persistent()*, *SessionEvents.deleted_to_persistent()* and *SessionEvents.loaded_as_persistent()* event hooks in order to intercept objects as they enter the persistent transition, and the *SessionEvents.persistent_to_detached()* and *SessionEvents.persistent_to_deleted()* hooks to intercept objects as they leave the persistent state.

The above function may be called for any *Session* in order to provide strong-referencing behavior on a per-*Session* basis:

```
from sqlalchemy.orm import Session

my_session = Session()
strong_reference_session(my_session)
```

It may also be called for any *sessionmaker*:

```
from sqlalchemy.orm import sessionmaker

maker = sessionmaker()
strong_reference_session(maker)
```

Merging

merge() transfers state from an outside object into a new or already existing instance within a session. It also reconciles the incoming data against the state of the database, producing a history stream which will be applied towards the next flush, or alternatively can be made to produce a simple "transfer" of state without producing change history or accessing the database. Usage is as follows:

```
merged_object = session.merge(existing_object)
```

When given an instance, it follows these steps:

- It examines the primary key of the instance. If it's present, it attempts to locate that instance in the local identity map. If the `load=True` flag is left at its default, it also checks the database for this primary key if not located locally.
- If the given instance has no primary key, or if no instance can be found with the primary key given, a new instance is created.
- The state of the given instance is then copied onto the located/newly created instance. For attributes which are present on the source instance, the value is transferred to the target instance. For mapped attributes which aren't present on the source, the attribute is expired on the target instance, discarding its existing value.

  If the `load=True` flag is left at its default, this copy process emits events and will load the target object's unloaded collections for each attribute present on the source object, so that the incoming state can be reconciled against what's present in the database. If `load` is passed as `False`, the incoming data is "stamped" directly without producing any history.
- The operation is cascaded to related objects and collections, as indicated by the `merge` cascade (see *Cascades*).
- The new instance is returned.

With *merge()*, the given "source" instance is not modified nor is it associated with the target *Session*, and remains available to be merged with any number of other *Session* objects. *merge()* is useful for taking the state of any kind of object structure without regard for its origins or current session associations and copying its state into a new session. Here's some examples:

- An application which reads an object structure from a file and wishes to save it to the database might parse the file, build up the structure, and then use *merge()* to save it to the database, ensuring that the data within the file is used to formulate the primary key of each element of the structure. Later, when the file has changed, the same process can be re-run, producing a slightly different object structure, which can then be `merged` in again, and the *Session* will automatically update the database to reflect those changes, loading each object from the database by primary key and then updating its state with the new state given.
- An application is storing objects in an in-memory cache, shared by many *Session* objects simultaneously. *merge()* is used each time an object is retrieved from the cache to create a local copy of it in each *Session* which requests it. The cached object remains detached; only its state is moved into copies of itself that are local to individual *Session* objects.

  In the caching use case, it's common to use the `load=False` flag to remove the overhead of reconciling the object's state with the database. There's also a "bulk" version of *merge()* called *merge_result()* that was designed to work with cache-extended *Query* objects - see the section *Dogpile Caching*.
- An application wants to transfer the state of a series of objects into a *Session* maintained by a worker thread or other concurrent system. *merge()* makes a copy of each object to be placed into this new *Session*. At the end of the operation, the parent thread/process maintains the objects it started with, and the thread/worker can proceed with local copies of those objects.

  In the "transfer between threads/processes" use case, the application may want to use the `load=False` flag as well to avoid overhead and redundant SQL queries as the data is transferred.

Merge Tips

merge() is an extremely useful method for many purposes. However, it deals with the intricate border between objects that are transient/detached and those that are persistent, as well as the automated transference of state. The wide variety of scenarios that can present themselves here often require a more careful approach to the state of objects. Common problems with merge usually involve some unexpected state regarding the object being passed to *merge()*.

Lets use the canonical example of the User and Address objects:

```
class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    name = Column(String(50), nullable=False)
    addresses = relationship("Address", backref="user")

class Address(Base):
    __tablename__ = 'address'

    id = Column(Integer, primary_key=True)
    email_address = Column(String(50), nullable=False)
    user_id = Column(Integer, ForeignKey('user.id'), nullable=False)
```

Assume a `User` object with one `Address`, already persistent:

```
>>> u1 = User(name='ed', addresses=[Address(email_address='ed@ed.com')])
>>> session.add(u1)
>>> session.commit()
```

We now create a1, an object outside the session, which we'd like to merge on top of the existing Address:

```
>>> existing_a1 = u1.addresses[0]
>>> a1 = Address(id=existing_a1.id)
```

A surprise would occur if we said this:

```
>>> a1.user = u1
>>> a1 = session.merge(a1)
>>> session.commit()
sqlalchemy.orm.exc.FlushError: New instance <Address at 0x1298f50>
with identity key (<class '__main__.Address'>, (1,)) conflicts with
persistent instance <Address at 0x12a25d0>
```

Why is that ? We weren't careful with our cascades. The assignment of a1.user to a persistent object cascaded to the backref of User.addresses and made our a1 object pending, as though we had added it. Now we have *two* Address objects in the session:

```
>>> a1 = Address()
>>> a1.user = u1
>>> a1 in session
True
>>> existing_a1 in session
True
>>> a1 is existing_a1
False
```

Above, our a1 is already pending in the session. The subsequent *merge()* operation essentially does nothing. Cascade can be configured via the *cascade* option on *relationship()*, although in this case it would mean removing the save-update cascade from the User.addresses relationship - and usually, that behavior is extremely convenient. The solution here would usually be to not assign a1.user to an object already persistent in the target session.

The cascade_backrefs=False option of *relationship()* will also prevent the Address from being added to the session via the a1.user = u1 assignment.

Further detail on cascade operation is at *Cascades*.

Another example of unexpected state:

```
>>> a1 = Address(id=existing_a1.id, user_id=u1.id)
>>> assert a1.user is None
>>> True
>>> a1 = session.merge(a1)
>>> session.commit()
sqlalchemy.exc.IntegrityError: (IntegrityError) address.user_id
may not be NULL
```

Here, we accessed a1.user, which returned its default value of None, which as a result of this access, has been placed in the __dict__ of our object a1. Normally, this operation creates no change event, so the user_id attribute takes precedence during a flush. But when we merge the Address object into the session, the operation is equivalent to:

```
>>> existing_a1.id = existing_a1.id
>>> existing_a1.user_id = u1.id
>>> existing_a1.user = None
```

Where above, both user_id and user are assigned to, and change events are emitted for both. The user association takes precedence, and None is applied to user_id, causing a failure.

Most *merge()* issues can be examined by first checking - is the object prematurely in the session ?

```
>>> a1 = Address(id=existing_a1, user_id=user.id)
>>> assert a1 not in session
>>> a1 = session.merge(a1)
```

Or is there state on the object that we don't want ? Examining `__dict__` is a quick way to check:

```
>>> a1 = Address(id=existing_a1, user_id=user.id)
>>> a1.user
>>> a1.__dict__
{'_sa_instance_state': <sqlalchemy.orm.state.InstanceState object at 0x1298d10>,
    'user_id': 1,
    'id': 1,
    'user': None}
>>> # we don't want user=None merged, remove it
>>> del a1.user
>>> a1 = session.merge(a1)
>>> # success
>>> session.commit()
```

Expunging

Expunge removes an object from the Session, sending persistent instances to the detached state, and pending instances to the transient state:

```
session.expunge(obj1)
```

To remove all items, call *expunge_all()* (this method was formerly known as `clear()`).

Refreshing / Expiring

Expiring means that the database-persisted data held inside a series of object attributes is erased, in such a way that when those attributes are next accessed, a SQL query is emitted which will refresh that data from the database.

When we talk about expiration of data we are usually talking about an object that is in the persistent state. For example, if we load an object as follows:

```
user = session.query(User).filter_by(name='user1').first()
```

The above `User` object is persistent, and has a series of attributes present; if we were to look inside its `__dict__`, we'd see that state loaded:

```
>>> user.__dict__
{
  'id': 1, 'name': u'user1',
  '_sa_instance_state': <...>,
}
```

where `id` and `name` refer to those columns in the database. `_sa_instance_state` is a non-database-persisted value used by SQLAlchemy internally (it refers to the *InstanceState* for the instance. While not directly relevant to this section, if we want to get at it, we should use the *inspect()* function to access it).

At this point, the state in our `User` object matches that of the loaded database row. But upon expiring the object using a method such as *Session.expire()*, we see that the state is removed:

```
>>> session.expire(user)
>>> user.__dict__
{'_sa_instance_state': <...>}
```

We see that while the internal "state" still hangs around, the values which correspond to the `id` and `name` columns are gone. If we were to access one of these columns and are watching SQL, we'd see this:

```
>>> print(user.name)
SELECT user.id AS user_id, user.name AS user_name
FROM user
WHERE user.id = ?
(1,)
user1
```

Above, upon accessing the expired attribute `user.name`, the ORM initiated a lazy load to retrieve the most recent state from the database, by emitting a SELECT for the user row to which this user refers. Afterwards, the `__dict__` is again populated:

```
>>> user.__dict__
{
  'id': 1, 'name': u'user1',
  '_sa_instance_state': <...>,
}
```

Note: While we are peeking inside of `__dict__` in order to see a bit of what SQLAlchemy does with object attributes, we **should not modify** the contents of `__dict__` directly, at least as far as those attributes which the SQLAlchemy ORM is maintaining (other attributes outside of SQLA's realm are fine). This is because SQLAlchemy uses descriptors in order to track the changes we make to an object, and when we modify `__dict__` directly, the ORM won't be able to track that we changed something.

Another key behavior of both *expire()* and *refresh()* is that all un-flushed changes on an object are discarded. That is, if we were to modify an attribute on our `User`:

```
>>> user.name = 'user2'
```

but then we call *expire()* without first calling *flush()*, our pending value of `'user2'` is discarded:

```
>>> session.expire(user)
>>> user.name
'user1'
```

The *expire()* method can be used to mark as "expired" all ORM-mapped attributes for an instance:

```
# expire all ORM-mapped attributes on obj1
session.expire(obj1)
```

it can also be passed a list of string attribute names, referring to specific attributes to be marked as expired:

```
# expire only attributes obj1.attr1, obj1.attr2
session.expire(obj1, ['attr1', 'attr2'])
```

The *refresh()* method has a similar interface, but instead of expiring, it emits an immediate SELECT for the object's row immediately:

```
# reload all attributes on obj1
session.refresh(obj1)
```

refresh() also accepts a list of string attribute names, but unlike *expire()*, expects at least one name to be that of a column-mapped attribute:

```
# reload obj1.attr1, obj1.attr2
session.refresh(obj1, ['attr1', 'attr2'])
```

The `Session.expire_all()` method allows us to essentially call `Session.expire()` on all objects contained within the `Session` at once:

```
session.expire_all()
```

What Actually Loads

The SELECT statement that's emitted when an object marked with `expire()` or loaded with `refresh()` varies based on several factors, including:

- The load of expired attributes is triggered from **column-mapped attributes only**. While any kind of attribute can be marked as expired, including a `relationship()` - mapped attribute, accessing an expired `relationship()` attribute will emit a load only for that attribute, using standard relationship-oriented lazy loading. Column-oriented attributes, even if expired, will not load as part of this operation, and instead will load when any column-oriented attribute is accessed.
- `relationship()`- mapped attributes will not load in response to expired column-based attributes being accessed.
- Regarding relationships, `refresh()` is more restrictive than `expire()` with regards to attributes that aren't column-mapped. Calling `refresh()` and passing a list of names that only includes relationship-mapped attributes will actually raise an error. In any case, non-eager-loading `relationship()` attributes will not be included in any refresh operation.
- `relationship()` attributes configured as "eager loading" via the `lazy` parameter will load in the case of `refresh()`, if either no attribute names are specified, or if their names are inclued in the list of attributes to be refreshed.
- Attributes that are configured as `deferred()` will not normally load, during either the expired-attribute load or during a refresh. An unloaded attribute that's `deferred()` instead loads on its own when directly accessed, or if part of a "group" of deferred attributes where an unloaded attribute in that group is accessed.
- For expired attributes that are loaded on access, a joined-inheritance table mapping will emit a SELECT that typically only includes those tables for which unloaded attributes are present. The action here is sophisticated enough to load only the parent or child table, for example, if the subset of columns that were originally expired encompass only one or the other of those tables.
- When `refresh()` is used on a joined-inheritance table mapping, the SELECT emitted will resemble that of when `Session.query()` is used on the target object's class. This is typically all those tables that are set up as part of the mapping.

When to Expire or Refresh

The `Session` uses the expiration feature automatically whenever the transaction referred to by the session ends. Meaning, whenever `Session.commit()` or `Session.rollback()` is called, all objects within the `Session` are expired, using a feature equivalent to that of the `Session.expire_all()` method. The rationale is that the end of a transaction is a demarcating point at which there is no more context available in order to know what the current state of the database is, as any number of other transactions may be affecting it. Only when a new transaction starts can we again have access to the current state of the database, at which point any number of changes may have occurred.

Transaction Isolation

Of course, most databases are capable of handling multiple transactions at once, even involving the same rows of data. When a relational database handles multiple transactions involving the same tables or rows, this is when the isolation aspect of the database comes into play. The isolation behavior of different databases varies considerably and even on a single database can be configured to behave in different ways (via the so-called isolation level setting). In that sense, the `Session` can't fully predict when the same SELECT statement, emitted a second time, will definitely return the data we already have, or will return new data. So as a best guess, it assumes that within the scope of a transaction, unless it is known that a SQL expression has been emitted to modify a particular row, there's no need to refresh a row unless explicitly told to do so.

The `Session.expire()` and `Session.refresh()` methods are used in those cases when one wants to force an object to re-load its data from the database, in those cases when it is known that the current state of data is possibly stale. Reasons for this might include:

- some SQL has been emitted within the transaction outside of the scope of the ORM's object handling, such as if a `Table.update()` construct were emitted using the `Session.execute()` method;
- if the application is attempting to acquire data that is known to have been modified in a concurrent transaction, and it is also known that the isolation rules in effect allow this data to be visible.

The second bullet has the important caveat that "it is also known that the isolation rules in effect allow this data to be visible." This means that it cannot be assumed that an UPDATE that happened on another database connection will yet be visible here locally; in many cases, it will not. This is why if one wishes to use `expire()` or `refresh()` in order to view data between ongoing transactions, an understanding of the isolation behavior in effect is essential.

See also:

`Session.expire()`

`Session.expire_all()`

`Session.refresh()`

isolation - glossary explanation of isolation which includes links to Wikipedia.

The SQLAlchemy Session In-Depth - a video + slides with an in-depth discussion of the object lifecycle including the role of data expiration.

2.5.3 Cascades

Mappers support the concept of configurable cascade behavior on `relationship()` constructs. This refers to how operations performed on a "parent" object relative to a particular `Session` should be propagated to items referred to by that relationship (e.g. "child" objects), and is affected by the `relationship.cascade` option.

The default behavior of cascade is limited to cascades of the so-called *save-update* and *merge* settings. The typical "alternative" setting for cascade is to add the *delete* and *delete-orphan* options; these settings are appropriate for related objects which only exist as long as they are attached to their parent, and are otherwise deleted.

Cascade behavior is configured using the by changing the `cascade` option on `relationship()`:

```
class Order(Base):
    __tablename__ = 'order'

    items = relationship("Item", cascade="all, delete-orphan")
    customer = relationship("User", cascade="save-update")
```

To set cascades on a backref, the same flag can be used with the `backref()` function, which ultimately feeds its arguments back into `relationship()`:

```
class Item(Base):
    __tablename__ = 'item'

    order = relationship("Order",
                         backref=backref("items", cascade="all, delete-orphan")
                    )
```

The Origins of Cascade

SQLAlchemy's notion of cascading behavior on relationships, as well as the options to configure them, are primarily derived from the similar feature in the Hibernate ORM; Hibernate refers to "cascade" in a few places such as in Example: Parent/Child. If cascades are confusing, we'll refer to their conclusion, stating "The sections we have just covered can be a bit confusing. However, in practice, it all works out nicely."

The default value of *cascade* is `save-update, merge`. The typical alternative setting for this parameter is either `all` or more commonly `all, delete-orphan`. The `all` symbol is a synonym for `save-update, merge, refresh-expire, expunge, delete`, and using it in conjunction with `delete-orphan` indicates that the child object should follow along with its parent in all cases, and be deleted once it is no longer associated with that parent.

The list of available values which can be specified for the *cascade* parameter are described in the following subsections.

save-update

`save-update` cascade indicates that when an object is placed into a *Session* via *Session.add()*, all the objects associated with it via this *relationship()* should also be added to that same *Session*. Suppose we have an object `user1` with two related objects `address1`, `address2`:

```
>>> user1 = User()
>>> address1, address2 = Address(), Address()
>>> user1.addresses = [address1, address2]
```

If we add `user1` to a *Session*, it will also add `address1`, `address2` implicitly:

```
>>> sess = Session()
>>> sess.add(user1)
>>> address1 in sess
True
```

`save-update` cascade also affects attribute operations for objects that are already present in a *Session*. If we add a third object, `address3` to the `user1.addresses` collection, it becomes part of the state of that *Session*:

```
>>> address3 = Address()
>>> user1.append(address3)
>>> address3 in sess
>>> True
```

`save-update` has the possibly surprising behavior which is that persistent objects which were *removed* from a collection or in some cases a scalar attribute may also be pulled into the *Session* of a parent object; this is so that the flush process may handle that related object appropriately. This case can usually only arise if an object is removed from one *Session* and added to another:

```
>>> user1 = sess1.query(User).filter_by(id=1).first()
>>> address1 = user1.addresses[0]
>>> sess1.close()    # user1, address1 no longer associated with sess1
```

```
>>> user1.addresses.remove(address1)  # address1 no longer associated with user1
>>> sess2 = Session()
>>> sess2.add(user1)    # ... but it still gets added to the new session,
>>> address1 in sess2  # because it's still "pending" for flush
True
```

The `save-update` cascade is on by default, and is typically taken for granted; it simplifies code by allowing a single call to `Session.add()` to register an entire structure of objects within that `Session` at once. While it can be disabled, there is usually not a need to do so.

One case where `save-update` cascade does sometimes get in the way is in that it takes place in both directions for bi-directional relationships, e.g. backrefs, meaning that the association of a child object with a particular parent can have the effect of the parent object being implicitly associated with that child object's `Session`; this pattern, as well as how to modify its behavior using the `cascade_backrefs` flag, is discussed in the section *Controlling Cascade on Backrefs*.

delete

The `delete` cascade indicates that when a "parent" object is marked for deletion, its related "child" objects should also be marked for deletion. If for example we we have a relationship `User.addresses` with `delete` cascade configured:

```
class User(Base):
    # ...

    addresses = relationship("Address", cascade="save-update, merge, delete")
```

If using the above mapping, we have a `User` object and two related `Address` objects:

```
>>> user1 = sess.query(User).filter_by(id=1).first()
>>> address1, address2 = user1.addresses
```

If we mark `user1` for deletion, after the flush operation proceeds, `address1` and `address2` will also be deleted:

```
>>> sess.delete(user1)
>>> sess.commit()
DELETE FROM address WHERE address.id = ?
((1,), (2,))
DELETE FROM user WHERE user.id = ?
(1,)
COMMIT
```

Alternatively, if our `User.addresses` relationship does *not* have `delete` cascade, SQLAlchemy's default behavior is to instead de-associate `address1` and `address2` from `user1` by setting their foreign key reference to `NULL`. Using a mapping as follows:

```
class User(Base):
    # ...

    addresses = relationship("Address")
```

Upon deletion of a parent `User` object, the rows in `address` are not deleted, but are instead de-associated:

```
>>> sess.delete(user1)
>>> sess.commit()
UPDATE address SET user_id=? WHERE address.id = ?
(None, 1)
UPDATE address SET user_id=? WHERE address.id = ?
```

```
(None, 2)
DELETE FROM user WHERE user.id = ?
(1,)
COMMIT
```

`delete` cascade is more often than not used in conjunction with *delete-orphan* cascade, which will emit a DELETE for the related row if the "child" object is deassociated from the parent. The combination of `delete` and `delete-orphan` cascade covers both situations where SQLAlchemy has to decide between setting a foreign key column to NULL versus deleting the row entirely.

ORM-level "delete" cascade vs. FOREIGN KEY level "ON DELETE" cascade

The behavior of SQLAlchemy's "delete" cascade has a lot of overlap with the `ON DELETE CASCADE` feature of a database foreign key, as well as with that of the `ON DELETE SET NULL` foreign key setting when "delete" cascade is not specified. Database level "ON DELETE" cascades are specific to the "FOREIGN KEY" construct of the relational database; SQLAlchemy allows configuration of these schema-level constructs at the DDL level using options on `ForeignKeyConstraint` which are described at *ON UPDATE and ON DELETE*.
It is important to note the differences between the ORM and the relational database's notion of "cascade" as well as how they integrate:

- A database level `ON DELETE` cascade is configured effectively on the **many-to-one** side of the relationship; that is, we configure it relative to the `FOREIGN KEY` constraint that is the "many" side of a relationship. At the ORM level, **this direction is reversed**. SQLAlchemy handles the deletion of "child" objects relative to a "parent" from the "parent" side, which means that `delete` and `delete-orphan` cascade are configured on the **one-to-many** side.
- Database level foreign keys with no `ON DELETE` setting are often used to **prevent** a parent row from being removed, as it would necessarily leave an unhandled related row present. If this behavior is desired in a one-to-many relationship, SQLAlchemy's default behavior of setting a foreign key to `NULL` can be caught in one of two ways:
  - The easiest and most common is just to set the foreign-key-holding column to `NOT NULL` at the database schema level. An attempt by SQLAlchemy to set the column to NULL will fail with a simple NOT NULL constraint exception.
  - The other, more special case way is to set the `passive_deletes` flag to the string `"all"`. This has the effect of entirely disabling SQLAlchemy's behavior of setting the foreign key column to NULL, and a DELETE will be emitted for the parent row without any affect on the child row, even if the child row is present in memory. This may be desirable in the case when database-level foreign key triggers, either special `ON DELETE` settings or otherwise, need to be activated in all cases when a parent row is deleted.
- Database level `ON DELETE` cascade is **vastly more efficient** than that of SQLAlchemy. The database can chain a series of cascade operations across many relationships at once; e.g. if row A is deleted, all the related rows in table B can be deleted, and all the C rows related to each of those B rows, and on and on, all within the scope of a single DELETE statement. SQLAlchemy on the other hand, in order to support the cascading delete operation fully, has to individually load each related collection in order to target all rows that then may have further related collections. That is, SQLAlchemy isn't sophisticated enough to emit a DELETE for all those related rows at once within this context.
- SQLAlchemy doesn't **need** to be this sophisticated, as we instead provide smooth integration with the database's own `ON DELETE` functionality, by using the `passive_deletes` option in conjunction with properly configured foreign key constraints. Under this behavior, SQLAlchemy only emits DELETE for those rows that are already locally present in the `Session`; for any collections that are unloaded, it leaves them to the database to handle, rather than emitting a SELECT for them. The section *Using Passive Deletes* provides an example of this use.
- While database-level `ON DELETE` functionality works only on the "many" side of a relationship, SQLAlchemy's "delete" cascade has **limited** ability to operate in the *reverse* direction as well, meaning it can be configured on the "many" side to delete an object on the "one" side when the reference on the "many" side is deleted. However this can easily result in constraint violations if there are other objects referring to this "one" side from the "many", so it typically is only useful when a relationship is in fact a "one to one". The `single_parent` flag should be used to establish an in-Python assertion for this case.

When using a `relationship()` that also includes a many-to-many table using the `secondary` option, SQLAlchemy's delete cascade handles the rows in this many-to-many table automatically. Just like, as described in *Deleting Rows from the Many to Many Table*, the addition or removal of an object from a many-to-many collection results in the INSERT or DELETE of a row in the many-to-many table, the `delete` cascade, when activated as the result of a parent object delete operation, will DELETE not just the row in the "child" table but also in the many-to-many table.

delete-orphan

`delete-orphan` cascade adds behavior to the `delete` cascade, such that a child object will be marked for deletion when it is de-associated from the parent, not just when the parent is marked for deletion. This is a common feature when dealing with a related object that is "owned" by its parent, with a NOT NULL foreign key, so that removal of the item from the parent collection results in its deletion.

`delete-orphan` cascade implies that each child object can only have one parent at a time, so is configured in the vast majority of cases on a one-to-many relationship. Setting it on a many-to-one or many-to-many relationship is more awkward; for this use case, SQLAlchemy requires that the *relationship()* be configured with the *single_parent* argument, establishes Python-side validation that ensures the object is associated with only one parent at a time.

merge

`merge` cascade indicates that the *Session.merge()* operation should be propagated from a parent that's the subject of the *Session.merge()* call down to referred objects. This cascade is also on by default.

refresh-expire

`refresh-expire` is an uncommon option, indicating that the *Session.expire()* operation should be propagated from a parent down to referred objects. When using *Session.refresh()*, the referred objects are expired only, but not actually refreshed.

expunge

`expunge` cascade indicates that when the parent object is removed from the *Session* using *Session.expunge()*, the operation should be propagated down to referred objects.

Controlling Cascade on Backrefs

The *save-update* cascade by default takes place on attribute change events emitted from backrefs. This is probably a confusing statement more easily described through demonstration; it means that, given a mapping such as this:

```
mapper(Order, order_table, properties={
    'items' : relationship(Item, backref='order')
})
```

If an `Order` is already in the session, and is assigned to the `order` attribute of an `Item`, the backref appends the `Order` to the `items` collection of that `Order`, resulting in the `save-update` cascade taking place:

```
>>> o1 = Order()
>>> session.add(o1)
>>> o1 in session
True

>>> i1 = Item()
>>> i1.order = o1
>>> i1 in o1.items
True
>>> i1 in session
True
```

This behavior can be disabled using the *cascade_backrefs* flag:

```
mapper(Order, order_table, properties={
    'items' : relationship(Item, backref='order',
                                cascade_backrefs=False)
})
```

So above, the assignment of `i1.order = o1` will append `i1` to the `items` collection of `o1`, but will not add `i1` to the session. You can, of course, *add()* `i1` to the session at a later point. This option may be helpful for situations where an object needs to be kept out of a session until it's construction is completed, but still needs to be given associations to objects which are already persistent in the target session.

2.5.4 Transactions and Connection Management

Managing Transactions

A newly constructed *Session* may be said to be in the "begin" state. In this state, the *Session* has not established any connection or transactional state with any of the *Engine* objects that may be associated with it.

The *Session* then receives requests to operate upon a database connection. Typically, this means it is called upon to execute SQL statements using a particular *Engine*, which may be via *Session.query()*, *Session.execute()*, or within a flush operation of pending data, which occurs when such state exists and *Session.commit()* or *Session.flush()* is called.

As these requests are received, each new *Engine* encountered is associated with an ongoing transactional state maintained by the *Session*. When the first *Engine* is operated upon, the *Session* can be said to have left the "begin" state and entered "transactional" state. For each *Engine* encountered, a *Connection* is associated with it, which is acquired via the *Engine.contextual_connect()* method. If a *Connection* was directly associated with the *Session* (see *Joining a Session into an External Transaction (such as for test suites)* for an example of this), it is added to the transactional state directly.

For each *Connection*, the *Session* also maintains a *Transaction* object, which is acquired by calling *Connection.begin()* on each *Connection*, or if the *Session* object has been established using the flag `twophase=True`, a *TwoPhaseTransaction* object acquired via *Connection.begin_twophase()*. These transactions are all committed or rolled back corresponding to the invocation of the *Session.commit()* and *Session.rollback()* methods. A commit operation will also call the *TwoPhaseTransaction.prepare()* method on all transactions if applicable.

When the transactional state is completed after a rollback or commit, the *Session* releases all *Transaction* and *Connection* resources, and goes back to the "begin" state, which will again invoke new *Connection* and *Transaction* objects as new requests to emit SQL statements are received.

The example below illustrates this lifecycle:

```
engine = create_engine("...")
Session = sessionmaker(bind=engine)

# new session.   no connections are in use.
session = Session()
try:
    # first query.  a Connection is acquired
    # from the Engine, and a Transaction
    # started.
    item1 = session.query(Item).get(1)

    # second query.  the same Connection/Transaction
    # are used.
    item2 = session.query(Item).get(2)
```

```
    # pending changes are created.
    item1.foo = 'bar'
    item2.bar = 'foo'

    # commit.  The pending changes above
    # are flushed via flush(), the Transaction
    # is committed, the Connection object closed
    # and discarded, the underlying DBAPI connection
    # returned to the connection pool.
    session.commit()
except:
    # on rollback, the same closure of state
    # as that of commit proceeds.
    session.rollback()
    raise
```

Using SAVEPOINT

SAVEPOINT transactions, if supported by the underlying engine, may be delineated using the `begin_nested()` method:

```
Session = sessionmaker()
session = Session()
session.add(u1)
session.add(u2)

session.begin_nested() # establish a savepoint
session.add(u3)
session.rollback()  # rolls back u3, keeps u1 and u2

session.commit() # commits u1 and u2
```

`begin_nested()` may be called any number of times, which will issue a new SAVEPOINT with a unique identifier for each call. For each `begin_nested()` call, a corresponding `rollback()` or `commit()` must be issued. (But note that if the return value is used as a context manager, i.e. in a with-statement, then this rollback/commit is issued by the context manager upon exiting the context, and so should not be added explicitly.)

When `begin_nested()` is called, a `flush()` is unconditionally issued (regardless of the `autoflush` setting). This is so that when a `rollback()` occurs, the full state of the session is expired, thus causing all subsequent attribute/instance access to reference the full state of the `Session` right before `begin_nested()` was called.

`begin_nested()`, in the same manner as the less often used `begin()` method, returns a transactional object which also works as a context manager. It can be succinctly used around individual record inserts in order to catch things like unique constraint exceptions:

```
for record in records:
    try:
        with session.begin_nested():
            session.merge(record)
    except:
        print "Skipped record %s" % record
session.commit()
```

Autocommit Mode

The example of `Session` transaction lifecycle illustrated at the start of *Managing Transactions* applies to a `Session` configured in the default mode of `autocommit=False`. Constructing a `Session` with `autocommit=True` produces a `Session` placed into "autocommit" mode, where each SQL statement invoked by a `Session.query()` or `Session.execute()` occurs using a new connection from the connection pool, discarding it after results have been iterated. The `Session.flush()` operation still occurs within the scope of a single transaction, though this transaction is closed out after the `Session.flush()` operation completes.

> **Warning:** "autocommit" mode should **not be considered for general use**. If used, it should always be combined with the usage of `Session.begin()` and `Session.commit()`, to ensure a transaction demarcation.
> Executing queries outside of a demarcated transaction is a legacy mode of usage, and can in some cases lead to concurrent connection checkouts.
> In the absence of a demarcated transaction, the `Session` cannot make appropriate decisions as to when autoflush should occur nor when auto-expiration should occur, so these features should be disabled with `autoflush=False, expire_on_commit=False`.

Modern usage of "autocommit" is for framework integrations that need to control specifically when the "begin" state occurs. A session which is configured with `autocommit=True` may be placed into the "begin" state using the `Session.begin()` method. After the cycle completes upon `Session.commit()` or `Session.rollback()`, connection and transaction resources are released and the `Session` goes back into "autocommit" mode, until `Session.begin()` is called again:

```
Session = sessionmaker(bind=engine, autocommit=True)
session = Session()
session.begin()
try:
    item1 = session.query(Item).get(1)
    item2 = session.query(Item).get(2)
    item1.foo = 'bar'
    item2.bar = 'foo'
    session.commit()
except:
    session.rollback()
    raise
```

The `Session.begin()` method also returns a transactional token which is compatible with the Python 2.6 `with` statement:

```
Session = sessionmaker(bind=engine, autocommit=True)
session = Session()
with session.begin():
    item1 = session.query(Item).get(1)
    item2 = session.query(Item).get(2)
    item1.foo = 'bar'
    item2.bar = 'foo'
```

Using Subtransactions with Autocommit A subtransaction indicates usage of the `Session.begin()` method in conjunction with the `subtransactions=True` flag. This produces a non-transactional, delimiting construct that allows nesting of calls to `begin()` and `commit()`. Its purpose is to allow the construction of code that can function within a transaction both independently of any external code that starts a transaction, as well as within a block that has already demarcated a transaction.

`subtransactions=True` is generally only useful in conjunction with autocommit, and is equivalent to the pattern described at *Nesting of Transaction Blocks*, where any number of functions can call `Connection.begin()` and

Transaction.commit() as though they are the initiator of the transaction, but in fact may be participating in an already ongoing transaction:

```
# method_a starts a transaction and calls method_b
def method_a(session):
    session.begin(subtransactions=True)
    try:
        method_b(session)
        session.commit()  # transaction is committed here
    except:
        session.rollback() # rolls back the transaction
        raise

# method_b also starts a transaction, but when
# called from method_a participates in the ongoing
# transaction.
def method_b(session):
    session.begin(subtransactions=True)
    try:
        session.add(SomeObject('bat', 'lala'))
        session.commit()  # transaction is not committed yet
    except:
        session.rollback() # rolls back the transaction, in this case
                           # the one that was initiated in method_a().
        raise

# create a Session and call method_a
session = Session(autocommit=True)
method_a(session)
session.close()
```

Subtransactions are used by the *Session.flush()* process to ensure that the flush operation takes place within a transaction, regardless of autocommit. When autocommit is disabled, it is still useful in that it forces the *Session* into a "pending rollback" state, as a failed flush cannot be resumed in mid-operation, where the end user still maintains the "scope" of the transaction overall.

Enabling Two-Phase Commit

For backends which support two-phase operaration (currently MySQL and PostgreSQL), the session can be instructed to use two-phase commit semantics. This will coordinate the committing of transactions across databases so that the transaction is either committed or rolled back in all databases. You can also *prepare()* the session for interacting with transactions not managed by SQLAlchemy. To use two phase transactions set the flag `twophase=True` on the session:

```
engine1 = create_engine('postgresql://db1')
engine2 = create_engine('postgresql://db2')

Session = sessionmaker(twophase=True)

# bind User operations to engine 1, Account operations to engine 2
Session.configure(binds={User:engine1, Account:engine2})

session = Session()

# .... work with accounts and users

# commit.  session will issue a flush to all DBs, and a prepare step to all DBs,
```

```
# before committing both transactions
session.commit()
```

Setting Transaction Isolation Levels

Isolation refers to the behavior of the transaction at the database level in relation to other transactions occurring concurrently. There are four well-known modes of isolation, and typically the Python DBAPI allows these to be set on a per-connection basis, either through explicit APIs or via database-specific calls.

SQLAlchemy's dialects support settable isolation modes on a per-`Engine` or per-`Connection` basis, using flags at both the `create_engine()` level as well as at the `Connection.execution_options()` level.

When using the ORM `Session`, it acts as a *facade* for engines and connections, but does not expose transaction isolation directly. So in order to affect transaction isolation level, we need to act upon the `Engine` or `Connection` as appropriate.

See also:

`create_engine.isolation_level`

SQLite Transaction Isolation

Postgresql Isolation Level

MySQL Isolation Level

Setting Isolation Engine-Wide To set up a `Session` or `sessionmaker` with a specific isolation level globally, use the `create_engine.isolation_level` parameter:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

eng = create_engine(
    "postgresql://scott:tiger@localhost/test",
    isolation_level='REPEATABLE_READ')

maker = sessionmaker(bind=eng)

session = maker()
```

Setting Isolation for Individual Sessions When we make a new `Session`, either using the constructor directly or when we call upon the callable produced by a `sessionmaker`, we can pass the `bind` argument directly, overriding the pre-existing bind. We can combine this with the `Engine.execution_options()` method in order to produce a copy of the original `Engine` that will add this option:

```
session = maker(
    bind=engine.execution_options(isolation_level='SERIALIZABLE'))
```

For the case where the `Session` or `sessionmaker` is configured with multiple "binds", we can either re-specify the `binds` argument fully, or if we want to only replace specific binds, we can use the `Session.bind_mapper()` or `Session.bind_table()` methods:

```
session = maker()
session.bind_mapper(
    User, user_engine.execution_options(isolation_level='SERIALIZABLE'))
```

We can also use the individual transaction method that follows.

Setting Isolation for Individual Transactions A key caveat regarding isolation level is that the setting cannot be safely modified on a *Connection* where a transaction has already started. Databases cannot change the isolation level of a transaction in progress, and some DBAPIs and SQLAlchemy dialects have inconsistent behaviors in this area. Some may implicitly emit a ROLLBACK and some may implicitly emit a COMMIT, others may ignore the setting until the next transaction. Therefore SQLAlchemy emits a warning if this option is set when a transaction is already in play. The *Session* object does not provide for us a *Connection* for use in a transaction where the transaction is not already begun. So here, we need to pass execution options to the *Session* at the start of a transaction by passing *Session.connection.execution_options* provided by the *Session.connection()* method:

```
from sqlalchemy.orm import Session

sess = Session(bind=engine)
sess.connection(execution_options={'isolation_level': 'SERIALIZABLE'})

# work with session

# commit transaction.  the connection is released
# and reverted to its previous isolation level.
sess.commit()
```

Above, we first produce a *Session* using either the constructor or a *sessionmaker*. Then we explicitly set up the start of a transaction by calling upon *Session.connection()*, which provides for execution options that will be passed to the connection before the transaction is begun. If we are working with a *Session* that has multiple binds or some other custom scheme for *Session.get_bind()*, we can pass additional arguments to *Session.connection()* in order to affect how the bind is procured:

```
sess = my_sesssionmaker()

# set up a transaction for the bind associated with
# the User mapper
sess.connection(
    mapper=User,
    execution_options={'isolation_level': 'SERIALIZABLE'})

# work with session

# commit transaction.  the connection is released
# and reverted to its previous isolation level.
sess.commit()
```

The *Session.connection.execution_options* argument is only accepted on the **first** call to *Session.connection()* for a particular bind within a transaction. If a transaction is already begun on the target connection, a warning is emitted:

```
>>> session = Session(eng)
>>> session.execute("select 1")
<sqlalchemy.engine.result.ResultProxy object at 0x1017a6c50>
>>> session.connection(execution_options={'isolation_level': 'SERIALIZABLE'})
sqlalchemy/orm/session.py:310: SAWarning: Connection is already established
for the given bind; execution_options ignored
```

New in version 0.9.9: Added the *Session.connection.execution_options* parameter to *Session.connection()*.

Tracking Transaction State with Events

See the section *Transaction Events* for an overview of the available event hooks for session transaction state changes.

Joining a Session into an External Transaction (such as for test suites)

If a `Connection` is being used which is already in a transactional state (i.e. has a `Transaction` established), a `Session` can be made to participate within that transaction by just binding the `Session` to that `Connection`. The usual rationale for this is a test suite that allows ORM code to work freely with a `Session`, including the ability to call `Session.commit()`, where afterwards the entire database interaction is rolled back:

```
from sqlalchemy.orm import sessionmaker
from sqlalchemy import create_engine
from unittest import TestCase

# global application scope.  create Session class, engine
Session = sessionmaker()

engine = create_engine('postgresql://...')

class SomeTest(TestCase):
    def setUp(self):
        # connect to the database
        self.connection = engine.connect()

        # begin a non-ORM transaction
        self.trans = self.connection.begin()

        # bind an individual Session to the connection
        self.session = Session(bind=self.connection)

    def test_something(self):
        # use the session in tests.

        self.session.add(Foo())
        self.session.commit()

    def tearDown(self):
        self.session.close()

        # rollback - everything that happened with the
        # Session above (including calls to commit())
        # is rolled back.
        self.trans.rollback()

        # return connection to the Engine
        self.connection.close()
```

Above, we issue `Session.commit()` as well as `Transaction.rollback()`. This is an example of where we take advantage of the `Connection` object's ability to maintain *subtransactions*, or nested begin/commit-or-rollback pairs where only the outermost begin/commit pair actually commits the transaction, or if the outermost block rolls back, everything is rolled back.

Supporting Tests with Rollbacks

The above recipe works well for any kind of database enabled test, except for a test that needs to actually invoke `Session.rollback()` within the scope of the test itself. The above recipe can be expanded, such that the `Session` always runs all operations within the scope of a SAVEPOINT, which is established at the start of each transaction, so that tests can also rollback the "transaction" as well while still remaining in the scope of a larger "transaction" that's never committed, using two extra events:

```
from sqlalchemy import event


class SomeTest(TestCase):

    def setUp(self):
        # connect to the database
        self.connection = engine.connect()

        # begin a non-ORM transaction
        self.trans = connection.begin()

        # bind an individual Session to the connection
        self.session = Session(bind=self.connection)

        # start the session in a SAVEPOINT...
        self.session.begin_nested()

        # then each time that SAVEPOINT ends, reopen it
        @event.listens_for(self.session, "after_transaction_end")
        def restart_savepoint(session, transaction):
            if transaction.nested and not transaction._parent.nested:

                # ensure that state is expired the way
                # session.commit() at the top level normally does
                # (optional step)
                session.expire_all()

                session.begin_nested()

    # ... the tearDown() method stays the same
```

2.5.5 Additional Persistence Techniques

Embedding SQL Insert/Update Expressions into a Flush

This feature allows the value of a database column to be set to a SQL expression instead of a literal value. It's especially useful for atomic updates, calling stored procedures, etc. All you do is assign an expression to an attribute:

```
class SomeClass(object):
    pass
mapper(SomeClass, some_table)

someobject = session.query(SomeClass).get(5)

# set 'value' attribute to a SQL expression adding one
someobject.value = some_table.c.value + 1
```

```
# issues "UPDATE some_table SET value=value+1"
session.commit()
```

This technique works both for INSERT and UPDATE statements. After the flush/commit operation, the `value` attribute on `someobject` above is expired, so that when next accessed the newly generated value will be loaded from the database.

Using SQL Expressions with Sessions

SQL expressions and strings can be executed via the *Session* within its transactional context. This is most easily accomplished using the *execute()* method, which returns a *ResultProxy* in the same manner as an *Engine* or *Connection*:

```
Session = sessionmaker(bind=engine)
session = Session()

# execute a string statement
result = session.execute("select * from table where id=:id", {'id':7})

# execute a SQL expression construct
result = session.execute(select([mytable]).where(mytable.c.id==7))
```

The current *Connection* held by the *Session* is accessible using the *connection()* method:

```
connection = session.connection()
```

The examples above deal with a *Session* that's bound to a single *Engine* or *Connection*. To execute statements using a *Session* which is bound either to multiple engines, or none at all (i.e. relies upon bound metadata), both *execute()* and *connection()* accept a `mapper` keyword argument, which is passed a mapped class or *Mapper* instance, which is used to locate the proper context for the desired engine:

```
Session = sessionmaker()
session = Session()

# need to specify mapper or class when executing
result = session.execute("select * from table where id=:id", {'id':7}, mapper=MyMappedClass)

result = session.execute(select([mytable], mytable.c.id==7), mapper=MyMappedClass)

connection = session.connection(MyMappedClass)
```

Forcing NULL on a column with a default

The ORM considers any attribute that was never set on an object as a "default" case; the attribute will be omitted from the INSERT statement:

```
class MyObject(Base):
    __tablename__ = 'my_table'
    id = Column(Integer, primary_key=True)
    data = Column(String(50), nullable=True)

obj = MyObject(id=1)
session.add(obj)
session.commit()  # INSERT with the 'data' column omitted; the database
                  # itself will persist this as the NULL value
```

Omitting a column from the INSERT means that the column will have the NULL value set, *unless* the column has a default set up, in which case the default value will be persisted. This holds true both from a pure SQL perspective with server-side defaults, as well as the behavior of SQLAlchemy's insert behavior with both client-side and server-side defaults:

```
class MyObject(Base):
    __tablename__ = 'my_table'
    id = Column(Integer, primary_key=True)
    data = Column(String(50), nullable=True, server_default="default")

obj = MyObject(id=1)
session.add(obj)
session.commit()  # INSERT with the 'data' column omitted; the database
                  # itself will persist this as the value 'default'
```

However, in the ORM, even if one assigns the Python value `None` explicitly to the object, this is treated the **same** as though the value were never assigned:

```
class MyObject(Base):
    __tablename__ = 'my_table'
    id = Column(Integer, primary_key=True)
    data = Column(String(50), nullable=True, server_default="default")

obj = MyObject(id=1, data=None)
session.add(obj)
session.commit()  # INSERT with the 'data' column explicitly set to None;
                  # the ORM still omits it from the statement and the
                  # database will still persist this as the value 'default'
```

The above operation will persist into the `data` column the server default value of `"default"` and not SQL NULL, even though `None` was passed; this is a long-standing behavior of the ORM that many applications hold as an assumption.

So what if we want to actually put NULL into this column, even though the column has a default value? There are two approaches. One is that on a per-instance level, we assign the attribute using the *null* SQL construct:

```
from sqlalchemy import null

obj = MyObject(id=1, data=null())
session.add(obj)
session.commit()  # INSERT with the 'data' column explicitly set as null();
                  # the ORM uses this directly, bypassing all client-
                  # and server-side defaults, and the database will
                  # persist this as the NULL value
```

The *null* SQL construct always translates into the SQL NULL value being directly present in the target INSERT statement.

If we'd like to be able to use the Python value `None` and have this also be persisted as NULL despite the presence of column defaults, we can configure this for the ORM using a Core-level modifier *TypeEngine.evaluates_none()*, which indicates a type where the ORM should treat the value `None` the same as any other value and pass it through, rather than omitting it as a "missing" value:

```
class MyObject(Base):
    __tablename__ = 'my_table'
    id = Column(Integer, primary_key=True)
    data = Column(
      String(50).evaluates_none(),  # indicate that None should always be passed
      nullable=True, server_default="default")
```

```
obj = MyObject(id=1, data=None)
session.add(obj)
session.commit()  # INSERT with the 'data' column explicitly set to None;
                  # the ORM uses this directly, bypassing all client-
                  # and server-side defaults, and the database will
                  # persist this as the NULL value
```

Evaluating None

The `TypeEngine.evaluates_none()` modifier is primarily intended to signal a type where the Python value "None" is significant, the primary example being a JSON type which may want to persist the JSON `null` value rather than SQL NULL. We are slightly repurposing it here in order to signal to the ORM that we'd like `None` to be passed into the type whenever present, even though no special type-level behaviors are assigned to it.

New in version 1.1: added the `TypeEngine.evaluates_none()` method in order to indicate that a "None" value should be treated as significant.

Partitioning Strategies

Simple Vertical Partitioning

Vertical partitioning places different kinds of objects, or different tables, across multiple databases:

```
engine1 = create_engine('postgresql://db1')
engine2 = create_engine('postgresql://db2')

Session = sessionmaker(twophase=True)

# bind User operations to engine 1, Account operations to engine 2
Session.configure(binds={User:engine1, Account:engine2})

session = Session()
```

Above, operations against either class will make usage of the `Engine` linked to that class. Upon a flush operation, similar rules take place to ensure each class is written to the right database.

The transactions among the multiple databases can optionally be coordinated via two phase commit, if the underlying backend supports it. See *Enabling Two-Phase Commit* for an example.

Custom Vertical Partitioning

More comprehensive rule-based class-level partitioning can be built by overriding the `Session.get_bind()` method. Below we illustrate a custom `Session` which delivers the following rules:

1. Flush operations are delivered to the engine named `master`.
2. Operations on objects that subclass `MyOtherClass` all occur on the `other` engine.
3. Read operations for all other classes occur on a random choice of the `slave1` or `slave2` database.

```
engines = {
    'master':create_engine("sqlite:///master.db"),
    'other':create_engine("sqlite:///other.db"),
    'slave1':create_engine("sqlite:///slave1.db"),
```

```
    'slave2':create_engine("sqlite:///slave2.db"),
}

from sqlalchemy.orm import Session, sessionmaker
import random

class RoutingSession(Session):
    def get_bind(self, mapper=None, clause=None):
        if mapper and issubclass(mapper.class_, MyOtherClass):
            return engines['other']
        elif self._flushing:
            return engines['master']
        else:
            return engines[
                random.choice(['slave1','slave2'])
            ]
```

The above *Session* class is plugged in using the `class_` argument to *sessionmaker*:

```
Session = sessionmaker(class_=RoutingSession)
```

This approach can be combined with multiple *MetaData* objects, using an approach such as that of using the declarative `__abstract__` keyword, described at *__abstract__*.

Horizontal Partitioning

Horizontal partitioning partitions the rows of a single table (or a set of tables) across multiple databases.

See the "sharding" example: *Horizontal Sharding*.

Bulk Operations

Note: Bulk Operations mode is a new series of operations made available on the *Session* object for the purpose of invoking INSERT and UPDATE statements with greatly reduced Python overhead, at the expense of much less functionality, automation, and error checking. As of SQLAlchemy 1.0, these features should be considered as "beta", and additionally are intended for advanced users.

New in version 1.0.0.

Bulk operations on the *Session* include *Session.bulk_save_objects()*, *Session.bulk_insert_mappings()*, and *Session.bulk_update_mappings()*. The purpose of these methods is to directly expose internal elements of the unit of work system, such that facilities for emitting INSERT and UPDATE statements given dictionaries or object states can be utilized alone, bypassing the normal unit of work mechanics of state, relationship and attribute management. The advantages to this approach is strictly one of reduced Python overhead:

- The flush() process, including the survey of all objects, their state, their cascade status, the status of all objects associated with them via *relationship()*, and the topological sort of all operations to be performed is completely bypassed. This reduces a great amount of Python overhead.
- The objects as given have no defined relationship to the target *Session*, even when the operation is complete, meaning there's no overhead in attaching them or managing their state in terms of the identity map or session.
- The *Session.bulk_insert_mappings()* and *Session.bulk_update_mappings()* methods accept lists of plain Python dictionaries, not objects; this further reduces a large amount of overhead associ-

ated with instantiating mapped objects and assigning state to them, which normally is also subject to expensive tracking of history on a per-attribute basis.

- The process of fetching primary keys after an INSERT also is disabled by default. When performed correctly, INSERT statements can now more readily be batched by the unit of work process into `executemany()` blocks, which perform vastly better than individual statement invocations.
- UPDATE statements can similarly be tailored such that all attributes are subject to the SET clase unconditionally, again making it much more likely that `executemany()` blocks can be used.

The performance behavior of the bulk routines should be studied using the *Performance* example suite. This is a series of example scripts which illustrate Python call-counts across a variety of scenarios, including bulk insert and update scenarios.

See also:

Performance - includes detailed examples of bulk operations contrasted against traditional Core and ORM methods, including performance metrics.

Usage

The methods each work in the context of the `Session` object's transaction, like any other:

```
s = Session()
objects = [
    User(name="u1"),
    User(name="u2"),
    User(name="u3")
]
s.bulk_save_objects(objects)
```

For `Session.bulk_insert_mappings()`, and `Session.bulk_update_mappings()`, dictionaries are passed:

```
s.bulk_insert_mappings(User,
  [dict(name="u1"), dict(name="u2"), dict(name="u3")]
)
```

See also:

`Session.bulk_save_objects()`

`Session.bulk_insert_mappings()`

`Session.bulk_update_mappings()`

Comparison to Core Insert / Update Constructs

The bulk methods offer performance that under particular circumstances can be close to that of using the core `Insert` and `Update` constructs in an "executemany" context (for a description of "executemany", see *Executing Multiple Statements* in the Core tutorial). In order to achieve this, the `Session.bulk_insert_mappings.return_defaults` flag should be disabled so that rows can be batched together. The example suite in *Performance* should be carefully studied in order to gain familiarity with how fast bulk performance can be achieved.

ORM Compatibility

The bulk insert / update methods lose a significant amount of functionality versus traditional ORM use. The following is a listing of features that are **not available** when using these methods:

- persistence along `relationship()` linkages
- sorting of rows within order of dependency; rows are inserted or updated directly in the order in which they are passed to the methods
- Session-management on the given objects, including attachment to the session, identity map management.
- Functionality related to primary key mutation, ON UPDATE cascade
- SQL expression inserts / updates (e.g. *Embedding SQL Insert/Update Expressions into a Flush*)
- ORM events such as `MapperEvents.before_insert()`, etc. The bulk session methods have no event support.

Features that **are available** include:

- INSERTs and UPDATEs of mapped objects
- Version identifier support
- Multi-table mappings, such as joined-inheritance - however, an object to be inserted across multiple tables either needs to have primary key identifiers fully populated ahead of time, else the `Session.bulk_save_objects.return_defaults` flag must be used, which will greatly reduce the performance benefits

2.5.6 Contextual/Thread-local Sessions

Recall from the section *When do I construct a Session, when do I commit it, and when do I close it?*, the concept of "session scopes" was introduced, with an emphasis on web applications and the practice of linking the scope of a `Session` with that of a web request. Most modern web frameworks include integration tools so that the scope of the `Session` can be managed automatically, and these tools should be used as they are available.

SQLAlchemy includes its own helper object, which helps with the establishment of user-defined `Session` scopes. It is also used by third-party integration systems to help construct their integration schemes.

The object is the `scoped_session` object, and it represents a **registry** of `Session` objects. If you're not familiar with the registry pattern, a good introduction can be found in Patterns of Enterprise Architecture.

Note: The `scoped_session` object is a very popular and useful object used by many SQLAlchemy applications. However, it is important to note that it presents **only one approach** to the issue of `Session` management. If you're new to SQLAlchemy, and especially if the term "thread-local variable" seems strange to you, we recommend that if possible you familiarize first with an off-the-shelf integration system such as Flask-SQLAlchemy or zope.sqlalchemy.

A `scoped_session` is constructed by calling it, passing it a **factory** which can create new `Session` objects. A factory is just something that produces a new object when called, and in the case of `Session`, the most common factory is the `sessionmaker`, introduced earlier in this section. Below we illustrate this usage:

```
>>> from sqlalchemy.orm import scoped_session
>>> from sqlalchemy.orm import sessionmaker

>>> session_factory = sessionmaker(bind=some_engine)
>>> Session = scoped_session(session_factory)
```

The `scoped_session` object we've created will now call upon the `sessionmaker` when we "call" the registry:

```
>>> some_session = Session()
```

Above, `some_session` is an instance of *Session*, which we can now use to talk to the database. This same *Session* is also present within the *scoped_session* registry we've created. If we call upon the registry a second time, we get back the **same** *Session*:

```
>>> some_other_session = Session()
>>> some_session is some_other_session
True
```

This pattern allows disparate sections of the application to call upon a global *scoped_session*, so that all those areas may share the same session without the need to pass it explicitly. The *Session* we've established in our registry will remain, until we explicitly tell our registry to dispose of it, by calling *scoped_session.remove()*:

```
>>> Session.remove()
```

The *scoped_session.remove()* method first calls *Session.close()* on the current *Session*, which has the effect of releasing any connection/transactional resources owned by the *Session* first, then discarding the *Session* itself. "Releasing" here means that connections are returned to their connection pool and any transactional state is rolled back, ultimately using the `rollback()` method of the underlying DBAPI connection.

At this point, the *scoped_session* object is "empty", and will create a **new** *Session* when called again. As illustrated below, this is not the same *Session* we had before:

```
>>> new_session = Session()
>>> new_session is some_session
False
```

The above series of steps illustrates the idea of the "registry" pattern in a nutshell. With that basic idea in hand, we can discuss some of the details of how this pattern proceeds.

Implicit Method Access

The job of the *scoped_session* is simple; hold onto a *Session* for all who ask for it. As a means of producing more transparent access to this *Session*, the *scoped_session* also includes **proxy behavior**, meaning that the registry itself can be treated just like a *Session* directly; when methods are called on this object, they are **proxied** to the underlying *Session* being maintained by the registry:

```
Session = scoped_session(some_factory)

# equivalent to:
#
# session = Session()
# print session.query(MyClass).all()
#
print Session.query(MyClass).all()
```

The above code accomplishes the same task as that of acquiring the current *Session* by calling upon the registry, then using that *Session*.

Thread-Local Scope

Users who are familiar with multithreaded programming will note that representing anything as a global variable is usually a bad idea, as it implies that the global object will be accessed by many threads concurrently. The *Session* object is entirely designed to be used in a **non-concurrent** fashion, which in terms of multithreading means "only in one thread at a time". So our above example of *scoped_session* usage, where the same *Session* object is

maintained across multiple calls, suggests that some process needs to be in place such that mutltiple calls across many threads don't actually get a handle to the same session. We call this notion **thread local storage**, which means, a special object is used that will maintain a distinct object per each application thread. Python provides this via the threading.local() construct. The `scoped_session` object by default uses this object as storage, so that a single `Session` is maintained for all who call upon the `scoped_session` registry, but only within the scope of a single thread. Callers who call upon the registry in a different thread get a `Session` instance that is local to that other thread.

Using this technique, the `scoped_session` provides a quick and relatively simple (if one is familiar with thread-local storage) way of providing a single, global object in an application that is safe to be called upon from multiple threads.

The `scoped_session.remove()` method, as always, removes the current `Session` associated with the thread, if any. However, one advantage of the `threading.local()` object is that if the application thread itself ends, the "storage" for that thread is also garbage collected. So it is in fact "safe" to use thread local scope with an application that spawns and tears down threads, without the need to call `scoped_session.remove()`. However, the scope of transactions themselves, i.e. ending them via `Session.commit()` or `Session.rollback()`, will usually still be something that must be explicitly arranged for at the appropriate time, unless the application actually ties the lifespan of a thread to the lifespan of a transaction.

Using Thread-Local Scope with Web Applications

As discussed in the section *When do I construct a Session, when do I commit it, and when do I close it?*, a web application is architected around the concept of a **web request**, and integrating such an application with the `Session` usually implies that the `Session` will be associated with that request. As it turns out, most Python web frameworks, with notable exceptions such as the asynchronous frameworks Twisted and Tornado, use threads in a simple way, such that a particular web request is received, processed, and completed within the scope of a single *worker thread*. When the request ends, the worker thread is released to a pool of workers where it is available to handle another request.

This simple correspondence of web request and thread means that to associate a `Session` with a thread implies it is also associated with the web request running within that thread, and vice versa, provided that the `Session` is created only after the web request begins and torn down just before the web request ends. So it is a common practice to use `scoped_session` as a quick way to integrate the `Session` with a web application. The sequence diagram below illustrates this flow:

```
Web Server          Web Framework        SQLAlchemy ORM Code
--------------      --------------       ------------------------------
startup        ->   Web framework        # Session registry is established
                    initializes          Session = scoped_session(sessionmaker())


incoming
web request    ->   web request     ->   # The registry is *optionally*
                    starts               # called upon explicitly to create
                                         # a Session local to the thread and/or request
                                         Session()

                                         # the Session registry can otherwise
                                         # be used at any time, creating the
                                         # request-local Session() if not present,
                                         # or returning the existing one
                                         Session.query(MyClass) # ...

                                         Session.add(some_object) # ...

                                         # if data was modified, commit the
                                         # transaction
                                         Session.commit()
```

```
                    web request ends  -> # the registry is instructed to
                                         # remove the Session
                                         Session.remove()

                    sends output      <-
outgoing web    <-
response
```

Using the above flow, the process of integrating the `Session` with the web application has exactly two requirements:

1. Create a single `scoped_session` registry when the web application first starts, ensuring that this object is accessible by the rest of the application.
2. Ensure that `scoped_session.remove()` is called when the web request ends, usually by integrating with the web framework's event system to establish an "on request end" event.

As noted earlier, the above pattern is **just one potential way** to integrate a `Session` with a web framework, one which in particular makes the significant assumption that the **web framework associates web requests with application threads**. It is however **strongly recommended that the integration tools provided with the web framework itself be used, if available**, instead of `scoped_session`.

In particular, while using a thread local can be convenient, it is preferable that the `Session` be associated **directly with the request**, rather than with the current thread. The next section on custom scopes details a more advanced configuration which can combine the usage of `scoped_session` with direct request based scope, or any kind of scope.

Using Custom Created Scopes

The `scoped_session` object's default behavior of "thread local" scope is only one of many options on how to "scope" a `Session`. A custom scope can be defined based on any existing system of getting at "the current thing we are working with".

Suppose a web framework defines a library function `get_current_request()`. An application built using this framework can call this function at any time, and the result will be some kind of `Request` object that represents the current request being processed. If the `Request` object is hashable, then this function can be easily integrated with `scoped_session` to associate the `Session` with the request. Below we illustrate this in conjunction with a hypothetical event marker provided by the web framework `on_request_end`, which allows code to be invoked whenever a request ends:

```
from my_web_framework import get_current_request, on_request_end
from sqlalchemy.orm import scoped_session, sessionmaker

Session = scoped_session(sessionmaker(bind=some_engine), scopefunc=get_current_request)

@on_request_end
def remove_session(req):
    Session.remove()
```

Above, we instantiate `scoped_session` in the usual way, except that we pass our request-returning function as the "scopefunc". This instructs `scoped_session` to use this function to generate a dictionary key whenever the registry is called upon to return the current `Session`. In this case it is particularly important that we ensure a reliable "remove" system is implemented, as this dictionary is not otherwise self-managed.

Contextual Session API

class `sqlalchemy.orm.scoping.`**`scoped_session`**(*session_factory*, *scopefunc=None*)

Provides scoped management of `Session` objects.

See *Contextual/Thread-local Sessions* for a tutorial.

__call__(***kw*)

Return the current `Session`, creating it using the `scoped_session.session_factory` if not present.

Parameters ****kw** – Keyword arguments will be passed to the `scoped_session.session_factory` callable, if an existing `Session` is not present. If the `Session` is present and keyword arguments have been passed, `InvalidRequestError` is raised.

__init__(*session_factory*, *scopefunc=None*)

Construct a new `scoped_session`.

Parameters

- **session_factory** – a factory to create new `Session` instances. This is usually, but not necessarily, an instance of `sessionmaker`.
- **scopefunc** – optional function which defines the current scope. If not passed, the `scoped_session` object assumes "thread-local" scope, and will use a Python `threading.local()` in order to maintain the current `Session`. If passed, the function should return a hashable token; this token will be used as the key in a dictionary in order to store and retrieve the current `Session`.

configure(***kwargs*)

reconfigure the `sessionmaker` used by this `scoped_session`.

See `sessionmaker.configure()`.

query_property(*query_cls=None*)

return a class property which produces a `Query` object against the class and the current `Session` when called.

e.g.:

```
Session = scoped_session(sessionmaker())

class MyClass(object):
    query = Session.query_property()

# after mappers are defined
result = MyClass.query.filter(MyClass.name=='foo').all()
```

Produces instances of the session's configured query class by default. To override and use a custom implementation, provide a `query_cls` callable. The callable will be invoked with the class's mapper as a positional argument and a session keyword argument.

There is no limit to the number of query properties placed on a class.

remove()

Dispose of the current `Session`, if present.

This will first call `Session.close()` method on the current `Session`, which releases any existing transactional/connection resources still being held; transactions specifically are rolled back. The `Session` is then discarded. Upon next usage within the same scope, the `scoped_session` will produce a new `Session` object.

session_factory = None

The *session_factory* provided to *__init__* is stored in this attribute and may be accessed at a later time. This can be useful when a new non-scoped `Session` or `Connection` to the database is needed.

class `sqlalchemy.util.`**`ScopedRegistry`**(*createfunc*, *scopefunc*)

A Registry that can store one or multiple instances of a single class on the basis of a "scope" function.

The object implements `__call__` as the "getter", so by calling `myregistry()` the contained object is returned for the current scope.

Parameters

- **`createfunc`** – a callable that returns a new object to be placed in the registry
- **`scopefunc`** – a callable that will return a key to store/retrieve an object.

`__init__`(*createfunc*, *scopefunc*)

Construct a new *`ScopedRegistry`*.

Parameters

- **`createfunc`** – A creation function that will generate a new value for the current scope, if none is present.
- **`scopefunc`** – A function that returns a hashable token representing the current scope (such as, current thread identifier).

`clear`()

Clear the current scope, if any.

`has`()

Return True if an object is present in the current scope.

`set`(*obj*)

Set the value for the current scope.

class `sqlalchemy.util.`**`ThreadLocalRegistry`**(*createfunc*)

Bases: `sqlalchemy.util._collections.ScopedRegistry`

A *`ScopedRegistry`* that uses a `threading.local()` variable for storage.

2.5.7 Tracking Object and Session Changes with Events

SQLAlchemy features an extensive *Event Listening* system used throughout the Core and ORM. Within the ORM, there are a wide variety of event listener hooks, which are documented at an API level at *ORM Events*. This collection of events has grown over the years to include lots of very useful new events as well as some older events that aren't as relevant as they once were. This section will attempt to introduce the major event hooks and when they might be used.

Persistence Events

Probably the most widely used series of events are the "persistence" events, which correspond to the *flush process*. The flush is where all the decisions are made about pending changes to objects and are then emitted out to the database in the form of INSERT, UPDATE, and DELETE staetments.

`before_flush()`

The *`SessionEvents.before_flush()`* hook is by far the most generally useful event to use when an application wants to ensure that additional persistence changes to the database are made when a flush proceeds. Use *`SessionEvents.before_flush()`* in order to operate upon objects to validate their state as well as to compose additional objects and references before they are persisted. Within this event, it is **safe to manipulate the Session's state**, that is, new objects can be attached to it, objects can be deleted, and indivual attributes on objects can be changed freely, and these changes will be pulled into the flush process when the event hook completes.

The typical `SessionEvents.before_flush()` hook will be tasked with scanning the collections `Session.new`, `Session.dirty` and `Session.deleted` in order to look for objects where something will be happening.

For illustrations of `SessionEvents.before_flush()`, see examples such as *Versioning with a History Table* and *Versioning using Temporal Rows*.

after_flush()

The `SessionEvents.after_flush()` hook is called after the SQL has been emitted for a flush process, but **before** the state of the objects that were flushed has been altered. That is, you can still inspect the `Session.new`, `Session.dirty` and `Session.deleted` collections to see what was just flushed, and you can also use history tracking features like the ones provided by `AttributeState` to see what changes were just persisted. In the `SessionEvents.after_flush()` event, additional SQL can be emitted to the database based on what's observed to have changed.

after_flush_postexec()

`SessionEvents.after_flush_postexec()` is called soon after `SessionEvents.after_flush()`, but is invoked **after** the state of the objects has been modified to account for the flush that just took place. The `Session.new`, `Session.dirty` and `Session.deleted` collections are normally completely empty here. Use `SessionEvents.after_flush_postexec()` to inspect the identity map for finalized objects and possibly emit additional SQL. In this hook, there is the ability to make new changes on objects, which means the `Session` will again go into a "dirty" state; the mechanics of the `Session` here will cause it to flush **again** if new changes are detected in this hook if the flush were invoked in the context of `Session.commit()`; otherwise, the pending changes will be bundled as part of the next normal flush. When the hook detects new changes within a `Session.commit()`, a counter ensures that an endless loop in this regard is stopped after 100 iterations, in the case that an `SessionEvents.after_flush_postexec()` hook continually adds new state to be flushed each time it is called.

Mapper-level Events

In addition to the flush-level hooks, there is also a suite of hooks that are more fine-grained, in that they are called on a per-object basis and are broken out based on INSERT, UPDATE or DELETE. These are the mapper persistence hooks, and they too are very popular, however these events need to be approached more cautiously, as they proceed within the context of the flush process that is already ongoing; many operations are not safe to proceed here.

The events are:

- `MapperEvents.before_insert()`
- `MapperEvents.after_insert()`
- `MapperEvents.before_update()`
- `MapperEvents.after_update()`
- `MapperEvents.before_delete()`
- `MapperEvents.after_delete()`

Each event is passed the `Mapper`, the mapped object itself, and the `Connection` which is being used to emit an INSERT, UPDATE or DELETE statement. The appeal of these events is clear, in that if an application wants to tie some activity to when a specific type of object is persisted with an INSERT, the hook is very specific; unlike the `SessionEvents.before_flush()` event, there's no need to search through collections like `Session.new` in order to find targets. However, the flush plan which represents the full list of every single INSERT, UPDATE, DELETE

statement to be emitted has *already been decided* when these events are called, and no changes may be made at this stage. Therefore the only changes that are even possible to the given objects are upon attributes **local** to the object's row. Any other change to the object or other objects will impact the state of the `Session`, which will fail to function properly.

Operations that are not supported within these mapper-level persistence events include:

- `Session.add()`
- `Session.delete()`
- Mapped collection append, add, remove, delete, discard, etc.
- Mapped relationship attribute set/del events, i.e. `someobject.related = someotherobject`

The reason the `Connection` is passed is that it is encouraged that **simple SQL operations take place here**, directly on the `Connection`, such as incrementing counters or inserting extra rows within log tables. When dealing with the `Connection`, it is expected that Core-level SQL operations will be used; e.g. those described in *SQL Expression Language Tutorial*.

There are also many per-object operations that don't need to be handled within a flush event at all. The most common alternative is to simply establish additional state along with an object inside its `__init__()` method, such as creating additional objects that are to be associated with the new object. Using validators as described in *Simple Validators* is another approach; these functions can intercept changes to attributes and establish additional state changes on the target object in response to the attribute change. With both of these approaches, the object is in the correct state before it ever gets to the flush step.

Object Lifecycle Events

Another use case for events is to track the lifecycle of objects. This refers to the states first introduced at *Quickie Intro to Object States*.

New in version 1.1: added a system of events that intercept all possible state transitions of an object within the `Session`.

All the states above can be tracked fully with events. Each event represents a distinct state transition, meaning, the starting state and the destination state are both part of what are tracked. With the exception of the initial transient event, all the events are in terms of the `Session` object or class, meaning they can be associated either with a specific `Session` object:

```
from sqlalchemy import event
from sqlalchemy.orm import Session

session = Session()

@event.listens_for(session, 'transient_to_pending')
def object_is_pending(session, obj):
    print("new pending: %s" % obj)
```

Or with the `Session` class itself, as well as with a specific `sessionmaker`, which is likely the most useful form:

```
from sqlalchemy import event
from sqlalchemy.orm import sessionmaker

maker = sessionmaker()

@event.listens_for(maker, 'transient_to_pending')
def object_is_pending(session, obj):
    print("new pending: %s" % obj)
```

The listeners can of course be stacked on top of one function, as is likely to be common. For example, to track all objects that are entering the persistent state:

```
@event.listens_for(maker, "pending_to_persistent")
@event.listens_for(maker, "deleted_to_persistent")
@event.listens_for(maker, "detached_to_persistent")
@event.listens_for(maker, "loaded_as_persistent")
def detect_all_persistent(session, instance):
    print("object is now persistent: %s" % instance)
```

Transient

All mapped objects when first constructed start out as transient. In this state, the object exists alone and doesn't have an association with any *Session*. For this initial state, there's no specific "transition" event since there is no *Session*, however if one wanted to intercept when any transient object is created, the *InstanceEvents.init()* method is probably the best event. This event is applied to a specific class or superclass. For example, to intercept all new objects for a particular declarative base:

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import event

Base = declarative_base()

@event.listens_for(Base, "init", propagate=True)
def intercept_init(instance, args, kwargs):
    print("new transient: %s" % instance)
```

Transient to Pending

The transient object becomes pending when it is first associated with a *Session* via the *Session.add()* or *Session.add_all()* method. An object may also become part of a *Session* as a result of a *"cascade"* from a referencing object that was explicitly added. The transient to pending transition is detectable using the *SessionEvents.transient_to_pending()* event:

```
@event.listens_for(sessionmaker, "transient_to_pending")
def intercept_transient_to_pending(session, object_):
    print("transient to pending: %s" % object_)
```

Pending to Persistent

The pending object becomes persistent when a flush proceeds and an INSERT statement takes place for the instance. The object now has an identity key. Track pending to persistent with the *SessionEvents.pending_to_persistent()* event:

```
@event.listens_for(sessionmaker, "pending_to_persistent")
def intercept_pending_to_persistent(session, object_):
    print("pending to persistent: %s" % object_)
```

Pending to Transient

The pending object can revert back to transient if the *Session.rollback()* method is called before the pending object has been flushed, or if the *Session.expunge()* method is called for the object before it is flushed. Track

pending to transient with the *SessionEvents.pending_to_transient()* event:

```
@event.listens_for(sessionmaker, "pending_to_transient")
def intercept_pending_to_transient(session, object_):
    print("transient to pending: %s" % object_)
```

Loaded as Persistent

Objects can appear in the *Session* directly in the persistent state when they are loaded from the database. Tracking this state transition is synonymous with tracking objects as they are loaded, and is synonomous with using the *InstanceEvents.load()* instance-level event. However, the *SessionEvents.loaded_as_persistent()* event is provided as a session-centric hook for intercepting objects as they enter the persistent state via this particular avenue:

```
@event.listens_for(sessionmaker, "loaded_as_persistent")
def intercept_loaded_as_persistent(session, object_):
    print("object loaded into persistent state: %s" % object_)
```

Persistent to Transient

The persistent object can revert to the transient state if the *Session.rollback()* method is called for a transaction where the object was first added as pending. In the case of the ROLLBACK, the INSERT statement that made this object persistent is rolled back, and the object is evicted from the *Session* to again become transient. Track objects that were reverted to transient from persistent using the *SessionEvents.persistent_to_transient()* event hook:

```
@event.listens_for(sessionmaker, "persistent_to_transient")
def intercept_persistent_to_transient(session, object_):
    print("persistent to transient: %s" % object_)
```

Persistent to Deleted

The persistent object enters the deleted state when an object marked for deletion is deleted from the database within the flush process. Note that this is **not the same** as when the *Session.delete()* method is called for a target object. The *Session.delete()* method only **marks** the object for deletion; the actual DELETE statement is not emitted until the flush proceeds. It is subsequent to the flush that the "deleted" state is present for the target object.

Within the "deleted" state, the object is only marginally associated with the *Session*. It is not present in the identity map nor is it present in the *Session.deleted* collection that refers to when it was pending for deletion.

From the "deleted" state, the object can go either to the detached state when the transaction is committed, or back to the persistent state if the transaction is instead rolled back.

Track the persistent to deleted transition with *SessionEvents.persistent_to_deleted()*:

```
@event.listens_for(sessionmaker, "persistent_to_deleted")
def intercept_persistent_to_deleted(session, object_):
    print("object was DELETEd, is now in deleted state: %s" % object_)
```

Deleted to Detached

The deleted object becomes detached when the session's transaction is committed. After the *Session.commit()* method is called, the database transaction is final and the *Session* now fully dis-

cards the deleted object and removes all associations to it. Track the deleted to detached transition using *SessionEvents.deleted_to_detached()*:

```
@event.listens_for(sessionmaker, "deleted_to_detached")
def intercept_deleted_to_detached(session, object_):
    print("deleted to detached: %s" % object_)
```

Note: While the object is in the deleted state, the *InstanceState.deleted* attribute, accessible using `inspect(object).deleted`, returns True. However when the object is detached, *InstanceState.deleted* will again return False. To detect that an object was deleted, regardless of whether or not it is detached, use the *InstanceState.was_deleted* accessor.

Persistent to Detached

The persistent object becomes detached when the object is de-associated with the *Session*, via the *Session.expunge()*, *Session.expunge_all()*, or *Session.close()* methods.

Note: An object may also become **implicitly detached** if its owning *Session* is dereferenced by the application and discarded due to garbage collection. In this case, **no event is emitted**.

Track objects as they move from persistent to detached using the *SessionEvents.persistent_to_detached()* event:

```
@event.listens_for(sessionmaker, "persistent_to_detached")
def intecept_persistent_to_detached(session, object_):
    print("object became detached: %s" % object_)
```

Detached to Persistent

The detached object becomes persistent when it is re-associated with a session using the *Session.add()* or equivalent method. Track objects moving back to persistent from detached using the *SessionEvents.detached_to_persistent()* event:

```
@event.listens_for(sessionmaker, "detached_to_persistent")
def intecept_detached_to_persistent(session, object_):
    print("object became persistent again: %s" % object_)
```

Deleted to Persistent

The deleted object can be reverted to the persistent state when the transaction in which it was DELETEd was rolled back using the *Session.rollback()* method. Track deleted objects moving back to the persistent state using the *SessionEvents.deleted_to_persistent()* event:

```
@event.listens_for(sessionmaker, "transient_to_pending")
def intercept_transient_to_pending(session, object_):
    print("transient to pending: %s" % object_)
```

Transaction Events

Transaction events allow an application to be notifed when transaction boundaries occur at the *Session* level as well as when the *Session* changes the transactional state on *Connection* objects.

- `SessionEvents.after_transaction_create()`, `SessionEvents.after_transaction_end()` - these events track the logical transaction scopes of the `Session` in a way that is not specific to individual database connections. These events are intended to help with integration of transaction-tracking systems such as `zope.sqlalchemy`. Use these events when the application needs to align some external scope with the transactional scope of the `Session`. These hooks mirror the "nested" transactional behavior of the `Session`, in that they track logical "subtransactions" as well as "nested" (e.g. SAVEPOINT) transactions.
- `SessionEvents.before_commit()`, `SessionEvents.after_commit()`, `SessionEvents.after_begin()`, `SessionEvents.after_rollback()`, `SessionEvents.after_soft_rollback()` - These events allow tracking of transaction events from the perspective of database connections. `SessionEvents.after_begin()` in particular is a per-connection event; a `Session` that maintains more than one connection will emit this event for each connection individually as those connections become used within the current transaction. The rollback and commit events then refer to when the DBAPI connections themselves have received rollback or commit instructions directly.

Attribute Change Events

The attribute change events allow interception of when specific attributes on an object are modified. These events include `AttributeEvents.set()`, `AttributeEvents.append()`, and `AttributeEvents.remove()`. These events are extremely useful, particularly for per-object validation operations; however, it is often much more convenient to use a "validator" hook, which uses these hooks behind the scenes; see *Simple Validators* for background on this. The attribute events are also behind the mechanics of backreferences. An example illustrating use of attribute events is in *Attribute Instrumentation*.

2.5.8 Session API

Session and sessionmaker()

class `sqlalchemy.orm.session.`**`sessionmaker`**(*bind=None*, *class_=<class 'sqlalchemy.orm.session.Session'>*, *autoflush=True*, *autocommit=False*, *expire_on_commit=True*, *info=None*, ***kw*)

Bases: `sqlalchemy.orm.session._SessionClassMethods`

A configurable `Session` factory.

The `sessionmaker` factory generates new `Session` objects when called, creating them given the configurational arguments established here.

e.g.:

```
# global scope
Session = sessionmaker(autoflush=False)

# later, in a local scope, create and use a session:
sess = Session()
```

Any keyword arguments sent to the constructor itself will override the "configured" keywords:

```
Session = sessionmaker()

# bind an individual session to a connection
sess = Session(bind=connection)
```

The class also includes a method `configure()`, which can be used to specify additional keyword arguments to the factory, which will take effect for subsequent `Session` objects generated. This is usually used to associate one or more `Engine` objects with an existing `sessionmaker` factory before it is first used:

```
# application starts
Session = sessionmaker()

# ... later
engine = create_engine('sqlite:///foo.db')
Session.configure(bind=engine)

sess = Session()
```

__call__(***local_kw*)

Produce a new `Session` object using the configuration established in this `sessionmaker`.

In Python, the `__call__` method is invoked on an object when it is "called" in the same way as a function:

```
Session = sessionmaker()
session = Session()  # invokes sessionmaker.__call__()
```

__init__(*bind=None*, *class_=<class 'sqlalchemy.orm.session.Session'>*, *autoflush=True*, *autocommit=False*, *expire_on_commit=True*, *info=None*, ***kw*)

Construct a new `sessionmaker`.

All arguments here except for `class_` correspond to arguments accepted by `Session` directly. See the `Session.__init__()` docstring for more details on parameters.

Parameters

- **bind** – a `Engine` or other `Connectable` with which newly created `Session` objects will be associated.
- **class_** – class to use in order to create new `Session` objects. Defaults to `Session`.
- **autoflush** – The autoflush setting to use with newly created `Session` objects.
- **autocommit** – The autocommit setting to use with newly created `Session` objects.
- **expire_on_commit=True** – the expire_on_commit setting to use with newly created `Session` objects.
- **info** – optional dictionary of information that will be available via `Session.info`. Note this dictionary is *updated*, not replaced, when the `info` parameter is specified to the specific `Session` construction operation.

  New in version 0.9.0.
- ****kw** – all other keyword arguments are passed to the constructor of newly created `Session` objects.

close_all()

inherited from the `close_all()` *method of* `_SessionClassMethods`

Close *all* sessions in memory.

configure(***new_kw*)

(Re)configure the arguments for this sessionmaker.

e.g.:

```
Session = sessionmaker()

Session.configure(bind=create_engine('sqlite://'))
```

identity_key(**args*, ***kwargs*)

inherited from the `identity_key()` *method of* `_SessionClassMethods`

Return an identity key.

This is an alias of `util.identity_key()`.

object_session(*instance*)

inherited from the `object_session()` *method of* `_SessionClassMethods`

Return the `Session` to which an object belongs.

This is an alias of `object_session()`.

class `sqlalchemy.orm.session.`**Session**(*bind=None*, *autoflush=True*, *expire_on_commit=True*, *_enable_transaction_accounting=True*, *autocommit=False*, *twophase=False*, *weak_identity_map=True*, *binds=None*, *extension=None*, *info=None*, *query_cls=<class 'sqlalchemy.orm.query.Query'>*)

Bases: `sqlalchemy.orm.session._SessionClassMethods`

Manages persistence operations for ORM-mapped objects.

The Session's usage paradigm is described at Using the Session.

__init__(*bind=None*, *autoflush=True*, *expire_on_commit=True*, *_enable_transaction_accounting=True*, *autocommit=False*, *twophase=False*, *weak_identity_map=True*, *binds=None*, *extension=None*, *info=None*, *query_cls=<class 'sqlalchemy.orm.query.Query'>*)

Construct a new Session.

See also the `sessionmaker` function which is used to generate a `Session`-producing callable with a given set of arguments.

Parameters

- **autocommit** –

  > **Warning:** The autocommit flag is **not for general use**, and if it is used, queries should only be invoked within the span of a `Session.begin()` / `Session.commit()` pair. Executing queries outside of a demarcated transaction is a legacy mode of usage, and can in some cases lead to concurrent connection checkouts.

  Defaults to `False`. When `True`, the `Session` does not keep a persistent transaction running, and will acquire connections from the engine on an as-needed basis, returning them immediately after their use. Flushes will begin and commit (or possibly rollback) their own transaction if no transaction is present. When using this mode, the `Session.begin()` method is used to explicitly start transactions.

  See also:

  Autocommit Mode

- **autoflush** – When `True`, all query operations will issue a `flush()` call to this `Session` before proceeding. This is a convenience feature so that `flush()` need not be called repeatedly in order for database queries to retrieve results. It's typical that `autoflush` is used in conjunction with `autocommit=False`. In this scenario, explicit calls to `flush()` are rarely needed; you usually only need to call `commit()` (which flushes) to finalize changes.

- **bind** – An optional *Engine* or *Connection* to which this Session should be bound. When specified, all SQL operations performed by this session will execute via this connectable.
- **binds** –

  An optional dictionary which contains more granular "bind" information than the bind parameter provides. This dictionary can map individual :class'.Table' instances as well as *Mapper* instances to individual *Engine* or *Connection* objects. Operations which proceed relative to a particular *Mapper* will consult this dictionary for the direct *Mapper* instance as well as the mapper's mapped_table attribute in order to locate a connectable to use. The full resolution is described in the *Session.get_bind()*. Usage looks like:

```
Session = sessionmaker(binds={
    SomeMappedClass: create_engine('postgresql://engine1'),
    somemapper: create_engine('postgresql://engine2'),
    some_table: create_engine('postgresql://engine3'),
    })
```

  Also see the *Session.bind_mapper()* and *Session.bind_table()* methods.

- **class_** – Specify an alternate class other than sqlalchemy.orm.session.Session which should be used by the returned class. This is the only argument that is local to the *sessionmaker* function, and is not sent directly to the constructor for Session.
- **_enable_transaction_accounting** – Defaults to True. A legacy-only flag which when False disables *all* 0.5-style object accounting on transaction boundaries, including auto-expiry of instances on rollback and commit, maintenance of the "new" and "deleted" lists upon rollback, and autoflush of pending changes upon *begin()*, all of which are interdependent.
- **expire_on_commit** – Defaults to True. When True, all instances will be fully expired after each *commit()*, so that all attribute/object access subsequent to a completed transaction will load from the most recent database state.
- **extension** – An optional *SessionExtension* instance, or a list of such instances, which will receive pre- and post- commit and flush events, as well as a post-rollback event. **Deprecated.** Please see *SessionEvents*.
- **info** – optional dictionary of arbitrary data to be associated with this *Session*. Is available via the *Session.info* attribute. Note the dictionary is copied at construction time so that modifications to the per- *Session* dictionary will be local to that *Session*.

  New in version 0.9.0.

- **query_cls** – Class which should be used to create new Query objects, as returned by the *query()* method. Defaults to *Query*.
- **twophase** – When True, all transactions will be started as a "two phase" transaction, i.e. using the "two phase" semantics of the database in use along with an XID. During a *commit()*, after *flush()* has been issued for all attached databases, the *prepare()* method on each database's *TwoPhaseTransaction* will be called. This allows each database to roll back the entire transaction, before each transaction is committed.
- **weak_identity_map** – Defaults to True - when set to False, objects placed in the *Session* will be strongly referenced until explicitly removed or the *Session* is closed. **Deprecated** - The strong reference identity map is legacy. See the recipe at *Session Referencing Behavior* for an event-based approach to maintaining strong identity references.

add(*instance*, *_warn=True*)
Place an object in the `Session`.

Its state will be persisted to the database on the next flush operation.

Repeated calls to `add()` will be ignored. The opposite of `add()` is `expunge()`.

add_all(*instances*)
Add the given collection of instances to this `Session`.

begin(*subtransactions=False*, *nested=False*)
Begin a transaction on this *Session*.

If this Session is already within a transaction, either a plain transaction or nested transaction, an error is raised, unless `subtransactions=True` or `nested=True` is specified.

The `subtransactions=True` flag indicates that this *begin()* can create a subtransaction if a transaction is already in progress. For documentation on subtransactions, please see *Using Subtransactions with Autocommit*.

The `nested` flag begins a SAVEPOINT transaction and is equivalent to calling *begin_nested()*. For documentation on SAVEPOINT transactions, please see *Using SAVEPOINT*.

begin_nested()
Begin a *nested* transaction on this Session.

The target database(s) must support SQL SAVEPOINTs or a SQLAlchemy-supported vendor implementation of the idea.

For documentation on SAVEPOINT transactions, please see *Using SAVEPOINT*.

bind_mapper(*mapper*, *bind*)
Associate a *Mapper* with a "bind", e.g. a *Engine* or *Connection*.

The given mapper is added to a lookup used by the *Session.get_bind()* method.

bind_table(*table*, *bind*)
Associate a *Table* with a "bind", e.g. a *Engine* or *Connection*.

The given mapper is added to a lookup used by the *Session.get_bind()* method.

bulk_insert_mappings(*mapper*, *mappings*, *return_defaults=False*)
Perform a bulk insert of the given list of mapping dictionaries.

The bulk insert feature allows plain Python dictionaries to be used as the source of simple INSERT operations which can be more easily grouped together into higher performing "executemany" operations. Using dictionaries, there is no "history" or session state management features in use, reducing latency when inserting large numbers of simple rows.

The values within the dictionaries as given are typically passed without modification into Core `Insert()` constructs, after organizing the values within them across the tables to which the given mapper is mapped.

New in version 1.0.0.

> **Warning:** The bulk insert feature allows for a lower-latency INSERT of rows at the expense of most other unit-of-work features. Features such as object management, relationship handling, and SQL clause support are **silently omitted** in favor of raw INSERT of records.
> **Please read the list of caveats at *Bulk Operations* before using this method, and fully test and confirm the functionality of all code developed using these systems.**

Parameters

- **mapper** – a mapped class, or the actual *Mapper* object, representing the single kind of object represented within the mapping list.
- **mappings** – a list of dictionaries, each one containing the state of the mapped row to be inserted, in terms of the attribute names on the mapped class. If the mapping refers to multiple tables, such as a joined-inheritance mapping, each dictionary must contain all keys to be populated into all tables.
- **return_defaults** – when True, rows that are missing values which generate defaults, namely integer primary key defaults and sequences, will be inserted **one at a time**, so that the primary key value is available. In particular this will allow joined-inheritance and other multi-table mappings to insert correctly without the need to provide primary key values ahead of time; however, *Session.bulk_insert_mappings.return_defaults* **greatly reduces the performance gains** of the method overall. If the rows to be inserted only refer to a single table, then there is no reason this flag should be set as the returned default information is not used.

See also:

Bulk Operations

Session.bulk_save_objects()

Session.bulk_update_mappings()

bulk_save_objects(*objects*, *return_defaults=False*, *update_changed_only=True*)

Perform a bulk save of the given list of objects.

The bulk save feature allows mapped objects to be used as the source of simple INSERT and UPDATE operations which can be more easily grouped together into higher performing "executemany" operations; the extraction of data from the objects is also performed using a lower-latency process that ignores whether or not attributes have actually been modified in the case of UPDATEs, and also ignores SQL expressions.

The objects as given are not added to the session and no additional state is established on them, unless the `return_defaults` flag is also set, in which case primary key attributes and server-side default values will be populated.

New in version 1.0.0.

> **Warning:** The bulk save feature allows for a lower-latency INSERT/UPDATE of rows at the expense of most other unit-of-work features. Features such as object management, relationship handling, and SQL clause support are **silently omitted** in favor of raw INSERT/UPDATES of records.
> **Please read the list of caveats at *Bulk Operations* before using this method, and fully test and confirm the functionality of all code developed using these systems.**

Parameters

- **objects** – a list of mapped object instances. The mapped objects are persisted as is, and are **not** associated with the *Session* afterwards.

  For each object, whether the object is sent as an INSERT or an UPDATE is dependent on the same rules used by the *Session* in traditional operation; if the object has the `InstanceState.key` attribute set, then the object is assumed to be "detached" and will result in an UPDATE. Otherwise, an INSERT is used.

  In the case of an UPDATE, statements are grouped based on which attributes have changed, and are thus to be the subject of each SET clause. If `update_changed_only` is False, then all attributes present within each object are applied to the UPDATE statement, which may help in allowing the statements to

be grouped together into a larger executemany(), and will also reduce the overhead of checking history on attributes.

- **return_defaults** – when True, rows that are missing values which generate defaults, namely integer primary key defaults and sequences, will be inserted **one at a time**, so that the primary key value is available. In particular this will allow joined-inheritance and other multi-table mappings to insert correctly without the need to provide primary key values ahead of time; however, `Session.bulk_save_objects.return_defaults` **greatly reduces the performance gains** of the method overall.
- **update_changed_only** – when True, UPDATE statements are rendered based on those attributes in each state that have logged changes. When False, all attributes present are rendered into the SET clause with the exception of primary key attributes.

See also:

Bulk Operations

`Session.bulk_insert_mappings()`

`Session.bulk_update_mappings()`

bulk_update_mappings(*mapper*, *mappings*)

Perform a bulk update of the given list of mapping dictionaries.

The bulk update feature allows plain Python dictionaries to be used as the source of simple UPDATE operations which can be more easily grouped together into higher performing "executemany" operations. Using dictionaries, there is no "history" or session state management features in use, reducing latency when updating large numbers of simple rows.

New in version 1.0.0.

> **Warning:** The bulk update feature allows for a lower-latency UPDATE of rows at the expense of most other unit-of-work features. Features such as object management, relationship handling, and SQL clause support are **silently omitted** in favor of raw UPDATES of records.
> **Please read the list of caveats at** *Bulk Operations* **before using this method, and fully test and confirm the functionality of all code developed using these systems.**

Parameters

- **mapper** – a mapped class, or the actual `Mapper` object, representing the single kind of object represented within the mapping list.
- **mappings** – a list of dictionaries, each one containing the state of the mapped row to be updated, in terms of the attribute names on the mapped class. If the mapping refers to multiple tables, such as a joined-inheritance mapping, each dictionary may contain keys corresponding to all tables. All those keys which are present and are not part of the primary key are applied to the SET clause of the UPDATE statement; the primary key values, which are required, are applied to the WHERE clause.

See also:

Bulk Operations

`Session.bulk_insert_mappings()`

`Session.bulk_save_objects()`

close()

Close this Session.

This clears all items and ends any transaction in progress.

If this session were created with `autocommit=False`, a new transaction is immediately begun. Note that this new transaction does not use any connection resources until they are first needed.

close_all()

inherited from the `close_all()` *method of* `_SessionClassMethods`

Close *all* sessions in memory.

commit()

Flush pending changes and commit the current transaction.

If no transaction is in progress, this method raises an `InvalidRequestError`.

By default, the `Session` also expires all database loaded state on all ORM-managed attributes after transaction commit. This so that subsequent operations load the most recent data from the database. This behavior can be disabled using the `expire_on_commit=False` option to `sessionmaker` or the `Session` constructor.

If a subtransaction is in effect (which occurs when begin() is called multiple times), the subtransaction will be closed, and the next call to `commit()` will operate on the enclosing transaction.

When using the `Session` in its default mode of `autocommit=False`, a new transaction will be begun immediately after the commit, but note that the newly begun transaction does *not* use any connection resources until the first SQL is actually emitted.

See also:

Committing

connection(*mapper=None*, *clause=None*, *bind=None*, *close_with_result=False*, *execution_options=None*, ***kw*)

Return a `Connection` object corresponding to this `Session` object's transactional state.

If this `Session` is configured with `autocommit=False`, either the `Connection` corresponding to the current transaction is returned, or if no transaction is in progress, a new one is begun and the `Connection` returned (note that no transactional state is established with the DBAPI until the first SQL statement is emitted).

Alternatively, if this `Session` is configured with `autocommit=True`, an ad-hoc `Connection` is returned using `Engine.contextual_connect()` on the underlying `Engine`.

Ambiguity in multi-bind or unbound `Session` objects can be resolved through any of the optional keyword arguments. This ultimately makes usage of the `get_bind()` method for resolution.

Parameters

- **bind** – Optional `Engine` to be used as the bind. If this engine is already involved in an ongoing transaction, that connection will be used. This argument takes precedence over `mapper`, `clause`.
- **mapper** – Optional `mapper()` mapped class, used to identify the appropriate bind. This argument takes precedence over `clause`.
- **clause** – A `ClauseElement` (i.e. `select()`, `text()`, etc.) which will be used to locate a bind, if a bind cannot otherwise be identified.
- **close_with_result** – Passed to `Engine.connect()`, indicating the `Connection` should be considered "single use", automatically closing when the first result set is closed. This flag only has an effect if this `Session` is configured with `autocommit=True` and does not already have a transaction in progress.
- **execution_options** – a dictionary of execution options that will be passed to `Connection.execution_options()`, **when the connection is first procured**

only. If the connection is already present within the `Session`, a warning is emitted and the arguments are ignored.

New in version 0.9.9.

See also:

Setting Transaction Isolation Levels

- **`**kw`** – Additional keyword arguments are sent to `get_bind()`, allowing additional arguments to be passed to custom implementations of `get_bind()`.

`delete`(*instance*)

Mark an instance as deleted.

The database delete operation occurs upon `flush()`.

`deleted`

The set of all instances marked as 'deleted' within this `Session`

`dirty`

The set of all persistent instances considered dirty.

E.g.:

```
some_mapped_object in session.dirty
```

Instances are considered dirty when they were modified but not deleted.

Note that this 'dirty' calculation is 'optimistic'; most attribute-setting or collection modification operations will mark an instance as 'dirty' and place it in this set, even if there is no net change to the attribute's value. At flush time, the value of each attribute is compared to its previously saved value, and if there's no net change, no SQL operation will occur (this is a more expensive operation so it's only done at flush time).

To check if an instance has actionable net changes to its attributes, use the `Session.is_modified()` method.

`enable_relationship_loading`(*obj*)

Associate an object with this `Session` for related object loading.

> **Warning:** `enable_relationship_loading()` exists to serve special use cases and is not recommended for general use.

Accesses of attributes mapped with `relationship()` will attempt to load a value from the database using this `Session` as the source of connectivity. The values will be loaded based on foreign key values present on this object - it follows that this functionality generally only works for many-to-one-relationships.

The object will be attached to this session, but will **not** participate in any persistence operations; its state for almost all purposes will remain either "transient" or "detached", except for the case of relationship loading.

Also note that backrefs will often not work as expected. Altering a relationship-bound attribute on the target object may not fire off a backref event, if the effective value is what was already loaded from a foreign-key-holding value.

The `Session.enable_relationship_loading()` method is similar to the `load_on_pending` flag on `relationship()`. Unlike that flag, `Session.enable_relationship_loading()` allows an object to remain transient while still being able to load related items.

To make a transient object associated with a *Session* via *Session.enable_relationship_loading()* pending, add it to the *Session* using *Session.add()* normally.

Session.enable_relationship_loading() does not improve behavior when the ORM is used normally - object references should be constructed at the object level, not at the foreign key level, so that they are present in an ordinary way before flush() proceeds. This method is not intended for general use.

New in version 0.8.

See also:

`load_on_pending` at *relationship()* - this flag allows per-relationship loading of many-to-ones on items that are pending.

execute(*clause*, *params=None*, *mapper=None*, *bind=None*, ***kw*)

Execute a SQL expression construct or string statement within the current transaction.

Returns a *ResultProxy* representing results of the statement execution, in the same manner as that of an *Engine* or *Connection*.

E.g.:

```
result = session.execute(
            user_table.select().where(user_table.c.id == 5)
        )
```

execute() accepts any executable clause construct, such as *select()*, *insert()*, *update()*, *delete()*, and *text()*. Plain SQL strings can be passed as well, which in the case of *Session.execute()* only will be interpreted the same as if it were passed via a *text()* construct. That is, the following usage:

```
result = session.execute(
            "SELECT * FROM user WHERE id=:param",
            {"param":5}
        )
```

is equivalent to:

```
from sqlalchemy import text
result = session.execute(
            text("SELECT * FROM user WHERE id=:param"),
            {"param":5}
        )
```

The second positional argument to *Session.execute()* is an optional parameter set. Similar to that of *Connection.execute()*, whether this is passed as a single dictionary, or a list of dictionaries, determines whether the DBAPI cursor's `execute()` or `executemany()` is used to execute the statement. An INSERT construct may be invoked for a single row:

```
result = session.execute(
    users.insert(), {"id": 7, "name": "somename"})
```

or for multiple rows:

```
result = session.execute(users.insert(), [
                         {"id": 7, "name": "somename7"},
                         {"id": 8, "name": "somename8"},
                         {"id": 9, "name": "somename9"}
                        ])
```

The statement is executed within the current transactional context of this `Session`. The `Connection` which is used to execute the statement can also be acquired directly by calling the `Session.connection()` method. Both methods use a rule-based resolution scheme in order to determine the `Connection`, which in the average case is derived directly from the "bind" of the `Session` itself, and in other cases can be based on the `mapper()` and `Table` objects passed to the method; see the documentation for `Session.get_bind()` for a full description of this scheme.

The `Session.execute()` method does *not* invoke autoflush.

The `ResultProxy` returned by the `Session.execute()` method is returned with the "close_with_result" flag set to true; the significance of this flag is that if this `Session` is autocommitting and does not have a transaction-dedicated `Connection` available, a temporary `Connection` is established for the statement execution, which is closed (meaning, returned to the connection pool) when the `ResultProxy` has consumed all available data. This applies *only* when the `Session` is configured with autocommit=True and no transaction has been started.

Parameters

- **clause** – An executable statement (i.e. an `Executable` expression such as `expression.select()`) or string SQL statement to be executed.
- **params** – Optional dictionary, or list of dictionaries, containing bound parameter values. If a single dictionary, single-row execution occurs; if a list of dictionaries, an "executemany" will be invoked. The keys in each dictionary must correspond to parameter names present in the statement.
- **mapper** – Optional `mapper()` or mapped class, used to identify the appropriate bind. This argument takes precedence over `clause` when locating a bind. See `Session.get_bind()` for more details.
- **bind** – Optional `Engine` to be used as the bind. If this engine is already involved in an ongoing transaction, that connection will be used. This argument takes precedence over `mapper` and `clause` when locating a bind.
- ****kw** – Additional keyword arguments are sent to `Session.get_bind()` to allow extensibility of "bind" schemes.

See also:

SQL Expression Language Tutorial - Tutorial on using Core SQL constructs.

Working with Engines and Connections - Further information on direct statement execution.

`Connection.execute()` - core level statement execution method, which is `Session.execute()` ultimately uses in order to execute the statement.

expire(*instance*, *attribute_names=None*)

Expire the attributes on an instance.

Marks the attributes of an instance as out of date. When an expired attribute is next accessed, a query will be issued to the `Session` object's current transactional context in order to load all expired attributes for the given instance. Note that a highly isolated transaction will return the same values as were previously read in that same transaction, regardless of changes in database state outside of that transaction.

To expire all objects in the `Session` simultaneously, use `Session.expire_all()`.

The `Session` object's default behavior is to expire all state whenever the `Session.rollback()` or `Session.commit()` methods are called, so that new state can be loaded for the new transaction. For this reason, calling `Session.expire()` only makes sense for the specific case that a non-ORM SQL statement was emitted in the current transaction.

Parameters

- **instance** – The instance to be refreshed.
- **attribute_names** – optional list of string attribute names indicating a subset of attributes to be expired.

See also:

Refreshing / Expiring - introductory material

`Session.expire()`

`Session.refresh()`

expire_all()

Expires all persistent instances within this Session.

When any attributes on a persistent instance is next accessed, a query will be issued using the `Session` object's current transactional context in order to load all expired attributes for the given instance. Note that a highly isolated transaction will return the same values as were previously read in that same transaction, regardless of changes in database state outside of that transaction.

To expire individual objects and individual attributes on those objects, use `Session.expire()`.

The `Session` object's default behavior is to expire all state whenever the `Session.rollback()` or `Session.commit()` methods are called, so that new state can be loaded for the new transaction. For this reason, calling `Session.expire_all()` should not be needed when autocommit is `False`, assuming the transaction is isolated.

See also:

Refreshing / Expiring - introductory material

`Session.expire()`

`Session.refresh()`

expunge(*instance*)

Remove the *instance* from this `Session`.

This will free all internal references to the instance. Cascading will be applied according to the *expunge* cascade rule.

expunge_all()

Remove all object instances from this `Session`.

This is equivalent to calling `expunge(obj)` on all objects in this `Session`.

flush(*objects=None*)

Flush all the object changes to the database.

Writes out all pending object creations, deletions and modifications to the database as INSERTs, DELETEs, UPDATEs, etc. Operations are automatically ordered by the Session's unit of work dependency solver.

Database operations will be issued in the current transactional context and do not affect the state of the transaction, unless an error occurs, in which case the entire transaction is rolled back. You may flush() as often as you like within a transaction to move changes from Python to the database's transaction buffer.

For `autocommit` Sessions with no active manual transaction, flush() will create a transaction on the fly that surrounds the entire set of operations into the flush.

Parameters **objects** – Optional; restricts the flush operation to operate only on elements that are in the given collection.

This feature is for an extremely narrow set of use cases where particular objects may need to be operated upon before the full flush() occurs. It is not intended for general use.

get_bind(*mapper=None*, *clause=None*)

Return a "bind" to which this `Session` is bound.

The "bind" is usually an instance of `Engine`, except in the case where the `Session` has been explicitly bound directly to a `Connection`.

For a multiply-bound or unbound `Session`, the `mapper` or `clause` arguments are used to determine the appropriate bind to return.

Note that the "mapper" argument is usually present when `Session.get_bind()` is called via an ORM operation such as a `Session.query()`, each individual INSERT/UPDATE/DELETE operation within a `Session.flush()`, call, etc.

The order of resolution is:

1. if mapper given and session.binds is present, locate a bind based on mapper.
2. if clause given and session.binds is present, locate a bind based on `Table` objects found in the given clause present in session.binds.
3. if session.bind is present, return that.
4. if clause given, attempt to return a bind linked to the `MetaData` ultimately associated with the clause.
5. if mapper given, attempt to return a bind linked to the `MetaData` ultimately associated with the `Table` or other selectable to which the mapper is mapped.
6. No bind can be found, `UnboundExecutionError` is raised.

Parameters

- **mapper** – Optional `mapper()` mapped class or instance of `Mapper`. The bind can be derived from a `Mapper` first by consulting the "binds" map associated with this `Session`, and secondly by consulting the `MetaData` associated with the `Table` to which the `Mapper` is mapped for a bind.
- **clause** – A `ClauseElement` (i.e. `select()`, `text()`, etc.). If the `mapper` argument is not present or could not produce a bind, the given expression construct will be searched for a bound element, typically a `Table` associated with bound `MetaData`.

identity_key(**args*, ***kwargs*)

inherited from the `identity_key()` *method of* `_SessionClassMethods`

Return an identity key.

This is an alias of `util.identity_key()`.

identity_map = None

A mapping of object identities to objects themselves.

Iterating through `Session.identity_map.values()` provides access to the full set of persistent objects (i.e., those that have row identity) currently in the session.

See also:

`identity_key()` - helper function to produce the keys used in this dictionary.

info

A user-modifiable dictionary.

The initial value of this dictionary can be populated using the `info` argument to the *Session* constructor or *sessionmaker* constructor or factory methods. The dictionary here is always local to this *Session* and can be modified independently of all other *Session* objects.

New in version 0.9.0.

invalidate()

Close this Session, using connection invalidation.

This is a variant of *Session.close()* that will additionally ensure that the *Connection.invalidate()* method will be called on all *Connection* objects. This can be called when the database is known to be in a state where the connections are no longer safe to be used.

E.g.:

```
try:
    sess = Session()
    sess.add(User())
    sess.commit()
except gevent.Timeout:
    sess.invalidate()
    raise
except:
    sess.rollback()
    raise
```

This clears all items and ends any transaction in progress.

If this session were created with `autocommit=False`, a new transaction is immediately begun. Note that this new transaction does not use any connection resources until they are first needed.

New in version 0.9.9.

is_active

True if this *Session* is in "transaction mode" and is not in "partial rollback" state.

The *Session* in its default mode of `autocommit=False` is essentially always in "transaction mode", in that a *SessionTransaction* is associated with it as soon as it is instantiated. This *SessionTransaction* is immediately replaced with a new one as soon as it is ended, due to a rollback, commit, or close operation.

"Transaction mode" does *not* indicate whether or not actual database connection resources are in use; the *SessionTransaction* object coordinates among zero or more actual database transactions, and starts out with none, accumulating individual DBAPI connections as different data sources are used within its scope. The best way to track when a particular *Session* has actually begun to use DBAPI resources is to implement a listener using the *SessionEvents.after_begin()* method, which will deliver both the *Session* as well as the target *Connection* to a user-defined event listener.

The "partial rollback" state refers to when an "inner" transaction, typically used during a flush, encounters an error and emits a rollback of the DBAPI connection. At this point, the *Session* is in "partial rollback" and awaits for the user to call *Session.rollback()*, in order to close out the transaction stack. It is in this "partial rollback" period that the *is_active* flag returns False. After the call to *Session.rollback()*, the *SessionTransaction* is replaced with a new one and *is_active* returns `True` again.

When a *Session* is used in `autocommit=True` mode, the *SessionTransaction* is only instantiated within the scope of a flush call, or when *Session.begin()* is called. So *is_active* will

always be `False` outside of a flush or `Session.begin()` block in this mode, and will be `True` within the `Session.begin()` block as long as it doesn't enter "partial rollback" state.

From all the above, it follows that the only purpose to this flag is for application frameworks that wish to detect is a "rollback" is necessary within a generic error handling routine, for `Session` objects that would otherwise be in "partial rollback" mode. In a typical integration case, this is also not necessary as it is standard practice to emit `Session.rollback()` unconditionally within the outermost exception catch.

To track the transactional state of a `Session` fully, use event listeners, primarily the `SessionEvents.after_begin()`, `SessionEvents.after_commit()`, `SessionEvents.after_rollback()` and related events.

is_modified(*instance*, *include_collections=True*, *passive=True*)

Return `True` if the given instance has locally modified attributes.

This method retrieves the history for each instrumented attribute on the instance and performs a comparison of the current value to its previously committed value, if any.

It is in effect a more expensive and accurate version of checking for the given instance in the `Session.dirty` collection; a full test for each attribute's net "dirty" status is performed.

E.g.:

```
return session.is_modified(someobject)
```

Changed in version 0.8: When using SQLAlchemy 0.7 and earlier, the `passive` flag should **always** be explicitly set to `True`, else SQL loads/autoflushes may proceed which can affect the modified state itself: `session.is_modified(someobject, passive=True)`. In 0.8 and above, the behavior is corrected and this flag is ignored.

A few caveats to this method apply:

- Instances present in the `Session.dirty` collection may report `False` when tested with this method. This is because the object may have received change events via attribute mutation, thus placing it in `Session.dirty`, but ultimately the state is the same as that loaded from the database, resulting in no net change here.
- Scalar attributes may not have recorded the previously set value when a new value was applied, if the attribute was not loaded, or was expired, at the time the new value was received - in these cases, the attribute is assumed to have a change, even if there is ultimately no net change against its database value. SQLAlchemy in most cases does not need the "old" value when a set event occurs, so it skips the expense of a SQL call if the old value isn't present, based on the assumption that an UPDATE of the scalar value is usually needed, and in those few cases where it isn't, is less expensive on average than issuing a defensive SELECT.

  The "old" value is fetched unconditionally upon set only if the attribute container has the `active_history` flag set to `True`. This flag is set typically for primary key attributes and scalar object references that are not a simple many-to-one. To set this flag for any arbitrary mapped column, use the `active_history` argument with `column_property()`.

  Parameters

  - **instance** – mapped instance to be tested for pending changes.
  - **include_collections** – Indicates if multivalued collections should be included in the operation. Setting this to `False` is a way to detect only local-column based properties (i.e. scalar columns or many-to-one foreign keys) that would result in an UPDATE for this instance upon flush.

- **passive** – Changed in version 0.8: Ignored for backwards compatibility. When using SQLAlchemy 0.7 and earlier, this flag should always be set to `True`.

merge(*instance*, *load=True*)

Copy the state of a given instance into a corresponding instance within this `Session`.

`Session.merge()` examines the primary key attributes of the source instance, and attempts to reconcile it with an instance of the same primary key in the session. If not found locally, it attempts to load the object from the database based on primary key, and if none can be located, creates a new instance. The state of each attribute on the source instance is then copied to the target instance. The resulting target instance is then returned by the method; the original source instance is left unmodified, and un-associated with the `Session` if not already.

This operation cascades to associated instances if the association is mapped with `cascade="merge"`.

See *Merging* for a detailed discussion of merging.

Changed in version 1.1: - `Session.merge()` will now reconcile pending objects with overlapping primary keys in the same way as persistent. See *Session.merge resolves pending conflicts the same as persistent* for discussion.

Parameters

- **instance** – Instance to be merged.
- **load** – Boolean, when False, `merge()` switches into a "high performance" mode which causes it to forego emitting history events as well as all database access. This flag is used for cases such as transferring graphs of objects into a `Session` from a second level cache, or to transfer just-loaded objects into the `Session` owned by a worker thread or process without re-querying the database.

  The `load=False` use case adds the caveat that the given object has to be in a "clean" state, that is, has no pending changes to be flushed - even if the incoming object is detached from any `Session`. This is so that when the merge operation populates local attributes and cascades to related objects and collections, the values can be "stamped" onto the target object as is, without generating any history or attribute events, and without the need to reconcile the incoming data with any existing related objects or collections that might not be loaded. The resulting objects from `load=False` are always produced as "clean", so it is only appropriate that the given objects should be "clean" as well, else this suggests a mis-use of the method.

new

The set of all instances marked as 'new' within this `Session`.

no_autoflush

Return a context manager that disables autoflush.

e.g.:

```
with session.no_autoflush:

    some_object = SomeClass()
    session.add(some_object)
    # won't autoflush
    some_object.related_thing = session.query(SomeRelated).first()
```

Operations that proceed within the `with:` block will not be subject to flushes occurring upon query access. This is useful when initializing a series of objects which involve existing database queries, where the uncompleted object should not yet be flushed.

New in version 0.7.6.

object_session(*instance*)

inherited from the `object_session()` *method of* `_SessionClassMethods`

Return the `Session` to which an object belongs.

This is an alias of `object_session()`.

prepare()

Prepare the current transaction in progress for two phase commit.

If no transaction is in progress, this method raises an `InvalidRequestError`.

Only root transactions of two phase sessions can be prepared. If the current transaction is not such, an `InvalidRequestError` is raised.

prune()

Remove unreferenced instances cached in the identity map.

Deprecated since version 0.7: The non-weak-referencing identity map feature is no longer needed.

Note that this method is only meaningful if "weak_identity_map" is set to False. The default weak identity map is self-pruning.

Removes any object in this Session's identity map that is not referenced in user code, modified, new or scheduled for deletion. Returns the number of objects pruned.

query(**entities*, ***kwargs*)

Return a new `Query` object corresponding to this `Session`.

refresh(*instance*, *attribute_names=None*, *lockmode=None*)

Expire and refresh the attributes on the given instance.

A query will be issued to the database and all attributes will be refreshed with their current database value.

Lazy-loaded relational attributes will remain lazily loaded, so that the instance-wide refresh operation will be followed immediately by the lazy load of that attribute.

Eagerly-loaded relational attributes will eagerly load within the single refresh operation.

Note that a highly isolated transaction will return the same values as were previously read in that same transaction, regardless of changes in database state outside of that transaction - usage of `refresh()` usually only makes sense if non-ORM SQL statement were emitted in the ongoing transaction, or if autocommit mode is turned on.

Parameters

- **attribute_names** – optional. An iterable collection of string attribute names indicating a subset of attributes to be refreshed.
- **lockmode** – Passed to the `Query` as used by `with_lockmode()`.

See also:

Refreshing / Expiring - introductory material

`Session.expire()`

`Session.expire_all()`

rollback()

Rollback the current transaction in progress.

If no transaction is in progress, this method is a pass-through.

This method rolls back the current transaction or nested transaction regardless of subtransactions being in effect. All subtransactions up to the first real transaction are closed. Subtransactions occur when `begin()` is called multiple times.

See also:

Rolling Back

scalar(*clause*, *params=None*, *mapper=None*, *bind=None*, ***kw*)
Like `execute()` but return a scalar result.

transaction = None
The current active or inactive `SessionTransaction`.

class sqlalchemy.orm.session.**SessionTransaction**(*session*, *parent=None*, *nested=False*)
A `Session`-level transaction.

`SessionTransaction` is a mostly behind-the-scenes object not normally referenced directly by application code. It coordinates among multiple `Connection` objects, maintaining a database transaction for each one individually, committing or rolling them back all at once. It also provides optional two-phase commit behavior which can augment this coordination operation.

The `Session.transaction` attribute of `Session` refers to the current `SessionTransaction` object in use, if any.

A `SessionTransaction` is associated with a `Session` in its default mode of `autocommit=False` immediately, associated with no database connections. As the `Session` is called upon to emit SQL on behalf of various `Engine` or `Connection` objects, a corresponding `Connection` and associated `Transaction` is added to a collection within the `SessionTransaction` object, becoming one of the connection/transaction pairs maintained by the `SessionTransaction`.

The lifespan of the `SessionTransaction` ends when the `Session.commit()`, `Session.rollback()` or `Session.close()` methods are called. At this point, the `SessionTransaction` removes its association with its parent `Session`. A `Session` that is in `autocommit=False` mode will create a new `SessionTransaction` to replace it immediately, whereas a `Session` that's in `autocommit=True` mode will remain without a `SessionTransaction` until the `Session.begin()` method is called.

Another detail of `SessionTransaction` behavior is that it is capable of "nesting". This means that the `Session.begin()` method can be called while an existing `SessionTransaction` is already present, producing a new `SessionTransaction` that temporarily replaces the parent `SessionTransaction`. When a `SessionTransaction` is produced as nested, it assigns itself to the `Session.transaction` attribute. When it is ended via `Session.commit()` or `Session.rollback()`, it restores its parent `SessionTransaction` back onto the `Session.transaction` attribute. The behavior is effectively a stack, where `Session.transaction` refers to the current head of the stack.

The purpose of this stack is to allow nesting of `Session.rollback()` or `Session.commit()` calls in context with various flavors of `Session.begin()`. This nesting behavior applies to when `Session.begin_nested()` is used to emit a SAVEPOINT transaction, and is also used to produce a so-called "subtransaction" which allows a block of code to use a begin/rollback/commit sequence regardless of whether or not its enclosing code block has begun a transaction. The `flush()` method, whether called explicitly or via autoflush, is the primary consumer of the "subtransaction" feature, in that it wishes to guarantee that it works within in a transaction block regardless of whether or not the `Session` is in transactional mode when the method is called.

See also:

`Session.rollback()`

`Session.commit()`

`Session.begin()`

`Session.begin_nested()`

`Session.is_active`

`SessionEvents.after_commit()`

`SessionEvents.after_rollback()`

`SessionEvents.after_soft_rollback()`

Session Utilites

`sqlalchemy.orm.session.`**`make_transient`**(*instance*)

Make the given instance 'transient'.

This will remove its association with any session and additionally will remove its "identity key", such that it's as though the object were newly constructed, except retaining its values. It also resets the "deleted" flag on the state if this object had been explicitly deleted by its session.

Attributes which were "expired" or deferred at the instance level are reverted to undefined, and will not trigger any loads.

`sqlalchemy.orm.session.`**`make_transient_to_detached`**(*instance*)

Make the given transient instance 'detached'.

All attribute history on the given instance will be reset as though the instance were freshly loaded from a query. Missing attributes will be marked as expired. The primary key attributes of the object, which are required, will be made into the "key" of the instance.

The object can then be added to a session, or merged possibly with the load=False flag, at which point it will look as if it were loaded that way, without emitting SQL.

This is a special use case function that differs from a normal call to `Session.merge()` in that a given persistent state can be manufactured without any SQL calls.

New in version 0.9.5.

See also:

`make_transient()`

`sqlalchemy.orm.session.`**`object_session`**(*instance*)

Return the `Session` to which the given instance belongs.

This is essentially the same as the `InstanceState.session` accessor. See that attribute for details.

`sqlalchemy.orm.util.`**`was_deleted`**(*object*)

Return True if the given object was deleted within a session flush.

This is regardless of whether or not the object is persistent or detached.

New in version 0.8.0.

See also:

`InstanceState.was_deleted`

Attribute and State Management Utilities

These functions are provided by the SQLAlchemy attribute instrumentation API to provide a detailed interface for dealing with instances, attribute values, and history. Some of them are useful when constructing event listener functions, such as those described in ORM Events.

`sqlalchemy.orm.util.`**`object_state`**(*instance*)

Given an object, return the `InstanceState` associated with the object.

Raises `sqlalchemy.orm.exc.UnmappedInstanceError` if no mapping is configured.

Equivalent functionality is available via the `inspect()` function as:

```
inspect(instance)
```

Using the inspection system will raise `sqlalchemy.exc.NoInspectionAvailable` if the instance is not part of a mapping.

`sqlalchemy.orm.attributes.`**`del_attribute`**(*instance*, *key*)

Delete the value of an attribute, firing history events.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required. Custom attribute management schemes will need to make usage of this method to establish attribute state as understood by SQLAlchemy.

`sqlalchemy.orm.attributes.`**`get_attribute`**(*instance*, *key*)

Get the value of an attribute, firing any callables required.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required. Custom attribute management schemes will need to make usage of this method to make usage of attribute state as understood by SQLAlchemy.

`sqlalchemy.orm.attributes.`**`get_history`**(*obj*, *key*, *passive=symbol('PASSIVE_OFF')*)

Return a `History` record for the given object and attribute key.

Parameters

- **`obj`** – an object whose class is instrumented by the attributes package.
- **`key`** – string attribute name.
- **`passive`** – indicates loading behavior for the attribute if the value is not already present. This is a bitflag attribute, which defaults to the symbol `PASSIVE_OFF` indicating all necessary SQL should be emitted.

`sqlalchemy.orm.attributes.`**`init_collection`**(*obj*, *key*)

Initialize a collection attribute and return the collection adapter.

This function is used to provide direct access to collection internals for a previously unloaded attribute. e.g.:

```
collection_adapter = init_collection(someobject, 'elements')
for elem in values:
    collection_adapter.append_without_event(elem)
```

For an easier way to do the above, see `set_committed_value()`.

obj is an instrumented object instance. An InstanceState is accepted directly for backwards compatibility but this usage is deprecated.

`sqlalchemy.orm.attributes.`**`flag_modified`**(*instance*, *key*)

Mark an attribute on an instance as 'modified'.

This sets the 'modified' flag on the instance and establishes an unconditional change event for the given attribute.

sqlalchemy.orm.attributes.**instance_state**()

Return the *InstanceState* for a given mapped object.

This function is the internal version of *object_state()*. The *object_state()* and/or the *inspect()* function is preferred here as they each emit an informative exception if the given object is not mapped.

sqlalchemy.orm.instrumentation.**is_instrumented**(*instance*, *key*)

Return True if the given attribute on the given instance is instrumented by the attributes package.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required.

sqlalchemy.orm.attributes.**set_attribute**(*instance*, *key*, *value*)

Set the value of an attribute, firing history events.

This function may be used regardless of instrumentation applied directly to the class, i.e. no descriptors are required. Custom attribute management schemes will need to make usage of this method to establish attribute state as understood by SQLAlchemy.

sqlalchemy.orm.attributes.**set_committed_value**(*instance*, *key*, *value*)

Set the value of an attribute with no history events.

Cancels any previous history present. The value should be a scalar value for scalar-holding attributes, or an iterable for any collection-holding attribute.

This is the same underlying method used when a lazy loader fires off and loads additional data from the database. In particular, this method can be used by application code which has loaded additional attributes or collections through separate queries, which can then be attached to an instance as though it were part of its original loaded state.

class sqlalchemy.orm.attributes.**History**

Bases: *sqlalchemy.orm.attributes.History*

A 3-tuple of added, unchanged and deleted values, representing the changes which have occurred on an instrumented attribute.

The easiest way to get a *History* object for a particular attribute on an object is to use the *inspect()* function:

```
from sqlalchemy import inspect

hist = inspect(myobject).attrs.myattribute.history
```

Each tuple member is an iterable sequence:

- added - the collection of items added to the attribute (the first tuple element).
- unchanged - the collection of items that have not changed on the attribute (the second tuple element).
- deleted - the collection of items that have been removed from the attribute (the third tuple element).

empty()

Return True if this *History* has no changes and no existing, unchanged state.

has_changes()

Return True if this *History* has changes.

non_added()

Return a collection of unchanged + deleted.

non_deleted()

Return a collection of added + unchanged.

sum()
Return a collection of added + unchanged + deleted.

2.6 Events and Internals

2.6.1 ORM Events

The ORM includes a wide variety of hooks available for subscription.

For an introduction to the most commonly used ORM events, see the section *Tracking Object and Session Changes with Events*. The event system in general is discussed at *Events*. Non-ORM events such as those regarding connections and low-level statement execution are described in *Core Events*.

Attribute Events

class sqlalchemy.orm.events.**AttributeEvents**
Bases: *sqlalchemy.event.base.Events*

Define events for object attributes.

These are typically defined on the class-bound descriptor for the target class.

e.g.:

```
from sqlalchemy import event

def my_append_listener(target, value, initiator):
    print "received append event for target: %s" % target

event.listen(MyClass.collection, 'append', my_append_listener)
```

Listeners have the option to return a possibly modified version of the value, when the `retval=True` flag is passed to *listen()*:

```
def validate_phone(target, value, oldvalue, initiator):
    "Strip non-numeric characters from a phone number"

    return re.sub(r'(?![0-9])', '', value)

# setup listener on UserContact.phone attribute, instructing
# it to use the return value
listen(UserContact.phone, 'set', validate_phone, retval=True)
```

A validation function like the above can also raise an exception such as `ValueError` to halt the operation.

Several modifiers are available to the *listen()* function.

Parameters

- **active_history=False** – When True, indicates that the "set" event would like to receive the "old" value being replaced unconditionally, even if this requires firing off database loads. Note that `active_history` can also be set directly via *column_property()* and *relationship()*.
- **propagate=False** – When True, the listener function will be established not just for the class attribute given, but for attributes of the same name on all current subclasses of that class, as well as all future subclasses of that class, using an additional listener that listens for instrumentation events.

- **raw=False** – When True, the "target" argument to the event will be the *InstanceState* management object, rather than the mapped instance itself.
- **retval=False** – when True, the user-defined event listening must return the "value" argument from the function. This gives the listening function the opportunity to change the value that is ultimately used for a "set" or "append" event.

append(*target*, *value*, *initiator*)

Receive a collection append event.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass.some_attribute, 'append')
def receive_append(target, value, initiator):
    "listen for the 'append' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the object instance receiving the event. If the listener is registered with `raw=True`, this will be the *InstanceState* object.
- **value** – the value being appended. If this listener is registered with `retval=True`, the listener function must return this value, or a new value which replaces it.
- **initiator** – An instance of *attributes.Event* representing the initiation of the event. May be modified from its original value by backref handlers in order to control chained event propagation.

  Changed in version 0.9.0: the `initiator` argument is now passed as a *attributes.Event* object, and may be modified by backref handlers within a chain of backref-linked events.

Returns if the event was registered with `retval=True`, the given value, or a new effective value, should be returned.

dispose_collection(*target*, *collection*, *collection_adpater*)

Receive a 'collection dispose' event.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass.some_attribute, 'dispose_collection')
def receive_dispose_collection(target, collection, collection_adpater):
    "listen for the 'dispose_collection' event"

    # ... (event handling logic) ...
```

This event is triggered for a collection-based attribute when a collection is replaced, that is:

```
u1.addresses.append(a1)

u1.addresses = [a2, a3]  # <- old collection is disposed
```

The mechanics of the event will typically include that the given collection is empty, even if it stored objects while being replaced.

New in version 1.0.0: the *AttributeEvents.init_collection()* and *AttributeEvents.dispose_collection()* events supersede the `collection.linker` hook.

init_collection(*target*, *collection*, *collection_adapter*)

Receive a 'collection init' event.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass.some_attribute, 'init_collection')
def receive_init_collection(target, collection, collection_adapter):
    "listen for the 'init_collection' event"

    # ... (event handling logic) ...
```

This event is triggered for a collection-based attribute, when the initial "empty collection" is first generated for a blank attribute, as well as for when the collection is replaced with a new one, such as via a set event.

E.g., given that `User.addresses` is a relationship-based collection, the event is triggered here:

```
u1 = User()
u1.addresses.append(a1)  #  <- new collection
```

and also during replace operations:

```
u1.addresses = [a2, a3]  #  <- new collection
```

Parameters

- **target** – the object instance receiving the event. If the listener is registered with `raw=True`, this will be the *InstanceState* object.
- **collection** – the new collection. This will always be generated from what was specified as *RelationshipProperty.collection_class*, and will always be empty.
- **collection_adpater** – the *CollectionAdapter* that will mediate internal access to the collection.

New in version 1.0.0: the *AttributeEvents.init_collection()* and *AttributeEvents.dispose_collection()* events supersede the `collection.linker` hook.

remove(*target*, *value*, *initiator*)

Receive a collection remove event.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass.some_attribute, 'remove')
def receive_remove(target, value, initiator):
    "listen for the 'remove' event"
```

```
        # ... (event handling logic) ...
```

Parameters

- **target** – the object instance receiving the event. If the listener is registered with `raw=True`, this will be the *InstanceState* object.
- **value** – the value being removed.
- **initiator** – An instance of *attributes.Event* representing the initiation of the event. May be modified from its original value by backref handlers in order to control chained event propagation.

  Changed in version 0.9.0: the `initiator` argument is now passed as a *attributes.Event* object, and may be modified by backref handlers within a chain of backref-linked events.

Returns No return value is defined for this event.

set (*target*, *value*, *oldvalue*, *initiator*)

Receive a scalar set event.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass.some_attribute, 'set')
def receive_set(target, value, oldvalue, initiator):
    "listen for the 'set' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeClass.some_attribute, 'set', named=True)
def receive_set(**kw):
    "listen for the 'set' event"
    target = kw['target']
    value = kw['value']

    # ... (event handling logic) ...
```

Parameters

- **target** – the object instance receiving the event. If the listener is registered with `raw=True`, this will be the *InstanceState* object.
- **value** – the value being set. If this listener is registered with `retval=True`, the listener function must return this value, or a new value which replaces it.
- **oldvalue** – the previous value being replaced. This may also be the symbol `NEVER_SET` or `NO_VALUE`. If the listener is registered with `active_history=True`, the previous value of the attribute will be loaded from the database if the existing value is currently unloaded or expired.
- **initiator** – An instance of *attributes.Event* representing the initiation of the event. May be modified from its original value by backref handlers in order to control chained event propagation.

Changed in version 0.9.0: the `initiator` argument is now passed as a *attributes.Event* object, and may be modified by backref handlers within a chain of backref-linked events.

Returns if the event was registered with `retval=True`, the given value, or a new effective value, should be returned.

Mapper Events

class `sqlalchemy.orm.events.`**`MapperEvents`**

Bases: *sqlalchemy.event.base.Events*

Define events specific to mappings.

e.g.:

```
from sqlalchemy import event

def my_before_insert_listener(mapper, connection, target):
    # execute a stored procedure upon INSERT,
    # apply the value to the row to be inserted
    target.calculated_value = connection.scalar(
                                "select my_special_function(%d)"
                                % target.special_number)

# associate the listener function with SomeClass,
# to execute during the "before_insert" hook
event.listen(
    SomeClass, 'before_insert', my_before_insert_listener)
```

Available targets include:

- mapped classes
- unmapped superclasses of mapped or to-be-mapped classes (using the `propagate=True` flag)
- *Mapper* objects
- the *Mapper* class itself and the *mapper()* function indicate listening for all mappers.

Changed in version 0.8.0: mapper events can be associated with unmapped superclasses of mapped classes.

Mapper events provide hooks into critical sections of the mapper, including those related to object instrumentation, object loading, and object persistence. In particular, the persistence methods *before_insert()*, and *before_update()* are popular places to augment the state being persisted - however, these methods operate with several significant restrictions. The user is encouraged to evaluate the *SessionEvents.before_flush()* and *SessionEvents.after_flush()* methods as more flexible and user-friendly hooks in which to apply additional database state during a flush.

When using *MapperEvents*, several modifiers are available to the *event.listen()* function.

Parameters

- **`propagate=False`** – When True, the event listener should be applied to all inheriting mappers and/or the mappers of inheriting classes, as well as any mapper which is the target of this listener.
- **`raw=False`** – When True, the "target" argument passed to applicable event listener functions will be the instance's *InstanceState* management object, rather than the mapped instance itself.

- **retval=False** – when True, the user-defined event function must have a return value, the purpose of which is either to control subsequent event propagation, or to otherwise alter the operation in progress by the mapper. Possible return values are:
  - `sqlalchemy.orm.interfaces.EXT_CONTINUE` - continue event processing normally.
  - `sqlalchemy.orm.interfaces.EXT_STOP` - cancel all subsequent event handlers in the chain.
  - other values - the return value specified by specific listeners.

after_configured()

Called after a series of mappers have been configured.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'after_configured')
def receive_after_configured():
    "listen for the 'after_configured' event"

    # ... (event handling logic) ...
```

The *MapperEvents.after_configured()* event is invoked each time the *orm.configure_mappers()* function is invoked, after the function has completed its work. *orm.configure_mappers()* is typically invoked automatically as mappings are first used, as well as each time new mappers have been made available and new mapper use is detected.

Contrast this event to the *MapperEvents.mapper_configured()* event, which is called on a per-mapper basis while the configuration operation proceeds; unlike that event, when this event is invoked, all cross-configurations (e.g. backrefs) will also have been made available for any mappers that were pending. Also constrast to *MapperEvents.before_configured()*, which is invoked before the series of mappers has been configured.

This event can **only** be applied to the *Mapper* class or *mapper()* function, and not to individual mappings or mapped classes. It is only invoked for all mappings as a whole:

```
from sqlalchemy.orm import mapper

@event.listens_for(mapper, "after_configured")
def go():
    # ...
```

Theoretically this event is called once per application, but is actually called any time new mappers have been affected by a *orm.configure_mappers()* call. If new mappings are constructed after existing ones have already been used, this event will likely be called again. To ensure that a particular event is only called once and no further, the `once=True` argument (new in 0.9.4) can be applied:

```
from sqlalchemy.orm import mapper

@event.listens_for(mapper, "after_configured", once=True)
def go():
    # ...
```

See also:

MapperEvents.mapper_configured()

MapperEvents.before_configured()

after_delete(*mapper*, *connection*, *target*)

Receive an object instance after a DELETE statement has been emitted corresponding to that instance.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'after_delete')
def receive_after_delete(mapper, connection, target):
    "listen for the 'after_delete' event"

    # ... (event handling logic) ...
```

This event is used to emit additional SQL statements on the given connection as well as to perform application specific bookkeeping related to a deletion event.

The event is often called for a batch of objects of the same class after their DELETE statements have been emitted at once in a previous step.

> **Warning:** Mapper-level flush events only allow **very limited operations**, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given `Connection`. **Please read fully** the notes at *Mapper-level Events* for guidelines on using these methods; generally, the `SessionEvents.before_flush()` method should be preferred for general on-flush changes.

Parameters

- **mapper** – the `Mapper` which is the target of this event.
- **connection** – the `Connection` being used to emit DELETE statements for this instance. This provides a handle into the current transaction on the target database specific to this instance.
- **target** – the mapped instance being deleted. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.

Returns No return value is supported by this event.

See also:

Persistence Events

after_insert(*mapper*, *connection*, *target*)

Receive an object instance after an INSERT statement is emitted corresponding to that instance.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'after_insert')
def receive_after_insert(mapper, connection, target):
    "listen for the 'after_insert' event"

    # ... (event handling logic) ...
```

This event is used to modify in-Python-only state on the instance after an INSERT occurs, as well as to emit additional SQL statements on the given connection.

The event is often called for a batch of objects of the same class after their INSERT statements have been emitted at once in a previous step. In the extremely rare case that this is not desirable, the `mapper()` can be configured with `batch=False`, which will cause batches of instances to be broken up into individual (and more poorly performing) event->persist->event steps.

> **Warning:** Mapper-level flush events only allow **very limited operations**, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given `Connection`. **Please read fully** the notes at *Mapper-level Events* for guidelines on using these methods; generally, the `SessionEvents.before_flush()` method should be preferred for general on-flush changes.

Parameters

- **mapper** – the `Mapper` which is the target of this event.
- **connection** – the `Connection` being used to emit INSERT statements for this instance. This provides a handle into the current transaction on the target database specific to this instance.
- **target** – the mapped instance being persisted. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.

Returns No return value is supported by this event.

See also:

Persistence Events

after_update(*mapper*, *connection*, *target*)

Receive an object instance after an UPDATE statement is emitted corresponding to that instance.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'after_update')
def receive_after_update(mapper, connection, target):
    "listen for the 'after_update' event"

    # ... (event handling logic) ...
```

This event is used to modify in-Python-only state on the instance after an UPDATE occurs, as well as to emit additional SQL statements on the given connection.

This method is called for all instances that are marked as "dirty", *even those which have no net changes to their column-based attributes*, and for which no UPDATE statement has proceeded. An object is marked as dirty when any of its column-based attributes have a "set attribute" operation called or when any of its collections are modified. If, at update time, no column-based attributes have any net changes, no UPDATE statement will be issued. This means that an instance being sent to `after_update()` is *not* a guarantee that an UPDATE statement has been issued.

To detect if the column-based attributes on the object have net changes, and therefore resulted in an UPDATE statement, use `object_session(instance).is_modified(instance, include_collections=False)`.

The event is often called for a batch of objects of the same class after their UPDATE statements have been emitted at once in a previous step. In the extremely rare case that this is not desirable, the `mapper()` can be configured with `batch=False`, which will cause batches of instances to be broken up into individual (and more poorly performing) event->persist->event steps.

> **Warning:** Mapper-level flush events only allow **very limited operations**, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given `Connection`. **Please read fully** the notes at *Mapper-level Events* for guidelines on using these methods; generally, the `SessionEvents.before_flush()` method should be preferred for general on-flush changes.

Parameters

- **mapper** – the `Mapper` which is the target of this event.
- **connection** – the `Connection` being used to emit UPDATE statements for this instance. This provides a handle into the current transaction on the target database specific to this instance.
- **target** – the mapped instance being persisted. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.

Returns No return value is supported by this event.

See also:

Persistence Events

before_configured()

Called before a series of mappers have been configured.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'before_configured')
def receive_before_configured():
    "listen for the 'before_configured' event"

    # ... (event handling logic) ...
```

The `MapperEvents.before_configured()` event is invoked each time the `orm.configure_mappers()` function is invoked, before the function has done any of its work. `orm.configure_mappers()` is typically invoked automatically as mappings are first used, as well as each time new mappers have been made available and new mapper use is detected.

This event can **only** be applied to the `Mapper` class or `mapper()` function, and not to individual mappings or mapped classes. It is only invoked for all mappings as a whole:

```
from sqlalchemy.orm import mapper

@event.listens_for(mapper, "before_configured")
def go():
    # ...
```

Constrast this event to `MapperEvents.after_configured()`, which is invoked after the series of mappers has been configured, as well as `MapperEvents.mapper_configured()`, which is invoked on a per-mapper basis as each one is configured to the extent possible.

Theoretically this event is called once per application, but is actually called any time new mappers are to be affected by a `orm.configure_mappers()` call. If new mappings are constructed after existing ones have already been used, this event will likely be called again. To ensure that a particular event is only called once and no further, the `once=True` argument (new in 0.9.4) can be applied:

```
from sqlalchemy.orm import mapper

@event.listens_for(mapper, "before_configured", once=True)
def go():
    # ...
```

New in version 0.9.3.

See also:

MapperEvents.mapper_configured()

MapperEvents.after_configured()

before_delete(*mapper*, *connection*, *target*)

Receive an object instance before a DELETE statement is emitted corresponding to that instance.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'before_delete')
def receive_before_delete(mapper, connection, target):
    "listen for the 'before_delete' event"

    # ... (event handling logic) ...
```

This event is used to emit additional SQL statements on the given connection as well as to perform application specific bookkeeping related to a deletion event.

The event is often called for a batch of objects of the same class before their DELETE statements are emitted at once in a later step.

> **Warning:** Mapper-level flush events only allow **very limited operations**, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given *Connection*. **Please read fully** the notes at *Mapper-level Events* for guidelines on using these methods; generally, the *SessionEvents.before_flush()* method should be preferred for general on-flush changes.

Parameters

- **mapper** – the *Mapper* which is the target of this event.
- **connection** – the *Connection* being used to emit DELETE statements for this instance. This provides a handle into the current transaction on the target database specific to this instance.
- **target** – the mapped instance being deleted. If the event is configured with `raw=True`, this will instead be the *InstanceState* state-management object associated with the instance.

Returns No return value is supported by this event.

See also:

Persistence Events

before_insert(*mapper*, *connection*, *target*)

Receive an object instance before an INSERT statement is emitted corresponding to that instance.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'before_insert')
def receive_before_insert(mapper, connection, target):
    "listen for the 'before_insert' event"

    # ... (event handling logic) ...
```

This event is used to modify local, non-object related attributes on the instance before an INSERT occurs, as well as to emit additional SQL statements on the given connection.

The event is often called for a batch of objects of the same class before their INSERT statements are emitted at once in a later step. In the extremely rare case that this is not desirable, the `mapper()` can be configured with `batch=False`, which will cause batches of instances to be broken up into individual (and more poorly performing) event->persist->event steps.

> **Warning:** Mapper-level flush events only allow **very limited operations**, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given `Connection`. **Please read fully** the notes at *Mapper-level Events* for guidelines on using these methods; generally, the `SessionEvents.before_flush()` method should be preferred for general on-flush changes.

Parameters

- **mapper** – the `Mapper` which is the target of this event.
- **connection** – the `Connection` being used to emit INSERT statements for this instance. This provides a handle into the current transaction on the target database specific to this instance.
- **target** – the mapped instance being persisted. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.

Returns No return value is supported by this event.

See also:

Persistence Events

before_update(*mapper*, *connection*, *target*)

Receive an object instance before an UPDATE statement is emitted corresponding to that instance.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'before_update')
def receive_before_update(mapper, connection, target):
    "listen for the 'before_update' event"

    # ... (event handling logic) ...
```

This event is used to modify local, non-object related attributes on the instance before an UPDATE occurs, as well as to emit additional SQL statements on the given connection.

This method is called for all instances that are marked as "dirty", *even those which have no net changes to their column-based attributes*. An object is marked as dirty when any of its column-based attributes have a "set attribute" operation called or when any of its collections are modified. If, at update time, no

column-based attributes have any net changes, no UPDATE statement will be issued. This means that an instance being sent to `before_update()` is *not* a guarantee that an UPDATE statement will be issued, although you can affect the outcome here by modifying attributes so that a net change in value does exist.

To detect if the column-based attributes on the object have net changes, and will therefore generate an UPDATE statement, use `object_session(instance).is_modified(instance, include_collections=False)`.

The event is often called for a batch of objects of the same class before their UPDATE statements are emitted at once in a later step. In the extremely rare case that this is not desirable, the `mapper()` can be configured with `batch=False`, which will cause batches of instances to be broken up into individual (and more poorly performing) event->persist->event steps.

> **Warning:** Mapper-level flush events only allow **very limited operations**, on attributes local to the row being operated upon only, as well as allowing any SQL to be emitted on the given `Connection`. **Please read fully** the notes at *Mapper-level Events* for guidelines on using these methods; generally, the `SessionEvents.before_flush()` method should be preferred for general on-flush changes.

Parameters

- **mapper** – the `Mapper` which is the target of this event.
- **connection** – the `Connection` being used to emit UPDATE statements for this instance. This provides a handle into the current transaction on the target database specific to this instance.
- **target** – the mapped instance being persisted. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.

Returns No return value is supported by this event.

See also:

Persistence Events

instrument_class(*mapper*, *class_*)

Receive a class when the mapper is first constructed, before instrumentation is applied to the mapped class.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'instrument_class')
def receive_instrument_class(mapper, class_):
    "listen for the 'instrument_class' event"

    # ... (event handling logic) ...
```

This event is the earliest phase of mapper construction. Most attributes of the mapper are not yet initialized.

This listener can either be applied to the `Mapper` class overall, or to any un-mapped class which serves as a base for classes that will be mapped (using the `propagate=True` flag):

```
Base = declarative_base()

@event.listens_for(Base, "instrument_class", propagate=True)
```

```
def on_new_class(mapper, cls_):
    " ... "
```

Parameters

- **mapper** – the *Mapper* which is the target of this event.
- **class_** – the mapped class.

mapper_configured(*mapper*, *class_*)

Called when a specific mapper has completed its own configuration within the scope of the *configure_mappers()* call.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'mapper_configured')
def receive_mapper_configured(mapper, class_):
    "listen for the 'mapper_configured' event"

    # ... (event handling logic) ...
```

The *MapperEvents.mapper_configured()* event is invoked for each mapper that is encountered when the *orm.configure_mappers()* function proceeds through the current list of not-yet-configured mappers. *orm.configure_mappers()* is typically invoked automatically as mappings are first used, as well as each time new mappers have been made available and new mapper use is detected.

When the event is called, the mapper should be in its final state, but **not including backrefs** that may be invoked from other mappers; they might still be pending within the configuration operation. Bidirectional relationships that are instead configured via the *orm.relationship.back_populates* argument *will* be fully available, since this style of relationship does not rely upon other possibly-not-configured mappers to know that they exist.

For an event that is guaranteed to have **all** mappers ready to go including backrefs that are defined only on other mappings, use the *MapperEvents.after_configured()* event; this event invokes only after all known mappings have been fully configured.

The *MapperEvents.mapper_configured()* event, unlike *MapperEvents.before_configured()* or *MapperEvents.after_configured()*, is called for each mapper/class individually, and the mapper is passed to the event itself. It also is called exactly once for a particular mapper. The event is therefore useful for configurational steps that benefit from being invoked just once on a specific mapper basis, which don't require that "backref" configurations are necessarily ready yet.

Parameters

- **mapper** – the *Mapper* which is the target of this event.
- **class_** – the mapped class.

See also:

MapperEvents.before_configured()

MapperEvents.after_configured()

Instance Events

class `sqlalchemy.orm.events.`**`InstanceEvents`**

Bases: *`sqlalchemy.event.base.Events`*

Define events specific to object lifecycle.

e.g.:

```
from sqlalchemy import event

def my_load_listener(target, context):
    print "on load!"

event.listen(SomeClass, 'load', my_load_listener)
```

Available targets include:

- mapped classes
- unmapped superclasses of mapped or to-be-mapped classes (using the `propagate=True` flag)
- *`Mapper`* objects
- the *`Mapper`* class itself and the *`mapper()`* function indicate listening for all mappers.

Changed in version 0.8.0: instance events can be associated with unmapped superclasses of mapped classes.

Instance events are closely related to mapper events, but are more specific to the instance and its instrumentation, rather than its system of persistence.

When using *`InstanceEvents`*, several modifiers are available to the *`event.listen()`* function.

Parameters

- **`propagate=False`** – When True, the event listener should be applied to all inheriting classes as well as the class which is the target of this listener.
- **`raw=False`** – When True, the "target" argument passed to applicable event listener functions will be the instance's *`InstanceState`* management object, rather than the mapped instance itself.

`expire`(*target*, *attrs*)

Receive an object instance after its attributes or some subset have been expired.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'expire')
def receive_expire(target, attrs):
    "listen for the 'expire' event"

    # ... (event handling logic) ...
```

'keys' is a list of attribute names. If None, the entire state was expired.

Parameters

- **`target`** – the mapped instance. If the event is configured with `raw=True`, this will instead be the *`InstanceState`* state-management object associated with the instance.

- **attrs** – sequence of attribute names which were expired, or None if all attributes were expired.

first_init (*manager*, *cls*)

Called when the first instance of a particular mapping is called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'first_init')
def receive_first_init(manager, cls):
    "listen for the 'first_init' event"

    # ... (event handling logic) ...
```

This event is called when the `__init__` method of a class is called the first time for that particular class. The event invokes before `__init__` actually proceeds as well as before the `InstanceEvents.init()` event is invoked.

init (*target*, *args*, *kwargs*)

Receive an instance when its constructor is called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'init')
def receive_init(target, args, kwargs):
    "listen for the 'init' event"

    # ... (event handling logic) ...
```

This method is only called during a userland construction of an object, in conjunction with the object's constructor, e.g. its `__init__` method. It is not called when an object is loaded from the database; see the `InstanceEvents.load()` event in order to intercept a database load.

The event is called before the actual `__init__` constructor of the object is called. The `kwargs` dictionary may be modified in-place in order to affect what is passed to `__init__`.

Parameters

- **target** – the mapped instance. If the event is configured with `raw=True`, this will instead be the `InstanceState` state-management object associated with the instance.
- **args** – positional arguments passed to the `__init__` method. This is passed as a tuple and is currently immutable.
- **kwargs** – keyword arguments passed to the `__init__` method. This structure *can* be altered in place.

See also:

`InstanceEvents.init_failure()`

`InstanceEvents.load()`

init_failure (*target*, *args*, *kwargs*)

Receive an instance when its constructor has been called, and raised an exception.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'init_failure')
def receive_init_failure(target, args, kwargs):
    "listen for the 'init_failure' event"

    # ... (event handling logic) ...
```

This method is only called during a userland construction of an object, in conjunction with the object's constructor, e.g. its `__init__` method. It is not called when an object is loaded from the database.

The event is invoked after an exception raised by the `__init__` method is caught. After the event is invoked, the original exception is re-raised outwards, so that the construction of the object still raises an exception. The actual exception and stack trace raised should be present in `sys.exc_info()`.

Parameters

- **target** – the mapped instance. If the event is configured with `raw=True`, this will instead be the *InstanceState* state-management object associated with the instance.
- **args** – positional arguments that were passed to the `__init__` method.
- **kwargs** – keyword arguments that were passed to the `__init__` method.

See also:

InstanceEvents.init()

InstanceEvents.load()

load(*target*, *context*)

Receive an object instance after it has been created via `__new__`, and after initial attribute population has occurred.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'load')
def receive_load(target, context):
    "listen for the 'load' event"

    # ... (event handling logic) ...
```

This typically occurs when the instance is created based on incoming result rows, and is only called once for that instance's lifetime.

Note that during a result-row load, this method is called upon the first row received for this instance. Note that some attributes and collections may or may not be loaded or even initialized, depending on what's present in the result rows.

Parameters

- **target** – the mapped instance. If the event is configured with `raw=True`, this will instead be the *InstanceState* state-management object associated with the instance.

- **context** – the *QueryContext* corresponding to the current *Query* in progress. This argument may be None if the load does not correspond to a *Query*, such as during *Session.merge()*.

See also:

InstanceEvents.init()

InstanceEvents.refresh()

SessionEvents.loaded_as_persistent()

pickle(*target*, *state_dict*)

Receive an object instance when its associated state is being pickled.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'pickle')
def receive_pickle(target, state_dict):
    "listen for the 'pickle' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the mapped instance. If the event is configured with raw=True, this will instead be the *InstanceState* state-management object associated with the instance.
- **state_dict** – the dictionary returned by InstanceState.__getstate__, containing the state to be pickled.

refresh(*target*, *context*, *attrs*)

Receive an object instance after one or more attributes have been refreshed from a query.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'refresh')
def receive_refresh(target, context, attrs):
    "listen for the 'refresh' event"

    # ... (event handling logic) ...
```

Contrast this to the *InstanceEvents.load()* method, which is invoked when the object is first loaded from a query.

Parameters

- **target** – the mapped instance. If the event is configured with raw=True, this will instead be the *InstanceState* state-management object associated with the instance.
- **context** – the *QueryContext* corresponding to the current *Query* in progress.
- **attrs** – sequence of attribute names which were populated, or None if all column-mapped, non-deferred attributes were populated.

See also:

InstanceEvents.load()

refresh_flush(*target*, *flush_context*, *attrs*)

Receive an object instance after one or more attributes have been refreshed within the persistence of the object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'refresh_flush')
def receive_refresh_flush(target, flush_context, attrs):
    "listen for the 'refresh_flush' event"

    # ... (event handling logic) ...
```

This event is the same as *InstanceEvents.refresh()* except it is invoked within the unit of work flush process, and the values here typically come from the process of handling an INSERT or UPDATE, such as via the RETURNING clause or from Python-side default values.

New in version 1.0.5.

Parameters

- **target** – the mapped instance. If the event is configured with `raw=True`, this will instead be the *InstanceState* state-management object associated with the instance.
- **flush_context** – Internal *UOWTransaction* object which handles the details of the flush.
- **attrs** – sequence of attribute names which were populated.

unpickle(*target*, *state_dict*)

Receive an object instance after its associated state has been unpickled.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeClass, 'unpickle')
def receive_unpickle(target, state_dict):
    "listen for the 'unpickle' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the mapped instance. If the event is configured with `raw=True`, this will instead be the *InstanceState* state-management object associated with the instance.
- **state_dict** – the dictionary sent to `InstanceState.__setstate__`, containing the state dictionary which was pickled.

Session Events

class sqlalchemy.orm.events.**SessionEvents**

Bases: *sqlalchemy.event.base.Events*

Define events specific to *Session* lifecycle.

e.g.:

```
from sqlalchemy import event
from sqlalchemy.orm import sessionmaker

def my_before_commit(session):
    print "before commit!"

Session = sessionmaker()

event.listen(Session, "before_commit", my_before_commit)
```

The *listen()* function will accept *Session* objects as well as the return result of *sessionmaker()* and *scoped_session()*.

Additionally, it accepts the *Session* class which will apply listeners to all *Session* instances globally.

after_attach(*session*, *instance*)

Execute after an instance is attached to a session.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_attach')
def receive_after_attach(session, instance):
    "listen for the 'after_attach' event"

    # ... (event handling logic) ...
```

This is called after an add, delete or merge.

Note: As of 0.8, this event fires off *after* the item has been fully associated with the session, which is different than previous releases. For event handlers that require the object not yet be part of session state (such as handlers which may autoflush while the target object is not yet complete) consider the new *before_attach()* event.

See also:

before_attach()

Object Lifecycle Events

after_begin(*session*, *transaction*, *connection*)

Execute after a transaction is begun on a connection

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_begin')
def receive_after_begin(session, transaction, connection):
```

```
        "listen for the 'after_begin' event"

        # ... (event handling logic) ...
```

Parameters

- **session** – The target *Session*.
- **transaction** – The *SessionTransaction*.
- **connection** – The *Connection* object which will be used for SQL statements.

See also:

before_commit()

after_commit()

after_transaction_create()

after_transaction_end()

after_bulk_delete(*delete_context*)

Execute after a bulk delete operation to the session.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style (arguments as of 0.9)
@event.listens_for(SomeSessionOrFactory, 'after_bulk_delete')
def receive_after_bulk_delete(delete_context):
    "listen for the 'after_bulk_delete' event"

    # ... (event handling logic) ...

# legacy calling style (pre-0.9)
@event.listens_for(SomeSessionOrFactory, 'after_bulk_delete')
def receive_after_bulk_delete(session, query, query_context, result):
    "listen for the 'after_bulk_delete' event"

    # ... (event handling logic) ...
```

Changed in version 0.9: The `after_bulk_delete` event now accepts the arguments `delete_context`. Listener functions which accept the previous argument signature(s) listed above will be automatically adapted to the new signature.

This is called as a result of the *Query.delete()* method.

Parameters **delete_context** – a "delete context" object which contains details about the update, including these attributes:

- `session` - the *Session* involved
- `query` -the *Query* object that this update operation was called upon.
- `context` The *QueryContext* object, corresponding to the invocation of an ORM query.
- `result` the *ResultProxy* returned as a result of the bulk DELETE operation.

after_bulk_update(*update_context*)

Execute after a bulk update operation to the session.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style (arguments as of 0.9)
@event.listens_for(SomeSessionOrFactory, 'after_bulk_update')
def receive_after_bulk_update(update_context):
    "listen for the 'after_bulk_update' event"

    # ... (event handling logic) ...

# legacy calling style (pre-0.9)
@event.listens_for(SomeSessionOrFactory, 'after_bulk_update')
def receive_after_bulk_update(session, query, query_context, result):
    "listen for the 'after_bulk_update' event"

    # ... (event handling logic) ...
```

Changed in version 0.9: The `after_bulk_update` event now accepts the arguments `update_context`. Listener functions which accept the previous argument signature(s) listed above will be automatically adapted to the new signature.

This is called as a result of the `Query.update()` method.

Parameters **update_context** – an "update context" object which contains details about the update, including these attributes:

- `session` - the `Session` involved
- `query` -the `Query` object that this update operation was called upon.
- `context` The `QueryContext` object, corresponding to the invocation of an ORM query.
- `result` the `ResultProxy` returned as a result of the bulk UPDATE operation.

after_commit(*session*)

Execute after a commit has occurred.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_commit')
def receive_after_commit(session):
    "listen for the 'after_commit' event"

    # ... (event handling logic) ...
```

Note: The `after_commit()` hook is *not* per-flush, that is, the `Session` can emit SQL to the database many times within the scope of a transaction. For interception of these events, use the `before_flush()`, `after_flush()`, or `after_flush_postexec()` events.

Note: The `Session` is not in an active transaction when the `after_commit()` event is invoked, and therefore can not emit SQL. To emit SQL corresponding to every transaction, use the `before_commit()` event.

Parameters **session** – The target `Session`.

See also:

before_commit()

after_begin()

after_transaction_create()

after_transaction_end()

after_flush(*session*, *flush_context*)

Execute after flush has completed, but before commit has been called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_flush')
def receive_after_flush(session, flush_context):
    "listen for the 'after_flush' event"

    # ... (event handling logic) ...
```

Note that the session's state is still in pre-flush, i.e. 'new', 'dirty', and 'deleted' lists still show pre-flush state as well as the history settings on instance attributes.

Parameters

- **session** – The target *Session*.
- **flush_context** – Internal *UOWTransaction* object which handles the details of the flush.

See also:

before_flush()

after_flush_postexec()

Persistence Events

after_flush_postexec(*session*, *flush_context*)

Execute after flush has completed, and after the post-exec state occurs.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_flush_postexec')
def receive_after_flush_postexec(session, flush_context):
    "listen for the 'after_flush_postexec' event"

    # ... (event handling logic) ...
```

This will be when the 'new', 'dirty', and 'deleted' lists are in their final state. An actual commit() may or may not have occurred, depending on whether or not the flush started its own transaction or participated in a larger transaction.

Parameters

- **session** – The target *Session*.

- **flush_context** – Internal *UOWTransaction* object which handles the details of the flush.

See also:

before_flush()

after_flush()

Persistence Events

after_rollback(*session*)

Execute after a real DBAPI rollback has occurred.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_rollback')
def receive_after_rollback(session):
    "listen for the 'after_rollback' event"

    # ... (event handling logic) ...
```

Note that this event only fires when the *actual* rollback against the database occurs - it does *not* fire each time the *Session.rollback()* method is called, if the underlying DBAPI transaction has already been rolled back. In many cases, the *Session* will not be in an "active" state during this event, as the current transaction is not valid. To acquire a *Session* which is active after the outermost rollback has proceeded, use the *SessionEvents.after_soft_rollback()* event, checking the *Session.is_active* flag.

Parameters **session** – The target *Session*.

after_soft_rollback(*session*, *previous_transaction*)

Execute after any rollback has occurred, including "soft" rollbacks that don't actually emit at the DBAPI level.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_soft_rollback')
def receive_after_soft_rollback(session, previous_transaction):
    "listen for the 'after_soft_rollback' event"

    # ... (event handling logic) ...
```

This corresponds to both nested and outer rollbacks, i.e. the innermost rollback that calls the DBAPI's rollback() method, as well as the enclosing rollback calls that only pop themselves from the transaction stack.

The given *Session* can be used to invoke SQL and *Session.query()* operations after an outermost rollback by first checking the *Session.is_active* flag:

```
@event.listens_for(Session, "after_soft_rollback")
def do_something(session, previous_transaction):
    if session.is_active:
        session.execute("select * from some_table")
```

Parameters

- **session** – The target *Session*.
- **previous_transaction** – The *SessionTransaction* transactional marker object which was just closed. The current *SessionTransaction* for the given *Session* is available via the *Session.transaction* attribute.

New in version 0.7.3.

after_transaction_create(*session*, *transaction*)

Execute when a new *SessionTransaction* is created.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_transaction_create')
def receive_after_transaction_create(session, transaction):
    "listen for the 'after_transaction_create' event"

    # ... (event handling logic) ...
```

This event differs from *after_begin()* in that it occurs for each *SessionTransaction* overall, as opposed to when transactions are begun on individual database connections. It is also invoked for nested transactions and subtransactions, and is always matched by a corresponding *after_transaction_end()* event (assuming normal operation of the *Session*).

Parameters

- **session** – the target *Session*.
- **transaction** – the target *SessionTransaction*.

New in version 0.8.

See also:

after_transaction_end()

after_transaction_end(*session*, *transaction*)

Execute when the span of a *SessionTransaction* ends.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'after_transaction_end')
def receive_after_transaction_end(session, transaction):
    "listen for the 'after_transaction_end' event"

    # ... (event handling logic) ...
```

This event differs from *after_commit()* in that it corresponds to all *SessionTransaction* objects in use, including those for nested transactions and subtransactions, and is always matched by a corresponding *after_transaction_create()* event.

Parameters

- **session** – the target *Session*.
- **transaction** – the target *SessionTransaction*.

New in version 0.8.

See also:

after_transaction_create()

before_attach(*session*, *instance*)

Execute before an instance is attached to a session.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'before_attach')
def receive_before_attach(session, instance):
    "listen for the 'before_attach' event"

    # ... (event handling logic) ...
```

This is called before an add, delete or merge causes the object to be part of the session.

New in version 0.8.: Note that *after_attach()* now fires off after the item is part of the session. *before_attach()* is provided for those cases where the item should not yet be part of the session state.

See also:

after_attach()

Object Lifecycle Events

before_commit(*session*)

Execute before commit is called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'before_commit')
def receive_before_commit(session):
    "listen for the 'before_commit' event"

    # ... (event handling logic) ...
```

Note: The *before_commit()* hook is *not* per-flush, that is, the *Session* can emit SQL to the database many times within the scope of a transaction. For interception of these events, use the *before_flush()*, *after_flush()*, or *after_flush_postexec()* events.

Parameters **session** – The target *Session*.

See also:

after_commit()

after_begin()

after_transaction_create()

after_transaction_end()

before_flush(*session*, *flush_context*, *instances*)

Execute before flush process has started.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'before_flush')
def receive_before_flush(session, flush_context, instances):
    "listen for the 'before_flush' event"

    # ... (event handling logic) ...
```

Parameters

- **session** – The target *Session*.
- **flush_context** – Internal *UOWTransaction* object which handles the details of the flush.
- **instances** – Usually None, this is the collection of objects which can be passed to the *Session.flush()* method (note this usage is deprecated).

See also:

after_flush()

after_flush_postexec()

Persistence Events

deleted_to_detached(*session*, *instance*)

Intercept the "deleted to detached" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'deleted_to_detached')
def receive_deleted_to_detached(session, instance):
    "listen for the 'deleted_to_detached' event"

    # ... (event handling logic) ...
```

This event is invoked when a deleted object is evicted from the session. The typical case when this occurs is when the transaction for a *Session* in which the object was deleted is committed; the object moves from the deleted state to the detached state.

It is also invoked for objects that were deleted in a flush when the *Session.expunge_all()* or *Session.close()* events are called, as well as if the object is individually expunged from its deleted state via *Session.expunge()*.

New in version 1.1.

See also:

Object Lifecycle Events

deleted_to_persistent(*session*, *instance*)

Intercept the "deleted to persistent" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'deleted_to_persistent')
def receive_deleted_to_persistent(session, instance):
    "listen for the 'deleted_to_persistent' event"

    # ... (event handling logic) ...
```

This transition occurs only when an object that's been deleted successfully in a flush is restored due to a call to `Session.rollback()`. The event is not called under any other circumstances.

New in version 1.1.

See also:

Object Lifecycle Events

detached_to_persistent(*session*, *instance*)

Intercept the "detached to persistent" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'detached_to_persistent')
def receive_detached_to_persistent(session, instance):
    "listen for the 'detached_to_persistent' event"

    # ... (event handling logic) ...
```

This event is a specialization of the `SessionEvents.after_attach()` event which is only invoked for this specific transition. It is invoked typically during the `Session.add()` call, as well as during the `Session.delete()` call if the object was not previously associated with the `Session` (note that an object marked as "deleted" remains in the "persistent" state until the flush proceeds).

Note: If the object becomes persistent as part of a call to `Session.delete()`, the object is **not** yet marked as deleted when this event is called. To detect deleted objects, check the `deleted` flag sent to the `SessionEvents.persistent_to_detached()` to event after the flush proceeds, or check the `Session.deleted` collection within the `SessionEvents.before_flush()` event if deleted objects need to be intercepted before the flush.

Parameters

- **session** – target `Session`
- **instance** – the ORM-mapped instance being operated upon.

New in version 1.1.

See also:

Object Lifecycle Events

loaded_as_persistent(*session*, *instance*)

Intercept the "loaded as peristent" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'loaded_as_persistent')
def receive_loaded_as_persistent(session, instance):
    "listen for the 'loaded_as_persistent' event"

    # ... (event handling logic) ...
```

This event is invoked within the ORM loading process, and is invoked very similarly to the `InstanceEvents.load()` event. However, the event here is linkable to a `Session` class or instance, rather than to a mapper or class hierarchy, and integrates with the other session lifecycle events smoothly. The object is guaranteed to be present in the session's identity map when this event is called.

Parameters

- **session** – target `Session`
- **instance** – the ORM-mapped instance being operated upon.

New in version 1.1.

See also:

Object Lifecycle Events

pending_to_persistent(*session*, *instance*)

Intercept the "pending to persistent"" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'pending_to_persistent')
def receive_pending_to_persistent(session, instance):
    "listen for the 'pending_to_persistent' event"

    # ... (event handling logic) ...
```

This event is invoked within the flush process, and is similar to scanning the `Session.new` collection within the `SessionEvents.after_flush()` event. However, in this case the object has already been moved to the persistent state when the event is called.

Parameters

- **session** – target `Session`
- **instance** – the ORM-mapped instance being operated upon.

New in version 1.1.

See also:

Object Lifecycle Events

pending_to_transient(*session*, *instance*)

Intercept the "pending to transient" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
```

```
@event.listens_for(SomeSessionOrFactory, 'pending_to_transient')
def receive_pending_to_transient(session, instance):
    "listen for the 'pending_to_transient' event"

    # ... (event handling logic) ...
```

This less common transition occurs when an pending object that has not been flushed is evicted from the session; this can occur when the `Session.rollback()` method rolls back the transaction, or when the `Session.expunge()` method is used.

Parameters

- **session** – target `Session`
- **instance** – the ORM-mapped instance being operated upon.

New in version 1.1.

See also:

Object Lifecycle Events

persistent_to_deleted(*session*, *instance*)

Intercept the "persistent to deleted" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'persistent_to_deleted')
def receive_persistent_to_deleted(session, instance):
    "listen for the 'persistent_to_deleted' event"

    # ... (event handling logic) ...
```

This event is invoked when a persistent object's identity is deleted from the database within a flush, however the object still remains associated with the `Session` until the transaction completes.

If the transaction is rolled back, the object moves again to the persistent state, and the `SessionEvents.deleted_to_persistent()` event is called. If the transaction is committed, the object becomes detached, which will emit the `SessionEvents.deleted_to_detached()` event.

Note that while the `Session.delete()` method is the primary public interface to mark an object as deleted, many objects get deleted due to cascade rules, which are not always determined until flush time. Therefore, there's no way to catch every object that will be deleted until the flush has proceeded. the `SessionEvents.persistent_to_deleted()` event is therefore invoked at the end of a flush.

New in version 1.1.

See also:

Object Lifecycle Events

persistent_to_detached(*session*, *instance*)

Intercept the "persistent to detached" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
```

```
@event.listens_for(SomeSessionOrFactory, 'persistent_to_detached')
def receive_persistent_to_detached(session, instance):
    "listen for the 'persistent_to_detached' event"

    # ... (event handling logic) ...
```

This event is invoked when a persistent object is evicted from the session. There are many conditions that cause this to happen, including:

- using a method such as `Session.expunge()` or `Session.close()`
- Calling the `Session.rollback()` method, when the object was part of an INSERT statement for that session's transaction

Parameters

- **session** – target `Session`
- **instance** – the ORM-mapped instance being operated upon.
- **deleted** – boolean. If True, indicates this object moved to the detached state because it was marked as deleted and flushed.

New in version 1.1.

See also:

Object Lifecycle Events

persistent_to_transient(*session*, *instance*)

Intercept the "persistent to transient" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'persistent_to_transient')
def receive_persistent_to_transient(session, instance):
    "listen for the 'persistent_to_transient' event"

    # ... (event handling logic) ...
```

This less common transition occurs when an pending object that has has been flushed is evicted from the session; this can occur when the `Session.rollback()` method rolls back the transaction.

Parameters

- **session** – target `Session`
- **instance** – the ORM-mapped instance being operated upon.

New in version 1.1.

See also:

Object Lifecycle Events

transient_to_pending(*session*, *instance*)

Intercept the "transient to pending" transition for a specific object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSessionOrFactory, 'transient_to_pending')
def receive_transient_to_pending(session, instance):
    "listen for the 'transient_to_pending' event"

    # ... (event handling logic) ...
```

This event is a specialization of the `SessionEvents.after_attach()` event which is only invoked for this specific transition. It is invoked typically during the `Session.add()` call.

Parameters

- **session** – target `Session`
- **instance** – the ORM-mapped instance being operated upon.

New in version 1.1.

See also:

Object Lifecycle Events

Query Events

class `sqlalchemy.orm.events.`**`QueryEvents`**

Bases: `sqlalchemy.event.base.Events`

Represent events within the construction of a `Query` object.

The events here are intended to be used with an as-yet-unreleased inspection system for `Query`. Some very basic operations are possible now, however the inspection system is intended to allow complex query manipulations to be automated.

New in version 1.0.0.

before_compile(*query*)

Receive the `Query` object before it is composed into a core `Select` object.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeQuery, 'before_compile')
def receive_before_compile(query):
    "listen for the 'before_compile' event"

    # ... (event handling logic) ...
```

This event is intended to allow changes to the query given:

```
@event.listens_for(Query, "before_compile", retval=True)
def no_deleted(query):
    for desc in query.column_descriptions:
        if desc['type'] is User:
            entity = desc['entity']
            query = query.filter(entity.deleted == False)
    return query
```

The event should normally be listened with the `retval=True` parameter set, so that the modified query may be returned.

Instrumentation Events

Defines SQLAlchemy's system of class instrumentation.

This module is usually not directly visible to user applications, but defines a large part of the ORM's interactivity.

instrumentation.py deals with registration of end-user classes for state tracking. It interacts closely with state.py and attributes.py which establish per-instance and per-class-attribute instrumentation, respectively.

The class instrumentation system can be customized on a per-class or global basis using the *sqlalchemy.ext.instrumentation* module, which provides the means to build and specify alternate instrumentation forms.

class `sqlalchemy.orm.events.`**`InstrumentationEvents`**

Bases: *sqlalchemy.event.base.Events*

Events related to class instrumentation events.

The listeners here support being established against any new style class, that is any object that is a subclass of 'type'. Events will then be fired off for events against that class. If the "propagate=True" flag is passed to event.listen(), the event will fire off for subclasses of that class as well.

The Python `type` builtin is also accepted as a target, which when used has the effect of events being emitted for all classes.

Note the "propagate" flag here is defaulted to `True`, unlike the other class level events where it defaults to `False`. This means that new subclasses will also be the subject of these events, when a listener is established on a superclass.

Changed in version 0.8: - events here will emit based on comparing the incoming class to the type of class passed to *event.listen()*. Previously, the event would fire for any class unconditionally regardless of what class was sent for listening, despite documentation which stated the contrary.

`attribute_instrument`(*cls*, *key*, *inst*)

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeBaseClass, 'attribute_instrument')
def receive_attribute_instrument(cls, key, inst):
    "listen for the 'attribute_instrument' event"

    # ... (event handling logic) ...
```

Called when an attribute is instrumented.

`class_instrument`(*cls*)

Called after the given class is instrumented.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeBaseClass, 'class_instrument')
def receive_class_instrument(cls):
    "listen for the 'class_instrument' event"
```

```
        # ... (event handling logic) ...
```

To get at the *ClassManager*, use `manager_of_class()`.

class_uninstrument(*cls*)

Called before the given class is uninstrumented.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeBaseClass, 'class_uninstrument')
def receive_class_uninstrument(cls):
    "listen for the 'class_uninstrument' event"

    # ... (event handling logic) ...
```

To get at the *ClassManager*, use `manager_of_class()`.

2.6.2 ORM Internals

Key ORM constructs, not otherwise covered in other sections, are listed here.

class sqlalchemy.orm.state.**AttributeState**(*state*, *key*)

Provide an inspection interface corresponding to a particular attribute on a particular mapped object.

The *AttributeState* object is accessed via the *InstanceState.attrs* collection of a particular *InstanceState*:

```
from sqlalchemy import inspect

insp = inspect(some_mapped_object)
attr_state = insp.attrs.some_attribute
```

history

Return the current pre-flush change history for this attribute, via the *History* interface.

This method will **not** emit loader callables if the value of the attribute is unloaded.

See also:

AttributeState.load_history() - retrieve history using loader callables if the value is not locally present.

attributes.get_history() - underlying function

load_history()

Return the current pre-flush change history for this attribute, via the *History* interface.

This method **will** emit loader callables if the value of the attribute is unloaded.

See also:

AttributeState.history

attributes.get_history() - underlying function

New in version 0.9.0.

loaded_value
The current value of this attribute as loaded from the database.

If the value has not been loaded, or is otherwise not present in the object's dictionary, returns NO_VALUE.

value
Return the value of this attribute.

This operation is equivalent to accessing the object's attribute directly or via `getattr()`, and will fire off any pending loader callables if needed.

class `sqlalchemy.orm.util.`**CascadeOptions**
Bases: `__builtin__.frozenset`

Keeps track of the options sent to relationship().cascade

class `sqlalchemy.orm.instrumentation.`**ClassManager**(*class_*)
Bases: `__builtin__.dict`

tracks state information at the class level.

__le__
inherited from the `__le__` *attribute of* `dict`

x.__le__(y) <==> x<=y

__lt__
inherited from the `__lt__` *attribute of* `dict`

x.__lt__(y) <==> x<y

__ne__
inherited from the `__ne__` *attribute of* `dict`

x.__ne__(y) <==> x!=y

clear() → None. Remove all items from D.
inherited from the `clear()` *method of* `dict`

copy() → a shallow copy of D
inherited from the `copy()` *method of* `dict`

dispose()
Dissasociate this manager from its class.

fromkeys(*S*[, *v*]) → New dict with keys from S and values equal to v.
inherited from the `fromkeys()` *method of* `dict`

v defaults to None.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.
inherited from the `get()` *method of* `dict`

has_key(*k*) → True if D has a key k, else False
inherited from the `has_key()` *method of* `dict`

has_parent(*state*, *key*, *optimistic=False*)
TODO

items() → list of D's (key, value) pairs, as 2-tuples
inherited from the `items()` *method of* `dict`

iteritems() → an iterator over the (key, value) items of D
inherited from the `iteritems()` *method of* `dict`

iterkeys() → an iterator over the keys of D
inherited from the `iterkeys()` *method of* `dict`

itervalues() → an iterator over the values of D
inherited from the `itervalues()` *method of* `dict`

keys() → list of D's keys
inherited from the `keys()` *method of* `dict`

manage()
Mark this instance as the manager for its class.

original_init
x.__init__(...) initializes x; see help(type(x)) for signature

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
inherited from the `pop()` *method of* `dict`

If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
inherited from the `popitem()` *method of* `dict`

2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D
inherited from the `setdefault()` *method of* `dict`

state_getter()
Return a (instance) -> InstanceState callable.

"state getter" callables should raise either KeyError or AttributeError if no InstanceState could be found for the instance.

unregister()
remove all instrumentation established by this ClassManager.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
inherited from the `update()` *method of* `dict`

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → list of D's values
inherited from the `values()` *method of* `dict`

viewitems() → a set-like object providing a view on D's items
inherited from the `viewitems()` *method of* `dict`

viewkeys() → a set-like object providing a view on D's keys
inherited from the `viewkeys()` *method of* `dict`

viewvalues() → an object providing a view on D's values
inherited from the `viewvalues()` *method of* `dict`

class `sqlalchemy.orm.properties.`**`ColumnProperty`**(**columns*, ***kwargs*)
Bases: `sqlalchemy.orm.interfaces.StrategizedProperty`

Describes an object attribute that corresponds to a table column.

Public constructor is the *`orm.column_property()`* function.

class **`Comparator`**(*prop*, *parentmapper*, *adapt_to_entity=None*)
Bases: `sqlalchemy.util.langhelpers.MemoizedSlots`, *`sqlalchemy.orm.interfaces.PropComparator`*

Produce boolean, comparison, and other operators for `ColumnProperty` attributes.

See the documentation for `PropComparator` for a brief overview.

See also:

`PropComparator`

`ColumnOperators`

Redefining and Creating New Operators

`TypeEngine.comparator_factory`

__eq__(*other*)
inherited from the `__eq__()` *method of* `ColumnOperators`

Implement the == operator.

In a column context, produces the clause `a = b`. If the target is `None`, produces `a IS NULL`.

__le__(*other*)
inherited from the `__le__()` *method of* `ColumnOperators`

Implement the <= operator.

In a column context, produces the clause `a <= b`.

__lt__(*other*)
inherited from the `__lt__()` *method of* `ColumnOperators`

Implement the < operator.

In a column context, produces the clause `a < b`.

__ne__(*other*)
inherited from the `__ne__()` *method of* `ColumnOperators`

Implement the != operator.

In a column context, produces the clause `a != b`. If the target is `None`, produces `a IS NOT NULL`.

adapt_to_entity(*adapt_to_entity*)
inherited from the `adapt_to_entity()` *method of* `PropComparator`

Return a copy of this PropComparator which will use the given `AliasedInsp` to produce corresponding expressions.

adapter
inherited from the `adapter` *attribute of* `PropComparator`

Produce a callable that adapts column expressions to suit an aliased version of this comparator.

all_()
inherited from the `all_()` *method of* `ColumnOperators`

Produce a `all_()` clause against the parent object.

New in version 1.1.

any(*criterion=None*, ***kwargs*)
inherited from the `any()` *method of* `PropComparator`

Return true if this collection contains any member that meets the given criterion.

The usual implementation of `any()` is `RelationshipProperty.Comparator.any()`.

Parameters

- **criterion** – an optional ClauseElement formulated against the member class' table or attributes.
- ****kwargs** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values.

any_()

inherited from the `any_()` *method of* `ColumnOperators`

Produce a `any_()` clause against the parent object.

New in version 1.1.

asc()

inherited from the `asc()` *method of* `ColumnOperators`

Produce a `asc()` clause against the parent object.

between(*cleft*, *cright*, *symmetric=False*)

inherited from the `between()` *method of* `ColumnOperators`

Produce a `between()` clause against the parent object, given the lower and upper range.

collate(*collation*)

inherited from the `collate()` *method of* `ColumnOperators`

Produce a `collate()` clause against the parent object, given the collation string.

concat(*other*)

inherited from the `concat()` *method of* `ColumnOperators`

Implement the 'concat' operator.

In a column context, produces the clause `a || b`, or uses the `concat()` operator on MySQL.

contains(*other*, ***kwargs*)

inherited from the `contains()` *method of* `ColumnOperators`

Implement the 'contains' operator.

In a column context, produces the clause `LIKE '%<other>%'`

desc()

inherited from the `desc()` *method of* `ColumnOperators`

Produce a `desc()` clause against the parent object.

distinct()

inherited from the `distinct()` *method of* `ColumnOperators`

Produce a `distinct()` clause against the parent object.

endswith(*other*, ***kwargs*)

inherited from the `endswith()` *method of* `ColumnOperators`

Implement the 'endswith' operator.

In a column context, produces the clause `LIKE '%<other>'`

has(*criterion=None*, ***kwargs*)

inherited from the `has()` *method of* `PropComparator`

Return true if this element references a member which meets the given criterion.

The usual implementation of `has()` is `RelationshipProperty.Comparator.has()`.

Parameters

- **criterion** – an optional ClauseElement formulated against the member class' table or attributes.
- ****kwargs** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values.

ilike(*other*, *escape=None*)

inherited from the `ilike()` *method of* `ColumnOperators`

Implement the `ilike` operator.

In a column context, produces the clause `a ILIKE other`.

E.g.:

```
select([sometable]).where(sometable.c.column.ilike("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the `ESCAPE` keyword, e.g.:

```
somecolumn.ilike("foo/%bar", escape="/")
```

See also:

`ColumnOperators.like()`

in_(*other*)

inherited from the `in_()` *method of* `ColumnOperators`

Implement the `in` operator.

In a column context, produces the clause `a IN other`. "other" may be a tuple/list of column expressions, or a `select()` construct.

is_(*other*)

inherited from the `is_()` *method of* `ColumnOperators`

Implement the `IS` operator.

Normally, `IS` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

`ColumnOperators.isnot()`

isnot(*other*)

inherited from the `isnot()` *method of* `ColumnOperators`

Implement the `IS NOT` operator.

Normally, `IS NOT` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS NOT` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.is_()

like(*other*, *escape=None*)
inherited from the like() method of ColumnOperators

Implement the `like` operator.

In a column context, produces the clause `a LIKE other`.

E.g.:

```
select([sometable]).where(sometable.c.column.like("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the `ESCAPE` keyword, e.g.:

```
somecolumn.like("foo/%bar", escape="/")
```

See also:

ColumnOperators.ilike()

match(*other*, ***kwargs*)
inherited from the match() method of ColumnOperators

Implements a database-specific 'match' operator.

match() attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

- Postgresql - renders `x @@ to_tsquery(y)`
- MySQL - renders `MATCH (x) AGAINST (y IN BOOLEAN MODE)`
- Oracle - renders `CONTAINS(x, y)`
- other backends may provide special implementations.
- Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

notilike(*other*, *escape=None*)
inherited from the notilike() method of ColumnOperators

implement the `NOT ILIKE` operator.

This is equivalent to using negation with *ColumnOperators.ilike()*, i.e. `~x.ilike(y)`.

New in version 0.8.

See also:

ColumnOperators.ilike()

notin_(*other*)
inherited from the notin_() method of ColumnOperators

implement the `NOT IN` operator.

This is equivalent to using negation with *ColumnOperators.in_()*, i.e. `~x.in_(y)`.

New in version 0.8.

See also:

ColumnOperators.in_()

notlike(*other*, *escape=None*)
inherited from the notlike() method of ColumnOperators

implement the `NOT LIKE` operator.

This is equivalent to using negation with *ColumnOperators.like()*, i.e. `~x.like(y)`.

New in version 0.8.

See also:

ColumnOperators.like()

nullsfirst()
inherited from the nullsfirst() method of ColumnOperators

Produce a *nullsfirst()* clause against the parent object.

nullslast()
inherited from the nullslast() method of ColumnOperators

Produce a *nullslast()* clause against the parent object.

of_type(*class_*)
inherited from the of_type() method of PropComparator

Redefine this object in terms of a polymorphic subclass.

Returns a new PropComparator from which further criterion can be evaluated.

e.g.:

```
query.join(Company.employees.of_type(Engineer)).\
   filter(Engineer.name=='foo')
```

Parameters **class_** – a class or mapper indicating that criterion will be against this specific subclass.

op(*opstring*, *precedence=0*, *is_comparison=False*)
inherited from the op() method of Operators

produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

This function can also be used to make bitwise operators explicit. For example:

```
somecolumn.op('&')(0xff)
```

is a bitwise AND of the value in `somecolumn`.

Parameters

- **operator** – a string which will be output as the infix operator between this element and the expression passed to the generated function.

- **precedence** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of `0` is lower than all operators except for the comma (`,`) and `AS` operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.

- **is_comparison** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like ==, >, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

  New in version 0.9.2: - added the `Operators.op.is_comparison` flag.

See also:

Redefining and Creating New Operators

Using custom operators in join conditions

startswith(*other*, ***kwargs*)

inherited from the `startswith()` *method of* `ColumnOperators`

Implement the `startwith` operator.

In a column context, produces the clause `LIKE '<other>%'`

`ColumnProperty.`**`__init__`**(**columns*, ***kwargs*)

Construct a new `ColumnProperty` object.

This constructor is mirrored as a public API function; see `column_property()` for a full usage and argument description.

`ColumnProperty.`**`cascade_iterator`**(*type_*, *state*, *visited_instances=None*, *halt_on=None*)

inherited from the `cascade_iterator()` *method of* `MapperProperty`

Iterate through instances related to the given instance for a particular 'cascade', starting with this MapperProperty.

Return an iterator3-tuples (instance, mapper, state).

Note that the 'cascade' collection on this MapperProperty is checked first for the given type before cascade_iterator is called.

This method typically only applies to RelationshipProperty.

`ColumnProperty.`**`class_attribute`**

inherited from the `class_attribute` *attribute of* `MapperProperty`

Return the class-bound descriptor corresponding to this `MapperProperty`.

This is basically a `getattr()` call:

```
return getattr(self.parent.class_, self.key)
```

I.e. if this `MapperProperty` were named `addresses`, and the class to which it is mapped is `User`, this sequence is possible:

```
>>> from sqlalchemy import inspect
>>> mapper = inspect(User)
>>> addresses_property = mapper.attrs.addresses
>>> addresses_property.class_attribute is User.addresses
True
```

```
>>> User.addresses.property is addresses_property
True
```

`ColumnProperty.expression`
Return the primary column or expression for this ColumnProperty.

`ColumnProperty.extension_type` = symbol('NOT_EXTENSION')

`ColumnProperty.init()`
inherited from the `init()` method of `MapperProperty`

Called after all mappers are created to assemble relationships between mappers and perform other post-mapper-creation initialization steps.

`ColumnProperty.set_parent(parent, init)`
inherited from the `set_parent()` method of `MapperProperty`

Set the parent mapper that references this MapperProperty.

This method is overridden by some subclasses to perform extra setup when the mapper is first known.

class `sqlalchemy.orm.properties.ComparableProperty(comparator_factory, descriptor=None, doc=None, info=None)`
Bases: `sqlalchemy.orm.descriptor_props.DescriptorProperty`

Instruments a Python property for use in query expressions.

`__init__(comparator_factory, descriptor=None, doc=None, info=None)`
Construct a new `ComparableProperty` object.

This constructor is mirrored as a public API function; see `comparable_property()` for a full usage and argument description.

class `sqlalchemy.orm.descriptor_props.CompositeProperty(class_, *attrs, **kwargs)`
Bases: `sqlalchemy.orm.descriptor_props.DescriptorProperty`

Defines a "composite" mapped attribute, representing a collection of columns as one attribute.

`CompositeProperty` is constructed using the `composite()` function.

See also:

Composite Column Types

class `Comparator(prop, parentmapper, adapt_to_entity=None)`
Bases: `sqlalchemy.orm.interfaces.PropComparator`

Produce boolean, comparison, and other operators for `CompositeProperty` attributes.

See the example in *Redefining Comparison Operations for Composites* for an overview of usage , as well as the documentation for `PropComparator`.

See also:

`PropComparator`

`ColumnOperators`

Redefining and Creating New Operators

`TypeEngine.comparator_factory`

`CompositeProperty.__init__(class_, *attrs, **kwargs)`
Construct a new `CompositeProperty` object.

This constructor is mirrored as a public API function; see `composite()` for a full usage and argument description.

`CompositeProperty.do_init()`
Initialization which occurs after the *CompositeProperty* has been associated with its parent mapper.

`CompositeProperty.get_history(state, dict_, passive=symbol('PASSIVE_OFF'))`
Provided for userland code that uses attributes.get_history().

class `sqlalchemy.orm.attributes.Event(attribute_impl, op)`
A token propagated throughout the course of a chain of attribute events.

Serves as an indicator of the source of the event and also provides a means of controlling propagation across a chain of attribute operations.

The *Event* object is sent as the `initiator` argument when dealing with the *AttributeEvents.append()*, *AttributeEvents.set()*, and *AttributeEvents.remove()* events.

The *Event* object is currently interpreted by the backref event handlers, and is used to control the propagation of operations across two mutually-dependent attributes.

New in version 0.9.0.

Variables

- ***impl*** – The `AttributeImpl` which is the current event initiator.
- ***op*** – The symbol `OP_APPEND`, `OP_REMOVE` or `OP_REPLACE`, indicating the source operation.

class `sqlalchemy.orm.identity.IdentityMap`

check_modified()
return True if any InstanceStates present have been marked as 'modified'.

class `sqlalchemy.orm.base.InspectionAttr`
A base class applied to all ORM objects that can be returned by the *inspect()* function.

The attributes defined here allow the usage of simple boolean checks to test basic facts about the object returned.

While the boolean checks here are basically the same as using the Python isinstance() function, the flags here can be used without the need to import all of these classes, and also such that the SQLAlchemy class system can change while leaving the flags here intact for forwards-compatibility.

extension_type = symbol('NOT_EXTENSION')
The extension type, if any. Defaults to *interfaces.NOT_EXTENSION*

New in version 0.8.0.

See also:

HYBRID_METHOD

HYBRID_PROPERTY

ASSOCIATION_PROXY

is_aliased_class = False
True if this object is an instance of *AliasedClass*.

is_attribute = False
True if this object is a Python descriptor.

This can refer to one of many types. Usually a *QueryableAttribute* which handles attributes events on behalf of a *MapperProperty*. But can also be an extension type such as *AssociationProxy*

or *hybrid_property*. The *InspectionAttr.extension_type* will refer to a constant identifying the specific subtype.

See also:

Mapper.all_orm_descriptors

is_clause_element = **False**
True if this object is an instance of *ClauseElement*.

is_instance = **False**
True if this object is an instance of *InstanceState*.

is_mapper = **False**
True if this object is an instance of *Mapper*.

is_property = **False**
True if this object is an instance of *MapperProperty*.

is_selectable = **False**
Return True if this object is an instance of *Selectable*.

class sqlalchemy.orm.base.**InspectionAttrInfo**
Bases: *sqlalchemy.orm.base.InspectionAttr*

Adds the `.info` attribute to *InspectionAttr*.

The rationale for *InspectionAttr* vs. *InspectionAttrInfo* is that the former is compatible as a mixin for classes that specify `__slots__`; this is essentially an implementation artifact.

info
Info dictionary associated with the object, allowing user-defined data to be associated with this *InspectionAttr*.

The dictionary is generated when first accessed. Alternatively, it can be specified as a constructor argument to the *column_property()*, *relationship()*, or *composite()* functions.

New in version 0.8: Added support for .info to all *MapperProperty* subclasses.

Changed in version 1.0.0: *MapperProperty.info* is also available on extension types via the *InspectionAttrInfo.info* attribute, so that it can apply to a wider variety of ORM and extension constructs.

See also:

QueryableAttribute.info

SchemaItem.info

class sqlalchemy.orm.state.**InstanceState**(*obj*, *manager*)
Bases: *sqlalchemy.orm.base.InspectionAttr*

tracks state information at the instance level.

The *InstanceState* is a key object used by the SQLAlchemy ORM in order to track the state of an object; it is created the moment an object is instantiated, typically as a result of instrumentation which SQLAlchemy applies to the `__init__()` method of the class.

InstanceState is also a semi-public object, available for runtime inspection as to the state of a mapped instance, including information such as its current status within a particular *Session* and details about data on individual attributes. The public API in order to acquire a *InstanceState* object is to use the *inspect()* system:

```
>>> from sqlalchemy import inspect
>>> insp = inspect(some_mapped_object)
```

See also:

Runtime Inspection API

attrs

Return a namespace representing each attribute on the mapped object, including its current value and history.

The returned object is an instance of `AttributeState`. This object allows inspection of the current data within an attribute as well as attribute history since the last flush.

callables = ()

A namespace where a per-state loader callable can be associated.

In SQLAlchemy 1.0, this is only used for lazy loaders / deferred loaders that were set up via query option.

Previously, callables was used also to indicate expired attributes by storing a link to the InstanceState itself in this dictionary. This role is now handled by the expired_attributes set.

deleted

Return true if the object is deleted.

An object that is in the deleted state is guaranteed to not be within the `Session.identity_map` of its parent `Session`; however if the session's transaction is rolled back, the object will be restored to the persistent state and the identity map.

Note: The `InstanceState.deleted` attribute refers to a specific state of the object that occurs between the "persistent" and "detached" states; once the object is detached, the `InstanceState.deleted` attribute **no longer returns True**; in order to detect that a state was deleted, regardless of whether or not the object is associated with a `Session`, use the `InstanceState.was_deleted` accessor.

See also:

Quickie Intro to Object States

detached

Return true if the object is detached.

See also:

Quickie Intro to Object States

dict

Return the instance dict used by the object.

Under normal circumstances, this is always synonymous with the `__dict__` attribute of the mapped object, unless an alternative instrumentation system has been configured.

In the case that the actual object has been garbage collected, this accessor returns a blank dictionary.

expired_attributes = None

The set of keys which are 'expired' to be loaded by the manager's deferred scalar loader, assuming no pending changes.

see also the `unmodified` collection which is intersected against this set when a refresh operation occurs.

has_identity

Return `True` if this object has an identity key.

This should always have the same value as the expression `state.persistent or state.detached`.

identity

Return the mapped identity of the mapped object. This is the primary key identity as persisted by the ORM which can always be passed directly to *Query.get()*.

Returns `None` if the object has no primary key identity.

Note: An object which is transient or pending does **not** have a mapped identity until it is flushed, even if its attributes include primary key values.

---

identity_key

Return the identity key for the mapped object.

This is the key used to locate the object within the *Session.identity_map* mapping. It contains the identity as returned by *identity* within it.

mapper

Return the *Mapper* used for this mapepd object.

object

Return the mapped object represented by this *InstanceState*.

pending

Return true if the object is pending.

See also:

Quickie Intro to Object States

persistent

Return true if the object is persistent.

An object that is in the persistent state is guaranteed to be within the *Session.identity_map* of its parent *Session*.

Changed in version 1.1: The *InstanceState.persistent* accessor no longer returns True for an object that was "deleted" within a flush; use the *InstanceState.deleted* accessor to detect this state. This allows the "persistent" state to guarantee membership in the identity map.

See also:

Quickie Intro to Object States

session

Return the owning *Session* for this instance, or `None` if none available.

Note that the result here can in some cases be *different* from that of `obj in session`; an object that's been deleted will report as not `in session`, however if the transaction is still in progress, this attribute will still refer to that session. Only when the transaction is completed does the object become fully detached under normal circumstances.

transient

Return true if the object is transient.

See also:

Quickie Intro to Object States

unloaded

Return the set of keys which do not have a loaded value.

This includes expired attributes and any other attribute that was never populated or modified.

unmodified
Return the set of keys which have no uncommitted changes

unmodified_intersection(*keys*)
Return self.unmodified.intersection(keys).

was_deleted
Return True if this object is or was previously in the "deleted" state and has not been reverted to persistent.

This flag returns True once the object was deleted in flush. When the object is expunged from the session either explicitly or via transaction commit and enters the "detached" state, this flag will continue to report True.

New in version 1.1: - added a local method form of `orm.util.was_deleted()`.

See also:

`InstanceState.deleted` - refers to the "deleted" state

`orm.util.was_deleted()` - standalone function

Quickie Intro to Object States

class sqlalchemy.orm.attributes.**InstrumentedAttribute**(*class_*, *key*, *impl=None*, *comparator=None*, *parententity=None*, *of_type=None*)

Bases: `sqlalchemy.orm.attributes.QueryableAttribute`

Class bound instrumented attribute which adds basic descriptor methods.

See `QueryableAttribute` for a description of most features.

__delete__(*instance*)

__get__(*instance*, *owner*)

__set__(*instance*, *value*)

sqlalchemy.orm.interfaces.**MANYTOONE = symbol('MANYTOONE')**
Indicates the many-to-one direction for a `relationship()`.

This symbol is typically used by the internals but may be exposed within certain API features.

sqlalchemy.orm.interfaces.**MANYTOMANY = symbol('MANYTOMANY')**
Indicates the many-to-many direction for a `relationship()`.

This symbol is typically used by the internals but may be exposed within certain API features.

class sqlalchemy.orm.interfaces.**MapperProperty**

Bases: sqlalchemy.orm.base._MappedAttribute, `sqlalchemy.orm.base.InspectionAttr`, sqlalchemy.util.langhelpers.MemoizedSlots

Represent a particular class attribute mapped by `Mapper`.

The most common occurrences of `MapperProperty` are the mapped `Column`, which is represented in a mapping as an instance of `ColumnProperty`, and a reference to another class produced by `relationship()`, represented in the mapping as an instance of `RelationshipProperty`.

info
Info dictionary associated with the object, allowing user-defined data to be associated with this `InspectionAttr`.

The dictionary is generated when first accessed. Alternatively, it can be specified as a constructor argument to the `column_property()`, `relationship()`, or `composite()` functions.

New in version 0.8: Added support for .info to all `MapperProperty` subclasses.

Changed in version 1.0.0: `InspectionAttr.info` moved from *MapperProperty* so that it can apply to a wider variety of ORM and extension constructs.

See also:

QueryableAttribute.info

SchemaItem.info

cascade = frozenset([])
The set of 'cascade' attribute names.

This collection is checked before the 'cascade_iterator' method is called.

The collection typically only applies to a RelationshipProperty.

cascade_iterator(*type_*, *state*, *visited_instances=None*, *halt_on=None*)
Iterate through instances related to the given instance for a particular 'cascade', starting with this MapperProperty.

Return an iterator3-tuples (instance, mapper, state).

Note that the 'cascade' collection on this MapperProperty is checked first for the given type before cascade_iterator is called.

This method typically only applies to RelationshipProperty.

class_attribute
Return the class-bound descriptor corresponding to this *MapperProperty*.

This is basically a `getattr()` call:

```
return getattr(self.parent.class_, self.key)
```

I.e. if this *MapperProperty* were named `addresses`, and the class to which it is mapped is `User`, this sequence is possible:

```
>>> from sqlalchemy import inspect
>>> mapper = inspect(User)
>>> addresses_property = mapper.attrs.addresses
>>> addresses_property.class_attribute is User.addresses
True
>>> User.addresses.property is addresses_property
True
```

create_row_processor(*context*, *path*, *mapper*, *result*, *adapter*, *populators*)
Produce row processing functions and append to the given set of populators lists.

do_init()
Perform subclass-specific initialization post-mapper-creation steps.

This is a template method called by the `MapperProperty` object's init() method.

init()
Called after all mappers are created to assemble relationships between mappers and perform other post-mapper-creation initialization steps.

instrument_class(*mapper*)
Hook called by the Mapper to the property to initiate instrumentation of the class attribute managed by this MapperProperty.

The MapperProperty here will typically call out to the attributes module to set up an InstrumentedAttribute.

This step is the first of two steps to set up an InstrumentedAttribute, and is called early in the mapper setup process.

The second step is typically the init_class_attribute step, called from StrategizedProperty via the post_instrument_class() hook. This step assigns additional state to the InstrumentedAttribute (specifically the "impl") which has been determined after the MapperProperty has determined what kind of persistence management it needs to do (e.g. scalar, object, collection, etc).

is_property = **True**
Part of the InspectionAttr interface; states this object is a mapper property.

merge(*session*, *source_state*, *source_dict*, *dest_state*, *dest_dict*, *load*, *_recursive*, *_resolve_conflict_map*)
Merge the attribute represented by this `MapperProperty` from source to destination object.

post_instrument_class(*mapper*)
Perform instrumentation adjustments that need to occur after init() has completed.

The given Mapper is the Mapper invoking the operation, which may not be the same Mapper as self.parent in an inheritance scenario; however, Mapper will always at least be a sub-mapper of self.parent.

This method is typically used by StrategizedProperty, which delegates it to LoaderStrategy.init_class_attribute() to perform final setup on the class-bound InstrumentedAttribute.

set_parent(*parent*, *init*)
Set the parent mapper that references this MapperProperty.

This method is overridden by some subclasses to perform extra setup when the mapper is first known.

setup(*context*, *entity*, *path*, *adapter*, ***kwargs*)
Called by Query for the purposes of constructing a SQL statement.

Each MapperProperty associated with the target mapper processes the statement referenced by the query context, adding columns and/or criterion as appropriate.

`sqlalchemy.orm.interfaces.`**NOT_EXTENSION** = **symbol('NOT_EXTENSION')**
Symbol indicating an `InspectionAttr` that's not part of sqlalchemy.ext.

Is assigned to the *`InspectionAttr.extension_type`* attibute.

`sqlalchemy.orm.interfaces.`**ONETOMANY** = **symbol('ONETOMANY')**
Indicates the one-to-many direction for a *`relationship()`*.

This symbol is typically used by the internals but may be exposed within certain API features.

class `sqlalchemy.orm.interfaces.`**PropComparator**(*prop*, *parentmapper*, *adapt_to_entity=None*)
Bases: *`sqlalchemy.sql.operators.ColumnOperators`*

Defines SQL operators for *`MapperProperty`* objects.

SQLAlchemy allows for operators to be redefined at both the Core and ORM level. *`PropComparator`* is the base class of operator redefinition for ORM-level operations, including those of *`ColumnProperty`*, *`RelationshipProperty`*, and *`CompositeProperty`*.

Note: With the advent of Hybrid properties introduced in SQLAlchemy 0.7, as well as Core-level operator redefinition in SQLAlchemy 0.8, the use case for user-defined *`PropComparator`* instances is extremely rare. See *Hybrid Attributes* as well as *Redefining and Creating New Operators*.

User-defined subclasses of *`PropComparator`* may be created. The built-in Python comparison and math operator methods, such as *`operators.ColumnOperators.__eq__()`*, *`operators.ColumnOperators.__lt__()`*, and *`operators.ColumnOperators.__add__()`*, can be overridden to provide new operator behavior. The custom *`PropComparator`* is passed to the

MapperProperty instance via the comparator_factory argument. In each case, the appropriate subclass of *PropComparator* should be used:

```
# definition of custom PropComparator subclasses

from sqlalchemy.orm.properties import \
                        ColumnProperty,\
                        CompositeProperty,\
                        RelationshipProperty

class MyColumnComparator(ColumnProperty.Comparator):
    def __eq__(self, other):
        return self.__clause_element__() == other

class MyRelationshipComparator(RelationshipProperty.Comparator):
    def any(self, expression):
        "define the 'any' operation"
        # ...

class MyCompositeComparator(CompositeProperty.Comparator):
    def __gt__(self, other):
        "redefine the 'greater than' operation"

        return sql.and_(*[a>b for a, b in
                          zip(self.__clause_element__().clauses,
                              other.__composite_values__())])


# application of custom PropComparator subclasses

from sqlalchemy.orm import column_property, relationship, composite
from sqlalchemy import Column, String

class SomeMappedClass(Base):
    some_column = column_property(Column("some_column", String),
                        comparator_factory=MyColumnComparator)

    some_relationship = relationship(SomeOtherClass,
                        comparator_factory=MyRelationshipComparator)

    some_composite = composite(
            Column("a", String), Column("b", String),
            comparator_factory=MyCompositeComparator
        )
```

Note that for column-level operator redefinition, it's usually simpler to define the operators at the Core level, using the *TypeEngine.comparator_factory* attribute. See *Redefining and Creating New Operators* for more detail.

See also:

ColumnProperty.Comparator

RelationshipProperty.Comparator

CompositeProperty.Comparator

ColumnOperators

Redefining and Creating New Operators

TypeEngine.comparator_factory

__eq__(*other*)
inherited from the __eq__() method of ColumnOperators

Implement the == operator.

In a column context, produces the clause `a = b`. If the target is `None`, produces `a IS NULL`.

__le__(*other*)
inherited from the __le__() method of ColumnOperators

Implement the <= operator.

In a column context, produces the clause `a <= b`.

__lt__(*other*)
inherited from the __lt__() method of ColumnOperators

Implement the < operator.

In a column context, produces the clause `a < b`.

__ne__(*other*)
inherited from the __ne__() method of ColumnOperators

Implement the != operator.

In a column context, produces the clause `a != b`. If the target is `None`, produces `a IS NOT NULL`.

adapt_to_entity(*adapt_to_entity*)
Return a copy of this PropComparator which will use the given *AliasedInsp* to produce corresponding expressions.

adapter
Produce a callable that adapts column expressions to suit an aliased version of this comparator.

all_()
inherited from the all_() method of ColumnOperators

Produce a *all_()* clause against the parent object.

New in version 1.1.

any(*criterion=None*, ***kwargs*)
Return true if this collection contains any member that meets the given criterion.

The usual implementation of `any()` is *RelationshipProperty.Comparator.any()*.

Parameters

- **criterion** – an optional ClauseElement formulated against the member class' table or attributes.
- ****kwargs** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values.

any_()
inherited from the any_() method of ColumnOperators

Produce a *any_()* clause against the parent object.

New in version 1.1.

asc()
inherited from the asc() method of ColumnOperators

Produce a `asc()` clause against the parent object.

between(*cleft*, *cright*, *symmetric=False*)
inherited from the `between()` *method of* `ColumnOperators`

Produce a `between()` clause against the parent object, given the lower and upper range.

collate(*collation*)
inherited from the `collate()` *method of* `ColumnOperators`

Produce a `collate()` clause against the parent object, given the collation string.

concat(*other*)
inherited from the `concat()` *method of* `ColumnOperators`

Implement the 'concat' operator.

In a column context, produces the clause `a || b`, or uses the `concat()` operator on MySQL.

contains(*other*, ***kwargs*)
inherited from the `contains()` *method of* `ColumnOperators`

Implement the 'contains' operator.

In a column context, produces the clause `LIKE '%<other>%'`

desc()
inherited from the `desc()` *method of* `ColumnOperators`

Produce a `desc()` clause against the parent object.

distinct()
inherited from the `distinct()` *method of* `ColumnOperators`

Produce a `distinct()` clause against the parent object.

endswith(*other*, ***kwargs*)
inherited from the `endswith()` *method of* `ColumnOperators`

Implement the 'endswith' operator.

In a column context, produces the clause `LIKE '%<other>'`

has(*criterion=None*, ***kwargs*)
Return true if this element references a member which meets the given criterion.

The usual implementation of `has()` is `RelationshipProperty.Comparator.has()`.

Parameters

- **criterion** – an optional ClauseElement formulated against the member class' table or attributes.
- ****kwargs** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values.

ilike(*other*, *escape=None*)
inherited from the `ilike()` *method of* `ColumnOperators`

Implement the `ilike` operator.

In a column context, produces the clause `a ILIKE other`.

E.g.:

```
select([sometable]).where(sometable.c.column.ilike("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the ESCAPE keyword, e.g.:

```
somecolumn.ilike("foo/%bar", escape="/")
```

See also:

ColumnOperators.like()

in_(*other*)

inherited from the in_() method of ColumnOperators

Implement the in operator.

In a column context, produces the clause a IN other. "other" may be a tuple/list of column expressions, or a *select()* construct.

is_(*other*)

inherited from the is_() method of ColumnOperators

Implement the IS operator.

Normally, IS is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.isnot()

isnot(*other*)

inherited from the isnot() method of ColumnOperators

Implement the IS NOT operator.

Normally, IS NOT is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS NOT may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.is_()

like(*other*, *escape=None*)

inherited from the like() method of ColumnOperators

Implement the like operator.

In a column context, produces the clause a LIKE other.

E.g.:

```
select([sometable]).where(sometable.c.column.like("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the ESCAPE keyword, e.g.:

```
somecolumn.like("foo/%bar", escape="/")
```

See also:

ColumnOperators.ilike()

match(*other*, ***kwargs*)

inherited from the match() method of ColumnOperators

Implements a database-specific 'match' operator.

match() attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

- Postgresql - renders `x @@ to_tsquery(y)`
- MySQL - renders `MATCH (x) AGAINST (y IN BOOLEAN MODE)`
- Oracle - renders `CONTAINS(x, y)`
- other backends may provide special implementations.
- Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

notilike(*other*, *escape=None*)

inherited from the notilike() method of ColumnOperators

implement the `NOT ILIKE` operator.

This is equivalent to using negation with *ColumnOperators.ilike()*, i.e. `~x.ilike(y)`.

New in version 0.8.

See also:

ColumnOperators.ilike()

notin_(*other*)

inherited from the notin_() method of ColumnOperators

implement the `NOT IN` operator.

This is equivalent to using negation with *ColumnOperators.in_()*, i.e. `~x.in_(y)`.

New in version 0.8.

See also:

ColumnOperators.in_()

notlike(*other*, *escape=None*)

inherited from the notlike() method of ColumnOperators

implement the `NOT LIKE` operator.

This is equivalent to using negation with *ColumnOperators.like()*, i.e. `~x.like(y)`.

New in version 0.8.

See also:

ColumnOperators.like()

nullsfirst()

inherited from the `nullsfirst()` *method of* `ColumnOperators`

Produce a `nullsfirst()` clause against the parent object.

nullslast()

inherited from the `nullslast()` *method of* `ColumnOperators`

Produce a `nullslast()` clause against the parent object.

of_type(*class_*)

Redefine this object in terms of a polymorphic subclass.

Returns a new PropComparator from which further criterion can be evaluated.

e.g.:

```
query.join(Company.employees.of_type(Engineer)).\
   filter(Engineer.name=='foo')
```

Parameters **class_** – a class or mapper indicating that criterion will be against this specific subclass.

op(*opstring*, *precedence=0*, *is_comparison=False*)

inherited from the `op()` *method of* `Operators`

produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

This function can also be used to make bitwise operators explicit. For example:

```
somecolumn.op('&')(0xff)
```

is a bitwise AND of the value in `somecolumn`.

Parameters

- **operator** – a string which will be output as the infix operator between this element and the expression passed to the generated function.
- **precedence** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of `0` is lower than all operators except for the comma (`,`) and `AS` operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.
- **is_comparison** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like ==, >, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

  New in version 0.9.2: - added the `Operators.op.is_comparison` flag.

See also:

Redefining and Creating New Operators

Using custom operators in join conditions

operate(*op*, **other*, ***kwargs*)

inherited from the `operate()` *method of* `Operators`

Operate on an argument.

This is the lowest level of operation, raises `NotImplementedError` by default.

Overriding this on a subclass can allow common behavior to be applied to all operations. For example, overriding `ColumnOperators` to apply `func.lower()` to the left and right side:

```
class MyComparator(ColumnOperators):
    def operate(self, op, other):
        return op(func.lower(self), func.lower(other))
```

Parameters

- **op** – Operator callable.
- ***other** – the 'other' side of the operation. Will be a single scalar for most operations.
- ****kwargs** – modifiers. These may be passed by special operators such as `ColumnOperators.contains()`.

reverse_operate(*op*, *other*, ***kwargs*)

inherited from the `reverse_operate()` *method of* `Operators`

Reverse operate on an argument.

Usage is the same as `operate()`.

startswith(*other*, ***kwargs*)

inherited from the `startswith()` *method of* `ColumnOperators`

Implement the `startwith` operator.

In a column context, produces the clause `LIKE '<other>%'`

class sqlalchemy.orm.properties.**RelationshipProperty**(*argument*, *secondary=None*, *primaryjoin=None*, *secondaryjoin=None*, *foreign_keys=None*, *uselist=None*, *order_by=False*, *backref=None*, *back_populates=None*, *post_update=False*, *cascade=False*, *extension=None*, *viewonly=False*, *lazy=True*, *collection_class=None*, *passive_deletes=False*, *passive_updates=True*, *remote_side=None*, *enable_typechecks=True*, *join_depth=None*, *comparator_factory=None*, *single_parent=False*, *innerjoin=False*, *distinct_target_key=None*, *doc=None*, *active_history=False*, *cascade_backrefs=True*, *load_on_pending=False*, *bake_queries=True*, *strategy_class=None*, *_local_remote_pairs=None*, *query_class=None*, *info=None*)

Bases: sqlalchemy.orm.interfaces.StrategizedProperty

Describes an object property that holds a single item or list of items that correspond to a related database table.

Public constructor is the *orm.relationship()* function.

See also:

Relationship Configuration

class **Comparator**(*prop*, *parentmapper*, *adapt_to_entity=None*, *of_type=None*)

Bases: *sqlalchemy.orm.interfaces.PropComparator*

Produce boolean, comparison, and other operators for *RelationshipProperty* attributes.

See the documentation for *PropComparator* for a brief overview of ORM level operator definition.

See also:

PropComparator

ColumnProperty.Comparator

ColumnOperators

Redefining and Creating New Operators

TypeEngine.comparator_factory

__eq__(*other*)

Implement the == operator.

In a many-to-one context, such as:

```
MyClass.some_prop == <some object>
```

this will typically produce a clause such as:

```
mytable.related_id == <some id>
```

Where `<some id>` is the primary key of the given object.

The == operator provides partial functionality for non- many-to-one comparisons:

- Comparisons against collections are not supported. Use `contains()`.
- Compared to a scalar one-to-many, will produce a clause that compares the target columns in the parent to the given target.
- Compared to a scalar many-to-many, an alias of the association table will be rendered as well, forming a natural join that is part of the main body of the query. This will not work for queries that go beyond simple AND conjunctions of comparisons, such as those which use OR. Use explicit joins, outerjoins, or `has()` for more comprehensive non-many-to-one scalar membership tests.
- Comparisons against `None` given in a one-to-many or many-to-many context produce a NOT EXISTS clause.

__init__(*prop*, *parentmapper*, *adapt_to_entity=None*, *of_type=None*)

Construction of `RelationshipProperty.Comparator` is internal to the ORM's attribute mechanics.

__le__(*other*)

inherited from the `__le__()` *method of* `ColumnOperators`

Implement the <= operator.

In a column context, produces the clause `a <= b`.

__lt__(*other*)

inherited from the `__lt__()` *method of* `ColumnOperators`

Implement the < operator.

In a column context, produces the clause `a < b`.

__ne__(*other*)

Implement the != operator.

In a many-to-one context, such as:

```
MyClass.some_prop != <some object>
```

This will typically produce a clause such as:

```
mytable.related_id != <some id>
```

Where `<some id>` is the primary key of the given object.

The != operator provides partial functionality for non- many-to-one comparisons:

- Comparisons against collections are not supported. Use `contains()` in conjunction with `not_()`.
- Compared to a scalar one-to-many, will produce a clause that compares the target columns in the parent to the given target.
- Compared to a scalar many-to-many, an alias of the association table will be rendered as well, forming a natural join that is part of the main body of the query. This will not work for queries that go beyond simple AND conjunctions of comparisons, such as those which use OR. Use

explicit joins, outerjoins, or *has()* in conjunction with *not_()* for more comprehensive non-many-to-one scalar membership tests.

- Comparisons against `None` given in a one-to-many or many-to-many context produce an EXISTS clause.

adapter

inherited from the *adapter* *attribute of* *PropComparator*

Produce a callable that adapts column expressions to suit an aliased version of this comparator.

all_()

inherited from the *all_()* *method of* *ColumnOperators*

Produce a *all_()* clause against the parent object.

New in version 1.1.

any(*criterion=None*, ***kwargs*)

Produce an expression that tests a collection against particular criterion, using EXISTS.

An expression like:

```
session.query(MyClass).filter(
    MyClass.somereference.any(SomeRelated.x==2)
)
```

Will produce a query like:

```
SELECT * FROM my_table WHERE
EXISTS (SELECT 1 FROM related WHERE related.my_id=my_table.id
AND related.x=2)
```

Because *any()* uses a correlated subquery, its performance is not nearly as good when compared against large target tables as that of using a join.

any() is particularly useful for testing for empty collections:

```
session.query(MyClass).filter(
    ~MyClass.somereference.any()
)
```

will produce:

```
SELECT * FROM my_table WHERE
NOT EXISTS (SELECT 1 FROM related WHERE
related.my_id=my_table.id)
```

any() is only valid for collections, i.e. a *relationship()* that has `uselist=True`. For scalar references, use *has()*.

any_()

inherited from the *any_()* *method of* *ColumnOperators*

Produce a *any_()* clause against the parent object.

New in version 1.1.

asc()

inherited from the *asc()* *method of* *ColumnOperators*

Produce a *asc()* clause against the parent object.

between(*cleft*, *cright*, *symmetric=False*)
inherited from the `between()` *method of* `ColumnOperators`

Produce a `between()` clause against the parent object, given the lower and upper range.

collate(*collation*)
inherited from the `collate()` *method of* `ColumnOperators`

Produce a `collate()` clause against the parent object, given the collation string.

concat(*other*)
inherited from the `concat()` *method of* `ColumnOperators`

Implement the 'concat' operator.

In a column context, produces the clause `a || b`, or uses the `concat()` operator on MySQL.

contains(*other*, ***kwargs*)
Return a simple expression that tests a collection for containment of a particular item.

`contains()` is only valid for a collection, i.e. a `relationship()` that implements one-to-many or many-to-many with `uselist=True`.

When used in a simple one-to-many context, an expression like:

```
MyClass.contains(other)
```

Produces a clause like:

```
mytable.id == <some id>
```

Where `<some id>` is the value of the foreign key attribute on `other` which refers to the primary key of its parent object. From this it follows that `contains()` is very useful when used with simple one-to-many operations.

For many-to-many operations, the behavior of `contains()` has more caveats. The association table will be rendered in the statement, producing an "implicit" join, that is, includes multiple tables in the FROM clause which are equated in the WHERE clause:

```
query(MyClass).filter(MyClass.contains(other))
```

Produces a query like:

```
SELECT * FROM my_table, my_association_table AS
my_association_table_1 WHERE
my_table.id = my_association_table_1.parent_id
AND my_association_table_1.child_id = <some id>
```

Where `<some id>` would be the primary key of `other`. From the above, it is clear that `contains()` will **not** work with many-to-many collections when used in queries that move beyond simple AND conjunctions, such as multiple `contains()` expressions joined by OR. In such cases subqueries or explicit "outer joins" will need to be used instead. See `any()` for a less-performant alternative using EXISTS, or refer to `Query.outerjoin()` as well as *Querying with Joins* for more details on constructing outer joins.

desc()
inherited from the `desc()` *method of* `ColumnOperators`

Produce a `desc()` clause against the parent object.

distinct()
inherited from the `distinct()` *method of* `ColumnOperators`

Produce a `distinct()` clause against the parent object.

endswith(*other*, ***kwargs*)
inherited from the `endswith()` *method of* `ColumnOperators`

Implement the 'endswith' operator.

In a column context, produces the clause `LIKE '%<other>'`

has(*criterion=None*, ***kwargs*)
Produce an expression that tests a scalar reference against particular criterion, using EXISTS.

An expression like:

```
session.query(MyClass).filter(
    MyClass.somereference.has(SomeRelated.x==2)
)
```

Will produce a query like:

```
SELECT * FROM my_table WHERE
EXISTS (SELECT 1 FROM related WHERE
related.id==my_table.related_id AND related.x=2)
```

Because `has()` uses a correlated subquery, its performance is not nearly as good when compared against large target tables as that of using a join.

`has()` is only valid for scalar references, i.e. a `relationship()` that has `uselist=False`. For collection references, use `any()`.

ilike(*other*, *escape=None*)
inherited from the `ilike()` *method of* `ColumnOperators`

Implement the `ilike` operator.

In a column context, produces the clause `a ILIKE other`.

E.g.:

```
select([sometable]).where(sometable.c.column.ilike("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the `ESCAPE` keyword, e.g.:

```
somecolumn.ilike("foo/%bar", escape="/")
```

See also:

`ColumnOperators.like()`

in_(*other*)
Produce an IN clause - this is not implemented for `relationship()`-based attributes at this time.

is_(*other*)
inherited from the `is_()` *method of* `ColumnOperators`

Implement the `IS` operator.

Normally, `IS` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.isnot()

isnot(*other*)

inherited from the isnot() method of ColumnOperators

Implement the `IS NOT` operator.

Normally, `IS NOT` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS NOT` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.is_()

like(*other*, *escape=None*)

inherited from the like() method of ColumnOperators

Implement the `like` operator.

In a column context, produces the clause `a LIKE other`.

E.g.:

```
select([sometable]).where(sometable.c.column.like("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the `ESCAPE` keyword, e.g.:

```
somecolumn.like("foo/%bar", escape="/")
```

See also:

ColumnOperators.ilike()

mapper

The target *Mapper* referred to by this *RelationshipProperty.Comparator*.

This is the "target" or "remote" side of the *relationship()*.

match(*other*, ***kwargs*)

inherited from the match() method of ColumnOperators

Implements a database-specific 'match' operator.

match() attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

- Postgresql - renders `x @@ to_tsquery(y)`
- MySQL - renders `MATCH (x) AGAINST (y IN BOOLEAN MODE)`
- Oracle - renders `CONTAINS(x, y)`
- other backends may provide special implementations.
- Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

notilike(*other*, *escape=None*)
inherited from the `notilike()` *method of* `ColumnOperators`

implement the `NOT ILIKE` operator.

This is equivalent to using negation with `ColumnOperators.ilike()`, i.e. `~x.ilike(y)`.

New in version 0.8.

See also:

`ColumnOperators.ilike()`

notin_(*other*)
inherited from the `notin_()` *method of* `ColumnOperators`

implement the `NOT IN` operator.

This is equivalent to using negation with `ColumnOperators.in_()`, i.e. `~x.in_(y)`.

New in version 0.8.

See also:

`ColumnOperators.in_()`

notlike(*other*, *escape=None*)
inherited from the `notlike()` *method of* `ColumnOperators`

implement the `NOT LIKE` operator.

This is equivalent to using negation with `ColumnOperators.like()`, i.e. `~x.like(y)`.

New in version 0.8.

See also:

`ColumnOperators.like()`

nullsfirst()
inherited from the `nullsfirst()` *method of* `ColumnOperators`

Produce a `nullsfirst()` clause against the parent object.

nullslast()
inherited from the `nullslast()` *method of* `ColumnOperators`

Produce a `nullslast()` clause against the parent object.

of_type(*cls*)
Produce a construct that represents a particular 'subtype' of attribute for the parent class.

Currently this is usable in conjunction with `Query.join()` and `Query.outerjoin()`.

op(*opstring*, *precedence=0*, *is_comparison=False*)
inherited from the `op()` *method of* `Operators`

produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

This function can also be used to make bitwise operators explicit. For example:

```
somecolumn.op('&')(0xff)
```

is a bitwise AND of the value in `somecolumn`.

Parameters

- **operator** – a string which will be output as the infix operator between this element and the expression passed to the generated function.
- **precedence** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of `0` is lower than all operators except for the comma (`,`) and `AS` operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.
- **is_comparison** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like ==, >, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

  New in version 0.9.2: - added the *`Operators.op.is_comparison`* flag.

See also:

Redefining and Creating New Operators

Using custom operators in join conditions

operate(*op*, **other*, ***kwargs*)

inherited from the `operate()` method of `Operators`

Operate on an argument.

This is the lowest level of operation, raises `NotImplementedError` by default.

Overriding this on a subclass can allow common behavior to be applied to all operations. For example, overriding *`ColumnOperators`* to apply `func.lower()` to the left and right side:

```
class MyComparator(ColumnOperators):
    def operate(self, op, other):
        return op(func.lower(self), func.lower(other))
```

Parameters

- **op** – Operator callable.
- ***other** – the 'other' side of the operation. Will be a single scalar for most operations.
- ****kwargs** – modifiers. These may be passed by special operators such as `ColumnOperators.contains()`.

reverse_operate(*op*, *other*, ***kwargs*)

inherited from the `reverse_operate()` method of `Operators`

Reverse operate on an argument.

Usage is the same as `operate()`.

startswith(*other*, ***kwargs*)
inherited from the `startswith()` *method of* `ColumnOperators`

Implement the `startwith` operator.

In a column context, produces the clause `LIKE '<other>%'`

`RelationshipProperty.`**`__init__`**(*argument*, *secondary=None*, *primaryjoin=None*, *secondaryjoin=None*, *foreign_keys=None*, *uselist=None*, *order_by=False*, *backref=None*, *back_populates=None*, *post_update=False*, *cascade=False*, *extension=None*, *viewonly=False*, *lazy=True*, *collection_class=None*, *passive_deletes=False*, *passive_updates=True*, *remote_side=None*, *enable_typechecks=True*, *join_depth=None*, *comparator_factory=None*, *single_parent=False*, *innerjoin=False*, *distinct_target_key=None*, *doc=None*, *active_history=False*, *cascade_backrefs=True*, *load_on_pending=False*, *bake_queries=True*, *strategy_class=None*, *_local_remote_pairs=None*, *query_class=None*, *info=None*)

Construct a new `RelationshipProperty` object.

This constructor is mirrored as a public API function; see `relationship()` for a full usage and argument description.

`RelationshipProperty.`**`cascade`**
Return the current cascade setting for this `RelationshipProperty`.

`RelationshipProperty.`**`class_attribute`**
inherited from the `class_attribute` *attribute of* `MapperProperty`

Return the class-bound descriptor corresponding to this `MapperProperty`.

This is basically a `getattr()` call:

```
return getattr(self.parent.class_, self.key)
```

I.e. if this `MapperProperty` were named `addresses`, and the class to which it is mapped is `User`, this sequence is possible:

```
>>> from sqlalchemy import inspect
>>> mapper = inspect(User)
>>> addresses_property = mapper.attrs.addresses
>>> addresses_property.class_attribute is User.addresses
True
>>> User.addresses.property is addresses_property
True
```

`RelationshipProperty.`**`extension_type`** = **symbol('NOT_EXTENSION')**

`RelationshipProperty.`**`init`**()
inherited from the `init()` *method of* `MapperProperty`

Called after all mappers are created to assemble relationships between mappers and perform other post-mapper-creation initialization steps.

`RelationshipProperty.`**`mapper`**
Return the targeted `Mapper` for this `RelationshipProperty`.

This is a lazy-initializing static attribute.

`RelationshipProperty.set_parent`(*parent*, *init*)
inherited from the `set_parent()` *method of* `MapperProperty`

Set the parent mapper that references this MapperProperty.

This method is overridden by some subclasses to perform extra setup when the mapper is first known.

`RelationshipProperty.table`
Return the selectable linked to this `RelationshipProperty` object's target `Mapper`.

Deprecated since version 0.7: Use .target

class `sqlalchemy.orm.descriptor_props.SynonymProperty`(*name*, *map_column=None*, *descriptor=None*, *comparator_factory=None*, *doc=None*, *info=None*)
Bases: `sqlalchemy.orm.descriptor_props.DescriptorProperty`

`__init__`(*name*, *map_column=None*, *descriptor=None*, *comparator_factory=None*, *doc=None*, *info=None*)
Construct a new `SynonymProperty` object.

This constructor is mirrored as a public API function; see `synonym()` for a full usage and argument description.

`cascade_iterator`(*type_*, *state*, *visited_instances=None*, *halt_on=None*)
inherited from the `cascade_iterator()` *method of* `MapperProperty`

Iterate through instances related to the given instance for a particular 'cascade', starting with this MapperProperty.

Return an iterator3-tuples (instance, mapper, state).

Note that the 'cascade' collection on this MapperProperty is checked first for the given type before cascade_iterator is called.

This method typically only applies to RelationshipProperty.

`class_attribute`
inherited from the `class_attribute` *attribute of* `MapperProperty`

Return the class-bound descriptor corresponding to this `MapperProperty`.

This is basically a `getattr()` call:

```
return getattr(self.parent.class_, self.key)
```

I.e. if this `MapperProperty` were named `addresses`, and the class to which it is mapped is `User`, this sequence is possible:

```
>>> from sqlalchemy import inspect
>>> mapper = inspect(User)
>>> addresses_property = mapper.attrs.addresses
>>> addresses_property.class_attribute is User.addresses
True
>>> User.addresses.property is addresses_property
True
```

`create_row_processor`(*context*, *path*, *mapper*, *result*, *adapter*, *populators*)
inherited from the `create_row_processor()` *method of* `MapperProperty`

Produce row processing functions and append to the given set of populators lists.

do_init()
inherited from the `do_init()` *method of* `MapperProperty`

Perform subclass-specific initialization post-mapper-creation steps.

This is a template method called by the `MapperProperty` object's init() method.

extension_type = symbol('NOT_EXTENSION')

init()
inherited from the `init()` *method of* `MapperProperty`

Called after all mappers are created to assemble relationships between mappers and perform other post-mapper-creation initialization steps.

merge(*session*, *source_state*, *source_dict*, *dest_state*, *dest_dict*, *load*, *_recursive*, *_resolve_conflict_map*)
inherited from the `merge()` *method of* `MapperProperty`

Merge the attribute represented by this `MapperProperty` from source to destination object.

post_instrument_class(*mapper*)
inherited from the `post_instrument_class()` *method of* `MapperProperty`

Perform instrumentation adjustments that need to occur after init() has completed.

The given Mapper is the Mapper invoking the operation, which may not be the same Mapper as self.parent in an inheritance scenario; however, Mapper will always at least be a sub-mapper of self.parent.

This method is typically used by StrategizedProperty, which delegates it to LoaderStrategy.init_class_attribute() to perform final setup on the class-bound InstrumentedAttribute.

setup(*context*, *entity*, *path*, *adapter*, ***kwargs*)
inherited from the `setup()` *method of* `MapperProperty`

Called by Query for the purposes of constructing a SQL statement.

Each MapperProperty associated with the target mapper processes the statement referenced by the query context, adding columns and/or criterion as appropriate.

class sqlalchemy.orm.query.**QueryContext**(*query*)

class sqlalchemy.orm.attributes.**QueryableAttribute**(*class_*, *key*, *impl=None*, *comparator=None*, *parententity=None*, *of_type=None*)

Bases: `sqlalchemy.orm.base._MappedAttribute`, `sqlalchemy.orm.base.InspectionAttr`, `sqlalchemy.orm.interfaces.PropComparator`

Base class for descriptor objects that intercept attribute events on behalf of a `MapperProperty` object. The actual `MapperProperty` is accessible via the `QueryableAttribute.property` attribute.

See also:

`InstrumentedAttribute`

`MapperProperty`

`Mapper.all_orm_descriptors`

`Mapper.attrs`

__eq__(*other*)
inherited from the `__eq__()` *method of* `ColumnOperators`

Implement the == operator.

In a column context, produces the clause `a = b`. If the target is `None`, produces `a IS NULL`.

__le__(*other*)
inherited from the `__le__()` *method of* `ColumnOperators`

Implement the <= operator.

In a column context, produces the clause `a <= b`.

__lt__(*other*)
inherited from the `__lt__()` *method of* `ColumnOperators`

Implement the < operator.

In a column context, produces the clause `a < b`.

__ne__(*other*)
inherited from the `__ne__()` *method of* `ColumnOperators`

Implement the != operator.

In a column context, produces the clause `a != b`. If the target is `None`, produces `a IS NOT NULL`.

adapter
inherited from the `adapter` *attribute of* `PropComparator`

Produce a callable that adapts column expressions to suit an aliased version of this comparator.

all_()
inherited from the `all_()` *method of* `ColumnOperators`

Produce a `all_()` clause against the parent object.

New in version 1.1.

any(*criterion=None*, ***kwargs*)
inherited from the `any()` *method of* `PropComparator`

Return true if this collection contains any member that meets the given criterion.

The usual implementation of `any()` is `RelationshipProperty.Comparator.any()`.

Parameters

- **criterion** – an optional ClauseElement formulated against the member class' table or attributes.
- ****kwargs** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values.

any_()
inherited from the `any_()` *method of* `ColumnOperators`

Produce a `any_()` clause against the parent object.

New in version 1.1.

asc()
inherited from the `asc()` *method of* `ColumnOperators`

Produce a `asc()` clause against the parent object.

between(*cleft*, *cright*, *symmetric=False*)
inherited from the `between()` *method of* `ColumnOperators`

Produce a `between()` clause against the parent object, given the lower and upper range.

collate(*collation*)
inherited from the `collate()` *method of* `ColumnOperators`

Produce a `collate()` clause against the parent object, given the collation string.

concat(*other*)
inherited from the `concat()` *method of* `ColumnOperators`

Implement the 'concat' operator.

In a column context, produces the clause `a || b`, or uses the `concat()` operator on MySQL.

contains(*other*, ***kwargs*)
inherited from the `contains()` *method of* `ColumnOperators`

Implement the 'contains' operator.

In a column context, produces the clause `LIKE '%<other>%'`

desc()
inherited from the `desc()` *method of* `ColumnOperators`

Produce a `desc()` clause against the parent object.

distinct()
inherited from the `distinct()` *method of* `ColumnOperators`

Produce a `distinct()` clause against the parent object.

endswith(*other*, ***kwargs*)
inherited from the `endswith()` *method of* `ColumnOperators`

Implement the 'endswith' operator.

In a column context, produces the clause `LIKE '%<other>'`

extension_type = symbol('NOT_EXTENSION')

has(*criterion=None*, ***kwargs*)
inherited from the `has()` *method of* `PropComparator`

Return true if this element references a member which meets the given criterion.

The usual implementation of `has()` is `RelationshipProperty.Comparator.has()`.

Parameters

- **criterion** – an optional ClauseElement formulated against the member class' table or attributes.
- ****kwargs** – key/value pairs corresponding to member class attribute names which will be compared via equality to the corresponding values.

ilike(*other*, *escape=None*)
inherited from the `ilike()` *method of* `ColumnOperators`

Implement the `ilike` operator.

In a column context, produces the clause `a ILIKE other`.

E.g.:

```
select([sometable]).where(sometable.c.column.ilike("%foobar%"))
```

Parameters

- **other** – expression to be compared

- **escape** – optional escape character, renders the ESCAPE keyword, e.g.:

```
somecolumn.ilike("foo/%bar", escape="/")
```

See also:

ColumnOperators.like()

in_(*other*)

inherited from the in_() method of ColumnOperators

Implement the in operator.

In a column context, produces the clause a IN other. "other" may be a tuple/list of column expressions, or a *select()* construct.

info

Return the 'info' dictionary for the underlying SQL element.

The behavior here is as follows:

- If the attribute is a column-mapped property, i.e. *ColumnProperty*, which is mapped directly to a schema-level *Column* object, this attribute will return the *SchemaItem.info* dictionary associated with the core-level *Column* object.
- If the attribute is a *ColumnProperty* but is mapped to any other kind of SQL expression other than a *Column*, the attribute will refer to the *MapperProperty.info* dictionary associated directly with the *ColumnProperty*, assuming the SQL expression itself does not have its own .info attribute (which should be the case, unless a user-defined SQL construct has defined one).
- If the attribute refers to any other kind of *MapperProperty*, including *RelationshipProperty*, the attribute will refer to the *MapperProperty.info* dictionary associated with that *MapperProperty*.
- To access the *MapperProperty.info* dictionary of the *MapperProperty* unconditionally, including for a *ColumnProperty* that's associated directly with a *schema.Column*, the attribute can be referred to using *QueryableAttribute.property* attribute, as MyClass.someattribute.property.info.

New in version 0.8.0.

See also:

SchemaItem.info

MapperProperty.info

is_(*other*)

inherited from the is_() method of ColumnOperators

Implement the IS operator.

Normally, IS is generated automatically when comparing to a value of None, which resolves to NULL. However, explicit usage of IS may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.isnot()

isnot(*other*)

inherited from the isnot() method of ColumnOperators

Implement the `IS NOT` operator.

Normally, `IS NOT` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS NOT` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.is_()

like(*other*, *escape=None*)
inherited from the *like()* *method of* *ColumnOperators*

Implement the `like` operator.

In a column context, produces the clause `a LIKE other`.

E.g.:

```
select([sometable]).where(sometable.c.column.like("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the `ESCAPE` keyword, e.g.:

```
somecolumn.like("foo/%bar", escape="/")
```

See also:

ColumnOperators.ilike()

match(*other*, ***kwargs*)
inherited from the *match()* *method of* *ColumnOperators*

Implements a database-specific 'match' operator.

match() attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

- Postgresql - renders `x @@ to_tsquery(y)`
- MySQL - renders `MATCH (x) AGAINST (y IN BOOLEAN MODE)`
- Oracle - renders `CONTAINS(x, y)`
- other backends may provide special implementations.
- Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

notilike(*other*, *escape=None*)
inherited from the *notilike()* *method of* *ColumnOperators*

implement the `NOT ILIKE` operator.

This is equivalent to using negation with *ColumnOperators.ilike()*, i.e. `~x.ilike(y)`.

New in version 0.8.

See also:

ColumnOperators.ilike()

notin_(*other*)

inherited from the `notin_()` *method of* `ColumnOperators`

implement the `NOT IN` operator.

This is equivalent to using negation with `ColumnOperators.in_()`, i.e. `~x.in_(y)`.

New in version 0.8.

See also:

`ColumnOperators.in_()`

notlike(*other*, *escape=None*)

inherited from the `notlike()` *method of* `ColumnOperators`

implement the `NOT LIKE` operator.

This is equivalent to using negation with `ColumnOperators.like()`, i.e. `~x.like(y)`.

New in version 0.8.

See also:

`ColumnOperators.like()`

nullsfirst()

inherited from the `nullsfirst()` *method of* `ColumnOperators`

Produce a `nullsfirst()` clause against the parent object.

nullslast()

inherited from the `nullslast()` *method of* `ColumnOperators`

Produce a `nullslast()` clause against the parent object.

op(*opstring*, *precedence=0*, *is_comparison=False*)

inherited from the `op()` *method of* `Operators`

produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

This function can also be used to make bitwise operators explicit. For example:

```
somecolumn.op('&')(0xff)
```

is a bitwise AND of the value in `somecolumn`.

Parameters

- **operator** – a string which will be output as the infix operator between this element and the expression passed to the generated function.
- **precedence** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of `0` is lower than all operators except for the comma (`,`) and `AS` operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.

- **is_comparison** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like ==, >, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

  New in version 0.9.2: - added the `Operators.op.is_comparison` flag.

See also:

Redefining and Creating New Operators

Using custom operators in join conditions

parent
Return an inspection instance representing the parent.

This will be either an instance of `Mapper` or `AliasedInsp`, depending upon the nature of the parent entity which this attribute is associated with.

property
Return the `MapperProperty` associated with this `QueryableAttribute`.

Return values here will commonly be instances of `ColumnProperty` or `RelationshipProperty`.

startswith(*other*, ***kwargs*)
inherited from the `startswith()` *method of* `ColumnOperators`

Implement the `startwith` operator.

In a column context, produces the clause `LIKE '<other>%'`

class `sqlalchemy.orm.session.`**UOWTransaction**(*session*)

filter_states_for_dep(*dep*, *states*)
Filter the given list of InstanceStates to those relevant to the given DependencyProcessor.

finalize_flush_changes()
mark processed objects as clean / deleted after a successful flush().

this method is called within the flush() method after the execute() method has succeeded and the transaction has been committed.

get_attribute_history(*state*, *key*, *passive=symbol('PASSIVE_NO_INITIALIZE')*)
facade to attributes.get_state_history(), including caching of results.

is_deleted(*state*)
return true if the given state is marked as deleted within this uowtransaction.

remove_state_actions(*state*)
remove pending actions for a state from the uowtransaction.

2.6.3 ORM Exceptions

SQLAlchemy ORM exceptions.

`sqlalchemy.orm.exc.`**ConcurrentModificationError**
alias of `StaleDataError`

exception `sqlalchemy.orm.exc.`**DetachedInstanceError**
An attempt to access unloaded attributes on a mapped instance that is detached.

exception `sqlalchemy.orm.exc.`**FlushError**
A invalid condition was detected during flush().

exception `sqlalchemy.orm.exc.`**`MultipleResultsFound`**
A single database result was required but more than one were found.

`sqlalchemy.orm.exc.`**`NO_STATE`** **= (<type 'exceptions.AttributeError'>, <type 'exceptions.KeyError'>)**
Exception types that may be raised by instrumentation implementations.

exception `sqlalchemy.orm.exc.`**`NoResultFound`**
A database result was required but none was found.

exception `sqlalchemy.orm.exc.`**`ObjectDeletedError`**(*state*, *msg=None*)
A refresh operation failed to retrieve the database row corresponding to an object's known primary key identity.

A refresh operation proceeds when an expired attribute is accessed on an object, or when `Query.get()` is used to retrieve an object which is, upon retrieval, detected as expired. A SELECT is emitted for the target row based on primary key; if no row is returned, this exception is raised.

The true meaning of this exception is simply that no row exists for the primary key identifier associated with a persistent object. The row may have been deleted, or in some cases the primary key updated to a new value, outside of the ORM's management of the target object.

exception `sqlalchemy.orm.exc.`**`ObjectDereferencedError`**
An operation cannot complete due to an object being garbage collected.

exception `sqlalchemy.orm.exc.`**`StaleDataError`**
An operation encountered database state that is unaccounted for.

Conditions which cause this to happen include:

- A flush may have attempted to update or delete rows and an unexpected number of rows were matched during the UPDATE or DELETE statement. Note that when version_id_col is used, rows in UPDATE or DELETE statements are also matched against the current known version identifier.
- A mapped object with version_id_col was refreshed, and the version number coming back from the database does not match that of the object itself.
- A object is detached from its parent object, however the object was previously attached to a different parent identity which was garbage collected, and a decision cannot be made if the new parent was really the most recent "parent".

  New in version 0.7.4.

exception `sqlalchemy.orm.exc.`**`UnmappedClassError`**(*cls*, *msg=None*)
An mapping operation was requested for an unknown class.

exception `sqlalchemy.orm.exc.`**`UnmappedColumnError`**
Mapping operation was requested on an unknown column.

exception `sqlalchemy.orm.exc.`**`UnmappedError`**
Base for exceptions that involve expected mappings not present.

exception `sqlalchemy.orm.exc.`**`UnmappedInstanceError`**(*obj*, *msg=None*)
An mapping operation was requested for an unknown instance.

2.6.4 Deprecated ORM Event Interfaces

This section describes the class-based ORM event interface which first existed in SQLAlchemy 0.1, which progressed with more kinds of events up until SQLAlchemy 0.5. The non-ORM analogue is described at *Deprecated Event Interfaces*.

Deprecated since version 0.7: As of SQLAlchemy 0.7, the new event system described in *Events* replaces the extension/proxy/listener system, providing a consistent interface to all events without the need for subclassing.

Mapper Events

class sqlalchemy.orm.interfaces.**MapperExtension**

Base implementation for *Mapper* event hooks.

Note: *MapperExtension* is deprecated. Please refer to *event.listen()* as well as *MapperEvents*.

New extension classes subclass *MapperExtension* and are specified using the `extension` mapper() argument, which is a single *MapperExtension* or a list of such:

```
from sqlalchemy.orm.interfaces import MapperExtension

class MyExtension(MapperExtension):
    def before_insert(self, mapper, connection, instance):
        print "instance %s before insert !" % instance

m = mapper(User, users_table, extension=MyExtension())
```

A single mapper can maintain a chain of `MapperExtension` objects. When a particular mapping event occurs, the corresponding method on each `MapperExtension` is invoked serially, and each method has the ability to halt the chain from proceeding further:

```
m = mapper(User, users_table, extension=[ext1, ext2, ext3])
```

Each `MapperExtension` method returns the symbol EXT_CONTINUE by default. This symbol generally means "move to the next `MapperExtension` for processing". For methods that return objects like translated rows or new object instances, EXT_CONTINUE means the result of the method should be ignored. In some cases it's required for a default mapper activity to be performed, such as adding a new instance to a result list.

The symbol EXT_STOP has significance within a chain of `MapperExtension` objects that the chain will be stopped when this symbol is returned. Like EXT_CONTINUE, it also has additional significance in some cases that a default mapper activity will not be performed.

after_delete(*mapper*, *connection*, *instance*)

Receive an object instance after that instance is deleted.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

after_insert(*mapper*, *connection*, *instance*)

Receive an object instance after that instance is inserted.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

after_update(*mapper*, *connection*, *instance*)

Receive an object instance after that instance is updated.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

before_delete(*mapper*, *connection*, *instance*)

Receive an object instance before that instance is deleted.

Note that *no* changes to the overall flush plan can be made here; and manipulation of the `Session` will not have the desired effect. To manipulate the `Session` within an extension, use `SessionExtension`.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

before_insert(*mapper*, *connection*, *instance*)
Receive an object instance before that instance is inserted into its table.

This is a good place to set up primary key values and such that aren't handled otherwise.

Column-based attributes can be modified within this method which will result in the new value being inserted. However *no* changes to the overall flush plan can be made, and manipulation of the `Session` will not have the desired effect. To manipulate the `Session` within an extension, use `SessionExtension`.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

before_update(*mapper*, *connection*, *instance*)
Receive an object instance before that instance is updated.

Note that this method is called for all instances that are marked as "dirty", even those which have no net changes to their column-based attributes. An object is marked as dirty when any of its column-based attributes have a "set attribute" operation called or when any of its collections are modified. If, at update time, no column-based attributes have any net changes, no UPDATE statement will be issued. This means that an instance being sent to before_update is *not* a guarantee that an UPDATE statement will be issued (although you can affect the outcome here).

To detect if the column-based attributes on the object have net changes, and will therefore generate an UPDATE statement, use `object_session(instance).is_modified(instance, include_collections=False)`.

Column-based attributes can be modified within this method which will result in the new value being updated. However *no* changes to the overall flush plan can be made, and manipulation of the `Session` will not have the desired effect. To manipulate the `Session` within an extension, use `SessionExtension`.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

init_failed(*mapper*, *class_*, *oldinit*, *instance*, *args*, *kwargs*)
Receive an instance when its constructor has been called, and raised an exception.

This method is only called during a userland construction of an object. It is not called when an object is loaded from the database.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

init_instance(*mapper*, *class_*, *oldinit*, *instance*, *args*, *kwargs*)
Receive an instance when its constructor is called.

This method is only called during a userland construction of an object. It is not called when an object is loaded from the database.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

instrument_class(*mapper*, *class_*)
Receive a class when the mapper is first constructed, and has applied instrumentation to the mapped class.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

reconstruct_instance(*mapper*, *instance*)
Receive an object instance after it has been created via `__new__`, and after initial attribute population has occurred.

This typically occurs when the instance is created based on incoming result rows, and is only called once for that instance's lifetime.

Note that during a result-row load, this method is called upon the first row received for this instance. Note that some attributes and collections may or may not be loaded or even initialized, depending on what's present in the result rows.

The return value is only significant within the `MapperExtension` chain; the parent mapper's behavior isn't modified by this method.

Session Events

class `sqlalchemy.orm.interfaces.`**`SessionExtension`**

Base implementation for *`Session`* event hooks.

Note: *`SessionExtension`* is deprecated. Please refer to *`event.listen()`* as well as *`SessionEvents`*.

Subclasses may be installed into a *`Session`* (or *`sessionmaker`*) using the `extension` keyword argument:

```
from sqlalchemy.orm.interfaces import SessionExtension

class MySessionExtension(SessionExtension):
    def before_commit(self, session):
        print "before commit!"

Session = sessionmaker(extension=MySessionExtension())
```

The same *`SessionExtension`* instance can be used with any number of sessions.

`after_attach`(*session*, *instance*)

Execute after an instance is attached to a session.

This is called after an add, delete or merge.

`after_begin`(*session*, *transaction*, *connection*)

Execute after a transaction is begun on a connection

transaction is the SessionTransaction. This method is called after an engine level transaction is begun on a connection.

`after_bulk_delete`(*session*, *query*, *query_context*, *result*)

Execute after a bulk delete operation to the session.

This is called after a session.query(...).delete()

query is the query object that this delete operation was called on. *query_context* was the query context object. *result* is the result object returned from the bulk operation.

`after_bulk_update`(*session*, *query*, *query_context*, *result*)

Execute after a bulk update operation to the session.

This is called after a session.query(...).update()

query is the query object that this update operation was called on. *query_context* was the query context object. *result* is the result object returned from the bulk operation.

`after_commit`(*session*)

Execute after a commit has occurred.

Note that this may not be per-flush if a longer running transaction is ongoing.

`after_flush`(*session*, *flush_context*)

Execute after flush has completed, but before commit has been called.

Note that the session's state is still in pre-flush, i.e. 'new', 'dirty', and 'deleted' lists still show pre-flush state as well as the history settings on instance attributes.

after_flush_postexec(*session*, *flush_context*)
Execute after flush has completed, and after the post-exec state occurs.

This will be when the 'new', 'dirty', and 'deleted' lists are in their final state. An actual commit() may or may not have occurred, depending on whether or not the flush started its own transaction or participated in a larger transaction.

after_rollback(*session*)
Execute after a rollback has occurred.

Note that this may not be per-flush if a longer running transaction is ongoing.

before_commit(*session*)
Execute right before commit is called.

Note that this may not be per-flush if a longer running transaction is ongoing.

before_flush(*session*, *flush_context*, *instances*)
Execute before flush process has started.

instances is an optional list of objects which were passed to the `flush()` method.

Attribute Events

class sqlalchemy.orm.interfaces.**AttributeExtension**
Base implementation for `AttributeImpl` event hooks, events that fire upon attribute mutations in user code.

Note: *AttributeExtension* is deprecated. Please refer to *event.listen()* as well as *AttributeEvents*.

AttributeExtension is used to listen for set, remove, and append events on individual mapped attributes. It is established on an individual mapped attribute using the *extension* argument, available on *column_property()*, *relationship()*, and others:

```
from sqlalchemy.orm.interfaces import AttributeExtension
from sqlalchemy.orm import mapper, relationship, column_property

class MyAttrExt(AttributeExtension):
    def append(self, state, value, initiator):
        print "append event !"
        return value

    def set(self, state, value, oldvalue, initiator):
        print "set event !"
        return value

mapper(SomeClass, sometable, properties={
    'foo':column_property(sometable.c.foo, extension=MyAttrExt()),
    'bar':relationship(Bar, extension=MyAttrExt())
})
```

Note that the *AttributeExtension* methods *append()* and *set()* need to return the `value` parameter. The returned value is used as the effective value, and allows the extension to change what is ultimately persisted.

AttributeExtension is assembled within the descriptors associated with a mapped class.

active_history = **True**
indicates that the set() method would like to receive the 'old' value, even if it means firing lazy callables.

Note that `active_history` can also be set directly via *column_property()* and *relationship()*.

append(*state*, *value*, *initiator*)
Receive a collection append event.

The returned value will be used as the actual value to be appended.

remove(*state*, *value*, *initiator*)
Receive a remove event.

No return value is defined.

set(*state*, *value*, *oldvalue*, *initiator*)
Receive a set event.

The returned value will be used as the actual value to be set.

2.7 ORM Extensions

SQLAlchemy has a variety of ORM extensions available, which add additional functionality to the core behavior.

The extensions build almost entirely on public core and ORM APIs and users should be encouraged to read their source code to further their understanding of their behavior. In particular the "Horizontal Sharding", "Hybrid Attributes", and "Mutation Tracking" extensions are very succinct.

2.7.1 Association Proxy

`associationproxy` is used to create a read/write view of a target attribute across a relationship. It essentially conceals the usage of a "middle" attribute between two endpoints, and can be used to cherry-pick fields from a collection of related objects or to reduce the verbosity of using the association object pattern. Applied creatively, the association proxy allows the construction of sophisticated collections and dictionary views of virtually any geometry, persisted to the database using standard, transparently configured relational patterns.

Simplifying Scalar Collections

Consider a many-to-many mapping between two classes, `User` and `Keyword`. Each `User` can have any number of `Keyword` objects, and vice-versa (the many-to-many pattern is described at *Many To Many*):

```
from sqlalchemy import Column, Integer, String, ForeignKey, Table
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(64))
    kw = relationship("Keyword", secondary=lambda: userkeywords_table)

    def __init__(self, name):
```

```
        self.name = name

class Keyword(Base):
    __tablename__ = 'keyword'
    id = Column(Integer, primary_key=True)
    keyword = Column('keyword', String(64))

    def __init__(self, keyword):
        self.keyword = keyword

userkeywords_table = Table('userkeywords', Base.metadata,
    Column('user_id', Integer, ForeignKey("user.id"),
           primary_key=True),
    Column('keyword_id', Integer, ForeignKey("keyword.id"),
           primary_key=True)
)
```

Reading and manipulating the collection of "keyword" strings associated with `User` requires traversal from each collection element to the `.keyword` attribute, which can be awkward:

```
>>> user = User('jek')
>>> user.kw.append(Keyword('cheese inspector'))
>>> print(user.kw)
[<__main__.Keyword object at 0x12bf830>]
>>> print(user.kw[0].keyword)
cheese inspector
>>> print([keyword.keyword for keyword in user.kw])
['cheese inspector']
```

The `association_proxy` is applied to the `User` class to produce a "view" of the `kw` relationship, which only exposes the string value of `.keyword` associated with each `Keyword` object:

```
from sqlalchemy.ext.associationproxy import association_proxy

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(64))
    kw = relationship("Keyword", secondary=lambda: userkeywords_table)

    def __init__(self, name):
        self.name = name

    # proxy the 'keyword' attribute from the 'kw' relationship
    keywords = association_proxy('kw', 'keyword')
```

We can now reference the `.keywords` collection as a listing of strings, which is both readable and writable. New `Keyword` objects are created for us transparently:

```
>>> user = User('jek')
>>> user.keywords.append('cheese inspector')
>>> user.keywords
['cheese inspector']
>>> user.keywords.append('snack ninja')
>>> user.kw
[<__main__.Keyword object at 0x12cdd30>, <__main__.Keyword object at 0x12cde30>]
```

The *AssociationProxy* object produced by the *association_proxy()* function is an instance of a Python descriptor. It is always declared with the user-defined class being mapped, regardless of whether Declarative or classical mappings via the *mapper()* function are used.

The proxy functions by operating upon the underlying mapped attribute or collection in response to operations, and changes made via the proxy are immediately apparent in the mapped attribute, as well as vice versa. The underlying attribute remains fully accessible.

When first accessed, the association proxy performs introspection operations on the target collection so that its behavior corresponds correctly. Details such as if the locally proxied attribute is a collection (as is typical) or a scalar reference, as well as if the collection acts like a set, list, or dictionary is taken into account, so that the proxy should act just like the underlying collection or attribute does.

Creation of New Values

When a list append() event (or set add(), dictionary __setitem__(), or scalar assignment event) is intercepted by the association proxy, it instantiates a new instance of the "intermediary" object using its constructor, passing as a single argument the given value. In our example above, an operation like:

```
user.keywords.append('cheese inspector')
```

Is translated by the association proxy into the operation:

```
user.kw.append(Keyword('cheese inspector'))
```

The example works here because we have designed the constructor for `Keyword` to accept a single positional argument, `keyword`. For those cases where a single-argument constructor isn't feasible, the association proxy's creational behavior can be customized using the `creator` argument, which references a callable (i.e. Python function) that will produce a new object instance given the singular argument. Below we illustrate this using a lambda as is typical:

```
class User(Base):
    # ...

    # use Keyword(keyword=kw) on append() events
    keywords = association_proxy('kw', 'keyword',
                    creator=lambda kw: Keyword(keyword=kw))
```

The `creator` function accepts a single argument in the case of a list- or set- based collection, or a scalar attribute. In the case of a dictionary-based collection, it accepts two arguments, "key" and "value". An example of this is below in *Proxying to Dictionary Based Collections*.

Simplifying Association Objects

The "association object" pattern is an extended form of a many-to-many relationship, and is described at *Association Object*. Association proxies are useful for keeping "association objects" out the way during regular use.

Suppose our `userkeywords` table above had additional columns which we'd like to map explicitly, but in most cases we don't require direct access to these attributes. Below, we illustrate a new mapping which introduces the `UserKeyword` class, which is mapped to the `userkeywords` table illustrated earlier. This class adds an additional column `special_key`, a value which we occasionally want to access, but not in the usual case. We create an association proxy on the `User` class called `keywords`, which will bridge the gap from the `user_keywords` collection of `User` to the `.keyword` attribute present on each `UserKeyword`:

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship, backref

from sqlalchemy.ext.associationproxy import association_proxy
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(64))

    # association proxy of "user_keywords" collection
    # to "keyword" attribute
    keywords = association_proxy('user_keywords', 'keyword')

    def __init__(self, name):
        self.name = name

class UserKeyword(Base):
    __tablename__ = 'user_keyword'
    user_id = Column(Integer, ForeignKey('user.id'), primary_key=True)
    keyword_id = Column(Integer, ForeignKey('keyword.id'), primary_key=True)
    special_key = Column(String(50))

    # bidirectional attribute/collection of "user"/"user_keywords"
    user = relationship(User,
                backref=backref("user_keywords",
                                cascade="all, delete-orphan")
            )

    # reference to the "Keyword" object
    keyword = relationship("Keyword")

    def __init__(self, keyword=None, user=None, special_key=None):
        self.user = user
        self.keyword = keyword
        self.special_key = special_key

class Keyword(Base):
    __tablename__ = 'keyword'
    id = Column(Integer, primary_key=True)
    keyword = Column('keyword', String(64))

    def __init__(self, keyword):
        self.keyword = keyword

    def __repr__(self):
        return 'Keyword(%s)' % repr(self.keyword)
```

With the above configuration, we can operate upon the `.keywords` collection of each `User` object, and the usage of `UserKeyword` is concealed:

```
>>> user = User('log')
>>> for kw in (Keyword('new_from_blammo'), Keyword('its_big')):
...     user.keywords.append(kw)
```

```
...
>>> print(user.keywords)
[Keyword('new_from_blammo'), Keyword('its_big')]
```

Where above, each `.keywords.append()` operation is equivalent to:

```
>>> user.user_keywords.append(UserKeyword(Keyword('its_heavy')))
```

The `UserKeyword` association object has two attributes here which are populated; the `.keyword` attribute is populated directly as a result of passing the `Keyword` object as the first argument. The `.user` argument is then assigned as the `UserKeyword` object is appended to the `User.user_keywords` collection, where the bidirectional relationship configured between `User.user_keywords` and `UserKeyword.user` results in a population of the `UserKeyword.user` attribute. The `special_key` argument above is left at its default value of `None`.

For those cases where we do want `special_key` to have a value, we create the `UserKeyword` object explicitly. Below we assign all three attributes, where the assignment of `.user` has the effect of the `UserKeyword` being appended to the `User.user_keywords` collection:

```
>>> UserKeyword(Keyword('its_wood'), user, special_key='my special key')
```

The association proxy returns to us a collection of `Keyword` objects represented by all these operations:

```
>>> user.keywords
[Keyword('new_from_blammo'), Keyword('its_big'), Keyword('its_heavy'), Keyword('its_wood')]
```

Proxying to Dictionary Based Collections

The association proxy can proxy to dictionary based collections as well. SQLAlchemy mappings usually use the *`attribute_mapped_collection()`* collection type to create dictionary collections, as well as the extended techniques described in *Custom Dictionary-Based Collections*.

The association proxy adjusts its behavior when it detects the usage of a dictionary-based collection. When new values are added to the dictionary, the association proxy instantiates the intermediary object by passing two arguments to the creation function instead of one, the key and the value. As always, this creation function defaults to the constructor of the intermediary class, and can be customized using the `creator` argument.

Below, we modify our `UserKeyword` example such that the `User.user_keywords` collection will now be mapped using a dictionary, where the `UserKeyword.special_key` argument will be used as the key for the dictionary. We then apply a `creator` argument to the `User.keywords` proxy so that these values are assigned appropriately when new elements are added to the dictionary:

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship, backref
from sqlalchemy.ext.associationproxy import association_proxy
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm.collections import attribute_mapped_collection

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(64))
```

```
    # proxy to 'user_keywords', instantiating UserKeyword
    # assigning the new key to 'special_key', values to
    # 'keyword'.
    keywords = association_proxy('user_keywords', 'keyword',
                    creator=lambda k, v:
                                UserKeyword(special_key=k, keyword=v)
                )

    def __init__(self, name):
        self.name = name

class UserKeyword(Base):
    __tablename__ = 'user_keyword'
    user_id = Column(Integer, ForeignKey('user.id'), primary_key=True)
    keyword_id = Column(Integer, ForeignKey('keyword.id'), primary_key=True)
    special_key = Column(String)

    # bidirectional user/user_keywords relationships, mapping
    # user_keywords with a dictionary against "special_key" as key.
    user = relationship(User, backref=backref(
                    "user_keywords",
                    collection_class=attribute_mapped_collection("special_key"),
                    cascade="all, delete-orphan"
                    )
                )
    keyword = relationship("Keyword")

class Keyword(Base):
    __tablename__ = 'keyword'
    id = Column(Integer, primary_key=True)
    keyword = Column('keyword', String(64))

    def __init__(self, keyword):
        self.keyword = keyword

    def __repr__(self):
        return 'Keyword(%s)' % repr(self.keyword)
```

We illustrate the `.keywords` collection as a dictionary, mapping the `UserKeyword.string_key` value to `Keyword` objects:

```
>>> user = User('log')

>>> user.keywords['sk1'] = Keyword('kw1')
>>> user.keywords['sk2'] = Keyword('kw2')

>>> print(user.keywords)
{'sk1': Keyword('kw1'), 'sk2': Keyword('kw2')}
```

Composite Association Proxies

Given our previous examples of proxying from relationship to scalar attribute, proxying across an association object, and proxying dictionaries, we can combine all three techniques together to give `User` a `keywords` dictionary that deals strictly with the string value of `special_key` mapped to the string `keyword`. Both the `UserKeyword` and `Keyword` classes are entirely concealed. This is achieved by building an association proxy on `User` that refers to an association proxy present on `UserKeyword`:

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship, backref

from sqlalchemy.ext.associationproxy import association_proxy
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm.collections import attribute_mapped_collection

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(64))

    # the same 'user_keywords'->'keyword' proxy as in
    # the basic dictionary example
    keywords = association_proxy(
                'user_keywords',
                'keyword',
                creator=lambda k, v:
                            UserKeyword(special_key=k, keyword=v)
                )

    def __init__(self, name):
        self.name = name

class UserKeyword(Base):
    __tablename__ = 'user_keyword'
    user_id = Column(Integer, ForeignKey('user.id'), primary_key=True)
    keyword_id = Column(Integer, ForeignKey('keyword.id'),
                                                    primary_key=True)
    special_key = Column(String)
    user = relationship(User, backref=backref(
            "user_keywords",
            collection_class=attribute_mapped_collection("special_key"),
            cascade="all, delete-orphan"
            )
        )

    # the relationship to Keyword is now called
    # 'kw'
    kw = relationship("Keyword")

    # 'keyword' is changed to be a proxy to the
    # 'keyword' attribute of 'Keyword'
    keyword = association_proxy('kw', 'keyword')

class Keyword(Base):
    __tablename__ = 'keyword'
    id = Column(Integer, primary_key=True)
    keyword = Column('keyword', String(64))

    def __init__(self, keyword):
        self.keyword = keyword
```

`User.keywords` is now a dictionary of string to string, where `UserKeyword` and `Keyword` objects are created and removed for us transparently using the association proxy. In the example below, we illustrate usage of the as-

signment operator, also appropriately handled by the association proxy, to apply a dictionary value to the collection at once:

```
>>> user = User('log')
>>> user.keywords = {
...     'sk1':'kw1',
...     'sk2':'kw2'
... }
>>> print(user.keywords)
{'sk1': 'kw1', 'sk2': 'kw2'}

>>> user.keywords['sk3'] = 'kw3'
>>> del user.keywords['sk2']
>>> print(user.keywords)
{'sk1': 'kw1', 'sk3': 'kw3'}

>>> # illustrate un-proxied usage
... print(user.user_keywords['sk3'].kw)
<__main__.Keyword object at 0x12ceb90>
```

One caveat with our example above is that because `Keyword` objects are created for each dictionary set operation, the example fails to maintain uniqueness for the `Keyword` objects on their string name, which is a typical requirement for a tagging scenario such as this one. For this use case the recipe UniqueObject, or a comparable creational strategy, is recommended, which will apply a "lookup first, then create" strategy to the constructor of the `Keyword` class, so that an already existing `Keyword` is returned if the given name is already present.

Querying with Association Proxies

The *`AssociationProxy`* features simple SQL construction capabilities which relate down to the underlying *`relationship()`* in use as well as the target attribute. For example, the *`RelationshipProperty.Comparator.any()`* and *`RelationshipProperty.Comparator.has()`* operations are available, and will produce a "nested" EXISTS clause, such as in our basic association object example:

```
>>> print(session.query(User).filter(User.keywords.any(keyword='jek')))
SELECT user.id AS user_id, user.name AS user_name
FROM user
WHERE EXISTS (SELECT 1
FROM user_keyword
WHERE user.id = user_keyword.user_id AND (EXISTS (SELECT 1
FROM keyword
WHERE keyword.id = user_keyword.keyword_id AND keyword.keyword = :keyword_1)))
```

For a proxy to a scalar attribute, `__eq__()` is supported:

```
>>> print(session.query(UserKeyword).filter(UserKeyword.keyword == 'jek'))
SELECT user_keyword.*
FROM user_keyword
WHERE EXISTS (SELECT 1
    FROM keyword
    WHERE keyword.id = user_keyword.keyword_id AND keyword.keyword = :keyword_1)
```

and `.contains()` is available for a proxy to a scalar collection:

```
>>> print(session.query(User).filter(User.keywords.contains('jek')))
SELECT user.*
FROM user
WHERE EXISTS (SELECT 1
FROM userkeywords, keyword
WHERE user.id = userkeywords.user_id
    AND keyword.id = userkeywords.keyword_id
    AND keyword.keyword = :keyword_1)
```

AssociationProxy can be used with *Query.join()* somewhat manually using the *attr* attribute in a star-args context:

```
q = session.query(User).join(*User.keywords.attr)
```

New in version 0.7.3: *attr* attribute in a star-args context.

attr is composed of *AssociationProxy.local_attr* and *AssociationProxy.remote_attr*, which are just synonyms for the actual proxied attributes, and can also be used for querying:

```
uka = aliased(UserKeyword)
ka = aliased(Keyword)
q = session.query(User).\
        join(uka, User.keywords.local_attr).\
        join(ka, User.keywords.remote_attr)
```

New in version 0.7.3: *AssociationProxy.local_attr* and *AssociationProxy.remote_attr*, synonyms for the actual proxied attributes, and usable for querying.

API Documentation

sqlalchemy.ext.associationproxy.**association_proxy**(*target_collection*, *attr*, ***kw*)

Return a Python property implementing a view of a target attribute which references an attribute on members of the target.

The returned value is an instance of *AssociationProxy*.

Implements a Python property representing a relationship as a collection of simpler values, or a scalar value. The proxied property will mimic the collection type of the target (list, dict or set), or, in the case of a one to one relationship, a simple scalar value.

Parameters

- **target_collection** – Name of the attribute we'll proxy to. This attribute is typically mapped by *relationship()* to link to a target collection, but can also be a many-to-one or non-scalar relationship.
- **attr** – Attribute on the associated instance or instances we'll proxy for.

  For example, given a target collection of [obj1, obj2], a list created by this proxy property would look like [getattr(obj1, *attr*), getattr(obj2, *attr*)]

  If the relationship is one-to-one or otherwise uselist=False, then simply: getattr(obj, *attr*)
- **creator** – optional.

  When new items are added to this proxied collection, new instances of the class collected by the target collection will be created. For list and set collections, the target class con-

structor will be called with the 'value' for the new instance. For dict types, two arguments are passed: key and value.

If you want to construct instances differently, supply a *creator* function that takes arguments as above and returns instances.

For scalar relationships, creator() will be called if the target is None. If the target is present, set operations are proxied to setattr() on the associated object.

If you have an associated object with multiple attributes, you may set up multiple association proxies mapping to different attributes. See the unit tests for examples, and for examples of how creator() functions can be used to construct the scalar relationship on-demand in this situation.

- ****kw** – Passes along any other keyword arguments to `AssociationProxy`.

class `sqlalchemy.ext.associationproxy.`**`AssociationProxy`**(*target_collection*, *attr*, *creator=None*, *getset_factory=None*, *proxy_factory=None*, *proxy_bulk_set=None*, *info=None*)

Bases: `sqlalchemy.orm.base.InspectionAttrInfo`

A descriptor that presents a read/write view of an object attribute.

`__init__`(*target_collection*, *attr*, *creator=None*, *getset_factory=None*, *proxy_factory=None*, *proxy_bulk_set=None*, *info=None*)

Construct a new `AssociationProxy`.

The `association_proxy()` function is provided as the usual entrypoint here, though `AssociationProxy` can be instantiated and/or subclassed directly.

Parameters

- **`target_collection`** – Name of the collection we'll proxy to, usually created with `relationship()`.
- **`attr`** – Attribute on the collected instances we'll proxy for. For example, given a target collection of [obj1, obj2], a list created by this proxy property would look like [getattr(obj1, attr), getattr(obj2, attr)]
- **`creator`** – Optional. When new items are added to this proxied collection, new instances of the class collected by the target collection will be created. For list and set collections, the target class constructor will be called with the 'value' for the new instance. For dict types, two arguments are passed: key and value.

  If you want to construct instances differently, supply a 'creator' function that takes arguments as above and returns instances.
- **`getset_factory`** – Optional. Proxied attribute access is automatically handled by routines that get and set values based on the *attr* argument for this proxy.

  If you would like to customize this behavior, you may supply a *getset_factory* callable that produces a tuple of *getter* and *setter* functions. The factory is called with two arguments, the abstract type of the underlying collection and this proxy instance.
- **`proxy_factory`** – Optional. The type of collection to emulate is determined by sniffing the target collection. If your collection type can't be determined by duck typing or you'd like to use a different collection implementation, you may supply a factory function to produce those collections. Only applicable to non-scalar relationships.

- **proxy_bulk_set** – Optional, use with proxy_factory. See the _set() method for details.
- **info** – optional, will be assigned to `AssociationProxy.info` if present.

  New in version 1.0.9.

any(*criterion=None*, ***kwargs*)

Produce a proxied 'any' expression using EXISTS.

This expression will be a composed product using the `RelationshipProperty.Comparator.any()` and/or `RelationshipProperty.Comparator.has()` operators of the underlying proxied attributes.

attr

Return a tuple of `(local_attr, remote_attr)`.

This attribute is convenient when specifying a join using `Query.join()` across two relationships:

```
sess.query(Parent).join(*Parent.proxied.attr)
```

New in version 0.7.3.

See also:

`AssociationProxy.local_attr`

`AssociationProxy.remote_attr`

contains(*obj*)

Produce a proxied 'contains' expression using EXISTS.

This expression will be a composed product using the `RelationshipProperty.Comparator.any()`, `RelationshipProperty.Comparator.has()`, and/or `RelationshipProperty.Comparator.contain` operators of the underlying proxied attributes.

extension_type = symbol('ASSOCIATION_PROXY')

has(*criterion=None*, ***kwargs*)

Produce a proxied 'has' expression using EXISTS.

This expression will be a composed product using the `RelationshipProperty.Comparator.any()` and/or `RelationshipProperty.Comparator.has()` operators of the underlying proxied attributes.

info

inherited from the `info` *attribute of* `InspectionAttrInfo`

Info dictionary associated with the object, allowing user-defined data to be associated with this `InspectionAttr`.

The dictionary is generated when first accessed. Alternatively, it can be specified as a constructor argument to the `column_property()`, `relationship()`, or `composite()` functions.

New in version 0.8: Added support for .info to all `MapperProperty` subclasses.

Changed in version 1.0.0: `MapperProperty.info` is also available on extension types via the `InspectionAttrInfo.info` attribute, so that it can apply to a wider variety of ORM and extension constructs.

See also:

`QueryableAttribute.info`

`SchemaItem.info`

is_aliased_class = **False**

is_attribute = **False**

is_clause_element = **False**

is_instance = **False**

is_mapper = **False**

is_property = **False**

is_selectable = **False**

local_attr

The 'local' *MapperProperty* referenced by this *AssociationProxy*.

New in version 0.7.3.

See also:

AssociationProxy.attr

AssociationProxy.remote_attr

remote_attr

The 'remote' *MapperProperty* referenced by this *AssociationProxy*.

New in version 0.7.3.

See also:

AssociationProxy.attr

AssociationProxy.local_attr

scalar

Return `True` if this *AssociationProxy* proxies a scalar relationship on the local side.

target_class

The intermediary class handled by this *AssociationProxy*.

Intercepted append/set/assignment events will result in the generation of new instances of this class.

`sqlalchemy.ext.associationproxy.`**ASSOCIATION_PROXY** = **symbol('ASSOCIATION_PROXY')**

2.7.2 Automap

Define an extension to the *sqlalchemy.ext.declarative* system which automatically generates mapped classes and relationships from a database schema, typically though not necessarily one which is reflected.

New in version 0.9.1: Added *sqlalchemy.ext.automap*.

It is hoped that the *AutomapBase* system provides a quick and modernized solution to the problem that the very famous SQLSoup also tries to solve, that of generating a quick and rudimentary object model from an existing database on the fly. By addressing the issue strictly at the mapper configuration level, and integrating fully with existing Declarative class techniques, *AutomapBase* seeks to provide a well-integrated approach to the issue of expediently auto-generating ad-hoc mappings.

Basic Use

The simplest usage is to reflect an existing database into a new model. We create a new *AutomapBase* class in a similar manner as to how we create a declarative base class, using *automap_base()*. We then call *AutomapBase.prepare()* on the resulting base class, asking it to reflect the schema and produce mappings:

```
from sqlalchemy.ext.automap import automap_base
from sqlalchemy.orm import Session
from sqlalchemy import create_engine

Base = automap_base()

# engine, suppose it has two tables 'user' and 'address' set up
engine = create_engine("sqlite:///mydatabase.db")

# reflect the tables
Base.prepare(engine, reflect=True)

# mapped classes are now created with names by default
# matching that of the table name.
User = Base.classes.user
Address = Base.classes.address

session = Session(engine)

# rudimentary relationships are produced
session.add(Address(email_address="foo@bar.com", user=User(name="foo")))
session.commit()

# collection-based relationships are by default named
# "<classname>_collection"
print (u1.address_collection)
```

Above, calling *AutomapBase.prepare()* while passing along the *AutomapBase.prepare.reflect* parameter indicates that the *MetaData.reflect()* method will be called on this declarative base classes' *MetaData* collection; then, each **viable** *Table* within the *MetaData* will get a new mapped class generated automatically. The *ForeignKeyConstraint* objects which link the various tables together will be used to produce new, bidirectional *relationship()* objects between classes. The classes and relationships follow along a default naming scheme that we can customize. At this point, our basic mapping consisting of related `User` and `Address` classes is ready to use in the traditional way.

Note: By **viable**, we mean that for a table to be mapped, it must specify a primary key. Additionally, if the table is detected as being a pure association table between two other tables, it will not be directly mapped and will instead be configured as a many-to-many table between the mappings for the two referring tables.

Generating Mappings from an Existing MetaData

We can pass a pre-declared *MetaData* object to *automap_base()*. This object can be constructed in any way, including programmatically, from a serialized file, or from itself being reflected using *MetaData.reflect()*. Below we illustrate a combination of reflection and explicit table declaration:

```
from sqlalchemy import create_engine, MetaData, Table, Column, ForeignKey
engine = create_engine("sqlite:///mydatabase.db")
```

```
# produce our own MetaData object
metadata = MetaData()

# we can reflect it ourselves from a database, using options
# such as 'only' to limit what tables we look at...
metadata.reflect(engine, only=['user', 'address'])

# ... or just define our own Table objects with it (or combine both)
Table('user_order', metadata,
                Column('id', Integer, primary_key=True),
                Column('user_id', ForeignKey('user.id'))
            )

# we can then produce a set of mappings from this MetaData.
Base = automap_base(metadata=metadata)

# calling prepare() just sets up mapped classes and relationships.
Base.prepare()

# mapped classes are ready
User, Address, Order = Base.classes.user, Base.classes.address,        Base.classes.user_order
```

Specifying Classes Explicitly

The *sqlalchemy.ext.automap* extension allows classes to be defined explicitly, in a way similar to that of the *DeferredReflection* class. Classes that extend from *AutomapBase* act like regular declarative classes, but are not immediately mapped after their construction, and are instead mapped when we call *AutomapBase.prepare()*. The *AutomapBase.prepare()* method will make use of the classes we've established based on the table name we use. If our schema contains tables `user` and `address`, we can define one or both of the classes to be used:

```
from sqlalchemy.ext.automap import automap_base
from sqlalchemy import create_engine

# automap base
Base = automap_base()

# pre-declare User for the 'user' table
class User(Base):
    __tablename__ = 'user'

    # override schema elements like Columns
    user_name = Column('name', String)

    # override relationships too, if desired.
    # we must use the same name that automap would use for the
    # relationship, and also must refer to the class name that automap will
    # generate for "address"
    address_collection = relationship("address", collection_class=set)

# reflect
engine = create_engine("sqlite:///mydatabase.db")
Base.prepare(engine, reflect=True)

# we still have Address generated from the tablename "address",
# but User is the same as Base.classes.User now
```

```
Address = Base.classes.address

u1 = session.query(User).first()
print (u1.address_collection)

# the backref is still there:
a1 = session.query(Address).first()
print (a1.user)
```

Above, one of the more intricate details is that we illustrated overriding one of the *relationship()* objects that automap would have created. To do this, we needed to make sure the names match up with what automap would normally generate, in that the relationship name would be `User.address_collection` and the name of the class referred to, from automap's perspective, is called `address`, even though we are referring to it as `Address` within our usage of this class.

Overriding Naming Schemes

sqlalchemy.ext.automap is tasked with producing mapped classes and relationship names based on a schema, which means it has decision points in how these names are determined. These three decision points are provided using functions which can be passed to the *AutomapBase.prepare()* method, and are known as *classname_for_table()*, *name_for_scalar_relationship()*, and *name_for_collection_relationship()*. Any or all of these functions are provided as in the example below, where we use a "camel case" scheme for class names and a "pluralizer" for collection names using the Inflect package:

```
import re
import inflect

def camelize_classname(base, tablename, table):
    "Produce a 'camelized' class name, e.g. "
    "'words_and_underscores' -> 'WordsAndUnderscores'"

    return str(tablename[0].upper() + \
            re.sub(r'_([a-z])', lambda m: m.group(1).upper(), tablename[1:]))

_pluralizer = inflect.engine()
def pluralize_collection(base, local_cls, referred_cls, constraint):
    "Produce an 'uncamelized', 'pluralized' class name, e.g. "
    "'SomeTerm' -> 'some_terms'"

    referred_name = referred_cls.__name__
    uncamelized = re.sub(r'[A-Z]',
                         lambda m: "_%s" % m.group(0).lower(),
                         referred_name)[1:]
    pluralized = _pluralizer.plural(uncamelized)
    return pluralized

from sqlalchemy.ext.automap import automap_base

Base = automap_base()

engine = create_engine("sqlite:///mydatabase.db")

Base.prepare(engine, reflect=True,
            classname_for_table=camelize_classname,
```

```
            name_for_collection_relationship=pluralize_collection
    )
```

From the above mapping, we would now have classes `User` and `Address`, where the collection from `User` to `Address` is called `User.addresses`:

```
User, Address = Base.classes.User, Base.classes.Address

u1 = User(addresses=[Address(email="foo@bar.com")])
```

Relationship Detection

The vast majority of what automap accomplishes is the generation of *relationship()* structures based on foreign keys. The mechanism by which this works for many-to-one and one-to-many relationships is as follows:

1. A given *Table*, known to be mapped to a particular class, is examined for *ForeignKeyConstraint* objects.
2. From each *ForeignKeyConstraint*, the remote *Table* object present is matched up to the class to which it is to be mapped, if any, else it is skipped.
3. As the *ForeignKeyConstraint* we are examining corresponds to a reference from the immediate mapped class, the relationship will be set up as a many-to-one referring to the referred class; a corresponding one-to-many backref will be created on the referred class referring to this class.
4. If any of the columns that are part of the *ForeignKeyConstraint* are not nullable (e.g. `nullable=False`), a *cascade* keyword argument of `all, delete-orphan` will be added to the keyword arguments to be passed to the relationship or backref. If the *ForeignKeyConstraint* reports that *ForeignKeyConstraint.ondelete* is set to `CASCADE` for a not null or `SET NULL` for a nullable set of columns, the option *passive_deletes* flag is set to `True` in the set of relationship keyword arguments. Note that not all backends support reflection of ON DELETE.

   New in version 1.0.0: - automap will detect non-nullable foreign key constraints when producing a one-to-many relationship and establish a default cascade of `all, delete-orphan` if so; additionally, if the constraint specifies *ForeignKeyConstraint.ondelete* of `CASCADE` for non-nullable or `SET NULL` for nullable columns, the `passive_deletes=True` option is also added.
5. The names of the relationships are determined using the *AutomapBase.prepare.name_for_scalar_relationship* and *AutomapBase.prepare.name_for_collection_relationship* callable functions. It is important to note that the default relationship naming derives the name from the **the actual class name**. If you've given a particular class an explicit name by declaring it, or specified an alternate class naming scheme, that's the name from which the relationship name will be derived.
6. The classes are inspected for an existing mapped property matching these names. If one is detected on one side, but none on the other side, *AutomapBase* attempts to create a relationship on the missing side, then uses the *relationship.back_populates* parameter in order to point the new relationship to the other side.
7. In the usual case where no relationship is on either side, *AutomapBase.prepare()* produces a *relationship()* on the "many-to-one" side and matches it to the other using the *relationship.backref* parameter.
8. Production of the *relationship()* and optionally the *backref()* is handed off to the *AutomapBase.prepare.generate_relationship* function, which can be supplied by the end-user in order to augment the arguments passed to *relationship()* or *backref()* or to make use of custom implementations of these functions.

Custom Relationship Arguments

The *AutomapBase.prepare.generate_relationship* hook can be used to add parameters to relationships. For most cases, we can make use of the existing *automap.generate_relationship()* function to return the object, after augmenting the given keyword dictionary with our own arguments.

Below is an illustration of how to send *relationship.cascade* and *relationship.passive_deletes* options along to all one-to-many relationships:

```
from sqlalchemy.ext.automap import generate_relationship

def _gen_relationship(base, direction, return_fn,
                              attrname, local_cls, referred_cls, **kw):
    if direction is interfaces.ONETOMANY:
        kw['cascade'] = 'all, delete-orphan'
        kw['passive_deletes'] = True
    # make use of the built-in function to actually return
    # the result.
    return generate_relationship(base, direction, return_fn,
                                 attrname, local_cls, referred_cls, **kw)

from sqlalchemy.ext.automap import automap_base
from sqlalchemy import create_engine

# automap base
Base = automap_base()

engine = create_engine("sqlite:///mydatabase.db")
Base.prepare(engine, reflect=True,
            generate_relationship=_gen_relationship)
```

Many-to-Many relationships

sqlalchemy.ext.automap will generate many-to-many relationships, e.g. those which contain a `secondary` argument. The process for producing these is as follows:

1. A given *Table* is examined for *ForeignKeyConstraint* objects, before any mapped class has been assigned to it.
2. If the table contains two and exactly two *ForeignKeyConstraint* objects, and all columns within this table are members of these two *ForeignKeyConstraint* objects, the table is assumed to be a "secondary" table, and will **not be mapped directly**.
3. The two (or one, for self-referential) external tables to which the *Table* refers to are matched to the classes to which they will be mapped, if any.
4. If mapped classes for both sides are located, a many-to-many bi-directional *relationship()* / *backref()* pair is created between the two classes.
5. The override logic for many-to-many works the same as that of one-to-many/ many-to-one; the *generate_relationship()* function is called upon to generate the strucures and existing attributes will be maintained.

Relationships with Inheritance

sqlalchemy.ext.automap will not generate any relationships between two classes that are in an inheritance

relationship. That is, with two classes given as follows:

```
class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    type = Column(String(50))
    __mapper_args__ = {
         'polymorphic_identity':'employee', 'polymorphic_on': type
    }

class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    __mapper_args__ = {
        'polymorphic_identity':'engineer',
    }
```

The foreign key from `Engineer` to `Employee` is used not for a relationship, but to establish joined inheritance between the two classes.

Note that this means automap will not generate *any* relationships for foreign keys that link from a subclass to a superclass. If a mapping has actual relationships from subclass to superclass as well, those need to be explicit. Below, as we have two separate foreign keys from `Engineer` to `Employee`, we need to set up both the relationship we want as well as the `inherit_condition`, as these are not things SQLAlchemy can guess:

```
class Employee(Base):
    __tablename__ = 'employee'
    id = Column(Integer, primary_key=True)
    type = Column(String(50))

    __mapper_args__ = {
        'polymorphic_identity':'employee', 'polymorphic_on':type
    }

class Engineer(Employee):
    __tablename__ = 'engineer'
    id = Column(Integer, ForeignKey('employee.id'), primary_key=True)
    favorite_employee_id = Column(Integer, ForeignKey('employee.id'))

    favorite_employee = relationship(Employee,
                                     foreign_keys=favorite_employee_id)

    __mapper_args__ = {
        'polymorphic_identity':'engineer',
        'inherit_condition': id == Employee.id
    }
```

Handling Simple Naming Conflicts

In the case of naming conflicts during mapping, override any of *classname_for_table()*, *name_for_scalar_relationship()*, and *name_for_collection_relationship()* as needed. For example, if automap is attempting to name a many-to-one relationship the same as an existing column, an alternate convention can be conditionally selected. Given a schema:

```
CREATE TABLE table_a (
    id INTEGER PRIMARY KEY
);

CREATE TABLE table_b (
    id INTEGER PRIMARY KEY,
    table_a INTEGER,
    FOREIGN KEY(table_a) REFERENCES table_a(id)
);
```

The above schema will first automap the `table_a` table as a class named `table_a`; it will then automap a relationship onto the class for `table_b` with the same name as this related class, e.g. `table_a`. This relationship name conflicts with the mapping column `table_b.table_a`, and will emit an error on mapping.

We can resolve this conflict by using an underscore as follows:

```
def name_for_scalar_relationship(base, local_cls, referred_cls, constraint):
    name = referred_cls.__name__.lower()
    local_table = local_cls.__table__
    if name in local_table.columns:
        newname = name + "_"
        warnings.warn(
            "Already detected name %s present.  using %s" %
            (name, newname))
        return newname
    return name


Base.prepare(engine, reflect=True,
    name_for_scalar_relationship=name_for_scalar_relationship)
```

Alternatively, we can change the name on the column side. The columns that are mapped can be modified using the technique described at *Naming Columns Distinctly from Attribute Names*, by assigning the column explicitly to a new name:

```
Base = automap_base()

class TableB(Base):
    __tablename__ = 'table_b'
    _table_a = Column('table_a', ForeignKey('table_a.id'))

Base.prepare(engine, reflect=True)
```

Using Automap with Explicit Declarations

As noted previously, automap has no dependency on reflection, and can make use of any collection of `Table` objects within a `MetaData` collection. From this, it follows that automap can also be used generate missing relationships given an otherwise complete model that fully defines table metadata:

```
from sqlalchemy.ext.automap import automap_base
from sqlalchemy import Column, Integer, String, ForeignKey

Base = automap_base()

class User(Base):
```

```
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    name = Column(String)

class Address(Base):
    __tablename__ = 'address'

    id = Column(Integer, primary_key=True)
    email = Column(String)
    user_id = Column(ForeignKey('user.id'))

# produce relationships
Base.prepare()

# mapping is complete, with "address_collection" and
# "user" relationships
a1 = Address(email='u1')
a2 = Address(email='u2')
u1 = User(address_collection=[a1, a2])
assert a1.user is u1
```

Above, given mostly complete `User` and `Address` mappings, the *ForeignKey* which we defined on `Address.user_id` allowed a bidirectional relationship pair `Address.user` and `User.address_collection` to be generated on the mapped classes.

Note that when subclassing *AutomapBase*, the *AutomapBase.prepare()* method is required; if not called, the classes we've declared are in an un-mapped state.

API Reference

`sqlalchemy.ext.automap.`**`automap_base`**(*declarative_base=None*, ***kw*)

Produce a declarative automap base.

This function produces a new base class that is a product of the *AutomapBase* class as well a declarative base produced by *declarative.declarative_base()*.

All parameters other than `declarative_base` are keyword arguments that are passed directly to the *declarative.declarative_base()* function.

Parameters

- **`declarative_base`** – an existing class produced by *declarative.declarative_base()*. When this is passed, the function no longer invokes *declarative.declarative_base()* itself, and all other keyword arguments are ignored.
- **`**kw`** – keyword arguments are passed along to *declarative.declarative_base()*.

class `sqlalchemy.ext.automap.`**`AutomapBase`**

Base class for an "automap" schema.

The *AutomapBase* class can be compared to the "declarative base" class that is produced by the *declarative.declarative_base()* function. In practice, the *AutomapBase* class is always used as a mixin along with an actual declarative base.

A new subclassable *AutomapBase* is typically instantated using the *automap_base()* function.

See also:

Automap

classes = None

An instance of `util.Properties` containing classes.

This object behaves much like the `.c` collection on a table. Classes are present under the name they were given, e.g.:

```
Base = automap_base()
Base.prepare(engine=some_engine, reflect=True)

User, Address = Base.classes.User, Base.classes.Address
```

classmethod prepare(*engine=None*, *reflect=False*, *classname_for_table=<function classname_for_table>*, *collection_class=<type 'list'>*, *name_for_scalar_relationship=<function name_for_scalar_relationship>*, *name_for_collection_relationship=<function name_for_collection_relationship>*, *generate_relationship=<function generate_relationship>*)

Extract mapped classes and relationships from the *MetaData* and perform mappings.

Parameters

- **engine** – an *Engine* or *Connection* with which to perform schema reflection, if specified. If the *AutomapBase.prepare.reflect* argument is False, this object is not used.
- **reflect** – if True, the *MetaData.reflect()* method is called on the *MetaData* associated with this *AutomapBase*. The *Engine* passed via *AutomapBase.prepare.engine* will be used to perform the reflection if present; else, the *MetaData* should already be bound to some engine else the operation will fail.
- **classname_for_table** – callable function which will be used to produce new class names, given a table name. Defaults to *classname_for_table()*.
- **name_for_scalar_relationship** – callable function which will be used to produce relationship names for scalar relationships. Defaults to *name_for_scalar_relationship()*.
- **name_for_collection_relationship** – callable function which will be used to produce relationship names for collection-oriented relationships. Defaults to *name_for_collection_relationship()*.
- **generate_relationship** – callable function which will be used to actually generate *relationship()* and *backref()* constructs. Defaults to *generate_relationship()*.
- **collection_class** – the Python collection class that will be used when a new *relationship()* object is created that represents a collection. Defaults to `list`.

`sqlalchemy.ext.automap.`**classname_for_table**(*base*, *tablename*, *table*)

Return the class name that should be used, given the name of a table.

The default implementation is:

```
return str(tablename)
```

Alternate implementations can be specified using the *AutomapBase.prepare.classname_for_table* parameter.

Parameters

- **base** – the *AutomapBase* class doing the prepare.
- **tablename** – string name of the *Table*.
- **table** – the *Table* object itself.

Returns

a string class name.

Note: In Python 2, the string used for the class name **must** be a non-Unicode object, e.g. a `str()` object. The `.name` attribute of *Table* is typically a Python unicode subclass, so the `str()` function should be applied to this name, after accounting for any non-ASCII characters.

sqlalchemy.ext.automap.**name_for_scalar_relationship**(*base*, *local_cls*, *referred_cls*, *constraint*)

Return the attribute name that should be used to refer from one class to another, for a scalar object reference.

The default implementation is:

```
return referred_cls.__name__.lower()
```

Alternate implementations can be specified using the *AutomapBase.prepare.name_for_scalar_relationship* parameter.

Parameters

- **base** – the *AutomapBase* class doing the prepare.
- **local_cls** – the class to be mapped on the local side.
- **referred_cls** – the class to be mapped on the referring side.
- **constraint** – the *ForeignKeyConstraint* that is being inspected to produce this relationship.

sqlalchemy.ext.automap.**name_for_collection_relationship**(*base*, *local_cls*, *referred_cls*, *constraint*)

Return the attribute name that should be used to refer from one class to another, for a collection reference.

The default implementation is:

```
return referred_cls.__name__.lower() + "_collection"
```

Alternate implementations can be specified using the *AutomapBase.prepare.name_for_collection_relationshi*
parameter.

Parameters

- **base** – the *AutomapBase* class doing the prepare.
- **local_cls** – the class to be mapped on the local side.
- **referred_cls** – the class to be mapped on the referring side.
- **constraint** – the *ForeignKeyConstraint* that is being inspected to produce this relationship.

sqlalchemy.ext.automap.**generate_relationship**(*base*, *direction*, *return_fn*, *attrname*, *local_cls*, *referred_cls*, ***kw*)

Generate a *relationship()* or *backref()* on behalf of two mapped classes.

An alternate implementation of this function can be specified using the *AutomapBase.prepare.generate_relationship* parameter.

The default implementation of this function is as follows:

```
if return_fn is backref:
    return return_fn(attrname, **kw)
elif return_fn is relationship:
    return return_fn(referred_cls, **kw)
else:
    raise TypeError("Unknown relationship function: %s" % return_fn)
```

Parameters

- **base** – the *AutomapBase* class doing the prepare.
- **direction** – indicate the "direction" of the relationship; this will be one of *ONETOMANY*, *MANYTOONE*, *MANYTOMANY*.
- **return_fn** – the function that is used by default to create the relationship. This will be either *relationship()* or *backref()*. The *backref()* function's result will be used to produce a new *relationship()* in a second step, so it is critical that user-defined implementations correctly differentiate between the two functions, if a custom relationship function is being used.
- **local_cls** – the "local" class to which this relationship or backref will be locally present.
- **referred_cls** – the "referred" class to which the relationship or backref refers to.
- ****kw** – all additional keyword arguments are passed along to the function.

Attrname the attribute name to which this relationship is being assigned. If the value of *generate_relationship.return_fn* is the *backref()* function, then this name is the name that is being assigned to the backref.

Returns a *relationship()* or *backref()* construct, as dictated by the *generate_relationship.return_fn* parameter.

2.7.3 Baked Queries

`baked` provides an alternative creational pattern for *Query* objects, which allows for caching of the object's construction and string-compilation steps. This means that for a particular *Query* building scenario that is used more than once, all of the Python function invocation involved in building the query from its initial construction up through generating a SQL string will only occur **once**, rather than for each time that query is built up and executed.

The rationale for this system is to greatly reduce Python interpreter overhead for everything that occurs **before the SQL is emitted**. The caching of the "baked" system does **not** in any way reduce SQL calls or cache the **return results** from the database. A technique that demonstates the caching of the SQL calls and result sets themselves is available in *Dogpile Caching*.

New in version 1.0.0.

Note: The *sqlalchemy.ext.baked* extension should be considered **experimental** as of 1.0.0. It provides a dramatically different system of producing queries which has yet to be proven at scale.

Synopsis

Usage of the baked system starts by producing a so-called "bakery", which represents storage for a particular series of query objects:

```
from sqlalchemy.ext import baked

bakery = baked.bakery()
```

The above "bakery" will store cached data in an LRU cache that defaults to 200 elements, noting that an ORM query will typically contain one entry for the ORM query as invoked, as well as one entry per database dialect for the SQL string.

The bakery allows us to build up a *Query* object by specifying its construction as a series of Python callables, which are typically lambdas. For succinct usage, it overrides the += operator so that a typical query build-up looks like the following:

```
from sqlalchemy import bindparam

def search_for_user(session, username, email=None):

    baked_query = bakery(lambda session: session.query(User))
    baked_query += lambda q: q.filter(User.name == bindparam('username'))

    baked_query += lambda q: q.order_by(User.id)

    if email:
        baked_query += lambda q: q.filter(User.email == bindparam('email'))

    result = baked_query(session).params(username=username, email=email).all()

    return result
```

Following are some observations about the above code:

1. The `baked_query` object is an instance of *BakedQuery*. This object is essentially the "builder" for a real orm *Query* object, but it is not itself the *actual* *Query* object.
2. The actual *Query* object is not built at all, until the very end of the function when *Result.all()* is called.
3. The steps that are added to the `baked_query` object are all expressed as Python functions, typically lambdas. The first lambda given to the *bakery()* function receives a *Session* as its argument. The remaining lambdas each receive a *Query* as their argument.
4. In the above code, even though our application may call upon `search_for_user()` many times, and even though within each invocation we build up an entirely new *BakedQuery* object, *all of the lambdas are only called once*. Each lambda is **never** called a second time for as long as this query is cached in the bakery.
5. The caching is achieved by storing references to the **lambda objects themselves** in order to formulate a cache key; that is, the fact that the Python interpreter assigns an in-Python identity to these functions is what determines how to identify the query on successive runs. For those invocations of `search_for_user()` where the `email` parameter is specified, the callable `lambda q: q.filter(User.email == bindparam('email'))` will be part of the cache key that's retrieved; when `email` is `None`, this callable is not part of the cache key.
6. Because the lambdas are all called only once, it is essential that no variables which may change across calls are referenced **within** the lambdas; instead, assuming these are values to be bound into the SQL

string, we use *bindparam()* to construct named parameters, where we apply their actual values later using *Result.params()*.

Performance

The baked query probably looks a little odd, a little bit awkward and a little bit verbose. However, the savings in Python performance for a query which is invoked lots of times in an application are very dramatic. The example suite `short_selects` demonstrated in *Performance* illustrates a comparison of queries which each return only one row, such as the following regular query:

```
session = Session(bind=engine)
for id_ in random.sample(ids, n):
    session.query(Customer).filter(Customer.id == id_).one()
```

compared to the equivalent "baked" query:

```
bakery = baked.bakery()
s = Session(bind=engine)
for id_ in random.sample(ids, n):
    q = bakery(lambda s: s.query(Customer))
    q += lambda q: q.filter(Customer.id == bindparam('id'))
    q(s).params(id=id_).one()
```

The difference in Python function call count for an iteration of 10000 calls to each block are:

```
test_baked_query : test a baked query of the full entity.
                   (10000 iterations); total fn calls 1951294

test_orm_query :   test a straight ORM query of the full entity.
                   (10000 iterations); total fn calls 7900535
```

In terms of number of seconds on a powerful laptop, this comes out as:

```
test_baked_query : test a baked query of the full entity.
                   (10000 iterations); total time 2.174126 sec

test_orm_query :   test a straight ORM query of the full entity.
                   (10000 iterations); total time 7.958516 sec
```

Note that this test very intentionally features queries that only return one row. For queries that return many rows, the performance advantage of the baked query will have less and less of an impact, proportional to the time spent fetching rows. It is critical to keep in mind that the **baked query feature only applies to building the query itself, not the fetching of results**. Using the baked feature is by no means a guarantee to a much faster application; it is only a potentially useful feature for those applications that have been measured as being impacted by this particular form of overhead.

Measure twice, cut once

For background on how to profile a SQLAlchemy application, please see the section *Performance*. It is essential that performance measurement techniques are used when attempting to improve the performance of an application.

Rationale

The "lambda" approach above is a superset of what would be a more traditional "parameterized" approach. Suppose we wished to build a simple system where we build a *Query* just once, then store it in a dictionary for re-use. This is possible right now by just building up the query, and removing its *Session* by calling `my_cached_query = query.with_session(None)`:

```
my_simple_cache = {}

def lookup(session, id_argument):
    if "my_key" not in my_simple_cache:
        query = session.query(Model).filter(Model.id == bindparam('id'))
        my_simple_cache["my_key"] = query.with_session(None)
    else:
        query = my_simple_cache["my_key"].with_session(session)

    return query.params(id=id_argument).all()
```

The above approach gets us a very minimal performance benefit. By re-using a *Query*, we save on the Python work within the `session.query(Model)` constructor as well as calling upon `filter(Model.id == bindparam('id'))`, which will skip for us the building up of the Core expression as well as sending it to *Query.filter()*. However, the approach still regenerates the full *Select* object every time when *Query.all()* is called and additionally this brand new *Select* is sent off to the string compilation step every time, which for a simple case like the above is probably about 70% of the overhead.

To reduce the additional overhead, we need some more specialized logic, some way to memoize the construction of the select object and the construction of the SQL. There is an example of this on the wiki in the section BakedQuery, a precursor to this feature, however in that system, we aren't caching the *construction* of the query. In order to remove all the overhead, we need to cache both the construction of the query as well as the SQL compilation. Let's assume we adapted the recipe in this way and made ourselves a method `.bake()` that pre-compiles the SQL for the query, producing a new object that can be invoked with minimal overhead. Our example becomes:

```
my_simple_cache = {}

def lookup(session, id_argument):

    if "my_key" not in my_simple_cache:
        query = session.query(Model).filter(Model.id == bindparam('id'))
        my_simple_cache["my_key"] = query.with_session(None).bake()
    else:
        query = my_simple_cache["my_key"].with_session(session)

    return query.params(id=id_argument).all()
```

Above, we've fixed the performance situation, but we still have this string cache key to deal with.

We can use the "bakery" approach to re-frame the above in a way that looks less unusual than the "building up lambdas" approach, and more like a simple improvement upon the simple "reuse a query" approach:

```
bakery = baked.bakery()

def lookup(session, id_argument):
    def create_model_query(session):
        return session.query(Model).filter(Model.id == bindparam('id'))
```

```
    parameterized_query = bakery.bake(create_model_query)
    return parameterized_query(session).params(id=id_argument).all()
```

Above, we use the "baked" system in a manner that is very similar to the simplistic "cache a query" system. However, it uses two fewer lines of code, does not need to manufacture a cache key of "my_key", and also includes the same feature as our custom "bake" function that caches 100% of the Python invocation work from the constructor of the query, to the filter call, to the production of the `Select` object, to the string compilation step.

From the above, if we ask ourselves, "what if lookup needs to make conditional decisions as to the structure of the query?", this is where hopefully it becomes apparent why "baked" is the way it is. Instead of a parameterized query building off from exactly one function (which is how we thought baked might work originally), we can build it from *any number* of functions. Consider our naive example, if we needed to have an additional clause in our query on a conditional basis:

```
my_simple_cache = {}

def lookup(session, id_argument, include_frobnizzle=False):
    if include_frobnizzle:
        cache_key = "my_key_with_frobnizzle"
    else:
        cache_key = "my_key_without_frobnizzle"

    if cache_key not in my_simple_cache:
        query = session.query(Model).filter(Model.id == bindparam('id'))
        if include_frobnizzle:
            query = query.filter(Model.frobnizzle == True)

        my_simple_cache[cache_key] = query.with_session(None).bake()
    else:
        query = my_simple_cache[cache_key].with_session(session)

    return query.params(id=id_argument).all()
```

Our "simple" parameterized system must now be tasked with generating cache keys which take into account whether or not the "include_frobnizzle" flag was passed, as the presence of this flag means that the generated SQL would be entirely different. It should be apparent that as the complexity of query building goes up, the task of caching these queries becomes burdensome very quickly. We can convert the above example into a direct use of "bakery" as follows:

```
bakery = baked.bakery()

def lookup(session, id_argument, include_frobnizzle=False):
    def create_model_query(session):
        return session.query(Model).filter(Model.id == bindparam('id'))

    parameterized_query = bakery.bake(create_model_query)

    if include_frobnizzle:
        def include_frobnizzle_in_query(query):
            return query.filter(Model.frobnizzle == True)

        parameterized_query = parameterized_query.with_criteria(
            include_frobnizzle_in_query)

    return parameterized_query(session).params(id=id_argument).all()
```

Above, we again cache not just the query object but all the work it needs to do in order to generate SQL. We also no longer need to deal with making sure we generate a cache key that accurately takes into account all of the structural modifications we've made; this is now handled automatically and without the chance of mistakes.

This code sample is a few lines shorter than the naive example, removes the need to deal with cache keys, and has the vast performance benefits of the full so-called "baked" feature. But still a little verbose! Hence we take methods like *BakedQuery.add_criteria()* and *BakedQuery.with_criteria()* and shorten them into operators, and encourage (though certainly not require!) using simple lambdas, only as a means to reduce verbosity:

```
bakery = baked.bakery()

def lookup(session, id_argument, include_frobnizzle=False):
    parameterized_query = bakery.bake(
        lambda s: s.query(Model).filter(Model.id == bindparam('id'))
      )

    if include_frobnizzle:
        parameterized_query += lambda q: q.filter(Model.frobnizzle == True)

    return parameterized_query(session).params(id=id_argument).all()
```

Where above, the approach is simpler to implement and much more similar in code flow to what a non-cached querying function would look like, hence making code easier to port.

The above description is essentially a summary of the design process used to arrive at the current "baked" approach. Starting from the "normal" approaches, the additional issues of cache key construction and management, removal of all redundant Python execution, and queries built up with conditionals needed to be addressed, leading to the final approach.

Lazy Loading Integration

The baked query can be integrated with SQLAlchemy's lazy loader feature transparently. A future release of SQLAlchemy may enable this by default, as its use within lazy loading is completely transparent. For now, to enable baked lazyloading for all lazyloaders systemwide, call upon the *bake_lazy_loaders()* function. This will impact all relationships that use the `lazy='select'` strategy as well as all use of the *lazyload()* per-query strategy.

"Baked" lazy loading may be enabled on a per-*relationship()* basis using the `baked_select` loader strategy:

```
class MyClass(Base):
    # ...

    widgets = relationship("Widget", lazy="baked_select")
```

The `baked_select` strategy is available once any part of the application has imported the `sqlalchemy.ext.baked` module. The "bakery" used by this feature is local to the mapper for `MyClass`.

For per-query use, the *baked_lazyload()* strategy may be used, which works like any other loader option.

Opting out with the bake_queries flag

The *relationship()* construct includes a flag *relationship.bake_queries* which when set to False will cause that relationship to opt out of the baked query system, when the application-wide *bake_lazy_loaders()* function has been called to enable baked query loaders by default.

API Documentation

`sqlalchemy.ext.baked.`**`bakery`**(*cls*, *size=200*)
Construct a new bakery.

class `sqlalchemy.ext.baked.`**`BakedQuery`**(*bakery*, *initial_fn*, *args=()*)
A builder object for `query.Query` objects.

`add_criteria`(*fn*, **args*)
Add a criteria function to this `BakedQuery`.

This is equivalent to using the += operator to modify a `BakedQuery` in-place.

classmethod `bakery`(*size=200*)
Construct a new bakery.

`for_session`(*session*)
Return a `Result` object for this `BakedQuery`.

This is equivalent to calling the `BakedQuery` as a Python callable, e.g. `result = my_baked_query(session)`.

`spoil`(*full=False*)
Cancel any query caching that will occur on this BakedQuery object.

The BakedQuery can continue to be used normally, however additional creational functions will not be cached; they will be called on every invocation.

This is to support the case where a particular step in constructing a baked query disqualifies the query from being cacheable, such as a variant that relies upon some uncacheable value.

Parameters **`full`** – if False, only functions added to this `BakedQuery` object subsequent to the spoil step will be non-cached; the state of the `BakedQuery` up until this point will be pulled from the cache. If True, then the entire `Query` object is built from scratch each time, with all creational functions being called on each invocation.

`with_criteria`(*fn*, **args*)
Add a criteria function to a `BakedQuery` cloned from this one.

This is equivalent to using the + operator to produce a new `BakedQuery` with modifications.

class `sqlalchemy.ext.baked.`**`Result`**(*bq*, *session*)
Invokes a `BakedQuery` against a `Session`.

The `Result` object is where the actual `query.Query` object gets created, or retrieved from the cache, against a target `Session`, and is then invoked for results.

`all`()
Return all rows.

Equivalent to `Query.all()`.

`first`()
Return the first row.

Equivalent to `Query.first()`.

`get`(*ident*)
Retrieve an object based on identity.

Equivalent to `Query.get()`.

`one`()
Return exactly one result or raise an exception.

Equivalent to `Query.one()`.

one_or_none()
Return one or zero results, or raise an exception for multiple rows.

Equivalent to `Query.one_or_none()`.

New in version 1.0.9.

params(**args*, ***kw*)
Specify parameters to be replaced into the string SQL statement.

sqlalchemy.ext.baked.**bake_lazy_loaders**()
Enable the use of baked queries for all lazyloaders systemwide.

This operation should be safe for all lazy loaders, and will reduce Python overhead for these operations.

sqlalchemy.ext.baked.**unbake_lazy_loaders**()
Disable the use of baked queries for all lazyloaders systemwide.

This operation reverts the changes produced by `bake_lazy_loaders()`.

sqlalchemy.ext.baked.**baked_lazyload**(**keys*)
Indicate that the given attribute should be loaded using "lazy" loading with a "baked" query used in the load.

sqlalchemy.ext.baked.**baked_lazyload_all**(**keys*)
Produce a standalone "all" option for `orm.baked_lazyload()`.

Deprecated since version 0.9.0: The "_all()" style is replaced by method chaining, e.g.:

```
session.query(MyClass).options(
    baked_lazyload("someattribute").baked_lazyload("anotherattribute")
)
```

2.7.4 Declarative

The Declarative system is the typically used system provided by the SQLAlchemy ORM in order to define classes mapped to relational database tables. However, as noted in *Classical Mappings*, Declarative is in fact a series of extensions that ride on top of the SQLAlchemy `mapper()` construct.

While the documentation typically refers to Declarative for most examples, the following sections will provide detailed information on how the Declarative API interacts with the basic `mapper()` and Core `Table` systems, as well as how sophisticated patterns can be built using systems such as mixins.

Basic Use

SQLAlchemy object-relational configuration involves the combination of `Table`, `mapper()`, and class objects to define a mapped class. `declarative` allows all three to be expressed at once within the class declaration. As much as possible, regular SQLAlchemy schema and ORM constructs are used directly, so that configuration between "classical" ORM usage and declarative remain highly similar.

As a simple example:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class SomeClass(Base):
    __tablename__ = 'some_table'
```

```
    id = Column(Integer, primary_key=True)
    name =  Column(String(50))
```

Above, the `declarative_base()` callable returns a new base class from which all mapped classes should inherit. When the class definition is completed, a new *Table* and *mapper()* will have been generated.

The resulting table and mapper are accessible via `__table__` and `__mapper__` attributes on the `SomeClass` class:

```
# access the mapped Table
SomeClass.__table__

# access the Mapper
SomeClass.__mapper__
```

Defining Attributes

In the previous example, the *Column* objects are automatically named with the name of the attribute to which they are assigned.

To name columns explicitly with a name distinct from their mapped attribute, just give the column a name. Below, column "some_table_id" is mapped to the "id" attribute of *SomeClass*, but in SQL will be represented as "some_table_id":

```
class SomeClass(Base):
    __tablename__ = 'some_table'
    id = Column("some_table_id", Integer, primary_key=True)
```

Attributes may be added to the class after its construction, and they will be added to the underlying *Table* and *mapper()* definitions as appropriate:

```
SomeClass.data = Column('data', Unicode)
SomeClass.related = relationship(RelatedInfo)
```

Classes which are constructed using declarative can interact freely with classes that are mapped explicitly with *mapper()*.

It is recommended, though not required, that all tables share the same underlying *MetaData* object, so that string-configured *ForeignKey* references can be resolved without issue.

Accessing the MetaData

The `declarative_base()` base class contains a *MetaData* object where newly defined *Table* objects are collected. This object is intended to be accessed directly for *MetaData*-specific operations. Such as, to issue CREATE statements for all tables:

```
engine = create_engine('sqlite://')
Base.metadata.create_all(engine)
```

`declarative_base()` can also receive a pre-existing *MetaData* object, which allows a declarative setup to be associated with an already existing traditional collection of *Table* objects:

```
mymetadata = MetaData()
Base = declarative_base(metadata=mymetadata)
```

Class Constructor

As a convenience feature, the `declarative_base()` sets a default constructor on classes which takes keyword arguments, and assigns them to the named attributes:

```
e = Engineer(primary_language='python')
```

Mapper Configuration

Declarative makes use of the *mapper()* function internally when it creates the mapping to the declared table. The options for *mapper()* are passed directly through via the `__mapper_args__` class attribute. As always, arguments which reference locally mapped columns can reference them directly from within the class declaration:

```
from datetime import datetime

class Widget(Base):
    __tablename__ = 'widgets'

    id = Column(Integer, primary_key=True)
    timestamp = Column(DateTime, nullable=False)

    __mapper_args__ = {
                    'version_id_col': timestamp,
                    'version_id_generator': lambda v:datetime.now()
                }
```

Defining SQL Expressions

See *SQL Expressions as Mapped Attributes* for examples on declaratively mapping attributes to SQL expressions.

Configuring Relationships

Relationships to other classes are done in the usual way, with the added feature that the class specified to *relationship()* may be a string name. The "class registry" associated with `Base` is used at mapper compilation time to resolve the name into the actual class object, which is expected to have been defined once the mapper configuration is used:

```
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String(50))
    addresses = relationship("Address", backref="user")

class Address(Base):
    __tablename__ = 'addresses'
```

```
    id = Column(Integer, primary_key=True)
    email = Column(String(50))
    user_id = Column(Integer, ForeignKey('users.id'))
```

Column constructs, since they are just that, are immediately usable, as below where we define a primary join condition on the `Address` class using them:

```
class Address(Base):
    __tablename__ = 'addresses'

    id = Column(Integer, primary_key=True)
    email = Column(String(50))
    user_id = Column(Integer, ForeignKey('users.id'))
    user = relationship(User, primaryjoin=user_id == User.id)
```

In addition to the main argument for *relationship()*, other arguments which depend upon the columns present on an as-yet undefined class may also be specified as strings. These strings are evaluated as Python expressions. The full namespace available within this evaluation includes all classes mapped for this declarative base, as well as the contents of the `sqlalchemy` package, including expression functions like *desc()* and *func*:

```
class User(Base):
    # ....
    addresses = relationship("Address",
                             order_by="desc(Address.email)",
                             primaryjoin="Address.user_id==User.id")
```

For the case where more than one module contains a class of the same name, string class names can also be specified as module-qualified paths within any of these string expressions:

```
class User(Base):
    # ....
    addresses = relationship("myapp.model.address.Address",
                             order_by="desc(myapp.model.address.Address.email)",
                             primaryjoin="myapp.model.address.Address.user_id=="
                                         "myapp.model.user.User.id")
```

The qualified path can be any partial path that removes ambiguity between the names. For example, to disambiguate between `myapp.model.address.Address` and `myapp.model.lookup.Address`, we can specify `address.Address` or `lookup.Address`:

```
class User(Base):
    # ....
    addresses = relationship("address.Address",
                             order_by="desc(address.Address.email)",
                             primaryjoin="address.Address.user_id=="
                                         "User.id")
```

New in version 0.8: module-qualified paths can be used when specifying string arguments with Declarative, in order to specify specific modules.

Two alternatives also exist to using string-based attributes. A lambda can also be used, which will be evaluated after all mappers have been configured:

```
class User(Base):
    # ...
    addresses = relationship(lambda: Address,
                             order_by=lambda: desc(Address.email),
                             primaryjoin=lambda: Address.user_id==User.id)
```

Or, the relationship can be added to the class explicitly after the classes are available:

```
User.addresses = relationship(Address,
                              primaryjoin=Address.user_id==User.id)
```

Configuring Many-to-Many Relationships

Many-to-many relationships are also declared in the same way with declarative as with traditional mappings. The `secondary` argument to *relationship()* is as usual passed a *Table* object, which is typically declared in the traditional way. The *Table* usually shares the *MetaData* object used by the declarative base:

```
keywords = Table(
    'keywords', Base.metadata,
    Column('author_id', Integer, ForeignKey('authors.id')),
    Column('keyword_id', Integer, ForeignKey('keywords.id'))
    )

class Author(Base):
    __tablename__ = 'authors'
    id = Column(Integer, primary_key=True)
    keywords = relationship("Keyword", secondary=keywords)
```

Like other *relationship()* arguments, a string is accepted as well, passing the string name of the table as defined in the `Base.metadata.tables` collection:

```
class Author(Base):
    __tablename__ = 'authors'
    id = Column(Integer, primary_key=True)
    keywords = relationship("Keyword", secondary="keywords")
```

As with traditional mapping, its generally not a good idea to use a *Table* as the "secondary" argument which is also mapped to a class, unless the *relationship()* is declared with `viewonly=True`. Otherwise, the unit-of-work system may attempt duplicate INSERT and DELETE statements against the underlying table.

Table Configuration

Table arguments other than the name, metadata, and mapped Column arguments are specified using the `__table_args__` class attribute. This attribute accommodates both positional as well as keyword arguments that are normally sent to the *Table* constructor. The attribute can be specified in one of two forms. One is as a dictionary:

```
class MyClass(Base):
    __tablename__ = 'sometable'
    __table_args__ = {'mysql_engine':'InnoDB'}
```

The other, a tuple, where each argument is positional (usually constraints):

```
class MyClass(Base):
    __tablename__ = 'sometable'
    __table_args__ = (
            ForeignKeyConstraint(['id'], ['remote_table.id']),
            UniqueConstraint('foo'),
            )
```

Keyword arguments can be specified with the above form by specifying the last argument as a dictionary:

```
class MyClass(Base):
    __tablename__ = 'sometable'
    __table_args__ = (
            ForeignKeyConstraint(['id'], ['remote_table.id']),
            UniqueConstraint('foo'),
            {'autoload':True}
            )
```

Using a Hybrid Approach with __table__

As an alternative to `__tablename__`, a direct *Table* construct may be used. The *Column* objects, which in this case require their names, will be added to the mapping just like a regular mapping to a table:

```
class MyClass(Base):
    __table__ = Table('my_table', Base.metadata,
        Column('id', Integer, primary_key=True),
        Column('name', String(50))
    )
```

`__table__` provides a more focused point of control for establishing table metadata, while still getting most of the benefits of using declarative. An application that uses reflection might want to load table metadata elsewhere and pass it to declarative classes:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
Base.metadata.reflect(some_engine)

class User(Base):
    __table__ = metadata.tables['user']

class Address(Base):
    __table__ = metadata.tables['address']
```

Some configuration schemes may find it more appropriate to use `__table__`, such as those which already take advantage of the data-driven nature of *Table* to customize and/or automate schema definition.

Note that when the `__table__` approach is used, the object is immediately usable as a plain *Table* within the class declaration body itself, as a Python class is only another syntactical block. Below this is illustrated by using the `id` column in the `primaryjoin` condition of a *relationship()*:

```
class MyClass(Base):
    __table__ = Table('my_table', Base.metadata,
        Column('id', Integer, primary_key=True),
```

```
        Column('name', String(50))
    )

    widgets = relationship(Widget,
                primaryjoin=Widget.myclass_id==__table__.c.id)
```

Similarly, mapped attributes which refer to `__table__` can be placed inline, as below where we assign the `name` column to the attribute `_name`, generating a synonym for `name`:

```
from sqlalchemy.ext.declarative import synonym_for

class MyClass(Base):
    __table__ = Table('my_table', Base.metadata,
        Column('id', Integer, primary_key=True),
        Column('name', String(50))
    )

    _name = __table__.c.name

    @synonym_for("_name")
    def name(self):
        return "Name: %s" % _name
```

Using Reflection with Declarative

It's easy to set up a *Table* that uses `autoload=True` in conjunction with a mapped class:

```
class MyClass(Base):
    __table__ = Table('mytable', Base.metadata,
                      autoload=True, autoload_with=some_engine)
```

However, one improvement that can be made here is to not require the *Engine* to be available when classes are being first declared. To achieve this, use the *DeferredReflection* mixin, which sets up mappings only after a special `prepare(engine)` step is called:

```
from sqlalchemy.ext.declarative import declarative_base, DeferredReflection

Base = declarative_base(cls=DeferredReflection)

class Foo(Base):
    __tablename__ = 'foo'
    bars = relationship("Bar")

class Bar(Base):
    __tablename__ = 'bar'

    # illustrate overriding of "bar.foo_id" to have
    # a foreign key constraint otherwise not
    # reflected, such as when using MySQL
    foo_id = Column(Integer, ForeignKey('foo.id'))

Base.prepare(e)
```

New in version 0.8: Added *DeferredReflection*.

Inheritance Configuration

Declarative supports all three forms of inheritance as intuitively as possible. The `inherits` mapper keyword argument is not needed as declarative will determine this from the class itself. The various "polymorphic" keyword arguments are specified using `__mapper_args__`.

Joined Table Inheritance

Joined table inheritance is defined as a subclass that defines its own table:

```
class Person(Base):
    __tablename__ = 'people'
    id = Column(Integer, primary_key=True)
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

class Engineer(Person):
    __tablename__ = 'engineers'
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
    id = Column(Integer, ForeignKey('people.id'), primary_key=True)
    primary_language = Column(String(50))
```

Note that above, the `Engineer.id` attribute, since it shares the same attribute name as the `Person.id` attribute, will in fact represent the `people.id` and `engineers.id` columns together, with the "Engineer.id" column taking precedence if queried directly. To provide the `Engineer` class with an attribute that represents only the `engineers.id` column, give it a different attribute name:

```
class Engineer(Person):
    __tablename__ = 'engineers'
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
    engineer_id = Column('id', Integer, ForeignKey('people.id'),
                                        primary_key=True)
    primary_language = Column(String(50))
```

Changed in version 0.7: joined table inheritance favors the subclass column over that of the superclass, such as querying above for `Engineer.id`. Prior to 0.7 this was the reverse.

Single Table Inheritance

Single table inheritance is defined as a subclass that does not have its own table; you just leave out the `__table__` and `__tablename__` attributes:

```
class Person(Base):
    __tablename__ = 'people'
    id = Column(Integer, primary_key=True)
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

class Engineer(Person):
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
    primary_language = Column(String(50))
```

When the above mappers are configured, the `Person` class is mapped to the `people` table *before* the `primary_language` column is defined, and this column will not be included in its own mapping. When `Engineer` then defines the `primary_language` column, the column is added to the `people` table so that it is included in the mapping for `Engineer` and is also part of the table's full set of columns. Columns which are not mapped to `Person` are also excluded from any other single or joined inheriting classes using the `exclude_properties` mapper argument. Below, `Manager` will have all the attributes of `Person` and `Manager` but *not* the `primary_language` attribute of `Engineer`:

```
class Manager(Person):
    __mapper_args__ = {'polymorphic_identity': 'manager'}
    golf_swing = Column(String(50))
```

The attribute exclusion logic is provided by the `exclude_properties` mapper argument, and declarative's default behavior can be disabled by passing an explicit `exclude_properties` collection (empty or otherwise) to the `__mapper_args__`.

Resolving Column Conflicts Note above that the `primary_language` and `golf_swing` columns are "moved up" to be applied to `Person.__table__`, as a result of their declaration on a subclass that has no table of its own. A tricky case comes up when two subclasses want to specify *the same* column, as below:

```
class Person(Base):
    __tablename__ = 'people'
    id = Column(Integer, primary_key=True)
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

class Engineer(Person):
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
    start_date = Column(DateTime)

class Manager(Person):
    __mapper_args__ = {'polymorphic_identity': 'manager'}
    start_date = Column(DateTime)
```

Above, the `start_date` column declared on both `Engineer` and `Manager` will result in an error:

```
sqlalchemy.exc.ArgumentError: Column 'start_date' on class
<class '__main__.Manager'> conflicts with existing
column 'people.start_date'
```

In a situation like this, Declarative can't be sure of the intent, especially if the `start_date` columns had, for example, different types. A situation like this can be resolved by using *`declared_attr`* to define the *`Column`* conditionally, taking care to return the **existing column** via the parent `__table__` if it already exists:

```
from sqlalchemy.ext.declarative import declared_attr

class Person(Base):
    __tablename__ = 'people'
    id = Column(Integer, primary_key=True)
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

class Engineer(Person):
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
```

```
    @declared_attr
    def start_date(cls):
        "Start date column, if not present already."
        return Person.__table__.c.get('start_date', Column(DateTime))

class Manager(Person):
    __mapper_args__ = {'polymorphic_identity': 'manager'}

    @declared_attr
    def start_date(cls):
        "Start date column, if not present already."
        return Person.__table__.c.get('start_date', Column(DateTime))
```

Above, when `Manager` is mapped, the `start_date` column is already present on the `Person` class. Declarative lets us return that *Column* as a result in this case, where it knows to skip re-assigning the same column. If the mapping is mis-configured such that the `start_date` column is accidentally re-assigned to a different table (such as, if we changed `Manager` to be joined inheritance without fixing `start_date`), an error is raised which indicates an existing *Column* is trying to be re-assigned to a different owning *Table*.

New in version 0.8: *declared_attr* can be used on a non-mixin class, and the returned *Column* or other mapped attribute will be applied to the mapping as any other attribute. Previously, the resulting attribute would be ignored, and also result in a warning being emitted when a subclass was created.

New in version 0.8: *declared_attr*, when used either with a mixin or non-mixin declarative class, can return an existing *Column* already assigned to the parent *Table*, to indicate that the re-assignment of the *Column* should be skipped, however should still be mapped on the target class, in order to resolve duplicate column conflicts.

The same concept can be used with mixin classes (see *Mixin and Custom Base Classes*):

```
class Person(Base):
    __tablename__ = 'people'
    id = Column(Integer, primary_key=True)
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

class HasStartDate(object):
    @declared_attr
    def start_date(cls):
        return cls.__table__.c.get('start_date', Column(DateTime))

class Engineer(HasStartDate, Person):
    __mapper_args__ = {'polymorphic_identity': 'engineer'}

class Manager(HasStartDate, Person):
    __mapper_args__ = {'polymorphic_identity': 'manager'}
```

The above mixin checks the local `__table__` attribute for the column. Because we're using single table inheritance, we're sure that in this case, `cls.__table__` refers to `People.__table__`. If we were mixing joined- and single-table inheritance, we might want our mixin to check more carefully if `cls.__table__` is really the *Table* we're looking for.

Concrete Table Inheritance

Concrete is defined as a subclass which has its own table and sets the `concrete` keyword argument to `True`:

```
class Person(Base):
    __tablename__ = 'people'
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

class Engineer(Person):
    __tablename__ = 'engineers'
    __mapper_args__ = {'concrete':True}
    id = Column(Integer, primary_key=True)
    primary_language = Column(String(50))
    name = Column(String(50))
```

Usage of an abstract base class is a little less straightforward as it requires usage of `polymorphic_union()`, which needs to be created with the `Table` objects before the class is built:

```
engineers = Table('engineers', Base.metadata,
                Column('id', Integer, primary_key=True),
                Column('name', String(50)),
                Column('primary_language', String(50))
            )
managers = Table('managers', Base.metadata,
                Column('id', Integer, primary_key=True),
                Column('name', String(50)),
                Column('golf_swing', String(50))
            )

punion = polymorphic_union({
    'engineer':engineers,
    'manager':managers
}, 'type', 'punion')

class Person(Base):
    __table__ = punion
    __mapper_args__ = {'polymorphic_on':punion.c.type}

class Engineer(Person):
    __table__ = engineers
    __mapper_args__ = {'polymorphic_identity':'engineer', 'concrete':True}

class Manager(Person):
    __table__ = managers
    __mapper_args__ = {'polymorphic_identity':'manager', 'concrete':True}
```

Using the Concrete Helpers Helper classes provides a simpler pattern for concrete inheritance. With these objects, the `__declare_first__` helper is used to configure the "polymorphic" loader for the mapper after all subclasses have been declared.

New in version 0.7.3.

An abstract base can be declared using the `AbstractConcreteBase` class:

```
from sqlalchemy.ext.declarative import AbstractConcreteBase

class Employee(AbstractConcreteBase, Base):
    pass
```

To have a concrete `employee` table, use *ConcreteBase* instead:

```
from sqlalchemy.ext.declarative import ConcreteBase

class Employee(ConcreteBase, Base):
    __tablename__ = 'employee'
    employee_id = Column(Integer, primary_key=True)
    name = Column(String(50))
    __mapper_args__ = {
                    'polymorphic_identity':'employee',
                    'concrete':True}
```

Either `Employee` base can be used in the normal fashion:

```
class Manager(Employee):
    __tablename__ = 'manager'
    employee_id = Column(Integer, primary_key=True)
    name = Column(String(50))
    manager_data = Column(String(40))
    __mapper_args__ = {
                    'polymorphic_identity':'manager',
                    'concrete':True}

class Engineer(Employee):
    __tablename__ = 'engineer'
    employee_id = Column(Integer, primary_key=True)
    name = Column(String(50))
    engineer_info = Column(String(40))
    __mapper_args__ = {'polymorphic_identity':'engineer',
                    'concrete':True}
```

The *AbstractConcreteBase* class is itself mapped, and can be used as a target of relationships:

```
class Company(Base):
    __tablename__ = 'company'

    id = Column(Integer, primary_key=True)
    employees = relationship("Employee",
                    primaryjoin="Company.id == Employee.company_id")
```

Changed in version 0.9.3: Support for use of *AbstractConcreteBase* as the target of a *relationship()* has been improved.

It can also be queried directly:

```
for employee in session.query(Employee).filter(Employee.name == 'qbert'):
    print(employee)
```

Mixin and Custom Base Classes

A common need when using *declarative* is to share some functionality, such as a set of common columns, some common table options, or other mapped properties, across many classes. The standard Python idioms for this is to have the classes inherit from a base which includes these common features.

When using *declarative*, this idiom is allowed via the usage of a custom declarative base class, as well as a "mixin" class which is inherited from in addition to the primary base. Declarative includes several helper features to make this work in terms of how mappings are declared. An example of some commonly mixed-in idioms is below:

```
from sqlalchemy.ext.declarative import declared_attr

class MyMixin(object):

    @declared_attr
    def __tablename__(cls):
        return cls.__name__.lower()

    __table_args__ = {'mysql_engine': 'InnoDB'}
    __mapper_args__= {'always_refresh': True}

    id =  Column(Integer, primary_key=True)

class MyModel(MyMixin, Base):
    name = Column(String(1000))
```

Where above, the class `MyModel` will contain an "id" column as the primary key, a `__tablename__` attribute that derives from the name of the class itself, as well as `__table_args__` and `__mapper_args__` defined by the `MyMixin` mixin class.

There's no fixed convention over whether `MyMixin` precedes `Base` or not. Normal Python method resolution rules apply, and the above example would work just as well with:

```
class MyModel(Base, MyMixin):
    name = Column(String(1000))
```

This works because `Base` here doesn't define any of the variables that `MyMixin` defines, i.e. `__tablename__`, `__table_args__`, `id`, etc. If the `Base` did define an attribute of the same name, the class placed first in the inherits list would determine which attribute is used on the newly defined class.

Augmenting the Base

In addition to using a pure mixin, most of the techniques in this section can also be applied to the base class itself, for patterns that should apply to all classes derived from a particular base. This is achieved using the `cls` argument of the *declarative_base()* function:

```
from sqlalchemy.ext.declarative import declared_attr

class Base(object):
    @declared_attr
    def __tablename__(cls):
        return cls.__name__.lower()

    __table_args__ = {'mysql_engine': 'InnoDB'}

    id =  Column(Integer, primary_key=True)

from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base(cls=Base)
```

```
class MyModel(Base):
    name = Column(String(1000))
```

Where above, `MyModel` and all other classes that derive from `Base` will have a table name derived from the class name, an `id` primary key column, as well as the "InnoDB" engine for MySQL.

Mixing in Columns

The most basic way to specify a column on a mixin is by simple declaration:

```
class TimestampMixin(object):
    created_at = Column(DateTime, default=func.now())

class MyModel(TimestampMixin, Base):
    __tablename__ = 'test'

    id =  Column(Integer, primary_key=True)
    name = Column(String(1000))
```

Where above, all declarative classes that include `TimestampMixin` will also have a column `created_at` that applies a timestamp to all row insertions.

Those familiar with the SQLAlchemy expression language know that the object identity of clause elements defines their role in a schema. Two `Table` objects `a` and `b` may both have a column called `id`, but the way these are differentiated is that `a.c.id` and `b.c.id` are two distinct Python objects, referencing their parent tables `a` and `b` respectively.

In the case of the mixin column, it seems that only one *`Column`* object is explicitly created, yet the ultimate `created_at` column above must exist as a distinct Python object for each separate destination class. To accomplish this, the declarative extension creates a **copy** of each *`Column`* object encountered on a class that is detected as a mixin.

This copy mechanism is limited to simple columns that have no foreign keys, as a *`ForeignKey`* itself contains references to columns which can't be properly recreated at this level. For columns that have foreign keys, as well as for the variety of mapper-level constructs that require destination-explicit context, the *`declared_attr`* decorator is provided so that patterns common to many classes can be defined as callables:

```
from sqlalchemy.ext.declarative import declared_attr

class ReferenceAddressMixin(object):
    @declared_attr
    def address_id(cls):
        return Column(Integer, ForeignKey('address.id'))

class User(ReferenceAddressMixin, Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
```

Where above, the `address_id` class-level callable is executed at the point at which the `User` class is constructed, and the declarative extension can use the resulting *`Column`* object as returned by the method without the need to copy it.

Changed in version 0.6.5: Rename `sqlalchemy.util.classproperty` into *`declared_attr`*.

Columns generated by *declared_attr* can also be referenced by `__mapper_args__` to a limited degree, currently by `polymorphic_on` and `version_id_col`; the declarative extension will resolve them at class construction time:

```
class MyMixin:
    @declared_attr
    def type_(cls):
        return Column(String(50))

    __mapper_args__= {'polymorphic_on':type_}

class MyModel(MyMixin, Base):
    __tablename__='test'
    id =  Column(Integer, primary_key=True)
```

Mixing in Relationships

Relationships created by *relationship()* are provided with declarative mixin classes exclusively using the *declared_attr* approach, eliminating any ambiguity which could arise when copying a relationship and its possibly column-bound contents. Below is an example which combines a foreign key column and a relationship so that two classes `Foo` and `Bar` can both be configured to reference a common target class via many-to-one:

```
class RefTargetMixin(object):
    @declared_attr
    def target_id(cls):
        return Column('target_id', ForeignKey('target.id'))

    @declared_attr
    def target(cls):
        return relationship("Target")

class Foo(RefTargetMixin, Base):
    __tablename__ = 'foo'
    id = Column(Integer, primary_key=True)

class Bar(RefTargetMixin, Base):
    __tablename__ = 'bar'
    id = Column(Integer, primary_key=True)

class Target(Base):
    __tablename__ = 'target'
    id = Column(Integer, primary_key=True)
```

Using Advanced Relationship Arguments (e.g. `primaryjoin`, etc.) *relationship()* definitions which require explicit primaryjoin, order_by etc. expressions should in all but the most simplistic cases use **late bound** forms for these arguments, meaning, using either the string form or a lambda. The reason for this is that the related *Column* objects which are to be configured using `@declared_attr` are not available to another `@declared_attr` attribute; while the methods will work and return new *Column* objects, those are not the *Column* objects that Declarative will be using as it calls the methods on its own, thus using *different* *Column* objects.

The canonical example is the primaryjoin condition that depends upon another mixed-in column:

```
class RefTargetMixin(object):
    @declared_attr
    def target_id(cls):
        return Column('target_id', ForeignKey('target.id'))

    @declared_attr
    def target(cls):
        return relationship(Target,
            primaryjoin=Target.id==cls.target_id   # this is *incorrect*
        )
```

Mapping a class using the above mixin, we will get an error like:

```
sqlalchemy.exc.InvalidRequestError: this ForeignKey's parent column is not
yet associated with a Table.
```

This is because the `target_id` *Column* we've called upon in our `target()` method is not the same *Column* that declarative is actually going to map to our table.

The condition above is resolved using a lambda:

```
class RefTargetMixin(object):
    @declared_attr
    def target_id(cls):
        return Column('target_id', ForeignKey('target.id'))

    @declared_attr
    def target(cls):
        return relationship(Target,
            primaryjoin=lambda: Target.id==cls.target_id
        )
```

or alternatively, the string form (which ultimately generates a lambda):

```
class RefTargetMixin(object):
    @declared_attr
    def target_id(cls):
        return Column('target_id', ForeignKey('target.id'))

    @declared_attr
    def target(cls):
        return relationship("Target",
            primaryjoin="Target.id==%s.target_id" % cls.__name__
        )
```

Mixing in deferred(), column_property(), and other MapperProperty classes

Like *relationship()*, all *MapperProperty* subclasses such as *deferred()*, *column_property()*, etc. ultimately involve references to columns, and therefore, when used with declarative mixins, have the *declared_attr* requirement so that no reliance on copying is needed:

```
class SomethingMixin(object):

    @declared_attr
```

```
    def dprop(cls):
        return deferred(Column(Integer))

class Something(SomethingMixin, Base):
    __tablename__ = "something"
```

The *column_property()* or other construct may refer to other columns from the mixin. These are copied ahead of time before the *declared_attr* is invoked:

```
class SomethingMixin(object):
    x = Column(Integer)

    y = Column(Integer)

    @declared_attr
    def x_plus_y(cls):
        return column_property(cls.x + cls.y)
```

Changed in version 1.0.0: mixin columns are copied to the final mapped class so that *declared_attr* methods can access the actual column that will be mapped.

Mixing in Association Proxy and Other Attributes

Mixins can specify user-defined attributes as well as other extension units such as *association_proxy()*. The usage of *declared_attr* is required in those cases where the attribute must be tailored specifically to the target subclass. An example is when constructing multiple *association_proxy()* attributes which each target a different type of child object. Below is an *association_proxy()* / mixin example which provides a scalar list of string values to an implementing class:

```
from sqlalchemy import Column, Integer, ForeignKey, String
from sqlalchemy.orm import relationship
from sqlalchemy.ext.associationproxy import association_proxy
from sqlalchemy.ext.declarative import declarative_base, declared_attr

Base = declarative_base()

class HasStringCollection(object):
    @declared_attr
    def _strings(cls):
        class StringAttribute(Base):
            __tablename__ = cls.string_table_name
            id = Column(Integer, primary_key=True)
            value = Column(String(50), nullable=False)
            parent_id = Column(Integer,
                               ForeignKey('%s.id' % cls.__tablename__),
                               nullable=False)
            def __init__(self, value):
                self.value = value

        return relationship(StringAttribute)

    @declared_attr
    def strings(cls):
        return association_proxy('_strings', 'value')
```

```
class TypeA(HasStringCollection, Base):
    __tablename__ = 'type_a'
    string_table_name = 'type_a_strings'
    id = Column(Integer(), primary_key=True)

class TypeB(HasStringCollection, Base):
    __tablename__ = 'type_b'
    string_table_name = 'type_b_strings'
    id = Column(Integer(), primary_key=True)
```

Above, the `HasStringCollection` mixin produces a *relationship()* which refers to a newly generated class called `StringAttribute`. The `StringAttribute` class is generated with its own *Table* definition which is local to the parent class making usage of the `HasStringCollection` mixin. It also produces an *association_proxy()* object which proxies references to the `strings` attribute onto the `value` attribute of each `StringAttribute` instance.

`TypeA` or `TypeB` can be instantiated given the constructor argument `strings`, a list of strings:

```
ta = TypeA(strings=['foo', 'bar'])
tb = TypeA(strings=['bat', 'bar'])
```

This list will generate a collection of `StringAttribute` objects, which are persisted into a table that's local to either the `type_a_strings` or `type_b_strings` table:

```
>>> print ta._strings
[<__main__.StringAttribute object at 0x10151cd90>,
    <__main__.StringAttribute object at 0x10151ce10>]
```

When constructing the *association_proxy()*, the *declared_attr* decorator must be used so that a distinct *association_proxy()* object is created for each of the `TypeA` and `TypeB` classes.

New in version 0.8: *declared_attr* is usable with non-mapped attributes, including user-defined attributes as well as *association_proxy()*.

Controlling table inheritance with mixins

The `__tablename__` attribute may be used to provide a function that will determine the name of the table used for each class in an inheritance hierarchy, as well as whether a class has its own distinct table.

This is achieved using the *declared_attr* indicator in conjunction with a method named `__tablename__()`. Declarative will always invoke *declared_attr* for the special names `__tablename__`, `__mapper_args__` and `__table_args__` function **for each mapped class in the hierarchy**. The function therefore needs to expect to receive each class individually and to provide the correct answer for each.

For example, to create a mixin that gives every class a simple table name based on class name:

```
from sqlalchemy.ext.declarative import declared_attr

class Tablename:
    @declared_attr
    def __tablename__(cls):
        return cls.__name__.lower()

class Person(Tablename, Base):
    id = Column(Integer, primary_key=True)
```

```
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

class Engineer(Person):
    __tablename__ = None
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
    primary_language = Column(String(50))
```

Alternatively, we can modify our `__tablename__` function to return `None` for subclasses, using *has_inherited_table()*. This has the effect of those subclasses being mapped with single table inheritance against the parent:

```
from sqlalchemy.ext.declarative import declared_attr
from sqlalchemy.ext.declarative import has_inherited_table

class Tablename(object):
    @declared_attr
    def __tablename__(cls):
        if has_inherited_table(cls):
            return None
        return cls.__name__.lower()

class Person(Tablename, Base):
    id = Column(Integer, primary_key=True)
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

class Engineer(Person):
    primary_language = Column(String(50))
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
```

Mixing in Columns in Inheritance Scenarios

In constrast to how `__tablename__` and other special names are handled when used with *declared_attr*, when we mix in columns and properties (e.g. relationships, column properties, etc.), the function is invoked for the **base class only** in the hierarchy. Below, only the `Person` class will receive a column called `id`; the mapping will fail on `Engineer`, which is not given a primary key:

```
class HasId(object):
    @declared_attr
    def id(cls):
        return Column('id', Integer, primary_key=True)

class Person(HasId, Base):
    __tablename__ = 'person'
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

class Engineer(Person):
    __tablename__ = 'engineer'
    primary_language = Column(String(50))
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
```

It is usually the case in joined-table inheritance that we want distinctly named columns on each subclass. However in this case, we may want to have an `id` column on every table, and have them refer to each other via foreign key. We

can achieve this as a mixin by using the *declared_attr.cascading* modifier, which indicates that the function should be invoked **for each class in the hierarchy**, just like it does for `__tablename__`:

```
class HasId(object):
    @declared_attr.cascading
    def id(cls):
        if has_inherited_table(cls):
            return Column('id',
                          Integer,
                          ForeignKey('person.id'), primary_key=True)
        else:
            return Column('id', Integer, primary_key=True)

class Person(HasId, Base):
    __tablename__ = 'person'
    discriminator = Column('type', String(50))
    __mapper_args__ = {'polymorphic_on': discriminator}

class Engineer(Person):
    __tablename__ = 'engineer'
    primary_language = Column(String(50))
    __mapper_args__ = {'polymorphic_identity': 'engineer'}
```

New in version 1.0.0: added *declared_attr.cascading*.

Combining Table/Mapper Arguments from Multiple Mixins

In the case of `__table_args__` or `__mapper_args__` specified with declarative mixins, you may want to combine some parameters from several mixins with those you wish to define on the class iteself. The *declared_attr* decorator can be used here to create user-defined collation routines that pull from multiple collections:

```
from sqlalchemy.ext.declarative import declared_attr

class MySQLSettings(object):
    __table_args__ = {'mysql_engine':'InnoDB'}

class MyOtherMixin(object):
    __table_args__ = {'info':'foo'}

class MyModel(MySQLSettings, MyOtherMixin, Base):
    __tablename__='my_model'

    @declared_attr
    def __table_args__(cls):
        args = dict()
        args.update(MySQLSettings.__table_args__)
        args.update(MyOtherMixin.__table_args__)
        return args

    id =  Column(Integer, primary_key=True)
```

Creating Indexes with Mixins

To define a named, potentially multicolumn `Index` that applies to all tables derived from a mixin, use the "inline" form of `Index` and establish it as part of `__table_args__`:

```
class MyMixin(object):
    a =  Column(Integer)
    b =  Column(Integer)

    @declared_attr
    def __table_args__(cls):
        return (Index('test_idx_%s' % cls.__tablename__, 'a', 'b'),)

class MyModel(MyMixin, Base):
    __tablename__ = 'atable'
    c =  Column(Integer,primary_key=True)
```

Declarative API

API Reference

`sqlalchemy.ext.declarative.`**`declarative_base`**(*bind=None, metadata=None, mapper=None, cls=<type 'object'>, name='Base', constructor=<function __init__>, class_registry=None, metaclass=<class 'sqlalchemy.ext.declarative.api.DeclarativeMeta'>*)

Construct a base class for declarative class definitions.

The new base class will be given a metaclass that produces appropriate `Table` objects and makes the appropriate `mapper()` calls based on the information provided declaratively in the class and any subclasses of the class.

Parameters

- **bind** – An optional `Connectable`, will be assigned the `bind` attribute on the `MetaData` instance.
- **metadata** – An optional `MetaData` instance. All `Table` objects implicitly declared by subclasses of the base will share this MetaData. A MetaData instance will be created if none is provided. The `MetaData` instance will be available via the *metadata* attribute of the generated declarative base class.
- **mapper** – An optional callable, defaults to `mapper()`. Will be used to map subclasses to their Tables.
- **cls** – Defaults to `object`. A type to use as the base for the generated declarative base class. May be a class or tuple of classes.
- **name** – Defaults to `Base`. The display name for the generated class. Customizing this is not required, but can improve clarity in tracebacks and debugging.
- **constructor** – Defaults to `_declarative_constructor()`, an __init__ implementation that assigns **kwargs for declared fields and relationships to an instance. If `None` is supplied, no __init__ will be provided and construction will fall back to cls.__init__ by way of the normal Python semantics.
- **class_registry** – optional dictionary that will serve as the registry of class names-> mapped classes when string names are used to identify classes inside of

relationship() and others. Allows two or more declarative base classes to share the same registry of class names for simplified inter-base relationships.

- **metaclass** – Defaults to `DeclarativeMeta`. A metaclass or __metaclass__ compatible callable to use as the meta type of the generated declarative base class.

See also:

as_declarative()

`sqlalchemy.ext.declarative.`**`as_declarative`**(***kw*)

Class decorator for *declarative_base()*.

Provides a syntactical shortcut to the `cls` argument sent to *declarative_base()*, allowing the base class to be converted in-place to a "declarative" base:

```
from sqlalchemy.ext.declarative import as_declarative

@as_declarative()
class Base(object):
    @declared_attr
    def __tablename__(cls):
        return cls.__name__.lower()
    id = Column(Integer, primary_key=True)

class MyMappedClass(Base):
    # ...
```

All keyword arguments passed to *as_declarative()* are passed along to *declarative_base()*.

New in version 0.8.3.

See also:

declarative_base()

class `sqlalchemy.ext.declarative.`**`declared_attr`**(*fget*, *cascading=False*)

Bases: `sqlalchemy.orm.base._MappedAttribute`, `__builtin__.property`

Mark a class-level method as representing the definition of a mapped property or special declarative member name.

@declared_attr turns the attribute into a scalar-like property that can be invoked from the uninstantiated class. Declarative treats attributes specifically marked with @declared_attr as returning a construct that is specific to mapping or declarative table configuration. The name of the attribute is that of what the non-dynamic version of the attribute would be.

@declared_attr is more often than not applicable to mixins, to define relationships that are to be applied to different implementors of the class:

```
class ProvidesUser(object):
    "A mixin that adds a 'user' relationship to classes."

    @declared_attr
    def user(self):
        return relationship("User")
```

It also can be applied to mapped classes, such as to provide a "polymorphic" scheme for inheritance:

```
class Employee(Base):
    id = Column(Integer, primary_key=True)
    type = Column(String(50), nullable=False)
```

```
        @declared_attr
        def __tablename__(cls):
            return cls.__name__.lower()

        @declared_attr
        def __mapper_args__(cls):
            if cls.__name__ == 'Employee':
                return {
                        "polymorphic_on":cls.type,
                        "polymorphic_identity":"Employee"
                }
            else:
                return {"polymorphic_identity":cls.__name__}
```

Changed in version 0.8: *declared_attr* can be used with non-ORM or extension attributes, such as user-defined attributes or *association_proxy()* objects, which will be assigned to the class at class construction time.

cascading

Mark a *declared_attr* as cascading.

This is a special-use modifier which indicates that a column or MapperProperty-based declared attribute should be configured distinctly per mapped subclass, within a mapped-inheritance scenario.

Below, both MyClass as well as MySubClass will have a distinct `id` Column object established:

```
        class HasSomeAttribute(object):
            @declared_attr.cascading
            def some_id(cls):
                if has_inherited_table(cls):
                    return Column(
                        ForeignKey('myclass.id'), primary_key=True)
                else:
                    return Column(Integer, primary_key=True)

                return Column('id', Integer, primary_key=True)

        class MyClass(HasSomeAttribute, Base):
            ""
            # ...

        class MySubClass(MyClass):
            ""
            # ...
```

The behavior of the above configuration is that `MySubClass` will refer to both its own `id` column as well as that of `MyClass` underneath the attribute named `some_id`.

See also:

Inheritance Configuration

Mixing in Columns in Inheritance Scenarios

`sqlalchemy.ext.declarative.api.`**`_declarative_constructor`**(*self*, ***kwargs*)

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

sqlalchemy.ext.declarative.**has_inherited_table**(*cls*)
Given a class, return True if any of the classes it inherits from has a mapped table, otherwise return False.

sqlalchemy.ext.declarative.**synonym_for**(*name*, *map_column=False*)
Decorator, make a Python @property a query synonym for a column.

A decorator version of *synonym()*. The function being decorated is the 'descriptor', otherwise passes its arguments through to synonym():

```
@synonym_for('col')
@property
def prop(self):
    return 'special sauce'
```

The regular synonym() is also usable directly in a declarative setting and may be convenient for read/write properties:

```
prop = synonym('col', descriptor=property(_read_prop, _write_prop))
```

sqlalchemy.ext.declarative.**comparable_using**(*comparator_factory*)
Decorator, allow a Python @property to be used in query criteria.

This is a decorator front end to comparable_property() that passes through the comparator_factory and the function being decorated:

```
@comparable_using(MyComparatorType)
@property
def prop(self):
    return 'special sauce'
```

The regular comparable_property() is also usable directly in a declarative setting and may be convenient for read/write properties:

```
prop = comparable_property(MyComparatorType)
```

sqlalchemy.ext.declarative.**instrument_declarative**(*cls*, *registry*, *metadata*)
Given a class, configure the class declaratively, using the given registry, which can be any dictionary, and MetaData object.

class sqlalchemy.ext.declarative.**AbstractConcreteBase**
Bases: sqlalchemy.ext.declarative.api.ConcreteBase

A helper class for 'concrete' declarative mappings.

AbstractConcreteBase will use the *polymorphic_union()* function automatically, against all tables mapped as a subclass to this class. The function is called via the __declare_last__() function, which is essentially a hook for the *after_configured()* event.

AbstractConcreteBase does produce a mapped class for the base class, however it is not persisted to any table; it is instead mapped directly to the "polymorphic" selectable directly and is only used for selecting. Compare to *ConcreteBase*, which does create a persisted table for the base class.

Example:

```
from sqlalchemy.ext.declarative import AbstractConcreteBase

class Employee(AbstractConcreteBase, Base):
    pass

class Manager(Employee):
    __tablename__ = 'manager'
    employee_id = Column(Integer, primary_key=True)
```

```
        name = Column(String(50))
        manager_data = Column(String(40))

        __mapper_args__ = {
            'polymorphic_identity':'manager',
            'concrete':True}
```

The abstract base class is handled by declarative in a special way; at class configuration time, it behaves like a declarative mixin or an `__abstract__` base class. Once classes are configured and mappings are produced, it then gets mapped itself, but after all of its decscendants. This is a very unique system of mapping not found in any other SQLAlchemy system.

Using this approach, we can specify columns and properties that will take place on mapped subclasses, in the way that we normally do as in *Mixin and Custom Base Classes*:

```
class Company(Base):
    __tablename__ = 'company'
    id = Column(Integer, primary_key=True)

class Employee(AbstractConcreteBase, Base):
    employee_id = Column(Integer, primary_key=True)

    @declared_attr
    def company_id(cls):
        return Column(ForeignKey('company.id'))

    @declared_attr
    def company(cls):
        return relationship("Company")

class Manager(Employee):
    __tablename__ = 'manager'

    name = Column(String(50))
    manager_data = Column(String(40))

    __mapper_args__ = {
        'polymorphic_identity':'manager',
        'concrete':True}
```

When we make use of our mappings however, both `Manager` and `Employee` will have an independently usable `.company` attribute:

```
session.query(Employee).filter(Employee.company.has(id=5))
```

Changed in version 1.0.0: - The mechanics of *`AbstractConcreteBase`* have been reworked to support relationships established directly on the abstract base, without any special configurational steps.

class `sqlalchemy.ext.declarative.`**`ConcreteBase`**

A helper class for 'concrete' declarative mappings.

`ConcreteBase` will use the *`polymorphic_union()`* function automatically, against all tables mapped as a subclass to this class. The function is called via the `__declare_last__()` function, which is essentially a hook for the *`after_configured()`* event.

`ConcreteBase` produces a mapped table for the class itself. Compare to *`AbstractConcreteBase`*, which does not.

Example:

```
from sqlalchemy.ext.declarative import ConcreteBase

class Employee(ConcreteBase, Base):
    __tablename__ = 'employee'
    employee_id = Column(Integer, primary_key=True)
    name = Column(String(50))
    __mapper_args__ = {
                    'polymorphic_identity':'employee',
                    'concrete':True}

class Manager(Employee):
    __tablename__ = 'manager'
    employee_id = Column(Integer, primary_key=True)
    name = Column(String(50))
    manager_data = Column(String(40))
    __mapper_args__ = {
                    'polymorphic_identity':'manager',
                    'concrete':True}
```

class `sqlalchemy.ext.declarative.`**`DeferredReflection`**

A helper class for construction of mappings based on a deferred reflection step.

Normally, declarative can be used with reflection by setting a *`Table`* object using autoload=True as the `__table__` attribute on a declarative class. The caveat is that the *`Table`* must be fully reflected, or at the very least have a primary key column, at the point at which a normal declarative mapping is constructed, meaning the *`Engine`* must be available at class declaration time.

The *`DeferredReflection`* mixin moves the construction of mappers to be at a later point, after a specific method is called which first reflects all *`Table`* objects created so far. Classes can define it as such:

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.ext.declarative import DeferredReflection
Base = declarative_base()

class MyClass(DeferredReflection, Base):
    __tablename__ = 'mytable'
```

Above, `MyClass` is not yet mapped. After a series of classes have been defined in the above fashion, all tables can be reflected and mappings created using *`prepare()`*:

```
engine = create_engine("someengine://...")
DeferredReflection.prepare(engine)
```

The *`DeferredReflection`* mixin can be applied to individual classes, used as the base for the declarative base itself, or used in a custom abstract class. Using an abstract base allows that only a subset of classes to be prepared for a particular prepare step, which is necessary for applications that use more than one engine. For example, if an application has two engines, you might use two bases, and prepare each separately, e.g.:

```
class ReflectedOne(DeferredReflection, Base):
    __abstract__ = True

class ReflectedTwo(DeferredReflection, Base):
    __abstract__ = True

class MyClass(ReflectedOne):
    __tablename__ = 'mytable'

class MyOtherClass(ReflectedOne):
    __tablename__ = 'myothertable'
```

```
class YetAnotherClass(ReflectedTwo):
    __tablename__ = 'yetanothertable'

# ... etc.
```

Above, the class hierarchies for `ReflectedOne` and `ReflectedTwo` can be configured separately:

```
ReflectedOne.prepare(engine_one)
ReflectedTwo.prepare(engine_two)
```

New in version 0.8.

classmethod prepare(*engine*)

Reflect all *Table* objects for all current *DeferredReflection* subclasses

Special Directives

__declare_last__() The `__declare_last__()` hook allows definition of a class level function that is automatically called by the *MapperEvents.after_configured()* event, which occurs after mappings are assumed to be completed and the 'configure' step has finished:

```
class MyClass(Base):
    @classmethod
    def __declare_last__(cls):
        ""
        # do something with mappings
```

New in version 0.7.3.

__declare_first__() Like `__declare_last__()`, but is called at the beginning of mapper configuration via the *MapperEvents.before_configured()* event:

```
class MyClass(Base):
    @classmethod
    def __declare_first__(cls):
        ""
        # do something before mappings are configured
```

New in version 0.9.3.

__abstract__ `__abstract__` causes declarative to skip the production of a table or mapper for the class entirely. A class can be added within a hierarchy in the same way as mixin (see *Mixin and Custom Base Classes*), allowing subclasses to extend just from the special class:

```
class SomeAbstractBase(Base):
    __abstract__ = True

    def some_helpful_method(self):
        ""

    @declared_attr
    def __mapper_args__(cls):
        return {"helpful mapper arguments":True}
```

```
class MyMappedClass(SomeAbstractBase):
    ""
```

One possible use of `__abstract__` is to use a distinct *MetaData* for different bases:

```
Base = declarative_base()

class DefaultBase(Base):
    __abstract__ = True
    metadata = MetaData()

class OtherBase(Base):
    __abstract__ = True
    metadata = MetaData()
```

Above, classes which inherit from `DefaultBase` will use one *MetaData* as the registry of tables, and those which inherit from `OtherBase` will use a different one. The tables themselves can then be created perhaps within distinct databases:

```
DefaultBase.metadata.create_all(some_engine)
OtherBase.metadata_create_all(some_other_engine)
```

New in version 0.7.3.

2.7.5 Mutation Tracking

Provide support for tracking of in-place changes to scalar values, which are propagated into ORM change events on owning parent objects.

New in version 0.7: *sqlalchemy.ext.mutable* replaces SQLAlchemy's legacy approach to in-place mutations of scalar values; see *Mutation event extension, supersedes "mutable=True"*.

Establishing Mutability on Scalar Column Values

A typical example of a "mutable" structure is a Python dictionary. Following the example introduced in *Column and Data Types*, we begin with a custom type that marshals Python dictionaries into JSON strings before being persisted:

```
from sqlalchemy.types import TypeDecorator, VARCHAR
import json

class JSONEncodedDict(TypeDecorator):
    "Represents an immutable structure as a json-encoded string."

    impl = VARCHAR

    def process_bind_param(self, value, dialect):
        if value is not None:
            value = json.dumps(value)
        return value

    def process_result_value(self, value, dialect):
        if value is not None:
            value = json.loads(value)
        return value
```

The usage of `json` is only for the purposes of example. The *sqlalchemy.ext.mutable* extension can be used with any type whose target Python type may be mutable, including *PickleType*, *postgresql.ARRAY*, etc.

When using the *sqlalchemy.ext.mutable* extension, the value itself tracks all parents which reference it. Below, we illustrate the a simple version of the *MutableDict* dictionary object, which applies the *Mutable* mixin to a plain Python dictionary:

```
from sqlalchemy.ext.mutable import Mutable

class MutableDict(Mutable, dict):
    @classmethod
    def coerce(cls, key, value):
        "Convert plain dictionaries to MutableDict."

        if not isinstance(value, MutableDict):
            if isinstance(value, dict):
                return MutableDict(value)

            # this call will raise ValueError
            return Mutable.coerce(key, value)
        else:
            return value

    def __setitem__(self, key, value):
        "Detect dictionary set events and emit change events."

        dict.__setitem__(self, key, value)
        self.changed()

    def __delitem__(self, key):
        "Detect dictionary del events and emit change events."

        dict.__delitem__(self, key)
        self.changed()
```

The above dictionary class takes the approach of subclassing the Python built-in `dict` to produce a dict subclass which routes all mutation events through `__setitem__`. There are variants on this approach, such as subclassing `UserDict.UserDict` or `collections.MutableMapping`; the part that's important to this example is that the *Mutable.changed()* method is called whenever an in-place change to the datastructure takes place.

We also redefine the *Mutable.coerce()* method which will be used to convert any values that are not instances of `MutableDict`, such as the plain dictionaries returned by the `json` module, into the appropriate type. Defining this method is optional; we could just as well created our `JSONEncodedDict` such that it always returns an instance of `MutableDict`, and additionally ensured that all calling code uses `MutableDict` explicitly. When *Mutable.coerce()* is not overridden, any values applied to a parent object which are not instances of the mutable type will raise a `ValueError`.

Our new `MutableDict` type offers a class method *as_mutable()* which we can use within column metadata to associate with types. This method grabs the given type object or class and associates a listener that will detect all future mappings of this type, applying event listening instrumentation to the mapped attribute. Such as, with classical table metadata:

```
from sqlalchemy import Table, Column, Integer

my_data = Table('my_data', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', MutableDict.as_mutable(JSONEncodedDict))
)
```

Above, *as_mutable()* returns an instance of `JSONEncodedDict` (if the type object was not an instance already), which will intercept any attributes which are mapped against this type. Below we establish a simple mapping against the `my_data` table:

```
from sqlalchemy import mapper

class MyDataClass(object):
    pass

# associates mutation listeners with MyDataClass.data
mapper(MyDataClass, my_data)
```

The `MyDataClass.data` member will now be notified of in place changes to its value.

There's no difference in usage when using declarative:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class MyDataClass(Base):
    __tablename__ = 'my_data'
    id = Column(Integer, primary_key=True)
    data = Column(MutableDict.as_mutable(JSONEncodedDict))
```

Any in-place changes to the `MyDataClass.data` member will flag the attribute as "dirty" on the parent object:

```
>>> from sqlalchemy.orm import Session

>>> sess = Session()
>>> m1 = MyDataClass(data={'value1':'foo'})
>>> sess.add(m1)
>>> sess.commit()

>>> m1.data['value1'] = 'bar'
>>> assert m1 in sess.dirty
True
```

The `MutableDict` can be associated with all future instances of `JSONEncodedDict` in one step, using *associate_with()*. This is similar to *as_mutable()* except it will intercept all occurrences of `MutableDict` in all mappings unconditionally, without the need to declare it individually:

```
MutableDict.associate_with(JSONEncodedDict)

class MyDataClass(Base):
    __tablename__ = 'my_data'
    id = Column(Integer, primary_key=True)
    data = Column(JSONEncodedDict)
```

Supporting Pickling

The key to the *sqlalchemy.ext.mutable* extension relies upon the placement of a `weakref.WeakKeyDictionary` upon the value object, which stores a mapping of parent mapped objects

keyed to the attribute name under which they are associated with this value. `WeakKeyDictionary` objects are not picklable, due to the fact that they contain weakrefs and function callbacks. In our case, this is a good thing, since if this dictionary were picklable, it could lead to an excessively large pickle size for our value objects that are pickled by themselves outside of the context of the parent. The developer responsibility here is only to provide a `__getstate__` method that excludes the *_parents()* collection from the pickle stream:

```
class MyMutableType(Mutable):
    def __getstate__(self):
        d = self.__dict__.copy()
        d.pop('_parents', None)
        return d
```

With our dictionary example, we need to return the contents of the dict itself (and also restore them on __setstate__):

```
class MutableDict(Mutable, dict):
    # ....

    def __getstate__(self):
        return dict(self)

    def __setstate__(self, state):
        self.update(state)
```

In the case that our mutable value object is pickled as it is attached to one or more parent objects that are also part of the pickle, the *Mutable* mixin will re-establish the *Mutable._parents* collection on each value object as the owning parents themselves are unpickled.

Establishing Mutability on Composites

Composites are a special ORM feature which allow a single scalar attribute to be assigned an object value which represents information "composed" from one or more columns from the underlying mapped table. The usual example is that of a geometric "point", and is introduced in *Composite Column Types*.

Changed in version 0.7: The internals of *orm.composite()* have been greatly simplified and in-place mutation detection is no longer enabled by default; instead, the user-defined value must detect changes on its own and propagate them to all owning parents. The *sqlalchemy.ext.mutable* extension provides the helper class *MutableComposite*, which is a slight variant on the *Mutable* class.

As is the case with *Mutable*, the user-defined composite class subclasses *MutableComposite* as a mixin, and detects and delivers change events to its parents via the *MutableComposite.changed()* method. In the case of a composite class, the detection is usually via the usage of Python descriptors (i.e. `@property`), or alternatively via the special Python method `__setattr__()`. Below we expand upon the `Point` class introduced in *Composite Column Types* to subclass *MutableComposite* and to also route attribute set events via `__setattr__` to the *MutableComposite.changed()* method:

```
from sqlalchemy.ext.mutable import MutableComposite

class Point(MutableComposite):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __setattr__(self, key, value):
        "Intercept set events"
```

```
        # set the attribute
        object.__setattr__(self, key, value)

        # alert all parents to the change
        self.changed()

    def __composite_values__(self):
        return self.x, self.y

    def __eq__(self, other):
        return isinstance(other, Point) and \
            other.x == self.x and \
            other.y == self.y

    def __ne__(self, other):
        return not self.__eq__(other)
```

The *MutableComposite* class uses a Python metaclass to automatically establish listeners for any usage of *orm.composite()* that specifies our `Point` type. Below, when `Point` is mapped to the `Vertex` class, listeners are established which will route change events from `Point` objects to each of the `Vertex.start` and `Vertex.end` attributes:

```
from sqlalchemy.orm import composite, mapper
from sqlalchemy import Table, Column

vertices = Table('vertices', metadata,
    Column('id', Integer, primary_key=True),
    Column('x1', Integer),
    Column('y1', Integer),
    Column('x2', Integer),
    Column('y2', Integer),
    )

class Vertex(object):
    pass

mapper(Vertex, vertices, properties={
    'start': composite(Point, vertices.c.x1, vertices.c.y1),
    'end': composite(Point, vertices.c.x2, vertices.c.y2)
})
```

Any in-place changes to the `Vertex.start` or `Vertex.end` members will flag the attribute as "dirty" on the parent object:

```
>>> from sqlalchemy.orm import Session

>>> sess = Session()
>>> v1 = Vertex(start=Point(3, 4), end=Point(12, 15))
>>> sess.add(v1)
>>> sess.commit()

>>> v1.end.x = 8
>>> assert v1 in sess.dirty
True
```

Coercing Mutable Composites

The *MutableBase.coerce()* method is also supported on composite types. In the case of *MutableComposite*, the *MutableBase.coerce()* method is only called for attribute set operations, not load operations. Overriding the *MutableBase.coerce()* method is essentially equivalent to using a *validates()* validation routine for all attributes which make use of the custom composite type:

```
class Point(MutableComposite):
    # other Point methods
    # ...

    def coerce(cls, key, value):
        if isinstance(value, tuple):
            value = Point(*value)
        elif not isinstance(value, Point):
            raise ValueError("tuple or Point expected")
        return value
```

New in version 0.7.10,0.8.0b2: Support for the *MutableBase.coerce()* method in conjunction with objects of type *MutableComposite*.

Supporting Pickling

As is the case with *Mutable*, the *MutableComposite* helper class uses a `weakref.WeakKeyDictionary` available via the *MutableBase._parents()* attribute which isn't picklable. If we need to pickle instances of `Point` or its owning class `Vertex`, we at least need to define a `__getstate__` that doesn't include the `_parents` dictionary. Below we define both a `__getstate__` and a `__setstate__` that package up the minimal form of our `Point` class:

```
class Point(MutableComposite):
    # ...

    def __getstate__(self):
        return self.x, self.y

    def __setstate__(self, state):
        self.x, self.y = state
```

As with *Mutable*, the *MutableComposite* augments the pickling process of the parent's object-relational state so that the *MutableBase._parents()* collection is restored to all `Point` objects.

API Reference

class `sqlalchemy.ext.mutable.`**`MutableBase`**

Common base class to *Mutable* and *MutableComposite*.

`_parents`

Dictionary of parent object->attribute name on the parent.

This attribute is a so-called "memoized" property. It initializes itself with a new `weakref.WeakKeyDictionary` the first time it is accessed, returning the same object upon subsequent access.

classmethod coerce(*key*, *value*)

Given a value, coerce it into the target type.

Can be overridden by custom subclasses to coerce incoming data into a particular type.

By default, raises `ValueError`.

This method is called in different scenarios depending on if the parent class is of type `Mutable` or of type `MutableComposite`. In the case of the former, it is called for both attribute-set operations as well as during ORM loading operations. For the latter, it is only called during attribute-set operations; the mechanics of the `composite()` construct handle coercion during load operations.

Parameters

- **key** – string name of the ORM-mapped attribute being set.
- **value** – the incoming value.

Returns the method should return the coerced value, or raise `ValueError` if the coercion cannot be completed.

class `sqlalchemy.ext.mutable.`**Mutable**

Bases: `sqlalchemy.ext.mutable.MutableBase`

Mixin that defines transparent propagation of change events to a parent object.

See the example in *Establishing Mutability on Scalar Column Values* for usage information.

__init__

inherited from the `__init__` *attribute of* `object`

x.__init__(...) initializes x; see help(type(x)) for signature

_get_listen_keys(*attribute*)

inherited from the `_get_listen_keys()` *method of* `MutableBase`

Given a descriptor attribute, return a `set()` of the attribute keys which indicate a change in the state of this attribute.

This is normally just `set([attribute.key])`, but can be overridden to provide for additional keys. E.g. a `MutableComposite` augments this set with the attribute keys associated with the columns that comprise the composite value.

This collection is consulted in the case of intercepting the `InstanceEvents.refresh()` and `InstanceEvents.refresh_flush()` events, which pass along a list of attribute names that have been refreshed; the list is compared against this set to determine if action needs to be taken.

New in version 1.0.5.

_listen_on_attribute(*attribute*, *coerce*, *parent_cls*)

inherited from the `_listen_on_attribute()` *method of* `MutableBase`

Establish this type as a mutation listener for the given mapped descriptor.

_parents

inherited from the `_parents` *attribute of* `MutableBase`

Dictionary of parent object->attribute name on the parent.

This attribute is a so-called "memoized" property. It initializes itself with a new `weakref.WeakKeyDictionary` the first time it is accessed, returning the same object upon subsequent access.

classmethod as_mutable(*sqltype*)

Associate a SQL type with this mutable Python type.

This establishes listeners that will detect ORM mappings against the given type, adding mutation event trackers to those mappings.

The type is returned, unconditionally as an instance, so that *as_mutable()* can be used inline:

```
Table('mytable', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', MyMutableType.as_mutable(PickleType))
)
```

Note that the returned type is always an instance, even if a class is given, and that only columns which are declared specifically with that type instance receive additional instrumentation.

To associate a particular mutable type with all occurrences of a particular type, use the *Mutable.associate_with()* classmethod of the particular *Mutable* subclass to establish a global association.

> **Warning:** The listeners established by this method are *global* to all mappers, and are *not* garbage collected. Only use *as_mutable()* for types that are permanent to an application, not with ad-hoc types else this will cause unbounded growth in memory usage.

classmethod associate_with(*sqltype*)

Associate this wrapper with all future mapped columns of the given type.

This is a convenience method that calls `associate_with_attribute` automatically.

> **Warning:** The listeners established by this method are *global* to all mappers, and are *not* garbage collected. Only use *associate_with()* for types that are permanent to an application, not with ad-hoc types else this will cause unbounded growth in memory usage.

classmethod associate_with_attribute(*attribute*)

Establish this type as a mutation listener for the given mapped descriptor.

changed()

Subclasses should call this method whenever change events occur.

coerce(*key*, *value*)

inherited from the *coerce()* *method of* *MutableBase*

Given a value, coerce it into the target type.

Can be overridden by custom subclasses to coerce incoming data into a particular type.

By default, raises `ValueError`.

This method is called in different scenarios depending on if the parent class is of type *Mutable* or of type *MutableComposite*. In the case of the former, it is called for both attribute-set operations as well as during ORM loading operations. For the latter, it is only called during attribute-set operations; the mechanics of the *composite()* construct handle coercion during load operations.

Parameters

- **key** – string name of the ORM-mapped attribute being set.
- **value** – the incoming value.

Returns the method should return the coerced value, or raise `ValueError` if the coercion cannot be completed.

class sqlalchemy.ext.mutable.**MutableComposite**

Bases: *sqlalchemy.ext.mutable.MutableBase*

Mixin that defines transparent propagation of change events on a SQLAlchemy "composite" object to its owning parent or parents.

See the example in *Establishing Mutability on Composites* for usage information.

changed()
Subclasses should call this method whenever change events occur.

class sqlalchemy.ext.mutable.**MutableDict**
Bases: *sqlalchemy.ext.mutable.Mutable*, __builtin__.dict

A dictionary type that implements *Mutable*.

New in version 0.8.

clear()

classmethod coerce(*key*, *value*)
Convert plain dictionary to instance of this class.

pop(**arg*)

popitem()

setdefault(*key*, *value*)

update(**a*, ***kw*)

2.7.6 Ordering List

A custom list that manages index/position information for contained elements.

author Jason Kirtland

`orderinglist` is a helper for mutable ordered relationships. It will intercept list operations performed on a *relationship()*-managed collection and automatically synchronize changes in list position onto a target scalar attribute.

Example: A `slide` table, where each row refers to zero or more entries in a related `bullet` table. The bullets within a slide are displayed in order based on the value of the `position` column in the `bullet` table. As entries are reordered in memory, the value of the `position` attribute should be updated to reflect the new sort order:

```
Base = declarative_base()

class Slide(Base):
    __tablename__ = 'slide'

    id = Column(Integer, primary_key=True)
    name = Column(String)

    bullets = relationship("Bullet", order_by="Bullet.position")

class Bullet(Base):
    __tablename__ = 'bullet'
    id = Column(Integer, primary_key=True)
    slide_id = Column(Integer, ForeignKey('slide.id'))
    position = Column(Integer)
    text = Column(String)
```

The standard relationship mapping will produce a list-like attribute on each `Slide` containing all related `Bullet` objects, but coping with changes in ordering is not handled automatically. When appending a `Bullet` into

`Slide.bullets`, the `Bullet.position` attribute will remain unset until manually assigned. When the `Bullet` is inserted into the middle of the list, the following `Bullet` objects will also need to be renumbered.

The *OrderingList* object automates this task, managing the `position` attribute on all `Bullet` objects in the collection. It is constructed using the *ordering_list()* factory:

```
from sqlalchemy.ext.orderinglist import ordering_list

Base = declarative_base()

class Slide(Base):
    __tablename__ = 'slide'

    id = Column(Integer, primary_key=True)
    name = Column(String)

    bullets = relationship("Bullet", order_by="Bullet.position",
                           collection_class=ordering_list('position'))

class Bullet(Base):
    __tablename__ = 'bullet'
    id = Column(Integer, primary_key=True)
    slide_id = Column(Integer, ForeignKey('slide.id'))
    position = Column(Integer)
    text = Column(String)
```

With the above mapping the `Bullet.position` attribute is managed:

```
s = Slide()
s.bullets.append(Bullet())
s.bullets.append(Bullet())
s.bullets[1].position
>>> 1
s.bullets.insert(1, Bullet())
s.bullets[2].position
>>> 2
```

The *OrderingList* construct only works with **changes** to a collection, and not the initial load from the database, and requires that the list be sorted when loaded. Therefore, be sure to specify `order_by` on the *relationship()* against the target ordering attribute, so that the ordering is correct when first loaded.

Warning: *OrderingList* only provides limited functionality when a primary key column or unique column is the target of the sort. Operations that are unsupported or are problematic include:

- two entries must trade values. This is not supported directly in the case of a primary key or unique constraint because it means at least one row would need to be temporarily removed first, or changed to a third, neutral value while the switch occurs.
- an entry must be deleted in order to make room for a new entry. SQLAlchemy's unit of work performs all INSERTs before DELETEs within a single flush. In the case of a primary key, it will trade an INSERT/DELETE of the same primary key for an UPDATE statement in order to lessen the impact of this limitation, however this does not take place for a UNIQUE column. A future feature will allow the "DELETE before INSERT" behavior to be possible, allevating this limitation, though this feature will require explicit configuration at the mapper level for sets of columns that are to be handled in this way.

ordering_list() takes the name of the related object's ordering attribute as an argument. By default, the zero-based integer index of the object's position in the *ordering_list()* is synchronized with the ordering at-

tribute: index 0 will get position 0, index 1 position 1, etc. To start numbering at 1 or some other integer, provide `count_from=1`.

API Reference

`sqlalchemy.ext.orderinglist.`**`ordering_list`**(*attr*, *count_from=None*, ***kw*)

Prepares an *`OrderingList`* factory for use in mapper definitions.

Returns an object suitable for use as an argument to a Mapper relationship's `collection_class` option. e.g.:

```
from sqlalchemy.ext.orderinglist import ordering_list

class Slide(Base):
    __tablename__ = 'slide'

    id = Column(Integer, primary_key=True)
    name = Column(String)

    bullets = relationship("Bullet", order_by="Bullet.position",
                           collection_class=ordering_list('position'))
```

Parameters

- **`attr`** – Name of the mapped attribute to use for storage and retrieval of ordering information
- **`count_from`** – Set up an integer-based ordering, starting at `count_from`. For example, `ordering_list('pos', count_from=1)` would create a 1-based list in SQL, storing the value in the 'pos' column. Ignored if `ordering_func` is supplied.

Additional arguments are passed to the *`OrderingList`* constructor.

`sqlalchemy.ext.orderinglist.`**`count_from_0`**(*index*, *collection*)

Numbering function: consecutive integers starting at 0.

`sqlalchemy.ext.orderinglist.`**`count_from_1`**(*index*, *collection*)

Numbering function: consecutive integers starting at 1.

`sqlalchemy.ext.orderinglist.`**`count_from_n_factory`**(*start*)

Numbering function: consecutive integers starting at arbitrary start.

class `sqlalchemy.ext.orderinglist.`**`OrderingList`**(*ordering_attr=None*, *ordering_func=None*, *reorder_on_append=False*)

Bases: `__builtin__.list`

A custom list that manages position information for its children.

The *`OrderingList`* object is normally set up using the *`ordering_list()`* factory function, used in conjunction with the *`relationship()`* function.

`__init__`(*ordering_attr=None*, *ordering_func=None*, *reorder_on_append=False*)

A custom list that manages position information for its children.

`OrderingList` is a `collection_class` list implementation that syncs position in a Python list with a position attribute on the mapped objects.

This implementation relies on the list starting in the proper order, so be **sure** to put an `order_by` on your relationship.

Parameters

- **ordering_attr** – Name of the attribute that stores the object's order in the relationship.
- **ordering_func** – Optional. A function that maps the position in the Python list to a value to store in the `ordering_attr`. Values returned are usually (but need not be!) integers.

  An `ordering_func` is called with two positional parameters: the index of the element in the list, and the list itself.

  If omitted, Python list indexes are used for the attribute values. Two basic prebuilt numbering functions are provided in this module: `count_from_0` and `count_from_1`. For more exotic examples like stepped numbering, alphabetical and Fibonacci numbering, see the unit tests.
- **reorder_on_append** – Default False. When appending an object with an existing (non-None) ordering value, that value will be left untouched unless `reorder_on_append` is true. This is an optimization to avoid a variety of dangerous unexpected database writes.

  SQLAlchemy will add instances to the list via append() when your object loads. If for some reason the result set from the database skips a step in the ordering (say, row '1' is missing but you get '2', '3', and '4'), reorder_on_append=True would immediately renumber the items to '1', '2', '3'. If you have multiple sessions making changes, any of whom happen to load this collection even in passing, all of the sessions would try to "clean up" the numbering in their commits, possibly causing all but one to fail with a concurrent modification error.

  Recommend leaving this with the default of False, and just call `reorder()` if you're doing `append()` operations with previously ordered instances or when doing some housekeeping after manual sql operations.

append(*entity*)
L.append(object) – append object to end

insert(*index*, *entity*)
L.insert(index, object) – insert object before index

pop([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

remove(*entity*)
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

reorder()
Synchronize ordering for the entire collection.

Sweeps through the list and ensures that each object has accurate ordering information set.

2.7.7 Horizontal Sharding

Horizontal sharding support.

Defines a rudimental 'horizontal sharding' system which allows a Session to distribute queries and persistence operations across multiple databases.

For a usage example, see the *Horizontal Sharding* example included in the source distribution.

API Documentation

class sqlalchemy.ext.horizontal_shard.**ShardedSession**(*shard_chooser*, *id_chooser*, *query_chooser*, *shards=None*, *query_cls=<class 'sqlalchemy.ext.horizontal_shard.ShardedQuery'>*, ***kwargs*)

Bases: *sqlalchemy.orm.session.Session*

__init__(*shard_chooser*, *id_chooser*, *query_chooser*, *shards=None*, *query_cls=<class 'sqlalchemy.ext.horizontal_shard.ShardedQuery'>*, ***kwargs*)

Construct a ShardedSession.

Parameters

- **shard_chooser** – A callable which, passed a Mapper, a mapped instance, and possibly a SQL clause, returns a shard ID. This id may be based off of the attributes present within the object, or on some round-robin scheme. If the scheme is based on a selection, it should set whatever state on the instance to mark it in the future as participating in that shard.
- **id_chooser** – A callable, passed a query and a tuple of identity values, which should return a list of shard ids where the ID might reside. The databases will be queried in the order of this listing.
- **query_chooser** – For a given Query, returns the list of shard_ids where the query should be issued. Results from all shards returned will be combined together into a single listing.
- **shards** – A dictionary of string shard names to *Engine* objects.

class sqlalchemy.ext.horizontal_shard.**ShardedQuery**(**args*, ***kwargs*)

Bases: *sqlalchemy.orm.query.Query*

set_shard(*shard_id*)

return a new query, limited to a single shard ID.

all subsequent operations with the returned query will be against the single shard regardless of other state.

2.7.8 Hybrid Attributes

Define attributes on ORM-mapped classes that have "hybrid" behavior.

"hybrid" means the attribute has distinct behaviors defined at the class level and at the instance level.

The *hybrid* extension provides a special form of method decorator, is around 50 lines of code and has almost no dependencies on the rest of SQLAlchemy. It can, in theory, work with any descriptor-based expression system.

Consider a mapping `Interval`, representing integer `start` and `end` values. We can define higher level functions on mapped classes that produce SQL expressions at the class level, and Python expression evaluation at the instance level. Below, each function decorated with *hybrid_method* or *hybrid_property* may receive `self` as an instance of the class, or as the class itself:

```
from sqlalchemy import Column, Integer
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import Session, aliased
from sqlalchemy.ext.hybrid import hybrid_property, hybrid_method

Base = declarative_base()
```

```
class Interval(Base):
    __tablename__ = 'interval'

    id = Column(Integer, primary_key=True)
    start = Column(Integer, nullable=False)
    end = Column(Integer, nullable=False)

    def __init__(self, start, end):
        self.start = start
        self.end = end

    @hybrid_property
    def length(self):
        return self.end - self.start

    @hybrid_method
    def contains(self, point):
        return (self.start <= point) & (point <= self.end)

    @hybrid_method
    def intersects(self, other):
        return self.contains(other.start) | self.contains(other.end)
```

Above, the `length` property returns the difference between the `end` and `start` attributes. With an instance of `Interval`, this subtraction occurs in Python, using normal Python descriptor mechanics:

```
>>> i1 = Interval(5, 10)
>>> i1.length
5
```

When dealing with the `Interval` class itself, the *hybrid_property* descriptor evaluates the function body given the `Interval` class as the argument, which when evaluated with SQLAlchemy expression mechanics returns a new SQL expression:

```
>>> print Interval.length
interval."end" - interval.start

>>> print Session().query(Interval).filter(Interval.length > 10)
SELECT interval.id AS interval_id, interval.start AS interval_start,
interval."end" AS interval_end
FROM interval
WHERE interval."end" - interval.start > :param_1
```

ORM methods such as *filter_by()* generally use `getattr()` to locate attributes, so can also be used with hybrid attributes:

```
>>> print Session().query(Interval).filter_by(length=5)
SELECT interval.id AS interval_id, interval.start AS interval_start,
interval."end" AS interval_end
FROM interval
WHERE interval."end" - interval.start = :param_1
```

The `Interval` class example also illustrates two methods, `contains()` and `intersects()`, decorated with *hybrid_method*. This decorator applies the same idea to methods that *hybrid_property* applies to attributes. The methods return boolean values, and take advantage of the Python `|` and `&` bitwise operators to produce equivalent instance-level and SQL expression-level boolean behavior:

```
>>> i1.contains(6)
True
>>> i1.contains(15)
False
>>> i1.intersects(Interval(7, 18))
True
>>> i1.intersects(Interval(25, 29))
False

>>> print Session().query(Interval).filter(Interval.contains(15))
SELECT interval.id AS interval_id, interval.start AS interval_start,
interval."end" AS interval_end
FROM interval
WHERE interval.start <= :start_1 AND interval."end" > :end_1

>>> ia = aliased(Interval)
>>> print Session().query(Interval, ia).filter(Interval.intersects(ia))
SELECT interval.id AS interval_id, interval.start AS interval_start,
interval."end" AS interval_end, interval_1.id AS interval_1_id,
interval_1.start AS interval_1_start, interval_1."end" AS interval_1_end
FROM interval, interval AS interval_1
WHERE interval.start <= interval_1.start
    AND interval."end" > interval_1.start
    OR interval.start <= interval_1."end"
    AND interval."end" > interval_1."end"
```

Defining Expression Behavior Distinct from Attribute Behavior

Our usage of the `&` and `|` bitwise operators above was fortunate, considering our functions operated on two boolean values to return a new one. In many cases, the construction of an in-Python function and a SQLAlchemy SQL expression have enough differences that two separate Python expressions should be defined. The *hybrid* decorators define the *hybrid_property.expression()* modifier for this purpose. As an example we'll define the radius of the interval, which requires the usage of the absolute value function:

```
from sqlalchemy import func

class Interval(object):
    # ...

    @hybrid_property
    def radius(self):
        return abs(self.length) / 2

    @radius.expression
    def radius(cls):
        return func.abs(cls.length) / 2
```

Above the Python function `abs()` is used for instance-level operations, the SQL function `ABS()` is used via the *func* object for class-level expressions:

```
>>> i1.radius
2

>>> print Session().query(Interval).filter(Interval.radius > 5)
```

```
SELECT interval.id AS interval_id, interval.start AS interval_start,
    interval."end" AS interval_end
FROM interval
WHERE abs(interval."end" - interval.start) / :abs_1 > :param_1
```

Defining Setters

Hybrid properties can also define setter methods. If we wanted `length` above, when set, to modify the endpoint value:

```
class Interval(object):
    # ...

    @hybrid_property
    def length(self):
        return self.end - self.start

    @length.setter
    def length(self, value):
        self.end = self.start + value
```

The `length(self, value)` method is now called upon set:

```
>>> i1 = Interval(5, 10)
>>> i1.length
5
>>> i1.length = 12
>>> i1.end
17
```

Working with Relationships

There's no essential difference when creating hybrids that work with related objects as opposed to column-based data. The need for distinct expressions tends to be greater. Two variants of we'll illustrate are the "join-dependent" hybrid, and the "correlated subquery" hybrid.

Join-Dependent Relationship Hybrid

Consider the following declarative mapping which relates a `User` to a `SavingsAccount`:

```
from sqlalchemy import Column, Integer, ForeignKey, Numeric, String
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.ext.hybrid import hybrid_property

Base = declarative_base()

class SavingsAccount(Base):
    __tablename__ = 'account'
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('user.id'), nullable=False)
    balance = Column(Numeric(15, 5))
```

```
class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(100), nullable=False)

    accounts = relationship("SavingsAccount", backref="owner")

    @hybrid_property
    def balance(self):
        if self.accounts:
            return self.accounts[0].balance
        else:
            return None

    @balance.setter
    def balance(self, value):
        if not self.accounts:
            account = Account(owner=self)
        else:
            account = self.accounts[0]
        account.balance = value

    @balance.expression
    def balance(cls):
        return SavingsAccount.balance
```

The above hybrid property `balance` works with the first `SavingsAccount` entry in the list of accounts for this user. The in-Python getter/setter methods can treat `accounts` as a Python list available on `self`.

However, at the expression level, it's expected that the `User` class will be used in an appropriate context such that an appropriate join to `SavingsAccount` will be present:

```
>>> print Session().query(User, User.balance).\
...     join(User.accounts).filter(User.balance > 5000)
SELECT "user".id AS user_id, "user".name AS user_name,
account.balance AS account_balance
FROM "user" JOIN account ON "user".id = account.user_id
WHERE account.balance > :balance_1
```

Note however, that while the instance level accessors need to worry about whether `self.accounts` is even present, this issue expresses itself differently at the SQL expression level, where we basically would use an outer join:

```
>>> from sqlalchemy import or_
>>> print (Session().query(User, User.balance).outerjoin(User.accounts).
...         filter(or_(User.balance < 5000, User.balance == None)))
SELECT "user".id AS user_id, "user".name AS user_name,
account.balance AS account_balance
FROM "user" LEFT OUTER JOIN account ON "user".id = account.user_id
WHERE account.balance <  :balance_1 OR account.balance IS NULL
```

Correlated Subquery Relationship Hybrid

We can, of course, forego being dependent on the enclosing query's usage of joins in favor of the correlated subquery, which can portably be packed into a single column expression. A correlated subquery is more portable, but often performs more poorly at the SQL level. Using the same technique illustrated at *Using column_property*, we can adjust

our `SavingsAccount` example to aggregate the balances for *all* accounts, and use a correlated subquery for the column expression:

```
from sqlalchemy import Column, Integer, ForeignKey, Numeric, String
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy import select, func

Base = declarative_base()

class SavingsAccount(Base):
    __tablename__ = 'account'
    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('user.id'), nullable=False)
    balance = Column(Numeric(15, 5))

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(100), nullable=False)

    accounts = relationship("SavingsAccount", backref="owner")

    @hybrid_property
    def balance(self):
        return sum(acc.balance for acc in self.accounts)

    @balance.expression
    def balance(cls):
        return select([func.sum(SavingsAccount.balance)]).\
                where(SavingsAccount.user_id==cls.id).\
                label('total_balance')
```

The above recipe will give us the `balance` column which renders a correlated SELECT:

```
>>> print s.query(User).filter(User.balance > 400)
SELECT "user".id AS user_id, "user".name AS user_name
FROM "user"
WHERE (SELECT sum(account.balance) AS sum_1
FROM account
WHERE account.user_id = "user".id) > :param_1
```

Building Custom Comparators

The hybrid property also includes a helper that allows construction of custom comparators. A comparator object allows one to customize the behavior of each SQLAlchemy expression operator individually. They are useful when creating custom types that have some highly idiosyncratic behavior on the SQL side.

The example class below allows case-insensitive comparisons on the attribute named `word_insensitive`:

```
from sqlalchemy.ext.hybrid import Comparator, hybrid_property
from sqlalchemy import func, Column, Integer, String
from sqlalchemy.orm import Session
from sqlalchemy.ext.declarative import declarative_base
```

```
Base = declarative_base()

class CaseInsensitiveComparator(Comparator):
    def __eq__(self, other):
        return func.lower(self.__clause_element__()) == func.lower(other)

class SearchWord(Base):
    __tablename__ = 'searchword'
    id = Column(Integer, primary_key=True)
    word = Column(String(255), nullable=False)

    @hybrid_property
    def word_insensitive(self):
        return self.word.lower()

    @word_insensitive.comparator
    def word_insensitive(cls):
        return CaseInsensitiveComparator(cls.word)
```

Above, SQL expressions against `word_insensitive` will apply the `LOWER()` SQL function to both sides:

```
>>> print Session().query(SearchWord).filter_by(word_insensitive="Trucks")
SELECT searchword.id AS searchword_id, searchword.word AS searchword_word
FROM searchword
WHERE lower(searchword.word) = lower(:lower_1)
```

The `CaseInsensitiveComparator` above implements part of the *ColumnOperators* interface. A "coercion" operation like lowercasing can be applied to all comparison operations (i.e. `eq`, `lt`, `gt`, etc.) using *Operators.operate()*:

```
class CaseInsensitiveComparator(Comparator):
    def operate(self, op, other):
        return op(func.lower(self.__clause_element__()), func.lower(other))
```

Hybrid Value Objects

Note in our previous example, if we were to compare the `word_insensitive` attribute of a `SearchWord` instance to a plain Python string, the plain Python string would not be coerced to lower case - the `CaseInsensitiveComparator` we built, being returned by `@word_insensitive.comparator`, only applies to the SQL side.

A more comprehensive form of the custom comparator is to construct a *Hybrid Value Object*. This technique applies the target value or expression to a value object which is then returned by the accessor in all cases. The value object allows control of all operations upon the value as well as how compared values are treated, both on the SQL expression side as well as the Python value side. Replacing the previous `CaseInsensitiveComparator` class with a new `CaseInsensitiveWord` class:

```
class CaseInsensitiveWord(Comparator):
    "Hybrid value representing a lower case representation of a word."

    def __init__(self, word):
        if isinstance(word, basestring):
            self.word = word.lower()
        elif isinstance(word, CaseInsensitiveWord):
```

```
            self.word = word.word
        else:
            self.word = func.lower(word)

    def operate(self, op, other):
        if not isinstance(other, CaseInsensitiveWord):
            other = CaseInsensitiveWord(other)
        return op(self.word, other.word)

    def __clause_element__(self):
        return self.word

    def __str__(self):
        return self.word

    key = 'word'
    "Label to apply to Query tuple results"
```

Above, the `CaseInsensitiveWord` object represents `self.word`, which may be a SQL function, or may be a Python native. By overriding `operate()` and `__clause_element__()` to work in terms of `self.word`, all comparison operations will work against the "converted" form of `word`, whether it be SQL side or Python side. Our `SearchWord` class can now deliver the `CaseInsensitiveWord` object unconditionally from a single hybrid call:

```
class SearchWord(Base):
    __tablename__ = 'searchword'
    id = Column(Integer, primary_key=True)
    word = Column(String(255), nullable=False)

    @hybrid_property
    def word_insensitive(self):
        return CaseInsensitiveWord(self.word)
```

The `word_insensitive` attribute now has case-insensitive comparison behavior universally, including SQL expression vs. Python expression (note the Python value is converted to lower case on the Python side here):

```
>>> print Session().query(SearchWord).filter_by(word_insensitive="Trucks")
SELECT searchword.id AS searchword_id, searchword.word AS searchword_word
FROM searchword
WHERE lower(searchword.word) = :lower_1
```

SQL expression versus SQL expression:

```
>>> sw1 = aliased(SearchWord)
>>> sw2 = aliased(SearchWord)
>>> print Session().query(
...                     sw1.word_insensitive,
...                     sw2.word_insensitive).\
...                         filter(
...                             sw1.word_insensitive > sw2.word_insensitive
...                         )
SELECT lower(searchword_1.word) AS lower_1,
lower(searchword_2.word) AS lower_2
FROM searchword AS searchword_1, searchword AS searchword_2
WHERE lower(searchword_1.word) > lower(searchword_2.word)
```

Python only expression:

```
>>> ws1 = SearchWord(word="SomeWord")
>>> ws1.word_insensitive == "sOmEwOrD"
True
>>> ws1.word_insensitive == "XOmEwOrX"
False
>>> print ws1.word_insensitive
someword
```

The Hybrid Value pattern is very useful for any kind of value that may have multiple representations, such as timestamps, time deltas, units of measurement, currencies and encrypted passwords.

See also:

Hybrids and Value Agnostic Types - on the techspot.zzzeek.org blog

Value Agnostic Types, Part II - on the techspot.zzzeek.org blog

Building Transformers

A *transformer* is an object which can receive a `Query` object and return a new one. The `Query` object includes a method `with_transformation()` that returns a new `Query` transformed by the given function.

We can combine this with the `Comparator` class to produce one type of recipe which can both set up the FROM clause of a query as well as assign filtering criterion.

Consider a mapped class `Node`, which assembles using adjacency list into a hierarchical tree pattern:

```
from sqlalchemy import Column, Integer, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()

class Node(Base):
    __tablename__ = 'node'
    id =Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('node.id'))
    parent = relationship("Node", remote_side=id)
```

Suppose we wanted to add an accessor `grandparent`. This would return the `parent` of `Node.parent`. When we have an instance of `Node`, this is simple:

```
from sqlalchemy.ext.hybrid import hybrid_property

class Node(Base):
    # ...

    @hybrid_property
    def grandparent(self):
        return self.parent.parent
```

For the expression, things are not so clear. We'd need to construct a `Query` where we `join()` twice along `Node.parent` to get to the `grandparent`. We can instead return a transforming callable that we'll combine with the `Comparator` class to receive any `Query` object, and return a new one that's joined to the `Node.parent` attribute and filtered based on the given criterion:

```
from sqlalchemy.ext.hybrid import Comparator

class GrandparentTransformer(Comparator):
    def operate(self, op, other):
        def transform(q):
            cls = self.__clause_element__()
            parent_alias = aliased(cls)
            return q.join(parent_alias, cls.parent).\
                        filter(op(parent_alias.parent, other))
        return transform

Base = declarative_base()

class Node(Base):
    __tablename__ = 'node'
    id =Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('node.id'))
    parent = relationship("Node", remote_side=id)

    @hybrid_property
    def grandparent(self):
        return self.parent.parent

    @grandparent.comparator
    def grandparent(cls):
        return GrandparentTransformer(cls)
```

The `GrandparentTransformer` overrides the core *Operators.operate()* method at the base of the *Comparator* hierarchy to return a query-transforming callable, which then runs the given comparison operation in a particular context. Such as, in the example above, the `operate` method is called, given the `Operators.eq` callable as well as the right side of the comparison `Node(id=5)`. A function `transform` is then returned which will transform a *Query* first to join to `Node.parent`, then to compare `parent_alias` using `Operators.eq` against the left and right sides, passing into `Query.filter`:

```
>>> from sqlalchemy.orm import Session
>>> session = Session()
>>> session.query(Node).\
...         with_transformation(Node.grandparent==Node(id=5)).\
...         all()
SELECT node.id AS node_id, node.parent_id AS node_parent_id
FROM node JOIN node AS node_1 ON node_1.id = node.parent_id
WHERE :param_1 = node_1.parent_id
```

We can modify the pattern to be more verbose but flexible by separating the "join" step from the "filter" step. The tricky part here is ensuring that successive instances of `GrandparentTransformer` use the same *AliasedClass* object against `Node`. Below we use a simple memoizing approach that associates a `GrandparentTransformer` with each class:

```
class Node(Base):

    # ...

    @grandparent.comparator
    def grandparent(cls):
        # memoize a GrandparentTransformer
        # per class
```

```
        if '_gp' not in cls.__dict__:
            cls._gp = GrandparentTransformer(cls)
        return cls._gp

class GrandparentTransformer(Comparator):

    def __init__(self, cls):
        self.parent_alias = aliased(cls)

    @property
    def join(self):
        def go(q):
            return q.join(self.parent_alias, Node.parent)
        return go

    def operate(self, op, other):
        return op(self.parent_alias.parent, other)
```

```
>>> session.query(Node).\
...             with_transformation(Node.grandparent.join).\
...             filter(Node.grandparent==Node(id=5))
SELECT node.id AS node_id, node.parent_id AS node_parent_id
FROM node JOIN node AS node_1 ON node_1.id = node.parent_id
WHERE :param_1 = node_1.parent_id
```

The "transformer" pattern is an experimental pattern that starts to make usage of some functional programming paradigms. While it's only recommended for advanced and/or patient developers, there's probably a whole lot of amazing things it can be used for.

API Reference

class `sqlalchemy.ext.hybrid.`**`hybrid_method`**(*func*, *expr=None*)

Bases: `sqlalchemy.orm.base.InspectionAttrInfo`

A decorator which allows definition of a Python object method with both instance-level and class-level behavior.

`__init__`(*func*, *expr=None*)

Create a new `hybrid_method`.

Usage is typically via decorator:

```
from sqlalchemy.ext.hybrid import hybrid_method

class SomeClass(object):
    @hybrid_method
    def value(self, x, y):
        return self._value + x + y

    @value.expression
    def value(self, x, y):
        return func.some_function(self._value, x, y)
```

`expression`(*expr*)

Provide a modifying decorator that defines a SQL-expression producing method.

class sqlalchemy.ext.hybrid.**hybrid_property**(*fget*, *fset=None*, *fdel=None*, *expr=None*)

Bases: *sqlalchemy.orm.base.InspectionAttrInfo*

A decorator which allows definition of a Python descriptor with both instance-level and class-level behavior.

__init__(*fget*, *fset=None*, *fdel=None*, *expr=None*)

Create a new *hybrid_property*.

Usage is typically via decorator:

```
from sqlalchemy.ext.hybrid import hybrid_property

class SomeClass(object):
    @hybrid_property
    def value(self):
        return self._value

    @value.setter
    def value(self, value):
        self._value = value
```

comparator(*comparator*)

Provide a modifying decorator that defines a custom comparator producing method.

The return value of the decorated method should be an instance of *Comparator*.

deleter(*fdel*)

Provide a modifying decorator that defines a value-deletion method.

expression(*expr*)

Provide a modifying decorator that defines a SQL-expression producing method.

setter(*fset*)

Provide a modifying decorator that defines a value-setter method.

class sqlalchemy.ext.hybrid.**Comparator**(*expression*)

Bases: *sqlalchemy.orm.interfaces.PropComparator*

A helper class that allows easy construction of custom *PropComparator* classes for usage with hybrids.

sqlalchemy.ext.hybrid.**HYBRID_METHOD = symbol('HYBRID_METHOD')**

sqlalchemy.ext.hybrid.**HYBRID_PROPERTY = symbol('HYBRID_PROPERTY')**

2.7.9 Alternate Class Instrumentation

Extensible class instrumentation.

The *sqlalchemy.ext.instrumentation* package provides for alternate systems of class instrumentation within the ORM. Class instrumentation refers to how the ORM places attributes on the class which maintain data and track changes to that data, as well as event hooks installed on the class.

Note: The extension package is provided for the benefit of integration with other object management packages, which already perform their own instrumentation. It is not intended for general use.

For examples of how the instrumentation extension is used, see the example *Attribute Instrumentation*.

Changed in version 0.8: The *sqlalchemy.orm.instrumentation* was split out so that all functionality having to do with non-standard instrumentation was moved out to *sqlalchemy.ext.instrumentation*. When imported, the module installs itself within *sqlalchemy.orm.instrumentation* so that it

takes effect, including recognition of `__sa_instrumentation_manager__` on mapped classes, as well *instrumentation_finders* being used to determine class instrumentation resolution.

API Reference

`sqlalchemy.ext.instrumentation.`**`INSTRUMENTATION_MANAGER`** = **'__sa_instrumentation_manager__'**

Attribute, elects custom instrumentation when present on a mapped class.

Allows a class to specify a slightly or wildly different technique for tracking changes made to mapped attributes and collections.

Only one instrumentation implementation is allowed in a given object inheritance hierarchy.

The value of this attribute must be a callable and will be passed a class object. The callable must return one of:

- An instance of an InstrumentationManager or subclass
- An object implementing all or some of InstrumentationManager (TODO)
- A dictionary of callables, implementing all or some of the above (TODO)
- An instance of a ClassManager or subclass

This attribute is consulted by SQLAlchemy instrumentation resolution, once the *sqlalchemy.ext.instrumentation* module has been imported. If custom finders are installed in the global instrumentation_finders list, they may or may not choose to honor this attribute.

class `sqlalchemy.orm.instrumentation.`**`InstrumentationFactory`**

Factory for new ClassManager instances.

class `sqlalchemy.ext.instrumentation.`**`InstrumentationManager`**(*class_*)

User-defined class instrumentation extension.

InstrumentationManager can be subclassed in order to change how class instrumentation proceeds. This class exists for the purposes of integration with other object management frameworks which would like to entirely modify the instrumentation methodology of the ORM, and is not intended for regular usage. For interception of class instrumentation events, see *InstrumentationEvents*.

The API for this class should be considered as semi-stable, and may change slightly with new releases.

Changed in version 0.8: *InstrumentationManager* was moved from *sqlalchemy.orm.instrumentation* to *sqlalchemy.ext.instrumentation*.

`dict_getter`(*class_*)

`dispose`(*class_*, *manager*)

`get_instance_dict`(*class_*, *instance*)

`initialize_instance_dict`(*class_*, *instance*)

`install_descriptor`(*class_*, *key*, *inst*)

`install_member`(*class_*, *key*, *implementation*)

`install_state`(*class_*, *instance*, *state*)

`instrument_attribute`(*class_*, *key*, *inst*)

`instrument_collection_class`(*class_*, *key*, *collection_class*)

`manage`(*class_*, *manager*)

`manager_getter`(*class_*)

`post_configure_attribute`(*class_*, *key*, *inst*)

remove_state(*class_*, *instance*)

state_getter(*class_*)

uninstall_descriptor(*class_*, *key*)

uninstall_member(*class_*, *key*)

sqlalchemy.ext.instrumentation.**instrumentation_finders = [<function find_native_user_instrumentation_ho**

An extensible sequence of callables which return instrumentation implementations

When a class is registered, each callable will be passed a class object. If None is returned, the next finder in the sequence is consulted. Otherwise the return must be an instrumentation factory that follows the same guidelines as sqlalchemy.ext.instrumentation.INSTRUMENTATION_MANAGER.

By default, the only finder is find_native_user_instrumentation_hook, which searches for INSTRUMENTATION_MANAGER. If all finders return None, standard ClassManager instrumentation is used.

class sqlalchemy.ext.instrumentation.**ExtendedInstrumentationRegistry**

Bases: *sqlalchemy.orm.instrumentation.InstrumentationFactory*

Extends *InstrumentationFactory* with additional bookkeeping, to accommodate multiple types of class managers.

2.8 ORM Examples

The SQLAlchemy distribution includes a variety of code examples illustrating a select set of patterns, some typical and some not so typical. All are runnable and can be found in the `/examples` directory of the distribution. Descriptions and source code for all can be found here.

Additional SQLAlchemy examples, some user contributed, are available on the wiki at http://www.sqlalchemy.org/trac/wiki/UsageRecipes.

2.8.1 Mapping Recipes

Adjacency List

An example of a dictionary-of-dictionaries structure mapped using an adjacency list model.

E.g.:

```
node = TreeNode('rootnode')
node.append('node1')
node.append('node3')
session.add(node)
session.commit()

dump_tree(node)
```

Listing of files:

- adjacency_list.py

Associations

Examples illustrating the usage of the "association object" pattern, where an intermediary class mediates the relationship between two classes that are associated in a many-to-many pattern.

Listing of files:

- `basic_association.py` - illustrate a many-to-many relationship between an "Order" and a collection of "Item" objects, associating a purchase price with each via an association object called "OrderItem"
- `dict_of_sets_with_default.py` - an advanced association proxy example which illustrates nesting of association proxies to produce multi-level Python collections, in this case a dictionary with string keys and sets of integers as values, which conceal the underlying mapped classes.
- `proxied_association.py` - same example as basic_association, adding in usage of *`sqlalchemy.ext.associationproxy`* to make explicit references to `OrderItem` optional.

Directed Graphs

An example of persistence for a directed graph structure. The graph is stored as a collection of edges, each referencing both a "lower" and an "upper" node in a table of nodes. Basic persistence and querying for lower- and upper- neighbors are illustrated:

```
n2 = Node(2)
n5 = Node(5)
n2.add_neighbor(n5)
print n2.higher_neighbors()
```

Listing of files:

- `directed_graph.py` - a directed graph example.

Dynamic Relations as Dictionaries

Illustrates how to place a dictionary-like facade on top of a "dynamic" relation, so that dictionary operations (assuming simple string keys) can operate upon a large collection without loading the full collection at once.

Listing of files:

- dynamic_dict.py

Generic Associations

Illustrates various methods of associating multiple types of parents with a particular child object.

The examples all use the declarative extension along with declarative mixins. Each one presents the identical use case at the end - two classes, `Customer` and `Supplier`, both subclassing the `HasAddresses` mixin, which ensures that the parent class is provided with an `addresses` collection which contains `Address` objects.

The discriminator_on_association.py and generic_fk.py scripts are modernized versions of recipes presented in the 2007 blog post Polymorphic Associations with SQLAlchemy.

Listing of files:

- `discriminator_on_association.py` - Illustrates a mixin which provides a generic association using a single target table and a single association table, referred to by all parent tables. The association table contains

a "discriminator" column which determines what type of parent object associates to each particular row in the association table.

- `table_per_association.py` - Illustrates a mixin which provides a generic association via a individually generated association tables for each parent class. The associated objects themselves are persisted in a single table shared among all parents.
- `table_per_related.py` - Illustrates a generic association which persists association objects within individual tables, each one generated to persist those objects on behalf of a particular parent class.
- `generic_fk.py` - Illustrates a so-called "generic foreign key", in a similar fashion to that of popular frameworks such as Django, ROR, etc. This approach bypasses standard referential integrity practices, in that the "foreign key" column is not actually constrained to refer to any particular table; instead, in-application logic is used to determine which table is referenced.

Large Collections

Large collection example.

Illustrates the options to use with *`relationship()`* when the list of related objects is very large, including:

- "dynamic" relationships which query slices of data as accessed
- how to use ON DELETE CASCADE in conjunction with `passive_deletes=True` to greatly improve the performance of related collection deletion.

Listing of files:

- large_collection.py

Materialized Paths

Illustrates the "materialized paths" pattern for hierarchical data using the SQLAlchemy ORM.

Listing of files:

- `materialized_paths.py` - Illustrates the "materialized paths" pattern.

Nested Sets

Illustrates a rudimentary way to implement the "nested sets" pattern for hierarchical data using the SQLAlchemy ORM.

Listing of files:

- `nested_sets.py` - Celko's "Nested Sets" Tree Structure.

Performance

A performance profiling suite for a variety of SQLAlchemy use cases.

Each suite focuses on a specific use case with a particular performance profile and associated implications:

- bulk inserts
- individual inserts, with or without transactions
- fetching large numbers of rows
- running lots of short queries

All suites include a variety of use patterns illustrating both Core and ORM use, and are generally sorted in order of performance from worst to greatest, inversely based on amount of functionality provided by SQLAlchemy, greatest to least (these two things generally correspond perfectly).

A command line tool is presented at the package level which allows individual suites to be run:

```
$ python -m examples.performance --help
usage: python -m examples.performance [-h] [--test TEST] [--dburl DBURL]
                                      [--num NUM] [--profile] [--dump]
                                      [--runsnake] [--echo]

                                      {bulk_inserts,large_resultsets,single_inserts}

positional arguments:
  {bulk_inserts,large_resultsets,single_inserts}
                        suite to run

optional arguments:
  -h, --help            show this help message and exit
  --test TEST           run specific test name
  --dburl DBURL         database URL, default sqlite:///profile.db
  --num NUM             Number of iterations/items/etc for tests; default is 0
                        module-specific
  --profile             run profiling and dump call counts
  --dump                dump full call profile (implies --profile)
  --runsnake            invoke runsnakerun (implies --profile)
  --echo                Echo SQL output
```

An example run looks like:

```
$ python -m examples.performance bulk_inserts
```

Or with options:

```
$ python -m examples.performance bulk_inserts \
    --dburl mysql+mysqldb://scott:tiger@localhost/test \
    --profile --num 1000
```

See also:

How can I profile a SQLAlchemy powered application?

File Listing

Listing of files:

- `bulk_updates.py` - This series of tests illustrates different ways to UPDATE a large number of rows in bulk.
- `short_selects.py` - This series of tests illustrates different ways to INSERT a large number of rows in bulk.
- `__main__.py` - Allows the examples/performance package to be run as a script.
- `large_resultsets.py` - In this series of tests, we are looking at time to load a large number of very small and simple rows.
- `bulk_inserts.py` - This series of tests illustrates different ways to INSERT a large number of rows in bulk.

- `single_inserts.py` - In this series of tests, we're looking at a method that inserts a row within a distinct transaction, and afterwards returns to essentially a "closed" state. This would be analogous to an API call that starts up a database connection, inserts the row, commits and closes.

Running all tests with time

This is the default form of run:

```
$ python -m examples.performance single_inserts
Tests to run: test_orm_commit, test_bulk_save,
              test_bulk_insert_dictionaries, test_core,
              test_core_query_caching, test_dbapi_raw_w_connect,
              test_dbapi_raw_w_pool

test_orm_commit : Individual INSERT/COMMIT pairs via the
    ORM (10000 iterations); total time 13.690218 sec
test_bulk_save : Individual INSERT/COMMIT pairs using
    the "bulk" API  (10000 iterations); total time 11.290371 sec
test_bulk_insert_dictionaries : Individual INSERT/COMMIT pairs using
    the "bulk" API with dictionaries (10000 iterations);
    total time 10.814626 sec
test_core : Individual INSERT/COMMIT pairs using Core.
    (10000 iterations); total time 9.665620 sec
test_core_query_caching : Individual INSERT/COMMIT pairs using Core
    with query caching (10000 iterations); total time 9.209010 sec
test_dbapi_raw_w_connect : Individual INSERT/COMMIT pairs w/ DBAPI +
    connection each time (10000 iterations); total time 9.551103 sec
test_dbapi_raw_w_pool : Individual INSERT/COMMIT pairs w/ DBAPI +
    connection pool (10000 iterations); total time 8.001813 sec
```

Dumping Profiles for Individual Tests

A Python profile output can be dumped for all tests, or more commonly individual tests:

```
$ python -m examples.performance single_inserts --test test_core --num 1000 --dump
Tests to run: test_core
test_core : Individual INSERT/COMMIT pairs using Core. (1000 iterations); total fn calls 186109
         186109 function calls (186102 primitive calls) in 1.089 seconds

   Ordered by: internal time, call count

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1000    0.634    0.001    0.634    0.001 {method 'commit' of 'sqlite3.Connection' objects}
     1000    0.154    0.000    0.154    0.000 {method 'execute' of 'sqlite3.Cursor' objects}
     1000    0.021    0.000    0.074    0.000 /Users/classic/dev/sqlalchemy/lib/sqlalchemy/sql/compil
     1000    0.015    0.000    0.034    0.000 /Users/classic/dev/sqlalchemy/lib/sqlalchemy/engine/def
        1    0.012    0.012    1.091    1.091 examples/performance/single_inserts.py:79(test_core)

    ...
```

Using RunSnake

This option requires the RunSnake command line tool be installed:

```
$ python -m examples.performance single_inserts --test test_core --num 1000 --runsnake
```

A graphical RunSnake output will be displayed.

Writing your Own Suites

The profiler suite system is extensible, and can be applied to your own set of tests. This is a valuable technique to use in deciding upon the proper approach for some performance-critical set of routines. For example, if we wanted to profile the difference between several kinds of loading, we can create a file `test_loads.py`, with the following content:

```
from examples.performance import Profiler
from sqlalchemy import Integer, Column, create_engine, ForeignKey
from sqlalchemy.orm import relationship, joinedload, subqueryload, Session
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
engine = None
session = None


class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    children = relationship("Child")


class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parent_id = Column(Integer, ForeignKey('parent.id'))


# Init with name of file, default number of items
Profiler.init("test_loads", 1000)


@Profiler.setup_once
def setup_once(dburl, echo, num):
    "setup once.  create an engine, insert fixture data"
    global engine
    engine = create_engine(dburl, echo=echo)
    Base.metadata.drop_all(engine)
    Base.metadata.create_all(engine)
    sess = Session(engine)
    sess.add_all([
        Parent(children=[Child() for j in range(100)])
        for i in range(num)
    ])
    sess.commit()


@Profiler.setup
def setup(dburl, echo, num):
    "setup per test.  create a new Session."
```

```
    global session
    session = Session(engine)
    # pre-connect so this part isn't profiled (if we choose)
    session.connection()


@Profiler.profile
def test_lazyload(n):
    "load everything, no eager loading."

    for parent in session.query(Parent):
        parent.children


@Profiler.profile
def test_joinedload(n):
    "load everything, joined eager loading."

    for parent in session.query(Parent).options(joinedload("children")):
        parent.children


@Profiler.profile
def test_subqueryload(n):
    "load everything, subquery eager loading."

    for parent in session.query(Parent).options(subqueryload("children")):
        parent.children

if __name__ == '__main__':
    Profiler.main()
```

We can run our new script directly:

```
$ python test_loads.py  --dburl postgresql+psycopg2://scott:tiger@localhost/test
Running setup once...
Tests to run: test_lazyload, test_joinedload, test_subqueryload
test_lazyload : load everything, no eager loading. (1000 iterations); total time 11.971159 sec
test_joinedload : load everything, joined eager loading. (1000 iterations); total time 2.754592 sec
test_subqueryload : load everything, subquery eager loading. (1000 iterations); total time 2.977696 s
```

As well as see RunSnake output for an individual test:

```
$ python test_loads.py  --num 100 --runsnake --test test_joinedload
```

Relationship Join Conditions

Examples of various *orm.relationship()* configurations, which make use of the `primaryjoin` argument to compose special types of join conditions.

Listing of files:

- `cast.py` - Illustrate a *relationship()* that joins two columns where those columns are not of the same type, and a CAST must be used on the SQL side in order to match them.

- `threeway.py` - Illustrate a "three way join" - where a primary table joins to a remote table via an association table, but then the primary table also needs to refer to some columns in the remote table directly.

XML Persistence

Illustrates three strategies for persisting and querying XML documents as represented by ElementTree in a relational database. The techniques do not apply any mappings to the ElementTree objects directly, so are compatible with the native cElementTree as well as lxml, and can be adapted to suit any kind of DOM representation system. Querying along xpath-like strings is illustrated as well.

E.g.:

```
# parse an XML file and persist in the database
doc = ElementTree.parse("test.xml")
session.add(Document(file, doc))
session.commit()

# locate documents with a certain path/attribute structure
for document in find_document('/somefile/header/field2[@attr=foo]'):
    # dump the XML
    print document
```

Listing of files:

- `pickle.py` - illustrates a quick and dirty way to persist an XML document expressed using ElementTree and pickle.
- `adjacency_list.py` - Illustrates an explicit way to persist an XML document expressed using ElementTree.
- `optimized_al.py` - Uses the same strategy as `adjacency_list.py`, but associates each DOM row with its owning document row, so that a full document of DOM nodes can be loaded using O(1) queries - the construction of the "hierarchy" is performed after the load in a non-recursive fashion and is more efficient.

Versioning Objects

Versioning with a History Table

Illustrates an extension which creates version tables for entities and stores records for each change. The given extensions generate an anonymous "history" class which represents historical versions of the target object.

Usage is illustrated via a unit test module `test_versioning.py`, which can be run via nose:

```
cd examples/versioning
nosetests -v
```

A fragment of example usage, using declarative:

```
from history_meta import Versioned, versioned_session

Base = declarative_base()

class SomeClass(Versioned, Base):
    __tablename__ = 'sometable'
```

```
    id = Column(Integer, primary_key=True)
    name = Column(String(50))

    def __eq__(self, other):
        assert type(other) is SomeClass and other.id == self.id

Session = sessionmaker(bind=engine)
versioned_session(Session)

sess = Session()
sc = SomeClass(name='sc1')
sess.add(sc)
sess.commit()

sc.name = 'sc1modified'
sess.commit()

assert sc.version == 2

SomeClassHistory = SomeClass.__history_mapper__.class_

assert sess.query(SomeClassHistory).\
            filter(SomeClassHistory.version == 1).\
            all() \
            == [SomeClassHistory(version=1, name='sc1')]
```

The `Versioned` mixin is designed to work with declarative. To use the extension with classical mappers, the `_history_mapper` function can be applied:

```
from history_meta import _history_mapper

m = mapper(SomeClass, sometable)
_history_mapper(m)

SomeHistoryClass = SomeClass.__history_mapper__.class_
```

Listing of files:

- `test_versioning.py` - Unit tests illustrating usage of the `history_meta.py` module functions.
- `history_meta.py` - Versioned mixin class and other utilities.

Versioning using Temporal Rows

Illustrates an extension which versions data by storing new rows for each change; that is, what would normally be an UPDATE becomes an INSERT.

Listing of files:

- `versioned_rows.py` - Illustrates a method to intercept changes on objects, turning an UPDATE statement on a single row into an INSERT statement, so that a new row is inserted with the new data, keeping the old row intact.
- `versioned_map.py` - A variant of the versioned_rows example. Here we store a dictionary of key/value pairs, storing the k/v's in a "vertical" fashion where each key gets a row. The value is split out into two separate datatypes, string and int - the range of datatype storage can be adjusted for individual needs.

Vertical Attribute Mapping

Illustrates "vertical table" mappings.

A "vertical table" refers to a technique where individual attributes of an object are stored as distinct rows in a table. The "vertical table" technique is used to persist objects which can have a varied set of attributes, at the expense of simple query control and brevity. It is commonly found in content/document management systems in order to represent user-created structures flexibly.

Two variants on the approach are given. In the second, each row references a "datatype" which contains information about the type of information stored in the attribute, such as integer, string, or date.

Example:

```
shrew = Animal(u'shrew')
shrew[u'cuteness'] = 5
shrew[u'weasel-like'] = False
shrew[u'poisonous'] = True

session.add(shrew)
session.flush()

q = (session.query(Animal).
     filter(Animal.facts.any(
       and_(AnimalFact.key == u'weasel-like',
            AnimalFact.value == True))))
print 'weasel-like animals', q.all()
```

Listing of files:

- `dictlike.py` - Mapping a vertical table as a dictionary.
- `dictlike-polymorphic.py` - Mapping a polymorphic-valued vertical table as a dictionary.

2.8.2 Inheritance Mapping Recipes

Basic Inheritance Mappings

Working examples of single-table, joined-table, and concrete-table inheritance as described in datamapping_inheritance.

Listing of files:

- `single.py` - Single-table inheritance example.
- `joined.py` - Joined-table (table-per-subclass) inheritance example.
- `concrete.py` - Concrete (table-per-class) inheritance example.

2.8.3 Special APIs

Attribute Instrumentation

Two examples illustrating modifications to SQLAlchemy's attribute management system.

Listing of files:

- `listen_for_events.py` - Illustrates how to attach events to all instrumented attributes and listen for change events.
- `custom_management.py` - Illustrates customized class instrumentation, using the *`sqlalchemy.ext.instrumentation`* extension package.

Horizontal Sharding

A basic example of using the SQLAlchemy Sharding API. Sharding refers to horizontally scaling data across multiple databases.

The basic components of a "sharded" mapping are:

- multiple databases, each assigned a 'shard id'
- a function which can return a single shard id, given an instance to be saved; this is called "shard_chooser"
- a function which can return a list of shard ids which apply to a particular instance identifier; this is called "id_chooser". If it returns all shard ids, all shards will be searched.
- a function which can return a list of shard ids to try, given a particular Query ("query_chooser"). If it returns all shard ids, all shards will be queried and the results joined together.

In this example, four sqlite databases will store information about weather data on a database-per-continent basis. We provide example shard_chooser, id_chooser and query_chooser functions. The query_chooser illustrates inspection of the SQL expression element in order to attempt to determine a single shard being requested.

The construction of generic sharding routines is an ambitious approach to the issue of organizing instances among multiple databases. For a more plain-spoken alternative, the "distinct entity" approach is a simple method of assigning objects to different tables (and potentially database nodes) in an explicit way - described on the wiki at EntityName.

Listing of files:

- attribute_shard.py

2.8.4 Extending the ORM

Dogpile Caching

Illustrates how to embed dogpile.cache functionality within the *`Query`* object, allowing full cache control as well as the ability to pull "lazy loaded" attributes from long term cache as well.

Changed in version 0.8: The example was modernized to use dogpile.cache, replacing Beaker as the caching library in use.

In this demo, the following techniques are illustrated:

- Using custom subclasses of *`Query`*
- Basic technique of circumventing Query to pull from a custom cache source instead of the database.
- Rudimental caching with dogpile.cache, using "regions" which allow global control over a fixed set of configurations.
- Using custom `MapperOption` objects to configure options on a Query, including the ability to invoke the options deep within an object graph when lazy loads occur.

E.g.:

```
# query for Person objects, specifying cache
q = Session.query(Person).options(FromCache("default"))

# specify that each Person's "addresses" collection comes from
# cache too
q = q.options(RelationshipCache(Person.addresses, "default"))

# query
print q.all()
```

To run, both SQLAlchemy and dogpile.cache must be installed or on the current PYTHONPATH. The demo will create a local directory for datafiles, insert initial data, and run. Running the demo a second time will utilize the cache files already present, and exactly one SQL statement against two tables will be emitted - the displayed result however will utilize dozens of lazyloads that all pull from cache.

The demo scripts themselves, in order of complexity, are run as Python modules so that relative imports work:

```
python -m examples.dogpile_caching.helloworld

python -m examples.dogpile_caching.relationship_caching

python -m examples.dogpile_caching.advanced

python -m examples.dogpile_caching.local_session_caching
```

Listing of files:

- `environment.py` - Establish data / cache file paths, and configurations, bootstrap fixture data if necessary.
- `caching_query.py` - Represent functions and classes which allow the usage of Dogpile caching with SQLAlchemy. Introduces a query option called FromCache.
- `model.py` - The datamodel, which represents Person that has multiple Address objects, each with PostalCode, City, Country.
- `fixture_data.py` - Installs some sample data. Here we have a handful of postal codes for a few US/Canadian cities. Then, 100 Person records are installed, each with a randomly selected postal code.
- `helloworld.py` - Illustrate how to load some data, and cache the results.
- `relationship_caching.py` - Illustrates how to add cache options on relationship endpoints, so that lazyloads load from cache.
- `advanced.py` - Illustrate usage of Query combined with the FromCache option, including front-end loading, cache invalidation and collection caching.
- `local_session_caching.py` - Grok everything so far ? This example creates a new dogpile.cache backend that will persist data in a dictionary which is local to the current session. remove() the session and the cache is gone.

PostGIS Integration

A naive example illustrating techniques to help embed PostGIS functionality.

This example was originally developed in the hopes that it would be extrapolated into a comprehensive PostGIS integration layer. We are pleased to announce that this has come to fruition as GeoAlchemy.

The example illustrates:

- a DDL extension which allows CREATE/DROP to work in conjunction with AddGeometryColumn/DropGeometryColumn
- a Geometry type, as well as a few subtypes, which convert result row values to a GIS-aware object, and also integrates with the DDL extension.
- a GIS-aware object which stores a raw geometry value and provides a factory for functions such as AsText().
- an ORM comparator which can override standard column methods on mapped objects to produce GIS operators.
- an attribute event listener that intercepts strings and converts to GeomFromText().
- a standalone operator example.

The implementation is limited to only public, well known and simple to use extension points.

E.g.:

```
print session.query(Road).filter(Road.road_geom.intersects(r1.road_geom)).all()
```

Listing of files:

- postgis.py

CHAPTER 3

SQLAlchemy Core

The breadth of SQLAlchemy's SQL rendering engine, DBAPI integration, transaction integration, and schema description services are documented here. In contrast to the ORM's domain-centric mode of usage, the SQL Expression Language provides a schema-centric usage paradigm.

3.1 SQL Expression Language Tutorial

The SQLAlchemy Expression Language presents a system of representing relational database structures and expressions using Python constructs. These constructs are modeled to resemble those of the underlying database as closely as possible, while providing a modicum of abstraction of the various implementation differences between database backends. While the constructs attempt to represent equivalent concepts between backends with consistent structures, they do not conceal useful concepts that are unique to particular subsets of backends. The Expression Language therefore presents a method of writing backend-neutral SQL expressions, but does not attempt to enforce that expressions are backend-neutral.

The Expression Language is in contrast to the Object Relational Mapper, which is a distinct API that builds on top of the Expression Language. Whereas the ORM, introduced in *Object Relational Tutorial*, presents a high level and abstracted pattern of usage, which itself is an example of applied usage of the Expression Language, the Expression Language presents a system of representing the primitive constructs of the relational database directly without opinion.

While there is overlap among the usage patterns of the ORM and the Expression Language, the similarities are more superficial than they may at first appear. One approaches the structure and content of data from the perspective of a user-defined domain model which is transparently persisted and refreshed from its underlying storage model. The other approaches it from the perspective of literal schema and SQL expression representations which are explicitly composed into messages consumed individually by the database.

A successful application may be constructed using the Expression Language exclusively, though the application will need to define its own system of translating application concepts into individual database messages and from individual database result sets. Alternatively, an application constructed with the ORM may, in advanced scenarios, make occasional usage of the Expression Language directly in certain areas where specific database interactions are required.

The following tutorial is in doctest format, meaning each >>> line represents something you can type at a Python command prompt, and the following text represents the expected return value. The tutorial has no prerequisites.

3.1.1 Version Check

A quick check to verify that we are on at least **version 1.1** of SQLAlchemy:

```
>>> import sqlalchemy
>>> sqlalchemy.__version__
1.1.0
```

3.1.2 Connecting

For this tutorial we will use an in-memory-only SQLite database. This is an easy way to test things without needing to have an actual database defined anywhere. To connect we use *create_engine()*:

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:', echo=True)
```

The `echo` flag is a shortcut to setting up SQLAlchemy logging, which is accomplished via Python's standard `logging` module. With it enabled, we'll see all the generated SQL produced. If you are working through this tutorial and want less output generated, set it to `False`. This tutorial will format the SQL behind a popup window so it doesn't get in our way; just click the "SQL" links to see what's being generated.

The return value of *create_engine()* is an instance of *Engine*, and it represents the core interface to the database, adapted through a dialect that handles the details of the database and DBAPI in use. In this case the SQLite dialect will interpret instructions to the Python built-in `sqlite3` module.

Lazy Connecting

The *Engine*, when first returned by *create_engine()*, has not actually tried to connect to the database yet; that happens only the first time it is asked to perform a task against the database.

The first time a method like *Engine.execute()* or *Engine.connect()* is called, the *Engine* establishes a real DBAPI connection to the database, which is then used to emit the SQL.

See also:

Database Urls - includes examples of *create_engine()* connecting to several kinds of databases with links to more information.

3.1.3 Define and Create Tables

The SQL Expression Language constructs its expressions in most cases against table columns. In SQLAlchemy, a column is most often represented by an object called *Column*, and in all cases a *Column* is associated with a *Table*. A collection of *Table* objects and their associated child objects is referred to as **database metadata**. In this tutorial we will explicitly lay out several *Table* objects, but note that SA can also "import" whole sets of *Table* objects automatically from an existing database (this process is called **table reflection**).

We define our tables all within a catalog called *MetaData*, using the *Table* construct, which resembles regular SQL CREATE TABLE statements. We'll make two tables, one of which represents "users" in an application, and another which represents zero or more "email addresses" for each row in the "users" table:

```
>>> from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey
>>> metadata = MetaData()
>>> users = Table('users', metadata,
...     Column('id', Integer, primary_key=True),
...     Column('name', String),
...     Column('fullname', String),
... )
```

```
>>> addresses = Table('addresses', metadata,
...   Column('id', Integer, primary_key=True),
...   Column('user_id', None, ForeignKey('users.id')),
...   Column('email_address', String, nullable=False)
...  )
```

All about how to define *Table* objects, as well as how to create them from an existing database automatically, is described in *Describing Databases with MetaData*.

Next, to tell the *MetaData* we'd actually like to create our selection of tables for real inside the SQLite database, we use *create_all()*, passing it the `engine` instance which points to our database. This will check for the presence of each table first before creating, so it's safe to call multiple times:

```
>>> metadata.create_all(engine)
SE...
CREATE TABLE users (
    id INTEGER NOT NULL,
    name VARCHAR,
    fullname VARCHAR,
    PRIMARY KEY (id)
)
()
COMMIT
CREATE TABLE addresses (
    id INTEGER NOT NULL,
    user_id INTEGER,
    email_address VARCHAR NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY(user_id) REFERENCES users (id)
)
()
COMMIT
```

Note: Users familiar with the syntax of CREATE TABLE may notice that the VARCHAR columns were generated without a length; on SQLite and Postgresql, this is a valid datatype, but on others, it's not allowed. So if running this tutorial on one of those databases, and you wish to use SQLAlchemy to issue CREATE TABLE, a "length" may be provided to the *String* type as below:

```
Column('name', String(50))
```

The length field on *String*, as well as similar precision/scale fields available on *Integer*, *Numeric*, etc. are not referenced by SQLAlchemy other than when creating tables.

Additionally, Firebird and Oracle require sequences to generate new primary key identifiers, and SQLAlchemy doesn't generate or assume these without being instructed. For that, you use the *Sequence* construct:

```
from sqlalchemy import Sequence
Column('id', Integer, Sequence('user_id_seq'), primary_key=True)
```

A full, foolproof *Table* is therefore:

```
users = Table('users', metadata,
   Column('id', Integer, Sequence('user_id_seq'), primary_key=True),
   Column('name', String(50)),
   Column('fullname', String(50)),
```

```
    Column('password', String(12))
)
```

We include this more verbose *Table* construct separately to highlight the difference between a minimal construct geared primarily towards in-Python usage only, versus one that will be used to emit CREATE TABLE statements on a particular set of backends with more stringent requirements.

3.1.4 Insert Expressions

The first SQL expression we'll create is the *Insert* construct, which represents an INSERT statement. This is typically created relative to its target table:

```
>>> ins = users.insert()
```

To see a sample of the SQL this construct produces, use the `str()` function:

```
>>> str(ins)
'INSERT INTO users (id, name, fullname) VALUES (:id, :name, :fullname)'
```

Notice above that the INSERT statement names every column in the `users` table. This can be limited by using the `values()` method, which establishes the VALUES clause of the INSERT explicitly:

```
>>> ins = users.insert().values(name='jack', fullname='Jack Jones')
>>> str(ins)
'INSERT INTO users (name, fullname) VALUES (:name, :fullname)'
```

Above, while the `values` method limited the VALUES clause to just two columns, the actual data we placed in `values` didn't get rendered into the string; instead we got named bind parameters. As it turns out, our data *is* stored within our *Insert* construct, but it typically only comes out when the statement is actually executed; since the data consists of literal values, SQLAlchemy automatically generates bind parameters for them. We can peek at this data for now by looking at the compiled form of the statement:

```
>>> ins.compile().params
{'fullname': 'Jack Jones', 'name': 'jack'}
```

3.1.5 Executing

The interesting part of an *Insert* is executing it. In this tutorial, we will generally focus on the most explicit method of executing a SQL construct, and later touch upon some "shortcut" ways to do it. The `engine` object we created is a repository for database connections capable of issuing SQL to the database. To acquire a connection, we use the `connect()` method:

```
>>> conn = engine.connect()
>>> conn
<sqlalchemy.engine.base.Connection object at 0x...>
```

The *Connection* object represents an actively checked out DBAPI connection resource. Lets feed it our *Insert* object and see what happens:

```
>>> result = conn.execute(ins)
INSERT INTO users (name, fullname) VALUES (?, ?)
('jack', 'Jack Jones')
COMMIT
```

So the INSERT statement was now issued to the database. Although we got positional "qmark" bind parameters instead of "named" bind parameters in the output. How come ? Because when executed, the *Connection* used the SQLite **dialect** to help generate the statement; when we use the `str()` function, the statement isn't aware of this dialect, and falls back onto a default which uses named parameters. We can view this manually as follows:

```
>>> ins.bind = engine
>>> str(ins)
'INSERT INTO users (name, fullname) VALUES (?, ?)'
```

What about the `result` variable we got when we called `execute()` ? As the SQLAlchemy *Connection* object references a DBAPI connection, the result, known as a *ResultProxy* object, is analogous to the DBAPI cursor object. In the case of an INSERT, we can get important information from it, such as the primary key values which were generated from our statement using *ResultProxy.inserted_primary_key*:

```
>>> result.inserted_primary_key
[1]
```

The value of `1` was automatically generated by SQLite, but only because we did not specify the `id` column in our *Insert* statement; otherwise, our explicit value would have been used. In either case, SQLAlchemy always knows how to get at a newly generated primary key value, even though the method of generating them is different across different databases; each database's *Dialect* knows the specific steps needed to determine the correct value (or values; note that *ResultProxy.inserted_primary_key* returns a list so that it supports composite primary keys). Methods here range from using `cursor.lastrowid`, to selecting from a database-specific function, to using `INSERT..RETURNING` syntax; this all occurs transparently.

3.1.6 Executing Multiple Statements

Our insert example above was intentionally a little drawn out to show some various behaviors of expression language constructs. In the usual case, an *Insert* statement is usually compiled against the parameters sent to the `execute()` method on *Connection*, so that there's no need to use the `values` keyword with *Insert*. Lets create a generic *Insert* statement again and use it in the "normal" way:

```
>>> ins = users.insert()
>>> conn.execute(ins, id=2, name='wendy', fullname='Wendy Williams')
INSERT INTO users (id, name, fullname) VALUES (?, ?, ?)
(2, 'wendy', 'Wendy Williams')
COMMIT
<sqlalchemy.engine.result.ResultProxy object at 0x...>
```

Above, because we specified all three columns in the `execute()` method, the compiled *Insert* included all three columns. The *Insert* statement is compiled at execution time based on the parameters we specified; if we specified fewer parameters, the *Insert* would have fewer entries in its VALUES clause.

To issue many inserts using DBAPI's `executemany()` method, we can send in a list of dictionaries each containing a distinct set of parameters to be inserted, as we do here to add some email addresses:

```
>>> conn.execute(addresses.insert(), [
...     {'user_id': 1, 'email_address' : 'jack@yahoo.com'},
...     {'user_id': 1, 'email_address' : 'jack@msn.com'},
...     {'user_id': 2, 'email_address' : 'www@www.org'},
...     {'user_id': 2, 'email_address' : 'wendy@aol.com'},
... ])
INSERT INTO addresses (user_id, email_address) VALUES (?, ?)
((1, 'jack@yahoo.com'), (1, 'jack@msn.com'), (2, 'www@www.org'), (2, 'wendy@aol.com'))
COMMIT
<sqlalchemy.engine.result.ResultProxy object at 0x...>
```

Above, we again relied upon SQLite's automatic generation of primary key identifiers for each `addresses` row.

When executing multiple sets of parameters, each dictionary must have the **same** set of keys; i.e. you cant have fewer keys in some dictionaries than others. This is because the *Insert* statement is compiled against the **first** dictionary in the list, and it's assumed that all subsequent argument dictionaries are compatible with that statement.

The "executemany" style of invocation is available for each of the *insert()*, *update()* and *delete()* constructs.

3.1.7 Selecting

We began with inserts just so that our test database had some data in it. The more interesting part of the data is selecting it! We'll cover UPDATE and DELETE statements later. The primary construct used to generate SELECT statements is the *select()* function:

```
>>> from sqlalchemy.sql import select
>>> s = select([users])
>>> result = conn.execute(s)
SELECT users.id, users.name, users.fullname
FROM users
()
```

Above, we issued a basic *select()* call, placing the `users` table within the COLUMNS clause of the select, and then executing. SQLAlchemy expanded the `users` table into the set of each of its columns, and also generated a FROM clause for us. The result returned is again a *ResultProxy* object, which acts much like a DBAPI cursor, including methods such as *fetchone()* and *fetchall()*. The easiest way to get rows from it is to just iterate:

```
>>> for row in result:
...     print(row)
(1, u'jack', u'Jack Jones')
(2, u'wendy', u'Wendy Williams')
```

Above, we see that printing each row produces a simple tuple-like result. We have more options at accessing the data in each row. One very common way is through dictionary access, using the string names of columns:

```
>>> result = conn.execute(s)
SELECT users.id, users.name, users.fullname
FROM users
()

>>> row = result.fetchone()
>>> print("name:", row['name'], "; fullname:", row['fullname'])
name: jack ; fullname: Jack Jones
```

Integer indexes work as well:

```
>>> row = result.fetchone()
>>> print("name:", row[1], "; fullname:", row[2])
name: wendy ; fullname: Wendy Williams
```

But another way, whose usefulness will become apparent later on, is to use the *Column* objects directly as keys:

```
>>> for row in conn.execute(s):
...     print("name:", row[users.c.name], "; fullname:", row[users.c.fullname])
SELECT users.id, users.name, users.fullname
FROM users
()
name: jack ; fullname: Jack Jones
name: wendy ; fullname: Wendy Williams
```

Result sets which have pending rows remaining should be explicitly closed before discarding. While the cursor and connection resources referenced by the *ResultProxy* will be respectively closed and returned to the connection pool when the object is garbage collected, it's better to make it explicit as some database APIs are very picky about such things:

```
>>> result.close()
```

If we'd like to more carefully control the columns which are placed in the COLUMNS clause of the select, we reference individual *Column* objects from our *Table*. These are available as named attributes off the `c` attribute of the *Table* object:

```
>>> s = select([users.c.name, users.c.fullname])
>>> result = conn.execute(s)
SELECT users.name, users.fullname
FROM users
()
>>> for row in result:
...     print(row)
(u'jack', u'Jack Jones')
(u'wendy', u'Wendy Williams')
```

Lets observe something interesting about the FROM clause. Whereas the generated statement contains two distinct sections, a "SELECT columns" part and a "FROM table" part, our *select()* construct only has a list containing columns. How does this work ? Let's try putting *two* tables into our *select()* statement:

```
>>> for row in conn.execute(select([users, addresses])):
...     print(row)
SELECT users.id, users.name, users.fullname, addresses.id, addresses.user_id, addresses.email_address
FROM users, addresses
()
(1, u'jack', u'Jack Jones', 1, 1, u'jack@yahoo.com')
(1, u'jack', u'Jack Jones', 2, 1, u'jack@msn.com')
(1, u'jack', u'Jack Jones', 3, 2, u'www@www.org')
(1, u'jack', u'Jack Jones', 4, 2, u'wendy@aol.com')
(2, u'wendy', u'Wendy Williams', 1, 1, u'jack@yahoo.com')
(2, u'wendy', u'Wendy Williams', 2, 1, u'jack@msn.com')
(2, u'wendy', u'Wendy Williams', 3, 2, u'www@www.org')
(2, u'wendy', u'Wendy Williams', 4, 2, u'wendy@aol.com')
```

It placed **both** tables into the FROM clause. But also, it made a real mess. Those who are familiar with SQL joins know that this is a **Cartesian product**; each row from the `users` table is produced against each row from the `addresses` table. So to put some sanity into this statement, we need a WHERE clause. We do that using *Select.where()*:

```
>>> s = select([users, addresses]).where(users.c.id == addresses.c.user_id)
>>> for row in conn.execute(s):
...     print(row)
SELECT users.id, users.name, users.fullname, addresses.id,
   addresses.user_id, addresses.email_address
FROM users, addresses
WHERE users.id = addresses.user_id
()
(1, u'jack', u'Jack Jones', 1, 1, u'jack@yahoo.com')
(1, u'jack', u'Jack Jones', 2, 1, u'jack@msn.com')
(2, u'wendy', u'Wendy Williams', 3, 2, u'www@www.org')
(2, u'wendy', u'Wendy Williams', 4, 2, u'wendy@aol.com')
```

So that looks a lot better, we added an expression to our *select()* which had the effect of adding `WHERE users.id = addresses.user_id` to our statement, and our results were managed down so that the join of `users` and `addresses` rows made sense. But let's look at that expression? It's using just a Python equality operator between two different *Column* objects. It should be clear that something is up. Saying `1 == 1` produces `True`, and `1 == 2` produces `False`, not a WHERE clause. So lets see exactly what that expression is doing:

```
>>> users.c.id == addresses.c.user_id
<sqlalchemy.sql.elements.BinaryExpression object at 0x...>
```

Wow, surprise ! This is neither a `True` nor a `False`. Well what is it ?

```
>>> str(users.c.id == addresses.c.user_id)
'users.id = addresses.user_id'
```

As you can see, the == operator is producing an object that is very much like the *Insert* and *select()* objects we've made so far, thanks to Python's `__eq__()` builtin; you call `str()` on it and it produces SQL. By now, one can see that everything we are working with is ultimately the same type of object. SQLAlchemy terms the base class of all of these expressions as *ColumnElement*.

3.1.8 Operators

Since we've stumbled upon SQLAlchemy's operator paradigm, let's go through some of its capabilities. We've seen how to equate two columns to each other:

```
>>> print(users.c.id == addresses.c.user_id)
users.id = addresses.user_id
```

If we use a literal value (a literal meaning, not a SQLAlchemy clause object), we get a bind parameter:

```
>>> print(users.c.id == 7)
users.id = :id_1
```

The `7` literal is embedded the resulting *ColumnElement*; we can use the same trick we did with the *Insert* object to see it:

```
>>> (users.c.id == 7).compile().params
{u'id_1': 7}
```

Most Python operators, as it turns out, produce a SQL expression here, like equals, not equals, etc.:

```
>>> print(users.c.id != 7)
users.id != :id_1

>>> # None converts to IS NULL
>>> print(users.c.name == None)
users.name IS NULL

>>> # reverse works too
>>> print('fred' > users.c.name)
users.name < :name_1
```

If we add two integer columns together, we get an addition expression:

```
>>> print(users.c.id + addresses.c.id)
users.id + addresses.id
```

Interestingly, the type of the `Column` is important! If we use + with two string based columns (recall we put types like `Integer` and `String` on our `Column` objects at the beginning), we get something different:

```
>>> print(users.c.name + users.c.fullname)
users.name || users.fullname
```

Where `||` is the string concatenation operator used on most databases. But not all of them. MySQL users, fear not:

```
>>> print((users.c.name + users.c.fullname).
...      compile(bind=create_engine('mysql://')))
concat(users.name, users.fullname)
```

The above illustrates the SQL that's generated for an `Engine` that's connected to a MySQL database; the `||` operator now compiles as MySQL's `concat()` function.

If you have come across an operator which really isn't available, you can always use the `ColumnOperators.op()` method; this generates whatever operator you need:

```
>>> print(users.c.name.op('tiddlywinks')('foo'))
users.name tiddlywinks :name_1
```

This function can also be used to make bitwise operators explicit. For example:

```
somecolumn.op('&')(0xff)
```

is a bitwise AND of the value in *somecolumn*.

Operator Customization

While `ColumnOperators.op()` is handy to get at a custom operator in a hurry, the Core supports fundamental customization and extension of the operator system at the type level. The behavior of existing operators can be modified on a per-type basis, and new operations can be defined which become available for all column expressions that are part of that particular type. See the section *Redefining and Creating New Operators* for a description.

3.1.9 Conjunctions

We'd like to show off some of our operators inside of `select()` constructs. But we need to lump them together a little more, so let's first introduce some conjunctions. Conjunctions are those little words like AND and OR that put things together. We'll also hit upon NOT. `and_()`, `or_()`, and `not_()` can work from the corresponding functions SQLAlchemy provides (notice we also throw in a `like()`):

```
>>> from sqlalchemy.sql import and_, or_, not_
>>> print(and_(
...         users.c.name.like('j%'),
...         users.c.id == addresses.c.user_id,
...         or_(
...              addresses.c.email_address == 'wendy@aol.com',
...              addresses.c.email_address == 'jack@yahoo.com'
...         ),
...         not_(users.c.id > 5)
...       )
...  )
users.name LIKE :name_1 AND users.id = addresses.user_id AND
(addresses.email_address = :email_address_1
   OR addresses.email_address = :email_address_2)
AND users.id <= :id_1
```

And you can also use the re-jiggered bitwise AND, OR and NOT operators, although because of Python operator precedence you have to watch your parenthesis:

```
>>> print(users.c.name.like('j%') & (users.c.id == addresses.c.user_id) &
...     (
...       (addresses.c.email_address == 'wendy@aol.com') | \
...       (addresses.c.email_address == 'jack@yahoo.com')
...     ) \
...     & ~(users.c.id>5)
... )
users.name LIKE :name_1 AND users.id = addresses.user_id AND
(addresses.email_address = :email_address_1
    OR addresses.email_address = :email_address_2)
AND users.id <= :id_1
```

So with all of this vocabulary, let's select all users who have an email address at AOL or MSN, whose name starts with a letter between "m" and "z", and we'll also generate a column containing their full name combined with their email address. We will add two new constructs to this statement, `between()` and `label()`. `between()` produces a BETWEEN clause, and `label()` is used in a column expression to produce labels using the `AS` keyword; it's recommended when selecting from expressions that otherwise would not have a name:

```
>>> s = select([(users.c.fullname +
...               ", " + addresses.c.email_address).
...                label('title')]).\
```

```
...         where(
...            and_(
...                users.c.id == addresses.c.user_id,
...                users.c.name.between('m', 'z'),
...                or_(
...                   addresses.c.email_address.like('%@aol.com'),
...                   addresses.c.email_address.like('%@msn.com')
...                )
...            )
...         )
>>> conn.execute(s).fetchall()
SELECT users.fullname || ? || addresses.email_address AS title
FROM users, addresses
WHERE users.id = addresses.user_id AND users.name BETWEEN ? AND ? AND
(addresses.email_address LIKE ? OR addresses.email_address LIKE ?)
(', ', 'm', 'z', '%@aol.com', '%@msn.com')
[(u'Wendy Williams, wendy@aol.com',)]
```

Once again, SQLAlchemy figured out the FROM clause for our statement. In fact it will determine the FROM clause based on all of its other bits; the columns clause, the where clause, and also some other elements which we haven't covered yet, which include ORDER BY, GROUP BY, and HAVING.

A shortcut to using `and_()` is to chain together multiple `where()` clauses. The above can also be written as:

```
>>> s = select([(users.c.fullname +
...                ", " + addresses.c.email_address).
...                 label('title')]).\
...         where(users.c.id == addresses.c.user_id).\
...         where(users.c.name.between('m', 'z')).\
...         where(
...                or_(
...                   addresses.c.email_address.like('%@aol.com'),
...                   addresses.c.email_address.like('%@msn.com')
...                )
...         )
>>> conn.execute(s).fetchall()
SELECT users.fullname || ? || addresses.email_address AS title
FROM users, addresses
WHERE users.id = addresses.user_id AND users.name BETWEEN ? AND ? AND
(addresses.email_address LIKE ? OR addresses.email_address LIKE ?)
(', ', 'm', 'z', '%@aol.com', '%@msn.com')
[(u'Wendy Williams, wendy@aol.com',)]
```

The way that we can build up a `select()` construct through successive method calls is called method chaining.

3.1.10 Using Textual SQL

Our last example really became a handful to type. Going from what one understands to be a textual SQL expression into a Python construct which groups components together in a programmatic style can be hard. That's why SQLAlchemy lets you just use strings, for those cases when the SQL is already known and there isn't a strong need for the statement to support dynamic features. The `text()` construct is used to compose a textual statement that is passed to the database mostly unchanged. Below, we create a `text()` object and execute it:

```
>>> from sqlalchemy.sql import text
>>> s = text(
```

```
...     "SELECT users.fullname || ', ' || addresses.email_address AS title "
...         "FROM users, addresses "
...         "WHERE users.id = addresses.user_id "
...         "AND users.name BETWEEN :x AND :y "
...         "AND (addresses.email_address LIKE :e1 "
...             "OR addresses.email_address LIKE :e2)")
>>> conn.execute(s, x='m', y='z', e1='%@aol.com', e2='%@msn.com').fetchall()
SELECT users.fullname || ', ' || addresses.email_address AS title
FROM users, addresses
WHERE users.id = addresses.user_id AND users.name BETWEEN ? AND ? AND
(addresses.email_address LIKE ? OR addresses.email_address LIKE ?)
('m', 'z', '%@aol.com', '%@msn.com')
[(u'Wendy Williams, wendy@aol.com',)]
```

Above, we can see that bound parameters are specified in *text()* using the named colon format; this format is consistent regardless of database backend. To send values in for the parameters, we passed them into the *execute()* method as additional arguments. Depending on how we are working, we can also send values to be associated directly with the *text()* construct using the *bindparams()* method; if we are using datatypes that need special handling as they are received in Python, or we'd like to compose our *text()* object into a larger expression, we may also wish to use the *columns()* method in order to specify column return types and names:

```
>>> s = text(
...     "SELECT users.fullname || ', ' || addresses.email_address AS title "
...         "FROM users, addresses "
...         "WHERE users.id = addresses.user_id "
...         "AND users.name BETWEEN :x AND :y "
...         "AND (addresses.email_address LIKE :e1 "
...             "OR addresses.email_address LIKE :e2)")
>>> s = s.columns(title=String)
>>> s = s.bindparams(x='m', y='z', e1='%@aol.com', e2='%@msn.com')
>>> conn.execute(s).fetchall()
SELECT users.fullname || ', ' || addresses.email_address AS title
FROM users, addresses
WHERE users.id = addresses.user_id AND users.name BETWEEN ? AND ? AND
(addresses.email_address LIKE ? OR addresses.email_address LIKE ?)
('m', 'z', '%@aol.com', '%@msn.com')
{stop}[(u'Wendy Williams, wendy@aol.com',)]
```

text() can also be used freely within a *select()* object, which accepts *text()* objects as an argument for most of its builder functions. Below, we combine the usage of *text()* within a *select()* object. The *select()* construct provides the "geometry" of the statement, and the *text()* construct provides the textual content within this form. We can build a statement without the need to refer to any pre-established *Table* metadata:

```
>>> s = select([
...         text("users.fullname || ', ' || addresses.email_address AS title")
...     ]).\
...         where(
...             and_(
...                 text("users.id = addresses.user_id"),
...                 text("users.name BETWEEN 'm' AND 'z'"),
...                 text(
...                     "(addresses.email_address LIKE :x "
...                     "OR addresses.email_address LIKE :y)")
...             )
...         ).select_from(text('users, addresses'))
>>> conn.execute(s, x='%@aol.com', y='%@msn.com').fetchall()
```

```
SELECT users.fullname || ', ' || addresses.email_address AS title
FROM users, addresses
WHERE users.id = addresses.user_id AND users.name BETWEEN 'm' AND 'z'
AND (addresses.email_address LIKE ? OR addresses.email_address LIKE ?)
('%@aol.com', '%@msn.com')
[(u'Wendy Williams, wendy@aol.com',)]
```

Why not use strings everywhere?

When we use literal strings, the Core can't adapt our SQL to work on different database backends. Above, our expression won't work with MySQL since MySQL doesn't have the `||` construct. If we only use `text()` to specify columns, our `select()` construct will have an empty `.c` collection that we'd normally use to create subqueries. We also lose typing information about result columns and bound parameters, which is often needed to correctly translate data values between Python and the database. Overall, the more `text()` we use, the less flexibility and ability for manipulation/transformation the statement will have.

See also:

Using Textual SQL - integrating ORM-level queries with `text()`

Changed in version 1.0.0: The `select()` construct emits warnings when string SQL fragments are coerced to `text()`, and `text()` should be used explicitly. See *Warnings emitted when coercing full SQL fragments into text()* for background.

Using More Specific Text with `table()`, `literal_column()`, and `column()`

We can move our level of structure back in the other direction too, by using `column()`, `literal_column()`, and `table()` for some of the key elements of our statement. Using these constructs, we can get some more expression capabilities than if we used `text()` directly, as they provide to the Core more information about how the strings they store are to be used, but still without the need to get into full `Table` based metadata. Below, we also specify the `String` datatype for two of the key `literal_column()` objects, so that the string-specific concatenation operator becomes available. We also use `literal_column()` in order to use table-qualified expressions, e.g. `users.fullname`, that will be rendered as is; using `column()` implies an individual column name that may be quoted:

```
>>> from sqlalchemy import select, and_, text, String
>>> from sqlalchemy.sql import table, literal_column
>>> s = select([
...     literal_column("users.fullname", String) +
...     ', ' +
...     literal_column("addresses.email_address").label("title")
... ]).\
...     where(
...         and_(
...             literal_column("users.id") == literal_column("addresses.user_id"),
...             text("users.name BETWEEN 'm' AND 'z'"),
...             text(
...                 "(addresses.email_address LIKE :x OR "
...                 "addresses.email_address LIKE :y)")
...         )
...     ).select_from(table('users')).select_from(table('addresses'))

>>> conn.execute(s, x='%@aol.com', y='%@msn.com').fetchall()
SELECT users.fullname || ? || addresses.email_address AS anon_1
```

```
FROM users, addresses
WHERE users.id = addresses.user_id
AND users.name BETWEEN 'm' AND 'z'
AND (addresses.email_address LIKE ? OR addresses.email_address LIKE ?)
(', ', '%@aol.com', '%@msn.com')
[(u'Wendy Williams, wendy@aol.com',)]
```

Ordering or Grouping by a Label

One place where we sometimes want to use a string as a shortcut is when our statement has some labeled column element that we want to refer to in a place such as the "ORDER BY" or "GROUP BY" clause; other candidates include fields within an "OVER" or "DISTINCT" clause. If we have such a label in our *select()* construct, we can refer to it directly by passing the string straight into `select.order_by()` or `select.group_by()`, among others. This will refer to the named label and also prevent the expression from being rendered twice:

```
>>> from sqlalchemy import func
>>> stmt = select([
...         addresses.c.user_id,
...         func.count(addresses.c.id).label('num_addresses')]).\
...         order_by("num_addresses")

>>> conn.execute(stmt).fetchall()
SELECT addresses.user_id, count(addresses.id) AS num_addresses
FROM addresses ORDER BY num_addresses
()
[(2, 4)]
```

We can use modifiers like *asc()* or *desc()* by passing the string name:

```
>>> from sqlalchemy import func, desc
>>> stmt = select([
...         addresses.c.user_id,
...         func.count(addresses.c.id).label('num_addresses')]).\
...         order_by(desc("num_addresses"))

>>> conn.execute(stmt).fetchall()
SELECT addresses.user_id, count(addresses.id) AS num_addresses
FROM addresses ORDER BY num_addresses DESC
()
[(2, 4)]
```

Note that the string feature here is very much tailored to when we have already used the *label()* method to create a specifically-named label. In other cases, we always want to refer to the *ColumnElement* object directly so that the expression system can make the most effective choices for rendering. Below, we illustrate how using the *ColumnElement* eliminates ambiguity when we want to order by a column name that appears more than once:

```
>>> u1a, u1b = users.alias(), users.alias()
>>> stmt = select([u1a, u1b]).\
...             where(u1a.c.name > u1b.c.name).\
...             order_by(u1a.c.name)  # using "name" here would be ambiguous

>>> conn.execute(stmt).fetchall()
SELECT users_1.id, users_1.name, users_1.fullname, users_2.id,
users_2.name, users_2.fullname
```

```
FROM users AS users_1, users AS users_2
WHERE users_1.name > users_2.name ORDER BY users_1.name
()
[(2, u'wendy', u'Wendy Williams', 1, u'jack', u'Jack Jones')]
```

3.1.11 Using Aliases

The alias in SQL corresponds to a "renamed" version of a table or SELECT statement, which occurs anytime you say "SELECT .. FROM sometable AS someothername". The `AS` creates a new name for the table. Aliases are a key construct as they allow any table or subquery to be referenced by a unique name. In the case of a table, this allows the same table to be named in the FROM clause multiple times. In the case of a SELECT statement, it provides a parent name for the columns represented by the statement, allowing them to be referenced relative to this name.

In SQLAlchemy, any *`Table`*, *`select()`* construct, or other selectable can be turned into an alias using the *`FromClause.alias()`* method, which produces a *`Alias`* construct. As an example, suppose we know that our user `jack` has two particular email addresses. How can we locate jack based on the combination of those two addresses? To accomplish this, we'd use a join to the `addresses` table, once for each address. We create two *`Alias`* constructs against `addresses`, and then use them both within a *`select()`* construct:

```
>>> a1 = addresses.alias()
>>> a2 = addresses.alias()
>>> s = select([users]).\
...        where(and_(
...            users.c.id == a1.c.user_id,
...            users.c.id == a2.c.user_id,
...            a1.c.email_address == 'jack@msn.com',
...            a2.c.email_address == 'jack@yahoo.com'
...        ))
>>> conn.execute(s).fetchall()
SELECT users.id, users.name, users.fullname
FROM users, addresses AS addresses_1, addresses AS addresses_2
WHERE users.id = addresses_1.user_id
    AND users.id = addresses_2.user_id
    AND addresses_1.email_address = ?
    AND addresses_2.email_address = ?
('jack@msn.com', 'jack@yahoo.com')
[(1, u'jack', u'Jack Jones')]
```

Note that the *`Alias`* construct generated the names `addresses_1` and `addresses_2` in the final SQL result. The generation of these names is determined by the position of the construct within the statement. If we created a query using only the second `a2` alias, the name would come out as `addresses_1`. The generation of the names is also *deterministic*, meaning the same SQLAlchemy statement construct will produce the identical SQL string each time it is rendered for a particular dialect.

Since on the outside, we refer to the alias using the *`Alias`* construct itself, we don't need to be concerned about the generated name. However, for the purposes of debugging, it can be specified by passing a string name to the *`FromClause.alias()`* method:

```
>>> a1 = addresses.alias('a1')
```

Aliases can of course be used for anything which you can SELECT from, including SELECT statements themselves. We can self-join the `users` table back to the *`select()`* we've created by making an alias of the entire statement. The `correlate(None)` directive is to avoid SQLAlchemy's attempt to "correlate" the inner `users` table with the outer one:

```
>>> a1 = s.correlate(None).alias()
>>> s = select([users.c.name]).where(users.c.id == a1.c.id)
>>> conn.execute(s).fetchall()
SELECT users.name
FROM users,
    (SELECT users.id AS id, users.name AS name, users.fullname AS fullname
        FROM users, addresses AS addresses_1, addresses AS addresses_2
        WHERE users.id = addresses_1.user_id AND users.id = addresses_2.user_id
        AND addresses_1.email_address = ?
        AND addresses_2.email_address = ?) AS anon_1
WHERE users.id = anon_1.id
('jack@msn.com', 'jack@yahoo.com')
[(u'jack',)]
```

3.1.12 Using Joins

We're halfway along to being able to construct any SELECT expression. The next cornerstone of the SELECT is the JOIN expression. We've already been doing joins in our examples, by just placing two tables in either the columns clause or the where clause of the *select()* construct. But if we want to make a real "JOIN" or "OUTERJOIN" construct, we use the *join()* and *outerjoin()* methods, most commonly accessed from the left table in the join:

```
>>> print(users.join(addresses))
users JOIN addresses ON users.id = addresses.user_id
```

The alert reader will see more surprises; SQLAlchemy figured out how to JOIN the two tables ! The ON condition of the join, as it's called, was automatically generated based on the *ForeignKey* object which we placed on the `addresses` table way at the beginning of this tutorial. Already the `join()` construct is looking like a much better way to join tables.

Of course you can join on whatever expression you want, such as if we want to join on all users who use the same name in their email address as their username:

```
>>> print(users.join(addresses,
...                  addresses.c.email_address.like(users.c.name + '%')
...              )
...  )
users JOIN addresses ON addresses.email_address LIKE (users.name || :name_1)
```

When we create a *select()* construct, SQLAlchemy looks around at the tables we've mentioned and then places them in the FROM clause of the statement. When we use JOINs however, we know what FROM clause we want, so here we make use of the *select_from()* method:

```
>>> s = select([users.c.fullname]).select_from(
...    users.join(addresses,
...               addresses.c.email_address.like(users.c.name + '%'))
...    )
>>> conn.execute(s).fetchall()
SELECT users.fullname
FROM users JOIN addresses ON addresses.email_address LIKE (users.name || ?)
('%',)
[(u'Jack Jones',), (u'Jack Jones',), (u'Wendy Williams',)]
```

The *outerjoin()* method creates `LEFT OUTER JOIN` constructs, and is used in the same way as *join()*:

```
>>> s = select([users.c.fullname]).select_from(users.outerjoin(addresses))
>>> print(s)
SELECT users.fullname
    FROM users
    LEFT OUTER JOIN addresses ON users.id = addresses.user_id
```

That's the output `outerjoin()` produces, unless, of course, you're stuck in a gig using Oracle prior to version 9, and you've set up your engine (which would be using `OracleDialect`) to use Oracle-specific SQL:

```
>>> from sqlalchemy.dialects.oracle import dialect as OracleDialect
>>> print(s.compile(dialect=OracleDialect(use_ansi=False)))
SELECT users.fullname
FROM users, addresses
WHERE users.id = addresses.user_id(+)
```

If you don't know what that SQL means, don't worry ! The secret tribe of Oracle DBAs don't want their black magic being found out ;).

See also:

expression.join()

expression.outerjoin()

Join

3.1.13 Everything Else

The concepts of creating SQL expressions have been introduced. What's left are more variants of the same themes. So now we'll catalog the rest of the important things we'll need to know.

Bind Parameter Objects

Throughout all these examples, SQLAlchemy is busy creating bind parameters wherever literal expressions occur. You can also specify your own bind parameters with your own names, and use the same statement repeatedly. The *bindparam()* construct is used to produce a bound parameter with a given name. While SQLAlchemy always refers to bound parameters by name on the API side, the database dialect converts to the appropriate named or positional style at execution time, as here where it converts to positional for SQLite:

```
>>> from sqlalchemy.sql import bindparam
>>> s = users.select(users.c.name == bindparam('username'))
>>> conn.execute(s, username='wendy').fetchall()
SELECT users.id, users.name, users.fullname
FROM users
WHERE users.name = ?
('wendy',)
[(2, u'wendy', u'Wendy Williams')]
```

Another important aspect of *bindparam()* is that it may be assigned a type. The type of the bind parameter will determine its behavior within expressions and also how the data bound to it is processed before being sent off to the database:

```
>>> s = users.select(users.c.name.like(bindparam('username', type_=String) + text("'%'")))
>>> conn.execute(s, username='wendy').fetchall()
SELECT users.id, users.name, users.fullname
FROM users
WHERE users.name LIKE (? || '%')
('wendy',)
[(2, u'wendy', u'Wendy Williams')]
```

bindparam() constructs of the same name can also be used multiple times, where only a single named value is needed in the execute parameters:

```
>>> s = select([users, addresses]).\
...     where(
...        or_(
...          users.c.name.like(
...                 bindparam('name', type_=String) + text("'%'")),
...          addresses.c.email_address.like(
...                 bindparam('name', type_=String) + text("'@%'"))
...        )
...     ).\
...     select_from(users.outerjoin(addresses)).\
...     order_by(addresses.c.id)
>>> conn.execute(s, name='jack').fetchall()
SELECT users.id, users.name, users.fullname, addresses.id,
    addresses.user_id, addresses.email_address
FROM users LEFT OUTER JOIN addresses ON users.id = addresses.user_id
WHERE users.name LIKE (? || '%') OR addresses.email_address LIKE (? || '@%')
ORDER BY addresses.id
('jack', 'jack')
[(1, u'jack', u'Jack Jones', 1, 1, u'jack@yahoo.com'), (1, u'jack', u'Jack Jones', 2, 1, u'jack@msn.c
```

See also:

bindparam()

Functions

SQL functions are created using the *func* keyword, which generates functions using attribute access:

```
>>> from sqlalchemy.sql import func
>>> print(func.now())
now()

>>> print(func.concat('x', 'y'))
concat(:concat_1, :concat_2)
```

By "generates", we mean that **any** SQL function is created based on the word you choose:

```
>>> print(func.xyz_my_goofy_function())
xyz_my_goofy_function()
```

Certain function names are known by SQLAlchemy, allowing special behavioral rules to be applied. Some for example are "ANSI" functions, which mean they don't get the parenthesis added after them, such as CURRENT_TIMESTAMP:

```
>>> print(func.current_timestamp())
CURRENT_TIMESTAMP
```

Functions are most typically used in the columns clause of a select statement, and can also be labeled as well as given a type. Labeling a function is recommended so that the result can be targeted in a result row based on a string name, and assigning it a type is required when you need result-set processing to occur, such as for Unicode conversion and date conversions. Below, we use the result function `scalar()` to just read the first column of the first row and then close the result; the label, even though present, is not important in this case:

```
>>> conn.execute(
...     select([
...            func.max(addresses.c.email_address, type_=String).
...                label('maxemail')
...           ])
...     ).scalar()
SELECT max(addresses.email_address) AS maxemail
FROM addresses
()
u'www@www.org'
```

Databases such as PostgreSQL and Oracle which support functions that return whole result sets can be assembled into selectable units, which can be used in statements. Such as, a database function `calculate()` which takes the parameters `x` and `y`, and returns three columns which we'd like to name `q`, `z` and `r`, we can construct using "lexical" column objects as well as bind parameters:

```
>>> from sqlalchemy.sql import column
>>> calculate = select([column('q'), column('z'), column('r')]).\
...        select_from(
...             func.calculate(
...                    bindparam('x'),
...                    bindparam('y')
...                )
...             )
>>> calc = calculate.alias()
>>> print(select([users]).where(users.c.id > calc.c.z))
SELECT users.id, users.name, users.fullname
FROM users, (SELECT q, z, r
FROM calculate(:x, :y)) AS anon_1
WHERE users.id > anon_1.z
```

If we wanted to use our `calculate` statement twice with different bind parameters, the *`unique_params()`* function will create copies for us, and mark the bind parameters as "unique" so that conflicting names are isolated. Note we also make two separate aliases of our selectable:

```
>>> calc1 = calculate.alias('c1').unique_params(x=17, y=45)
>>> calc2 = calculate.alias('c2').unique_params(x=5, y=12)
>>> s = select([users]).\
...         where(users.c.id.between(calc1.c.z, calc2.c.z))
>>> print(s)
SELECT users.id, users.name, users.fullname
FROM users,
    (SELECT q, z, r FROM calculate(:x_1, :y_1)) AS c1,
    (SELECT q, z, r FROM calculate(:x_2, :y_2)) AS c2
WHERE users.id BETWEEN c1.z AND c2.z
```

```
>>> s.compile().params
{u'x_2': 5, u'y_2': 12, u'y_1': 45, u'x_1': 17}
```

See also:

func

Window Functions

Any *FunctionElement*, including functions generated by *func*, can be turned into a "window function", that is an OVER clause, using the *FunctionElement.over()* method:

```
>>> s = select([
...         users.c.id,
...         func.row_number().over(order_by=users.c.name)
...     ])
>>> print(s)
SELECT users.id, row_number() OVER (ORDER BY users.name) AS anon_1
FROM users
```

See also:

over()

FunctionElement.over()

Unions and Other Set Operations

Unions come in two flavors, UNION and UNION ALL, which are available via module level functions *union()* and *union_all()*:

```
>>> from sqlalchemy.sql import union
>>> u = union(
...     addresses.select().
...             where(addresses.c.email_address == 'foo@bar.com'),
...    addresses.select().
...             where(addresses.c.email_address.like('%@yahoo.com')),
... ).order_by(addresses.c.email_address)

>>> conn.execute(u).fetchall()
SELECT addresses.id, addresses.user_id, addresses.email_address
FROM addresses
WHERE addresses.email_address = ?
UNION
SELECT addresses.id, addresses.user_id, addresses.email_address
FROM addresses
WHERE addresses.email_address LIKE ? ORDER BY addresses.email_address
('foo@bar.com', '%@yahoo.com')
[(1, 1, u'jack@yahoo.com')]
```

Also available, though not supported on all databases, are *intersect()*, *intersect_all()*, *except_()*, and *except_all()*:

```
>>> from sqlalchemy.sql import except_
>>> u = except_(
...     addresses.select().
...             where(addresses.c.email_address.like('%@%.com')),
...     addresses.select().
...             where(addresses.c.email_address.like('%@msn.com'))
... )

>>> conn.execute(u).fetchall()
SELECT addresses.id, addresses.user_id, addresses.email_address
FROM addresses
WHERE addresses.email_address LIKE ?
EXCEPT
SELECT addresses.id, addresses.user_id, addresses.email_address
FROM addresses
WHERE addresses.email_address LIKE ?
('%@%.com', '%@msn.com')
[(1, 1, u'jack@yahoo.com'), (4, 2, u'wendy@aol.com')]
```

A common issue with so-called "compound" selectables arises due to the fact that they nest with parenthesis. SQLite in particular doesn't like a statement that starts with parenthesis. So when nesting a "compound" inside a "compound", it's often necessary to apply `.alias().select()` to the first element of the outermost compound, if that element is also a compound. For example, to nest a "union" and a "select" inside of "except_", SQLite will want the "union" to be stated as a subquery:

```
>>> u = except_(
...     union(
...         addresses.select().
...             where(addresses.c.email_address.like('%@yahoo.com')),
...         addresses.select().
...             where(addresses.c.email_address.like('%@msn.com'))
...      ).alias().select(),    # apply subquery here
...     addresses.select(addresses.c.email_address.like('%@msn.com'))
... )
>>> conn.execute(u).fetchall()
SELECT anon_1.id, anon_1.user_id, anon_1.email_address
FROM (SELECT addresses.id AS id, addresses.user_id AS user_id,
    addresses.email_address AS email_address
    FROM addresses
    WHERE addresses.email_address LIKE ?
    UNION
    SELECT addresses.id AS id,
        addresses.user_id AS user_id,
        addresses.email_address AS email_address
    FROM addresses
    WHERE addresses.email_address LIKE ?) AS anon_1
EXCEPT
SELECT addresses.id, addresses.user_id, addresses.email_address
FROM addresses
WHERE addresses.email_address LIKE ?
('%@yahoo.com', '%@msn.com', '%@msn.com')
[(1, 1, u'jack@yahoo.com')]
```

See also:

union()

union_all()

intersect()

intersect_all()

except_()

except_all()

Scalar Selects

A scalar select is a SELECT that returns exactly one row and one column. It can then be used as a column expression. A scalar select is often a correlated subquery, which relies upon the enclosing SELECT statement in order to acquire at least one of its FROM clauses.

The *select()* construct can be modified to act as a column expression by calling either the *as_scalar()* or *label()* method:

```
>>> stmt = select([func.count(addresses.c.id)]).\
...             where(users.c.id == addresses.c.user_id).\
...             as_scalar()
```

The above construct is now a *ScalarSelect* object, and is no longer part of the *FromClause* hierarchy; it instead is within the *ColumnElement* family of expression constructs. We can place this construct the same as any other column within another *select()*:

```
>>> conn.execute(select([users.c.name, stmt])).fetchall()
SELECT users.name, (SELECT count(addresses.id) AS count_1
FROM addresses
WHERE users.id = addresses.user_id) AS anon_1
FROM users
()
[(u'jack', 2), (u'wendy', 2)]
```

To apply a non-anonymous column name to our scalar select, we create it using *SelectBase.label()* instead:

```
>>> stmt = select([func.count(addresses.c.id)]).\
...             where(users.c.id == addresses.c.user_id).\
...             label("address_count")
>>> conn.execute(select([users.c.name, stmt])).fetchall()
SELECT users.name, (SELECT count(addresses.id) AS count_1
FROM addresses
WHERE users.id = addresses.user_id) AS address_count
FROM users
()
[(u'jack', 2), (u'wendy', 2)]
```

See also:

Select.as_scalar()

Select.label()

Correlated Subqueries

Notice in the examples on *Scalar Selects*, the FROM clause of each embedded select did not contain the `users` table in its FROM clause. This is because SQLAlchemy automatically correlates embedded FROM objects to that of an

enclosing query, if present, and if the inner SELECT statement would still have at least one FROM clause of its own. For example:

```
>>> stmt = select([addresses.c.user_id]).\
...             where(addresses.c.user_id == users.c.id).\
...             where(addresses.c.email_address == 'jack@yahoo.com')
>>> enclosing_stmt = select([users.c.name]).where(users.c.id == stmt)
>>> conn.execute(enclosing_stmt).fetchall()
SELECT users.name
FROM users
WHERE users.id = (SELECT addresses.user_id
    FROM addresses
    WHERE addresses.user_id = users.id
    AND addresses.email_address = ?)
('jack@yahoo.com',)
[(u'jack',)]
```

Auto-correlation will usually do what's expected, however it can also be controlled. For example, if we wanted a statement to correlate only to the `addresses` table but not the `users` table, even if both were present in the enclosing SELECT, we use the *`correlate()`* method to specify those FROM clauses that may be correlated:

```
>>> stmt = select([users.c.id]).\
...             where(users.c.id == addresses.c.user_id).\
...             where(users.c.name == 'jack').\
...             correlate(addresses)
>>> enclosing_stmt = select(
...         [users.c.name, addresses.c.email_address]).\
...     select_from(users.join(addresses)).\
...     where(users.c.id == stmt)
>>> conn.execute(enclosing_stmt).fetchall()
SELECT users.name, addresses.email_address
 FROM users JOIN addresses ON users.id = addresses.user_id
 WHERE users.id = (SELECT users.id
 FROM users
 WHERE users.id = addresses.user_id AND users.name = ?)
 ('jack',)
 [(u'jack', u'jack@yahoo.com'), (u'jack', u'jack@msn.com')]
```

To entirely disable a statement from correlating, we can pass `None` as the argument:

```
>>> stmt = select([users.c.id]).\
...             where(users.c.name == 'wendy').\
...             correlate(None)
>>> enclosing_stmt = select([users.c.name]).\
...     where(users.c.id == stmt)
>>> conn.execute(enclosing_stmt).fetchall()
SELECT users.name
 FROM users
 WHERE users.id = (SELECT users.id
  FROM users
  WHERE users.name = ?)
('wendy',)
[(u'wendy',)]
```

We can also control correlation via exclusion, using the *`Select.correlate_except()`* method. Such as, we can write our SELECT for the `users` table by telling it to correlate all FROM clauses except for `users`:

```
>>> stmt = select([users.c.id]).\
...             where(users.c.id == addresses.c.user_id).\
...             where(users.c.name == 'jack').\
...             correlate_except(users)
>>> enclosing_stmt = select(
...         [users.c.name, addresses.c.email_address]).\
...     select_from(users.join(addresses)).\
...     where(users.c.id == stmt)
>>> conn.execute(enclosing_stmt).fetchall()
SELECT users.name, addresses.email_address
 FROM users JOIN addresses ON users.id = addresses.user_id
 WHERE users.id = (SELECT users.id
 FROM users
 WHERE users.id = addresses.user_id AND users.name = ?)
 ('jack',)
 [(u'jack', u'jack@yahoo.com'), (u'jack', u'jack@msn.com')]
```

Ordering, Grouping, Limiting, Offset...ing...

Ordering is done by passing column expressions to the `order_by()` method:

```
>>> stmt = select([users.c.name]).order_by(users.c.name)
>>> conn.execute(stmt).fetchall()
SELECT users.name
FROM users ORDER BY users.name
()
[(u'jack',), (u'wendy',)]
```

Ascending or descending can be controlled using the *asc()* and *desc()* modifiers:

```
>>> stmt = select([users.c.name]).order_by(users.c.name.desc())
>>> conn.execute(stmt).fetchall()
SELECT users.name
FROM users ORDER BY users.name DESC
()
[(u'wendy',), (u'jack',)]
```

Grouping refers to the GROUP BY clause, and is usually used in conjunction with aggregate functions to establish groups of rows to be aggregated. This is provided via the `group_by()` method:

```
>>> stmt = select([users.c.name, func.count(addresses.c.id)]).\
...             select_from(users.join(addresses)).\
...             group_by(users.c.name)
>>> conn.execute(stmt).fetchall()
SELECT users.name, count(addresses.id) AS count_1
FROM users JOIN addresses
    ON users.id = addresses.user_id
GROUP BY users.name
()
[(u'jack', 2), (u'wendy', 2)]
```

HAVING can be used to filter results on an aggregate value, after GROUP BY has been applied. It's available here via the *having()* method:

```
>>> stmt = select([users.c.name, func.count(addresses.c.id)]).\
...             select_from(users.join(addresses)).\
...             group_by(users.c.name).\
...             having(func.length(users.c.name) > 4)
>>> conn.execute(stmt).fetchall()
SELECT users.name, count(addresses.id) AS count_1
FROM users JOIN addresses
    ON users.id = addresses.user_id
GROUP BY users.name
HAVING length(users.name) > ?
(4,)
[(u'wendy', 2)]
```

A common system of dealing with duplicates in composed SELECT statements is the DISTINCT modifier. A simple DISTINCT clause can be added using the `Select.distinct()` method:

```
>>> stmt = select([users.c.name]).\
...             where(addresses.c.email_address.
...                    contains(users.c.name)).\
...             distinct()
>>> conn.execute(stmt).fetchall()
SELECT DISTINCT users.name
FROM users, addresses
WHERE (addresses.email_address LIKE '%%' || users.name || '%%')
()
[(u'jack',), (u'wendy',)]
```

Most database backends support a system of limiting how many rows are returned, and the majority also feature a means of starting to return rows after a given "offset". While common backends like Postgresql, MySQL and SQLite support LIMIT and OFFSET keywords, other backends need to refer to more esoteric features such as "window functions" and row ids to achieve the same effect. The `limit()` and `offset()` methods provide an easy abstraction into the current backend's methodology:

```
>>> stmt = select([users.c.name, addresses.c.email_address]).\
...             select_from(users.join(addresses)).\
...             limit(1).offset(1)
>>> conn.execute(stmt).fetchall()
SELECT users.name, addresses.email_address
FROM users JOIN addresses ON users.id = addresses.user_id
 LIMIT ? OFFSET ?
(1, 1)
[(u'jack', u'jack@msn.com')]
```

3.1.14 Inserts, Updates and Deletes

We've seen `insert()` demonstrated earlier in this tutorial. Where `insert()` produces INSERT, the `update()` method produces UPDATE. Both of these constructs feature a method called `values()` which specifies the VALUES or SET clause of the statement.

The `values()` method accommodates any column expression as a value:

```
>>> stmt = users.update().\
...             values(fullname="Fullname: " + users.c.name)
>>> conn.execute(stmt)
```

```
UPDATE users SET fullname=(? || users.name)
('Fullname: ',)
COMMIT
<sqlalchemy.engine.result.ResultProxy object at 0x...>
```

When using *insert()* or *update()* in an "execute many" context, we may also want to specify named bound parameters which we can refer to in the argument list. The two constructs will automatically generate bound placeholders for any column names passed in the dictionaries sent to *execute()* at execution time. However, if we wish to use explicitly targeted named parameters with composed expressions, we need to use the *bindparam()* construct. When using *bindparam()* with *insert()* or *update()*, the names of the table's columns themselves are reserved for the "automatic" generation of bind names. We can combine the usage of implicitly available bind names and explicitly named parameters as in the example below:

```
>>> stmt = users.insert().\
...         values(name=bindparam('_name') + " .. name")
>>> conn.execute(stmt, [
...        {'id':4, '_name':'name1'},
...        {'id':5, '_name':'name2'},
...        {'id':6, '_name':'name3'},
...     ])
INSERT INTO users (id, name) VALUES (?, (? || ?))
((4, 'name1', ' .. name'), (5, 'name2', ' .. name'), (6, 'name3', ' .. name'))
COMMIT
<sqlalchemy.engine.result.ResultProxy object at 0x...>
```

An UPDATE statement is emitted using the *update()* construct. This works much like an INSERT, except there is an additional WHERE clause that can be specified:

```
>>> stmt = users.update().\
...             where(users.c.name == 'jack').\
...             values(name='ed')

>>> conn.execute(stmt)
UPDATE users SET name=? WHERE users.name = ?
('ed', 'jack')
COMMIT
<sqlalchemy.engine.result.ResultProxy object at 0x...>
```

When using *update()* in an "executemany" context, we may wish to also use explicitly named bound parameters in the WHERE clause. Again, *bindparam()* is the construct used to achieve this:

```
>>> stmt = users.update().\
...             where(users.c.name == bindparam('oldname')).\
...             values(name=bindparam('newname'))
>>> conn.execute(stmt, [
...     {'oldname':'jack', 'newname':'ed'},
...     {'oldname':'wendy', 'newname':'mary'},
...     {'oldname':'jim', 'newname':'jake'},
...     ])
UPDATE users SET name=? WHERE users.name = ?
(('ed', 'jack'), ('mary', 'wendy'), ('jake', 'jim'))
COMMIT
<sqlalchemy.engine.result.ResultProxy object at 0x...>
```

Correlated Updates

A correlated update lets you update a table using selection from another table, or the same table:

```
>>> stmt = select([addresses.c.email_address]).\
...             where(addresses.c.user_id == users.c.id).\
...             limit(1)
>>> conn.execute(users.update().values(fullname=stmt))
UPDATE users SET fullname=(SELECT addresses.email_address
    FROM addresses
    WHERE addresses.user_id = users.id
    LIMIT ? OFFSET ?)
(1, 0)
COMMIT
<sqlalchemy.engine.result.ResultProxy object at 0x...>
```

Multiple Table Updates

New in version 0.7.4.

The Postgresql, Microsoft SQL Server, and MySQL backends all support UPDATE statements that refer to multiple tables. For PG and MSSQL, this is the "UPDATE FROM" syntax, which updates one table at a time, but can reference additional tables in an additional "FROM" clause that can then be referenced in the WHERE clause directly. On MySQL, multiple tables can be embedded into a single UPDATE statement separated by a comma. The SQLAlchemy *update()* construct supports both of these modes implicitly, by specifying multiple tables in the WHERE clause:

```
stmt = users.update().\
        values(name='ed wood').\
        where(users.c.id == addresses.c.id).\
        where(addresses.c.email_address.startswith('ed%'))
conn.execute(stmt)
```

The resulting SQL from the above statement would render as:

```
UPDATE users SET name=:name FROM addresses
WHERE users.id = addresses.id AND
addresses.email_address LIKE :email_address_1 || '%%'
```

When using MySQL, columns from each table can be assigned to in the SET clause directly, using the dictionary form passed to *Update.values()*:

```
stmt = users.update().\
        values({
            users.c.name:'ed wood',
            addresses.c.email_address:'ed.wood@foo.com'
        }).\
        where(users.c.id == addresses.c.id).\
        where(addresses.c.email_address.startswith('ed%'))
```

The tables are referenced explicitly in the SET clause:

```
UPDATE users, addresses SET addresses.email_address=%s,
        users.name=%s WHERE users.id = addresses.id
        AND addresses.email_address LIKE concat(%s, '%%')
```

SQLAlchemy doesn't do anything special when these constructs are used on a non-supporting database. The `UPDATE FROM` syntax generates by default when multiple tables are present, and the statement will be rejected by the database if this syntax is not supported.

Parameter-Ordered Updates

The default behavior of the *update()* construct when rendering the SET clauses is to render them using the column ordering given in the originating *Table* object. This is an important behavior, since it means that the rendering of a particular UPDATE statement with particular columns will be rendered the same each time, which has an impact on query caching systems that rely on the form of the statement, either client side or server side. Since the parameters themselves are passed to the *Update.values()* method as Python dictionary keys, there is no other fixed ordering available.

However in some cases, the order of parameters rendered in the SET clause of an UPDATE statement can be significant. The main example of this is when using MySQL and providing updates to column values based on that of other column values. The end result of the following statement:

```
UPDATE some_table SET x = y + 10, y = 20
```

Will have a different result than:

```
UPDATE some_table SET y = 20, x = y + 10
```

This because on MySQL, the individual SET clauses are fully evaluated on a per-value basis, as opposed to on a per-row basis, and as each SET clause is evaluated, the values embedded in the row are changing.

To suit this specific use case, the *preserve_parameter_order* flag may be used. When using this flag, we supply a **Python list of 2-tuples** as the argument to the *Update.values()* method:

```
stmt = some_table.update(preserve_parameter_order=True).\
    values([(some_table.c.y, 20), (some_table.c.x, some_table.c.y + 10)])
```

The list of 2-tuples is essentially the same structure as a Python dictionary except it is ordered. Using the above form, we are assured that the "y" column's SET clause will render first, then the "x" column's SET clause.

New in version 1.0.10: Added support for explicit ordering of UPDATE parameters using the *preserve_parameter_order* flag.

Deletes

Finally, a delete. This is accomplished easily enough using the *delete()* construct:

```
>>> conn.execute(addresses.delete())
DELETE FROM addresses
()
COMMIT
<sqlalchemy.engine.result.ResultProxy object at 0x...>

>>> conn.execute(users.delete().where(users.c.name > 'm'))
DELETE FROM users WHERE users.name > ?
('m',)
COMMIT
<sqlalchemy.engine.result.ResultProxy object at 0x...>
```

Matched Row Counts

Both of `update()` and `delete()` are associated with *matched row counts*. This is a number indicating the number of rows that were matched by the WHERE clause. Note that by "matched", this includes rows where no UPDATE actually took place. The value is available as `rowcount`:

```
>>> result = conn.execute(users.delete())
DELETE FROM users
()
COMMIT
>>> result.rowcount
1
```

3.1.15 Further Reference

Expression Language Reference: *SQL Statements and Expressions API*

Database Metadata Reference: *Describing Databases with MetaData*

Engine Reference: Engine Configuration

Connection Reference: *Working with Engines and Connections*

Types Reference: *Column and Data Types*

3.2 SQL Statements and Expressions API

This section presents the API reference for the SQL Expression Language. For a full introduction to its usage, see *SQL Expression Language Tutorial*.

3.2.1 Column Elements and Expressions

The most fundamental part of the SQL expression API are the "column elements", which allow for basic SQL expression support. The core of all SQL expression constructs is the `ClauseElement`, which is the base for several sub-branches. The `ColumnElement` class is the fundamental unit used to construct any kind of typed SQL expression.

`sqlalchemy.sql.expression.`**`all_`**`(`*expr*`)`

Produce an ALL expression.

This may apply to an array type for some dialects (e.g. postgresql), or to a subquery for others (e.g. mysql). e.g.:

```
# postgresql '5 = ALL (somearray)'
expr = 5 == all_(mytable.c.somearray)

# mysql '5 = ALL (SELECT value FROM table)'
expr = 5 == all_(select([table.c.value]))
```

New in version 1.1.

See also:

`expression.any_()`

sqlalchemy.sql.expression.**and_**(**clauses*)

Produce a conjunction of expressions joined by AND.

E.g.:

```
from sqlalchemy import and_

stmt = select([users_table]).where(
                and_(
                    users_table.c.name == 'wendy',
                    users_table.c.enrolled == True
                )
            )
```

The *and_()* conjunction is also available using the Python & operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
                (users_table.c.name == 'wendy') &
                (users_table.c.enrolled == True)
            )
```

The *and_()* operation is also implicit in some cases; the *Select.where()* method for example can be invoked multiple times against a statement, which will have the effect of each clause being combined using *and_()*:

```
stmt = select([users_table]).\
            where(users_table.c.name == 'wendy').\
            where(users_table.c.enrolled == True)
```

See also:

or_()

sqlalchemy.sql.expression.**any_**(*expr*)

Produce an ANY expression.

This may apply to an array type for some dialects (e.g. postgresql), or to a subquery for others (e.g. mysql). e.g.:

```
# postgresql '5 = ANY (somearray)'
expr = 5 == any_(mytable.c.somearray)

# mysql '5 = ANY (SELECT value FROM table)'
expr = 5 == any_(select([table.c.value]))
```

New in version 1.1.

See also:

expression.all_()

sqlalchemy.sql.expression.**asc**(*column*)

Produce an ascending ORDER BY clause element.

e.g.:

```
from sqlalchemy import asc
stmt = select([users_table]).order_by(asc(users_table.c.name))
```

will produce SQL as:

```
SELECT id, name FROM user ORDER BY name ASC
```

The *asc()* function is a standalone version of the *ColumnElement.asc()* method available on all SQL expressions, e.g.:

```
stmt = select([users_table]).order_by(users_table.c.name.asc())
```

Parameters **column** – A *ColumnElement* (e.g. scalar SQL expression) with which to apply the *asc()* operation.

See also:

desc()

nullsfirst()

nullslast()

Select.order_by()

sqlalchemy.sql.expression.**between**(*expr*, *lower_bound*, *upper_bound*, *symmetric=False*)

Produce a BETWEEN predicate clause.

E.g.:

```
from sqlalchemy import between
stmt = select([users_table]).where(between(users_table.c.id, 5, 7))
```

Would produce SQL resembling:

```
SELECT id, name FROM user WHERE id BETWEEN :id_1 AND :id_2
```

The *between()* function is a standalone version of the *ColumnElement.between()* method available on all SQL expressions, as in:

```
stmt = select([users_table]).where(users_table.c.id.between(5, 7))
```

All arguments passed to *between()*, including the left side column expression, are coerced from Python scalar values if a the value is not a *ColumnElement* subclass. For example, three fixed values can be compared as in:

```
print(between(5, 3, 7))
```

Which would produce:

```
:param_1 BETWEEN :param_2 AND :param_3
```

Parameters

- **expr** – a column expression, typically a *ColumnElement* instance or alternatively a Python scalar expression to be coerced into a column expression, serving as the left side of the BETWEEN expression.
- **lower_bound** – a column or Python scalar expression serving as the lower bound of the right side of the BETWEEN expression.
- **upper_bound** – a column or Python scalar expression serving as the upper bound of the right side of the BETWEEN expression.
- **symmetric** – if True, will render " BETWEEN SYMMETRIC ". Note that not all databases support this syntax.

New in version 0.9.5.

See also:

ColumnElement.between()

sqlalchemy.sql.expression.**bindparam**(*key*, *value=symbol('NO_ARG')*, *type_=None*, *unique=False*, *required=symbol('NO_ARG')*, *quote=None*, *callable_=None*, *isoutparam=False*, *_compared_to_operator=None*, *_compared_to_type=None*)

Produce a "bound expression".

The return value is an instance of *BindParameter*; this is a *ColumnElement* subclass which represents a so-called "placeholder" value in a SQL expression, the value of which is supplied at the point at which the statement in executed against a database connection.

In SQLAlchemy, the *bindparam()* construct has the ability to carry along the actual value that will be ultimately used at expression time. In this way, it serves not just as a "placeholder" for eventual population, but also as a means of representing so-called "unsafe" values which should not be rendered directly in a SQL statement, but rather should be passed along to the DBAPI as values which need to be correctly escaped and potentially handled for type-safety.

When using *bindparam()* explicitly, the use case is typically one of traditional deferment of parameters; the *bindparam()* construct accepts a name which can then be referred to at execution time:

```
from sqlalchemy import bindparam

stmt = select([users_table]).\
            where(users_table.c.name == bindparam('username'))
```

The above statement, when rendered, will produce SQL similar to:

```
SELECT id, name FROM user WHERE name = :username
```

In order to populate the value of `:username` above, the value would typically be applied at execution time to a method like *Connection.execute()*:

```
result = connection.execute(stmt, username='wendy')
```

Explicit use of *bindparam()* is also common when producing UPDATE or DELETE statements that are to be invoked multiple times, where the WHERE criterion of the statement is to change on each invocation, such as:

```
stmt = (users_table.update().
        where(user_table.c.name == bindparam('username')).
        values(fullname=bindparam('fullname'))
        )

connection.execute(
    stmt, [{"username": "wendy", "fullname": "Wendy Smith"},
           {"username": "jack", "fullname": "Jack Jones"},
           ]
)
```

SQLAlchemy's Core expression system makes wide use of *bindparam()* in an implicit sense. It is typical that Python literal values passed to virtually all SQL expression functions are coerced into fixed *bindparam()* constructs. For example, given a comparison operation such as:

```
expr = users_table.c.name == 'Wendy'
```

The above expression will produce a *BinaryExpression* construct, where the left side is the *Column* object representing the `name` column, and the right side is a *BindParameter* representing the literal value:

```
print(repr(expr.right))
BindParameter('%(4327771088 name)s', 'Wendy', type_=String())
```

The expression above will render SQL such as:

```
user.name = :name_1
```

Where the `:name_1` parameter name is an anonymous name. The actual string `Wendy` is not in the rendered string, but is carried along where it is later used within statement execution. If we invoke a statement like the following:

```
stmt = select([users_table]).where(users_table.c.name == 'Wendy')
result = connection.execute(stmt)
```

We would see SQL logging output as:

```
SELECT "user".id, "user".name
FROM "user"
WHERE "user".name = %(name_1)s
{'name_1': 'Wendy'}
```

Above, we see that `Wendy` is passed as a parameter to the database, while the placeholder `:name_1` is rendered in the appropriate form for the target database, in this case the Postgresql database.

Similarly, *bindparam()* is invoked automatically when working with CRUD statements as far as the "VALUES" portion is concerned. The *insert()* construct produces an `INSERT` expression which will, at statement execution time, generate bound placeholders based on the arguments passed, as in:

```
stmt = users_table.insert()
result = connection.execute(stmt, name='Wendy')
```

The above will produce SQL output as:

```
INSERT INTO "user" (name) VALUES (%(name)s)
{'name': 'Wendy'}
```

The *Insert* construct, at compilation/execution time, rendered a single *bindparam()* mirroring the column name `name` as a result of the single `name` parameter we passed to the *Connection.execute()* method.

Parameters

- **key** – the key (e.g. the name) for this bind param. Will be used in the generated SQL statement for dialects that use named parameters. This value may be modified when part of a compilation operation, if other *BindParameter* objects exist with the same key, or if its length is too long and truncation is required.
- **value** – Initial value for this bind param. Will be used at statement execution time as the value for this parameter passed to the DBAPI, if no other value is indicated to the statement execution method for this particular parameter name. Defaults to `None`.
- **callable_** – A callable function that takes the place of "value". The function will be called at statement execution time to determine the ultimate value. Used for scenarios where the actual bind value cannot be determined at the point at which the clause construct is created, but embedded bind values are still desirable.
- **type_** – A *TypeEngine* class or instance representing an optional datatype for this *bindparam()*. If not passed, a type may be determined automatically for the bind,

based on the given value; for example, trivial Python types such as `str`, `int`, `bool` may result in the *String*, *Integer* or *Boolean* types being autoamtically selected.

The type of a *bindparam()* is significant especially in that the type will apply pre-processing to the value before it is passed to the database. For example, a *bindparam()* which refers to a datetime value, and is specified as holding the *DateTime* type, may apply conversion needed to the value (such as stringification on SQLite) before passing the value to the database.

- **unique** – if True, the key name of this *BindParameter* will be modified if another *BindParameter* of the same name already has been located within the containing expression. This flag is used generally by the internals when producing so-called "anonymous" bound expressions, it isn't generally applicable to explicitly-named *bindparam()* constructs.
- **required** – If `True`, a value is required at execution time. If not passed, it defaults to `True` if neither *bindparam.value* or *bindparam.callable* were passed. If either of these parameters are present, then *bindparam.required* defaults to `False`.

  Changed in version 0.8: If the `required` flag is not specified, it will be set automatically to `True` or `False` depending on whether or not the `value` or `callable` parameters were specified.
- **quote** – True if this parameter name requires quoting and is not currently known as a SQLAlchemy reserved word; this currently only applies to the Oracle backend, where bound names must sometimes be quoted.
- **isoutparam** – if True, the parameter should be treated like a stored procedure "OUT" parameter. This applies to backends such as Oracle which support OUT parameters.

See also:

Bind Parameter Objects

Insert Expressions

outparam()

`sqlalchemy.sql.expression.`**`case`**(*whens*, *value=None*, *else_=None*)

Produce a `CASE` expression.

The `CASE` construct in SQL is a conditional object that acts somewhat analogously to an "if/then" construct in other languages. It returns an instance of *Case*.

case() in its usual form is passed a list of "when" constructs, that is, a list of conditions and results as tuples:

```
from sqlalchemy import case

stmt = select([users_table]).\
            where(
                case(
                    [
                        (users_table.c.name == 'wendy', 'W'),
                        (users_table.c.name == 'jack', 'J')
                    ],
                    else_='E'
                )
            )
```

The above statement will produce SQL resembling:

```
SELECT id, name FROM user
WHERE CASE
    WHEN (name = :name_1) THEN :param_1
    WHEN (name = :name_2) THEN :param_2
    ELSE :param_3
END
```

When simple equality expressions of several values against a single parent column are needed, *case()* also has a "shorthand" format used via the *case.value* parameter, which is passed a column expression to be compared. In this form, the *case.whens* parameter is passed as a dictionary containing expressions to be compared against keyed to result expressions. The statement below is equivalent to the preceding statement:

```
stmt = select([users_table]).\
            where(
                case(
                    {"wendy": "W", "jack": "J"},
                    value=users_table.c.name,
                    else_='E'
                )
            )
```

The values which are accepted as result values in *case.whens* as well as with *case.else_* are coerced from Python literals into *bindparam()* constructs. SQL expressions, e.g. *ColumnElement* constructs, are accepted as well. To coerce a literal string expression into a constant expression rendered inline, use the *literal_column()* construct, as in:

```
from sqlalchemy import case, literal_column

case(
    [
        (
            orderline.c.qty > 100,
            literal_column("'greaterthan100'")
        ),
        (
            orderline.c.qty > 10,
            literal_column("'greaterthan10'")
        )
    ],
    else_=literal_column("'lessthan10'")
)
```

The above will render the given constants without using bound parameters for the result values (but still for the comparison values), as in:

```
CASE
    WHEN (orderline.qty > :qty_1) THEN 'greaterthan100'
    WHEN (orderline.qty > :qty_2) THEN 'greaterthan10'
    ELSE 'lessthan10'
END
```

Parameters

- **whens** – The criteria to be compared against, *case.whens* accepts two different forms, based on whether or not *case.value* is used.

  In the first form, it accepts a list of 2-tuples; each 2-tuple consists of `(<sql expression>, <value>)`, where the SQL expression is a boolean expression and "value" is a resulting value, e.g.:

```
case([
    (users_table.c.name == 'wendy', 'W'),
    (users_table.c.name == 'jack', 'J')
])
```

In the second form, it accepts a Python dictionary of comparison values mapped to a resulting value; this form requires *case.value* to be present, and values will be compared using the == operator, e.g.:

```
case(
    {"wendy": "W", "jack": "J"},
    value=users_table.c.name
)
```

- **value** – An optional SQL expression which will be used as a fixed "comparison point" for candidate values within a dictionary passed to *case.whens*.
- **else_** – An optional SQL expression which will be the evaluated result of the CASE construct if all expressions within *case.whens* evaluate to false. When omitted, most databases will produce a result of NULL if none of the "when" expressions evaluate to true.

sqlalchemy.sql.expression.**cast**(*expression*, *type_*)

Produce a CAST expression.

cast() returns an instance of *Cast*.

E.g.:

```
from sqlalchemy import cast, Numeric

stmt = select([
            cast(product_table.c.unit_price, Numeric(10, 4))
        ])
```

The above statement will produce SQL resembling:

```
SELECT CAST(unit_price AS NUMERIC(10, 4)) FROM product
```

The *cast()* function performs two distinct functions when used. The first is that it renders the CAST expression within the resulting SQL string. The second is that it associates the given type (e.g. *TypeEngine* class or instance) with the column expression on the Python side, which means the expression will take on the expression operator behavior associated with that type, as well as the bound-value handling and result-row-handling behavior of the type.

Changed in version 0.9.0: *cast()* now applies the given type to the expression such that it takes effect on the bound-value, e.g. the Python-to-database direction, in addition to the result handling, e.g. database-to-Python, direction.

An alternative to *cast()* is the *type_coerce()* function. This function performs the second task of associating an expression with a specific type, but does not render the CAST expression in SQL.

Parameters

- **expression** – A SQL expression, such as a *ColumnElement* expression or a Python string which will be coerced into a bound literal value.
- **type_** – A *TypeEngine* class or instance indicating the type to which the CAST should apply.

See also:

type_coerce() - Python-side type coercion without emitting CAST.

sqlalchemy.sql.expression.**column**(*text*, *type_=None*, *is_literal=False*, *_selectable=None*)

Produce a *ColumnClause* object.

The *ColumnClause* is a lightweight analogue to the *Column* class. The *column()* function can be invoked with just a name alone, as in:

```
from sqlalchemy import column

id, name = column("id"), column("name")
stmt = select([id, name]).select_from("user")
```

The above statement would produce SQL like:

```
SELECT id, name FROM user
```

Once constructed, *column()* may be used like any other SQL expression element such as within *select()* constructs:

```
from sqlalchemy.sql import column

id, name = column("id"), column("name")
stmt = select([id, name]).select_from("user")
```

The text handled by *column()* is assumed to be handled like the name of a database column; if the string contains mixed case, special characters, or matches a known reserved word on the target backend, the column expression will render using the quoting behavior determined by the backend. To produce a textual SQL expression that is rendered exactly without any quoting, use *literal_column()* instead, or pass `True` as the value of *column.is_literal*. Additionally, full SQL statements are best handled using the *text()* construct.

column() can be used in a table-like fashion by combining it with the *table()* function (which is the lightweight analogue to *Table*) to produce a working table construct with minimal boilerplate:

```
from sqlalchemy import table, column, select

user = table("user",
        column("id"),
        column("name"),
        column("description"),
)

stmt = select([user.c.description]).where(user.c.name == 'wendy')
```

A *column()* / *table()* construct like that illustrated above can be created in an ad-hoc fashion and is not associated with any *schema.MetaData*, DDL, or events, unlike its *Table* counterpart.

Changed in version 1.0.0: *expression.column()* can now be imported from the plain `sqlalchemy` namespace like any other SQL element.

Parameters

- **text** – the text of the element.
- **type** – *types.TypeEngine* object which can associate this *ColumnClause* with a type.
- **is_literal** – if True, the *ColumnClause* is assumed to be an exact expression that will be delivered to the output with no quoting rules applied regardless of case sensi-

tive settings. the *literal_column()* function essentially invokes *column()* while passing `is_literal=True`.

See also:

Column

literal_column()

table()

text()

Using More Specific Text with table(), literal_column(), and column()

`sqlalchemy.sql.expression.`**`collate`**(*expression*, *collation*)

Return the clause `expression COLLATE collation`.

e.g.:

```
collate(mycolumn, 'utf8_bin')
```

produces:

```
mycolumn COLLATE utf8_bin
```

`sqlalchemy.sql.expression.`**`desc`**(*column*)

Produce a descending `ORDER BY` clause element.

e.g.:

```
from sqlalchemy import desc

stmt = select([users_table]).order_by(desc(users_table.c.name))
```

will produce SQL as:

```
SELECT id, name FROM user ORDER BY name DESC
```

The *desc()* function is a standalone version of the *ColumnElement.desc()* method available on all SQL expressions, e.g.:

```
stmt = select([users_table]).order_by(users_table.c.name.desc())
```

Parameters **column** – A *ColumnElement* (e.g. scalar SQL expression) with which to apply the *desc()* operation.

See also:

asc()

nullsfirst()

nullslast()

Select.order_by()

`sqlalchemy.sql.expression.`**`distinct`**(*expr*)

Produce an column-expression-level unary `DISTINCT` clause.

This applies the `DISTINCT` keyword to an individual column expression, and is typically contained within an aggregate function, as in:

```
from sqlalchemy import distinct, func
stmt = select([func.count(distinct(users_table.c.name))])
```

The above would produce an expression resembling:

```
SELECT COUNT(DISTINCT name) FROM user
```

The *distinct()* function is also available as a column-level method, e.g. *ColumnElement.distinct()*, as in:

```
stmt = select([func.count(users_table.c.name.distinct())])
```

The *distinct()* operator is different from the *Select.distinct()* method of *Select*, which produces a `SELECT` statement with `DISTINCT` applied to the result set as a whole, e.g. a `SELECT DISTINCT` expression. See that method for further information.

See also:

ColumnElement.distinct()

Select.distinct()

func

`sqlalchemy.sql.expression.`**`extract`**(*field*, *expr*, ***kwargs*)

Return a *Extract* construct.

This is typically available as *extract()* as well as `func.extract` from the *func* namespace.

`sqlalchemy.sql.expression.`**`false`**`()`

Return a *False_* construct.

E.g.:

```
>>> from sqlalchemy import false
>>> print select([t.c.x]).where(false())
SELECT x FROM t WHERE false
```

A backend which does not support true/false constants will render as an expression against 1 or 0:

```
>>> print select([t.c.x]).where(false())
SELECT x FROM t WHERE 0 = 1
```

The *true()* and *false()* constants also feature "short circuit" operation within an *and_()* or *or_()* conjunction:

```
>>> print select([t.c.x]).where(or_(t.c.x > 5, true()))
SELECT x FROM t WHERE true

>>> print select([t.c.x]).where(and_(t.c.x > 5, false()))
SELECT x FROM t WHERE false
```

Changed in version 0.9: *true()* and *false()* feature better integrated behavior within conjunctions and on dialects that don't support true/false constants.

See also:

true()

`sqlalchemy.sql.expression.`**`func = <sqlalchemy.sql.functions._FunctionGenerator object>`**

Generate SQL function expressions.

func is a special object instance which generates SQL functions based on name-based attributes, e.g.:

```
>>> print func.count(1)
count(:param_1)
```

The element is a column-oriented SQL element like any other, and is used in that way:

```
>>> print select([func.count(table.c.id)])
SELECT count(sometable.id) FROM sometable
```

Any name can be given to `func`. If the function name is unknown to SQLAlchemy, it will be rendered exactly as is. For common SQL functions which SQLAlchemy is aware of, the name may be interpreted as a *generic function* which will be compiled appropriately to the target database:

```
>>> print func.current_timestamp()
CURRENT_TIMESTAMP
```

To call functions which are present in dot-separated packages, specify them in the same manner:

```
>>> print func.stats.yield_curve(5, 10)
stats.yield_curve(:yield_curve_1, :yield_curve_2)
```

SQLAlchemy can be made aware of the return type of functions to enable type-specific lexical and result-based behavior. For example, to ensure that a string-based function returns a Unicode value and is similarly treated as a string in expressions, specify `Unicode` as the type:

```
>>> print func.my_string(u'hi', type_=Unicode) + ' ' + \
... func.my_string(u'there', type_=Unicode)
my_string(:my_string_1) || :my_string_2 || my_string(:my_string_3)
```

The object returned by a `func` call is usually an instance of `Function`. This object meets the "column" interface, including comparison and labeling functions. The object can also be passed the `execute()` method of a `Connection` or `Engine`, where it will be wrapped inside of a SELECT statement first:

```
print connection.execute(func.current_timestamp()).scalar()
```

In a few exception cases, the `func` accessor will redirect a name to a built-in expression such as `cast()` or `extract()`, as these names have well-known meaning but are not exactly the same as "functions" from a SQLAlchemy perspective.

New in version 0.8: `func` can return non-function expression constructs for common quasi-functional names like `cast()` and `extract()`.

Functions which are interpreted as "generic" functions know how to calculate their return type automatically. For a listing of known generic functions, see *SQL and Generic Functions*.

Note: The `func` construct has only limited support for calling standalone "stored procedures", especially those with special parameterization concerns.

See the section *Calling Stored Procedures* for details on how to use the DBAPI-level `callproc()` method for fully traditional stored procedures.

`sqlalchemy.sql.expression.`**`funcfilter`**(*func*, **criterion*)

Produce a `FunctionFilter` object against a function.

Used against aggregate and window functions, for database backends that support the "FILTER" clause.

E.g.:

```
from sqlalchemy import funcfilter
funcfilter(func.count(1), MyClass.name == 'some name')
```

Would produce "COUNT(1) FILTER (WHERE myclass.name = 'some name')".

This function is also available from the `func` construct itself via the `FunctionElement.filter()` method.

New in version 1.0.0.

See also:

`FunctionElement.filter()`

`sqlalchemy.sql.expression.`**`label`**(*name*, *element*, *type_=None*)

Return a `Label` object for the given `ColumnElement`.

A label changes the name of an element in the columns clause of a `SELECT` statement, typically via the `AS` SQL keyword.

This functionality is more conveniently available via the `ColumnElement.label()` method on `ColumnElement`.

Parameters

- **name** – label name
- **obj** – a `ColumnElement`.

`sqlalchemy.sql.expression.`**`literal`**(*value*, *type_=None*)

Return a literal clause, bound to a bind parameter.

Literal clauses are created automatically when non- `ClauseElement` objects (such as strings, ints, dates, etc.) are used in a comparison operation with a `ColumnElement` subclass, such as a `Column` object. Use this function to force the generation of a literal clause, which will be created as a `BindParameter` with a bound value.

Parameters

- **value** – the value to be bound. Can be any Python object supported by the underlying DB-API, or is translatable via the given type argument.
- **type_** – an optional `TypeEngine` which will provide bind-parameter translation for this literal.

`sqlalchemy.sql.expression.`**`literal_column`**(*text*, *type_=None*)

Produce a `ColumnClause` object that has the `column.is_literal` flag set to True.

`literal_column()` is similar to `column()`, except that it is more often used as a "standalone" column expression that renders exactly as stated; while `column()` stores a string name that will be assumed to be part of a table and may be quoted as such, `literal_column()` can be that, or any other arbitrary column-oriented expression.

Parameters

- **text** – the text of the expression; can be any SQL expression. Quoting rules will not be applied. To specify a column-name expression which should be subject to quoting rules, use the `column()` function.
- **type_** – an optional `TypeEngine` object which will provide result-set translation and additional expression semantics for this column. If left as None the type will be NullType.

See also:

`column()`

`text()`

Using More Specific Text with table(), literal_column(), and column()

sqlalchemy.sql.expression.**not_**(*clause*)

Return a negation of the given clause, i.e. `NOT(clause)`.

The ~ operator is also overloaded on all *ColumnElement* subclasses to produce the same result.

sqlalchemy.sql.expression.**null**()

Return a constant *Null* construct.

sqlalchemy.sql.expression.**nullsfirst**(*column*)

Produce the `NULLS FIRST` modifier for an `ORDER BY` expression.

nullsfirst() is intended to modify the expression produced by *asc()* or *desc()*, and indicates how NULL values should be handled when they are encountered during ordering:

```
from sqlalchemy import desc, nullsfirst

stmt = select([users_table]).\
            order_by(nullsfirst(desc(users_table.c.name)))
```

The SQL expression from the above would resemble:

```
SELECT id, name FROM user ORDER BY name DESC NULLS FIRST
```

Like *asc()* and *desc()*, *nullsfirst()* is typically invoked from the column expression itself using *ColumnElement.nullsfirst()*, rather than as its standalone function version, as in:

```
stmt = (select([users_table]).
        order_by(users_table.c.name.desc().nullsfirst())
        )
```

See also:

asc()

desc()

nullslast()

Select.order_by()

sqlalchemy.sql.expression.**nullslast**(*column*)

Produce the `NULLS LAST` modifier for an `ORDER BY` expression.

nullslast() is intended to modify the expression produced by *asc()* or *desc()*, and indicates how NULL values should be handled when they are encountered during ordering:

```
from sqlalchemy import desc, nullslast

stmt = select([users_table]).\
            order_by(nullslast(desc(users_table.c.name)))
```

The SQL expression from the above would resemble:

```
SELECT id, name FROM user ORDER BY name DESC NULLS LAST
```

Like *asc()* and *desc()*, *nullslast()* is typically invoked from the column expression itself using *ColumnElement.nullslast()*, rather than as its standalone function version, as in:

```
stmt = select([users_table]).\
            order_by(users_table.c.name.desc().nullslast())
```

See also:

asc()

desc()

nullsfirst()

Select.order_by()

sqlalchemy.sql.expression.**or_**(**clauses*)

Produce a conjunction of expressions joined by `OR`.

E.g.:

```
from sqlalchemy import or_

stmt = select([users_table]).where(
                or_(
                    users_table.c.name == 'wendy',
                    users_table.c.name == 'jack'
                )
            )
```

The *or_()* conjunction is also available using the Python `|` operator (though note that compound expressions need to be parenthesized in order to function with Python operator precedence behavior):

```
stmt = select([users_table]).where(
                (users_table.c.name == 'wendy') |
                (users_table.c.name == 'jack')
            )
```

See also:

and_()

sqlalchemy.sql.expression.**outparam**(*key*, *type_=None*)

Create an 'OUT' parameter for usage in functions (stored procedures), for databases which support them.

The `outparam` can be used like a regular function parameter. The "output" value will be available from the *ResultProxy* object via its `out_parameters` attribute, which returns a dictionary containing the values.

sqlalchemy.sql.expression.**over**(*element*, *partition_by=None*, *order_by=None*)

Produce an *Over* object against a function.

Used against aggregate or so-called "window" functions, for database backends that support window functions.

over() is usually called using the *FunctionElement.over()* method, e.g.:

```
func.row_number().over(order_by='x')
```

Would produce `ROW_NUMBER() OVER(ORDER BY x)`.

Parameters

- **element** – a *FunctionElement*, *WithinGroup*, or other compatible construct.
- **partition_by** – a column element or string, or a list of such, that will be used as the PARTITION BY clause of the OVER construct.
- **order_by** – a column element or string, or a list of such, that will be used as the ORDER BY clause of the OVER construct.

This function is also available from the *func* construct itself via the *FunctionElement.over()* method.

New in version 0.7.

See also:

expression.func

expression.within_group()

sqlalchemy.sql.expression.**text**(*text*, *bind=None*, *bindparams=None*, *typemap=None*, *autocommit=None*)

Construct a new *TextClause* clause, representing a textual SQL string directly.

E.g.:

```
from sqlalchemy import text

t = text("SELECT * FROM users")
result = connection.execute(t)
```

The advantages *text()* provides over a plain string are backend-neutral support for bind parameters, per-statement execution options, as well as bind parameter and result-column typing behavior, allowing SQLAlchemy type constructs to play a role when executing a statement that is specified literally. The construct can also be provided with a `.c` collection of column elements, allowing it to be embedded in other SQL expression constructs as a subquery.

Bind parameters are specified by name, using the format `:name`. E.g.:

```
t = text("SELECT * FROM users WHERE id=:user_id")
result = connection.execute(t, user_id=12)
```

For SQL statements where a colon is required verbatim, as within an inline string, use a backslash to escape:

```
t = text("SELECT * FROM users WHERE name='\:username'")
```

The *TextClause* construct includes methods which can provide information about the bound parameters as well as the column values which would be returned from the textual statement, assuming it's an executable SELECT type of statement. The *TextClause.bindparams()* method is used to provide bound parameter detail, and *TextClause.columns()* method allows specification of return columns including names and types:

```
t = text("SELECT * FROM users WHERE id=:user_id").\
        bindparams(user_id=7).\
        columns(id=Integer, name=String)

for id, name in connection.execute(t):
    print(id, name)
```

The *text()* construct is used internally in cases when a literal string is specified for part of a larger query, such as when a string is specified to the *Select.where()* method of *Select*. In those cases, the same bind parameter syntax is applied:

```
s = select([users.c.id, users.c.name]).where("id=:user_id")
result = connection.execute(s, user_id=12)
```

Using *text()* explicitly usually implies the construction of a full, standalone statement. As such, SQLAlchemy refers to it as an *Executable* object, and it supports the *Executable.execution_options()* method. For example, a *text()* construct that should be subject to "autocommit" can be set explicitly so using the *Connection.execution_options.autocommit* option:

```
t = text("EXEC my_procedural_thing()").\
        execution_options(autocommit=True)
```

Note that SQLAlchemy's usual "autocommit" behavior applies to *text()* constructs implicitly - that is, statements which begin with a phrase such as `INSERT`, `UPDATE`, `DELETE`, or a variety of other phrases specific to certain backends, will be eligible for autocommit if no transaction is in progress.

Parameters

- **text** – the text of the SQL statement to be created. use `:<param>` to specify bind parameters; they will be compiled to their engine-specific format.
- **autocommit** – Deprecated. Use .execution_options(autocommit=<True|False>) to set the autocommit option.
- **bind** – an optional connection or engine to be used for this text query.
- **bindparams** – Deprecated. A list of *bindparam()* instances used to provide information about parameters embedded in the statement. This argument now invokes the *TextClause.bindparams()* method on the construct before returning it. E.g.:

```
stmt = text("SELECT * FROM table WHERE id=:id",
            bindparams=[bindparam('id', value=5, type_=Integer)])
```

Is equivalent to:

```
stmt = text("SELECT * FROM table WHERE id=:id").\
            bindparams(bindparam('id', value=5, type_=Integer))
```

Deprecated since version 0.9.0: the *TextClause.bindparams()* method supersedes the `bindparams` argument to *text()*.

- **typemap** – Deprecated. A dictionary mapping the names of columns represented in the columns clause of a `SELECT` statement to type objects, which will be used to perform post-processing on columns within the result set. This parameter now invokes the *TextClause.columns()* method, which returns a *TextAsFrom* construct that gains a `.c` collection and can be embedded in other expressions. E.g.:

```
stmt = text("SELECT * FROM table",
                typemap={'id': Integer, 'name': String},
            )
```

Is equivalent to:

```
stmt = text("SELECT * FROM table").columns(id=Integer,
                                           name=String)
```

Or alternatively:

```
from sqlalchemy.sql import column
stmt = text("SELECT * FROM table").columns(
                      column('id', Integer),
                      column('name', String)
                    )
```

Deprecated since version 0.9.0: the *TextClause.columns()* method supersedes the `typemap` argument to *text()*.

`sqlalchemy.sql.expression.`**`true`**`()`

Return a constant *True_* construct.

E.g.:

```
>>> from sqlalchemy import true
>>> print select([t.c.x]).where(true())
SELECT x FROM t WHERE true
```

A backend which does not support true/false constants will render as an expression against 1 or 0:

```
>>> print select([t.c.x]).where(true())
SELECT x FROM t WHERE 1 = 1
```

The *true()* and *false()* constants also feature "short circuit" operation within an *and_()* or *or_()* conjunction:

```
>>> print select([t.c.x]).where(or_(t.c.x > 5, true()))
SELECT x FROM t WHERE true

>>> print select([t.c.x]).where(and_(t.c.x > 5, false()))
SELECT x FROM t WHERE false
```

Changed in version 0.9: *true()* and *false()* feature better integrated behavior within conjunctions and on dialects that don't support true/false constants.

See also:

false()

sqlalchemy.sql.expression.**tuple_**(**clauses*, ***kw*)

Return a *Tuple*.

Main usage is to produce a composite IN construct:

```
from sqlalchemy import tuple_

tuple_(table.c.col1, table.c.col2).in_(
    [(1, 2), (5, 12), (10, 19)]
)
```

> **Warning:** The composite IN construct is not supported by all backends, and is currently known to work on Postgresql and MySQL, but not SQLite. Unsupported backends will raise a subclass of *DBAPIError* when such an expression is invoked.

sqlalchemy.sql.expression.**type_coerce**(*expression*, *type_*)

Associate a SQL expression with a particular type, without rendering `CAST`.

E.g.:

```
from sqlalchemy import type_coerce

stmt = select([
    type_coerce(log_table.date_string, StringDateTime())
])
```

The above construct will produce a *TypeCoerce* object, which renders SQL that labels the expression, but otherwise does not modify its value on the SQL side:

```
SELECT date_string AS anon_1 FROM log
```

When result rows are fetched, the `StringDateTime` type will be applied to result rows on behalf of the `date_string` column. The rationale for the "anon_1" label is so that the type-coerced column remains separate in the list of result columns vs. other type-coerced or direct values of the target column. In order to provide a named label for the expression, use *ColumnElement.label()*:

```
stmt = select([
    type_coerce(
        log_table.date_string, StringDateTime()).label('date')
])
```

A type that features bound-value handling will also have that behavior take effect when literal values or *bindparam()* constructs are passed to *type_coerce()* as targets. For example, if a type implements the *TypeEngine.bind_expression()* method or *TypeEngine.bind_processor()* method or equivalent, these functions will take effect at statement compilation/execution time when a literal value is passed, as in:

```
# bound-value handling of MyStringType will be applied to the
# literal value "some string"
stmt = select([type_coerce("some string", MyStringType)])
```

type_coerce() is similar to the *cast()* function, except that it does not render the `CAST` expression in the resulting statement.

Parameters

- **expression** – A SQL expression, such as a *ColumnElement* expression or a Python string which will be coerced into a bound literal value.
- **type_** – A *TypeEngine* class or instance indicating the type to which the expression is coerced.

See also:

cast()

`sqlalchemy.sql.expression.`**`within_group`**(*element*, **order_by*)

Produce a *WithinGroup* object against a function.

Used against so-called "ordered set aggregate" and "hypothetical set aggregate" functions, including *percentile_cont*, *rank*, *dense_rank*, etc.

within_group() is usually called using the *FunctionElement.within_group()* method, e.g.:

```
from sqlalchemy import within_group
stmt = select([
    department.c.id,
    func.percentile_cont(0.5).within_group(
        department.c.salary.desc()
    )
])
```

The above statement would produce SQL similar to `SELECT department.id, percentile_cont(0.5) WITHIN GROUP (ORDER BY department.salary DESC)`.

Parameters

- **element** – a *FunctionElement* construct, typically generated by *func*.
- ***order_by** – one or more column elements that will be used as the ORDER BY clause of the WITHIN GROUP construct.

New in version 1.1.

See also:

expression.func

expression.over()

class `sqlalchemy.sql.expression.`**`BinaryExpression`**(*left*, *right*, *operator*, *type_=None*, *negate=None*, *modifiers=None*)

Bases: *sqlalchemy.sql.expression.ColumnElement*

Represent an expression that is `LEFT <operator> RIGHT`.

A `BinaryExpression` is generated automatically whenever two column expressions are used in a Python binary expression:

```
>>> from sqlalchemy.sql import column
>>> column('a') + column('b')
<sqlalchemy.sql.expression.BinaryExpression object at 0x101029dd0>
>>> print column('a') + column('b')
a + b
```

compare(*other*, ***kw*)

Compare this `BinaryExpression` against the given `BinaryExpression`.

class sqlalchemy.sql.expression.**BindParameter**(*key*, *value=symbol('NO_ARG')*, *type_=None*, *unique=False*, *required=symbol('NO_ARG')*, *quote=None*, *callable_=None*, *isoutparam=False*, *_compared_to_operator=None*, *_compared_to_type=None*)

Bases: `sqlalchemy.sql.expression.ColumnElement`

Represent a "bound expression".

`BindParameter` is invoked explicitly using the `bindparam()` function, as in:

```
from sqlalchemy import bindparam

stmt = select([users_table]).\
            where(users_table.c.name == bindparam('username'))
```

Detailed discussion of how `BindParameter` is used is at `bindparam()`.

See also:

`bindparam()`

__init__(*key*, *value=symbol('NO_ARG')*, *type_=None*, *unique=False*, *required=symbol('NO_ARG')*, *quote=None*, *callable_=None*, *isoutparam=False*, *_compared_to_operator=None*, *_compared_to_type=None*)

Construct a new `BindParameter` object.

This constructor is mirrored as a public API function; see `bindparam()` for a full usage and argument description.

compare(*other*, ***kw*)

Compare this `BindParameter` to the given clause.

effective_value

Return the value of this bound parameter, taking into account if the `callable` parameter was set.

The `callable` value will be evaluated and returned if present, else `value`.

class sqlalchemy.sql.expression.**Case**(*whens*, *value=None*, *else_=None*)

Bases: `sqlalchemy.sql.expression.ColumnElement`

Represent a `CASE` expression.

`Case` is produced using the `case()` factory function, as in:

```
from sqlalchemy import case

stmt = select([users_table]).\
            where(
                case(
```

```
                        [
                            (users_table.c.name == 'wendy', 'W'),
                            (users_table.c.name == 'jack', 'J')
                        ],
                        else_='E'
                    )
                )
```

Details on `Case` usage is at `case()`.

See also:

`case()`

__init__(*whens*, *value=None*, *else_=None*)

Construct a new `Case` object.

This constructor is mirrored as a public API function; see `case()` for a full usage and argument description.

class sqlalchemy.sql.expression.**Cast**(*expression*, *type_*)

Bases: `sqlalchemy.sql.expression.ColumnElement`

Represent a `CAST` expression.

`Cast` is produced using the `cast()` factory function, as in:

```
from sqlalchemy import cast, Numeric

stmt = select([
            cast(product_table.c.unit_price, Numeric(10, 4))
        ])
```

Details on `Cast` usage is at `cast()`.

See also:

`cast()`

__init__(*expression*, *type_*)

Construct a new `Cast` object.

This constructor is mirrored as a public API function; see `cast()` for a full usage and argument description.

class sqlalchemy.sql.expression.**ClauseElement**

Bases: `sqlalchemy.sql.visitors.Visitable`

Base class for elements of a programmatically constructed SQL expression.

compare(*other*, ***kw*)

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

**kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see `ColumnElement`)

compile(*bind=None*, *dialect=None*, ***kw*)

Compile this SQL expression.

The return value is a `Compiled` object. Calling `str()` or `unicode()` on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the `params` accessor.

Parameters

- **bind** – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.
- **column_keys** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- **dialect** – A `Dialect` instance from which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- **inline** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

How do I render SQL expressions as strings, possibly with bound parameters inlined?

get_children(***kwargs*)

Return immediate child elements of this `ClauseElement`.

This is used for visit traversal.

**kwargs may contain flags that change the collection that is returned, for example to return a subset of items in order to cut down on larger traversals, or to return child items from a different context (such as schema-level collections instead of clause-level).

params(**optionaldict*, ***kwargs*)

Return a copy with `bindparam()` elements replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

self_group(*against=None*)

Apply a 'grouping' to this `ClauseElement`.

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by `select()` constructs when placed into the FROM clause of another `select()`. (Note that subqueries should be normally created using the `Select.alias()` method, as many platforms require nested SELECT statements to be named).

As expressions are composed together, the application of `self_group()` is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like `x OR (y AND z)` - AND takes precedence over OR.

The base `self_group()` method of `ClauseElement` just returns self.

unique_params(**optionaldict*, ***kwargs*)

Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

class sqlalchemy.sql.expression.**ClauseList**(**clauses*, ***kwargs*)

Bases: `sqlalchemy.sql.expression.ClauseElement`

Describe a list of clauses, separated by an operator.

By default, is comma-separated, such as a column listing.

compare(*other*, ***kw*)

Compare this `ClauseList` to the given `ClauseList`, including a comparison of all the clause items.

class sqlalchemy.sql.expression.**ColumnClause**(*text*, *type_=None*, *is_literal=False*, *_selectable=None*)

Bases: `sqlalchemy.sql.expression.Immutable`, `sqlalchemy.sql.expression.ColumnElement`

Represents a column expression from any textual string.

The `ColumnClause`, a lightweight analogue to the `Column` class, is typically invoked using the `column()` function, as in:

```
from sqlalchemy import column

id, name = column("id"), column("name")
stmt = select([id, name]).select_from("user")
```

The above statement would produce SQL like:

```
SELECT id, name FROM user
```

`ColumnClause` is the immediate superclass of the schema-specific `Column` object. While the `Column` class has all the same capabilities as `ColumnClause`, the `ColumnClause` class is usable by itself in those cases where behavioral requirements are limited to simple SQL expression generation. The object has none of the associations with schema-level metadata or with execution-time behavior that `Column` does, so in that sense is a "lightweight" version of `Column`.

Full details on `ColumnClause` usage is at `column()`.

See also:

`column()`

`Column`

__init__(*text*, *type_=None*, *is_literal=False*, *_selectable=None*)

Construct a new `ColumnClause` object.

This constructor is mirrored as a public API function; see *column()* for a full usage and argument description.

class sqlalchemy.sql.expression.**ColumnCollection**(**columns*)

Bases: sqlalchemy.util._collections.OrderedProperties

An ordered dictionary that stores a list of ColumnElement instances.

Overrides the __eq__() method to produce SQL clauses between sets of correlated columns.

add(*column*)

Add a column to this collection.

The key attribute of the column will be used as the hash key for this dictionary.

replace(*column*)

add the given column to this collection, removing unaliased versions of this column as well as existing columns with the same key.

e.g.:

```
t = Table('sometable', metadata, Column('col1', Integer))
t.columns.replace(Column('col1', Integer, key='columnone'))
```

will remove the original 'col1' from the collection, and add the new column under the name 'columnname'.

Used by schema.Column to override columns during table reflection.

class sqlalchemy.sql.expression.**ColumnElement**

Bases: *sqlalchemy.sql.operators.ColumnOperators*, *sqlalchemy.sql.expression.ClauseElement*

Represent a column-oriented SQL expression suitable for usage in the "columns" clause, WHERE clause etc. of a statement.

While the most familiar kind of *ColumnElement* is the *Column* object, *ColumnElement* serves as the basis for any unit that may be present in a SQL expression, including the expressions themselves, SQL functions, bound parameters, literal expressions, keywords such as NULL, etc. *ColumnElement* is the ultimate base class for all such elements.

A wide variety of SQLAlchemy Core functions work at the SQL expression level, and are intended to accept instances of *ColumnElement* as arguments. These functions will typically document that they accept a "SQL expression" as an argument. What this means in terms of SQLAlchemy usually refers to an input which is either already in the form of a *ColumnElement* object, or a value which can be **coerced** into one. The coercion rules followed by most, but not all, SQLAlchemy Core functions with regards to SQL expressions are as follows:

- a literal Python value, such as a string, integer or floating point value, boolean, datetime, Decimal object, or virtually any other Python object, will be coerced into a "literal bound value". This generally means that a *bindparam()* will be produced featuring the given value embedded into the construct; the resulting *BindParameter* object is an instance of *ColumnElement*. The Python value will ultimately be sent to the DBAPI at execution time as a paramterized argument to the execute() or executemany() methods, after SQLAlchemy type-specific converters (e.g. those provided by any associated *TypeEngine* objects) are applied to the value.
- any special object value, typically ORM-level constructs, which feature a method called __clause_element__(). The Core expression system looks for this method when an object of otherwise unknown type is passed to a function that is looking to coerce the argument into a *ColumnElement* expression. The __clause_element__() method, if present, should return a *ColumnElement* instance. The primary use of __clause_element__() within SQLAlchemy is that of class-bound attributes on ORM-mapped classes; a User class which contains a mapped attribute named .name will have a method User.name.__clause_element__() which when invoked returns the *Column* called name associated with the mapped table.

- The Python `None` value is typically interpreted as `NULL`, which in SQLAlchemy Core produces an instance of *null()*.

A *ColumnElement* provides the ability to generate new *ColumnElement* objects using Python expressions. This means that Python operators such as ==, != and < are overloaded to mimic SQL operations, and allow the instantiation of further *ColumnElement* instances which are composed from other, more fundamental *ColumnElement* objects. For example, two *ColumnClause* objects can be added together with the addition operator + to produce a *BinaryExpression*. Both *ColumnClause* and *BinaryExpression* are subclasses of *ColumnElement*:

```
>>> from sqlalchemy.sql import column
>>> column('a') + column('b')
<sqlalchemy.sql.expression.BinaryExpression object at 0x101029dd0>
>>> print column('a') + column('b')
a + b
```

See also:

Column

expression.column()

__eq__(*other*)

inherited from the __eq__() method of ColumnOperators

Implement the == operator.

In a column context, produces the clause `a = b`. If the target is `None`, produces `a IS NULL`.

__init__

inherited from the __init__ attribute of object

x.__init__(...) initializes x; see help(type(x)) for signature

__le__(*other*)

inherited from the __le__() method of ColumnOperators

Implement the <= operator.

In a column context, produces the clause `a <= b`.

__lt__(*other*)

inherited from the __lt__() method of ColumnOperators

Implement the < operator.

In a column context, produces the clause `a < b`.

__ne__(*other*)

inherited from the __ne__() method of ColumnOperators

Implement the != operator.

In a column context, produces the clause `a != b`. If the target is `None`, produces `a IS NOT NULL`.

all_()

inherited from the all_() method of ColumnOperators

Produce a *all_()* clause against the parent object.

New in version 1.1.

anon_label

provides a constant 'anonymous label' for this ColumnElement.

This is a label() expression which will be named at compile time. The same label() is returned each time anon_label is called so that expressions can reference anon_label multiple times, producing the same label name at compile time.

the compiler uses this function automatically at compile time for expressions that are known to be 'unnamed' like binary expressions and function calls.

any_()
inherited from the `any_()` *method of* `ColumnOperators`

Produce a `any_()` clause against the parent object.

New in version 1.1.

asc()
inherited from the `asc()` *method of* `ColumnOperators`

Produce a `asc()` clause against the parent object.

base_columns

between(*cleft*, *cright*, *symmetric=False*)
inherited from the `between()` *method of* `ColumnOperators`

Produce a `between()` clause against the parent object, given the lower and upper range.

bind = None

cast(*type_*)
Produce a type cast, i.e. `CAST(<expression> AS <type>)`.

This is a shortcut to the `cast()` function.

New in version 1.0.7.

collate(*collation*)
inherited from the `collate()` *method of* `ColumnOperators`

Produce a `collate()` clause against the parent object, given the collation string.

comparator

compare(*other*, *use_proxies=False*, *equivalents=None*, ***kw*)
Compare this ColumnElement to another.

Special arguments understood:

Parameters

- **use_proxies** – when True, consider two columns that share a common base column as equivalent (i.e. shares_lineage())
- **equivalents** – a dictionary of columns as keys mapped to sets of columns. If the given "other" column is present in this dictionary, if any of the columns in the corresponding set() pass the comparison test, the result is True. This is used to expand the comparison to other columns that may be known to be equivalent to this one via foreign key or other criterion.

compile(*bind=None*, *dialect=None*, ***kw*)
inherited from the `compile()` *method of* `ClauseElement`

Compile this SQL expression.

The return value is a `Compiled` object. Calling `str()` or `unicode()` on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the `params` accessor.

Parameters

- **bind** – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.
- **column_keys** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- **dialect** – A `Dialect` instance from which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- **inline** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

How do I render SQL expressions as strings, possibly with bound parameters inlined?

concat(*other*)

inherited from the `concat()` *method of* `ColumnOperators`

Implement the 'concat' operator.

In a column context, produces the clause `a || b`, or uses the `concat()` operator on MySQL.

contains(*other*, ***kwargs*)

inherited from the `contains()` *method of* `ColumnOperators`

Implement the 'contains' operator.

In a column context, produces the clause `LIKE '%<other>%'`

desc()

inherited from the `desc()` *method of* `ColumnOperators`

Produce a `desc()` clause against the parent object.

description = None

distinct()

inherited from the `distinct()` *method of* `ColumnOperators`

Produce a `distinct()` clause against the parent object.

endswith(*other*, ***kwargs*)
inherited from the `endswith()` *method of* `ColumnOperators`

Implement the 'endswith' operator.

In a column context, produces the clause `LIKE '%<other>'`

expression
Return a column expression.

Part of the inspection interface; returns self.

foreign_keys = []

get_children(***kwargs*)
inherited from the `get_children()` *method of* `ClauseElement`

Return immediate child elements of this `ClauseElement`.

This is used for visit traversal.

**kwargs may contain flags that change the collection that is returned, for example to return a subset of items in order to cut down on larger traversals, or to return child items from a different context (such as schema-level collections instead of clause-level).

ilike(*other*, *escape=None*)
inherited from the `ilike()` *method of* `ColumnOperators`

Implement the `ilike` operator.

In a column context, produces the clause `a ILIKE other`.

E.g.:

```
select([sometable]).where(sometable.c.column.ilike("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the `ESCAPE` keyword, e.g.:

```
somecolumn.ilike("foo/%bar", escape="/")
```

See also:

`ColumnOperators.like()`

in_(*other*)
inherited from the `in_()` *method of* `ColumnOperators`

Implement the `in` operator.

In a column context, produces the clause `a IN other`. "other" may be a tuple/list of column expressions, or a `select()` construct.

is_(*other*)
inherited from the `is_()` *method of* `ColumnOperators`

Implement the `IS` operator.

Normally, `IS` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.isnot()

is_clause_element = **True**

is_selectable = **False**

isnot(*other*)

inherited from the isnot() method of ColumnOperators

Implement the `IS NOT` operator.

Normally, `IS NOT` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS NOT` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.is_()

key = **None**

the 'key' that in some circumstances refers to this object in a Python namespace.

This typically refers to the "key" of the column as present in the `.c` collection of a selectable, e.g. sometable.c["somekey"] would return a Column with a .key of "somekey".

label(*name*)

Produce a column label, i.e. `<columnname> AS <name>`.

This is a shortcut to the *label()* function.

if 'name' is None, an anonymous label name will be generated.

like(*other*, *escape=None*)

inherited from the like() method of ColumnOperators

Implement the `like` operator.

In a column context, produces the clause `a LIKE other`.

E.g.:

```
select([sometable]).where(sometable.c.column.like("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the `ESCAPE` keyword, e.g.:

```
somecolumn.like("foo/%bar", escape="/")
```

See also:

ColumnOperators.ilike()

match(*other*, ***kwargs*)

inherited from the match() method of ColumnOperators

Implements a database-specific 'match' operator.

match() attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

- Postgresql - renders `x @@ to_tsquery(y)`
- MySQL - renders `MATCH (x) AGAINST (y IN BOOLEAN MODE)`
- Oracle - renders `CONTAINS(x, y)`
- other backends may provide special implementations.
- Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

notilike(*other*, *escape=None*)

inherited from the `notilike()` *method of* `ColumnOperators`

implement the `NOT ILIKE` operator.

This is equivalent to using negation with `ColumnOperators.ilike()`, i.e. `~x.ilike(y)`.

New in version 0.8.

See also:

`ColumnOperators.ilike()`

notin_(*other*)

inherited from the `notin_()` *method of* `ColumnOperators`

implement the `NOT IN` operator.

This is equivalent to using negation with `ColumnOperators.in_()`, i.e. `~x.in_(y)`.

New in version 0.8.

See also:

`ColumnOperators.in_()`

notlike(*other*, *escape=None*)

inherited from the `notlike()` *method of* `ColumnOperators`

implement the `NOT LIKE` operator.

This is equivalent to using negation with `ColumnOperators.like()`, i.e. `~x.like(y)`.

New in version 0.8.

See also:

`ColumnOperators.like()`

nullsfirst()

inherited from the `nullsfirst()` *method of* `ColumnOperators`

Produce a `nullsfirst()` clause against the parent object.

nullslast()

inherited from the `nullslast()` *method of* `ColumnOperators`

Produce a `nullslast()` clause against the parent object.

op(*opstring*, *precedence=0*, *is_comparison=False*)

inherited from the `op()` *method of* `Operators`

produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

This function can also be used to make bitwise operators explicit. For example:

```
somecolumn.op('&')(0xff)
```

is a bitwise AND of the value in `somecolumn`.

Parameters

- **operator** – a string which will be output as the infix operator between this element and the expression passed to the generated function.
- **precedence** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of `0` is lower than all operators except for the comma (`,`) and `AS` operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.
- **is_comparison** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like ==, >, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

  New in version 0.9.2: - added the `Operators.op.is_comparison` flag.

See also:

Redefining and Creating New Operators

Using custom operators in join conditions

operate(*op*, **other*, ***kwargs*)

params(**optionaldict*, ***kwargs*)

inherited from the `params()` *method of* `ClauseElement`

Return a copy with `bindparam()` elements replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

primary_key = False

proxy_set

reverse_operate(*op*, *other*, ***kwargs*)

self_group(*against=None*)

shares_lineage(*othercolumn*)

Return True if the given `ColumnElement` has a common ancestor to this `ColumnElement`.

startswith(*other*, ***kwargs*)
inherited from the `startswith()` *method of* `ColumnOperators`

Implement the `startwith` operator.

In a column context, produces the clause `LIKE '<other>%'`

supports_execution = **False**

timetuple = **None**

type

unique_params(**optionaldict*, ***kwargs*)
inherited from the `unique_params()` *method of* `ClauseElement`

Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

class `sqlalchemy.sql.operators.`**ColumnOperators**
Bases: `sqlalchemy.sql.operators.Operators`

Defines boolean, comparison, and other operators for `ColumnElement` expressions.

By default, all methods call down to `operate()` or `reverse_operate()`, passing in the appropriate operator function from the Python builtin `operator` module or a SQLAlchemy-specific operator function from `sqlalchemy.expression.operators`. For example the `__eq__` function:

```
def __eq__(self, other):
    return self.operate(operators.eq, other)
```

Where `operators.eq` is essentially:

```
def eq(a, b):
    return a == b
```

The core column expression unit `ColumnElement` overrides `Operators.operate()` and others to return further `ColumnElement` constructs, so that the == operation above is replaced by a clause construct.

See also:

Redefining and Creating New Operators

`TypeEngine.comparator_factory`

`ColumnOperators`

`PropComparator`

__add__(*other*)
Implement the + operator.

In a column context, produces the clause `a + b` if the parent object has non-string affinity. If the parent object has a string affinity, produces the concatenation operator, `a || b` - see `ColumnOperators.concat()`.

__and__(*other*)
inherited from the `__and__()` *method of* `Operators`

Implement the `&` operator.

When used with SQL expressions, results in an AND operation, equivalent to `and_()`, that is:

```
a & b
```

is equivalent to:

```
from sqlalchemy import and_
and_(a, b)
```

Care should be taken when using `&` regarding operator precedence; the `&` operator has the highest precedence. The operands should be enclosed in parenthesis if they contain further sub expressions:

```
(a == 2) & (b == 4)
```

__delattr__

inherited from the `__delattr__` *attribute of* `object`

x.__delattr__('name') <==> del x.name

__div__(*other*)

Implement the / operator.

In a column context, produces the clause `a / b`.

__eq__(*other*)

Implement the == operator.

In a column context, produces the clause `a = b`. If the target is `None`, produces `a IS NULL`.

__format__()

inherited from the `__format__()` *method of* `object`

default object formatter

__ge__(*other*)

Implement the >= operator.

In a column context, produces the clause `a >= b`.

__getattribute__

inherited from the `__getattribute__` *attribute of* `object`

x.__getattribute__('name') <==> x.name

__getitem__(*index*)

Implement the [] operator.

This can be used by some database-specific types such as Postgresql ARRAY and HSTORE.

__gt__(*other*)

Implement the > operator.

In a column context, produces the clause `a > b`.

__hash__

__init__

inherited from the `__init__` *attribute of* `object`

x.__init__(...) initializes x; see help(type(x)) for signature

__invert__()

inherited from the `__invert__()` *method of* `Operators`

Implement the ~ operator.

When used with SQL expressions, results in a NOT operation, equivalent to `not_()`, that is:

```
~a
```

is equivalent to:

```
from sqlalchemy import not_
not_(a)
```

__le__(*other*)

Implement the <= operator.

In a column context, produces the clause `a <= b`.

__lshift__(*other*)

implement the << operator.

Not used by SQLAlchemy core, this is provided for custom operator systems which want to use << as an extension point.

__lt__(*other*)

Implement the < operator.

In a column context, produces the clause `a < b`.

__mod__(*other*)

Implement the % operator.

In a column context, produces the clause `a % b`.

__mul__(*other*)

Implement the * operator.

In a column context, produces the clause `a * b`.

__ne__(*other*)

Implement the != operator.

In a column context, produces the clause `a != b`. If the target is `None`, produces `a IS NOT NULL`.

__neg__()

Implement the − operator.

In a column context, produces the clause `-a`.

__new__(*S*, ...) → a new object with type S, a subtype of T

inherited from the `__new__()` *method of* `object`

__or__(*other*)

inherited from the `__or__()` *method of* `Operators`

Implement the | operator.

When used with SQL expressions, results in an OR operation, equivalent to `or_()`, that is:

```
a | b
```

is equivalent to:

```
from sqlalchemy import or_
or_(a, b)
```

Care should be taken when using | regarding operator precedence; the | operator has the highest precedence. The operands should be enclosed in parenthesis if they contain further sub expressions:

```
(a == 2) | (b == 4)
```

__radd__(*other*)
Implement the + operator in reverse.

See *ColumnOperators.__add__()*.

__rdiv__(*other*)
Implement the / operator in reverse.

See *ColumnOperators.__div__()*.

__reduce__()
inherited from the `__reduce__()` *method of* `object`

helper for pickle

__reduce_ex__()
inherited from the `__reduce_ex__()` *method of* `object`

helper for pickle

__repr__
inherited from the `__repr__` *attribute of* `object`

__rmod__(*other*)
Implement the % operator in reverse.

See *ColumnOperators.__mod__()*.

__rmul__(*other*)
Implement the * operator in reverse.

See *ColumnOperators.__mul__()*.

__rshift__(*other*)
implement the >> operator.

Not used by SQLAlchemy core, this is provided for custom operator systems which want to use >> as an extension point.

__rsub__(*other*)
Implement the − operator in reverse.

See *ColumnOperators.__sub__()*.

__rtruediv__(*other*)
Implement the // operator in reverse.

See *ColumnOperators.__truediv__()*.

__setattr__
inherited from the `__setattr__` *attribute of* `object`

x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int
inherited from the `__sizeof__()` *method of* `object`

size of object in memory, in bytes

__str__
inherited from the `__str__` *attribute of* `object`

__sub__(*other*)
Implement the − operator.

In a column context, produces the clause `a - b`.

__subclasshook__()
inherited from the `__subclasshook__()` *method of* `object`

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__truediv__(*other*)
Implement the `//` operator.

In a column context, produces the clause `a / b`.

all_()
Produce a *all_()* clause against the parent object.

New in version 1.1.

any_()
Produce a *any_()* clause against the parent object.

New in version 1.1.

asc()
Produce a *asc()* clause against the parent object.

between(*cleft*, *cright*, *symmetric=False*)
Produce a *between()* clause against the parent object, given the lower and upper range.

collate(*collation*)
Produce a *collate()* clause against the parent object, given the collation string.

concat(*other*)
Implement the 'concat' operator.

In a column context, produces the clause `a || b`, or uses the `concat()` operator on MySQL.

contains(*other*, ***kwargs*)
Implement the 'contains' operator.

In a column context, produces the clause `LIKE '%<other>%'`

desc()
Produce a *desc()* clause against the parent object.

distinct()
Produce a *distinct()* clause against the parent object.

endswith(*other*, ***kwargs*)
Implement the 'endswith' operator.

In a column context, produces the clause `LIKE '%<other>'`

ilike(*other*, *escape=None*)
Implement the `ilike` operator.

In a column context, produces the clause `a ILIKE other`.

E.g.:

```
select([sometable]).where(sometable.c.column.ilike("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the `ESCAPE` keyword, e.g.:

```
somecolumn.ilike("foo/%bar", escape="/")
```

See also:

ColumnOperators.like()

in_(*other*)

Implement the `in` operator.

In a column context, produces the clause `a IN other`. "other" may be a tuple/list of column expressions, or a *select()* construct.

is_(*other*)

Implement the `IS` operator.

Normally, `IS` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.isnot()

isnot(*other*)

Implement the `IS NOT` operator.

Normally, `IS NOT` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS NOT` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.is_()

like(*other*, *escape=None*)

Implement the `like` operator.

In a column context, produces the clause `a LIKE other`.

E.g.:

```
select([sometable]).where(sometable.c.column.like("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the `ESCAPE` keyword, e.g.:

```
somecolumn.like("foo/%bar", escape="/")
```

See also:

ColumnOperators.ilike()

match(*other*, ***kwargs*)

Implements a database-specific 'match' operator.

match() attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

- Postgresql - renders `x @@ to_tsquery(y)`
- MySQL - renders `MATCH (x) AGAINST (y IN BOOLEAN MODE)`
- Oracle - renders `CONTAINS(x, y)`
- other backends may provide special implementations.
- Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

notilike(*other*, *escape=None*)

implement the `NOT ILIKE` operator.

This is equivalent to using negation with *ColumnOperators.ilike()*, i.e. `~x.ilike(y)`.

New in version 0.8.

See also:

ColumnOperators.ilike()

notin_(*other*)

implement the `NOT IN` operator.

This is equivalent to using negation with *ColumnOperators.in_()*, i.e. `~x.in_(y)`.

New in version 0.8.

See also:

ColumnOperators.in_()

notlike(*other*, *escape=None*)

implement the `NOT LIKE` operator.

This is equivalent to using negation with *ColumnOperators.like()*, i.e. `~x.like(y)`.

New in version 0.8.

See also:

ColumnOperators.like()

nullsfirst()

Produce a *nullsfirst()* clause against the parent object.

nullslast()

Produce a *nullslast()* clause against the parent object.

op(*opstring*, *precedence=0*, *is_comparison=False*)

inherited from the op() method of Operators

produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

This function can also be used to make bitwise operators explicit. For example:

```
somecolumn.op('&')(0xff)
```

is a bitwise AND of the value in `somecolumn`.

Parameters

- **operator** – a string which will be output as the infix operator between this element and the expression passed to the generated function.
- **precedence** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of `0` is lower than all operators except for the comma (`,`) and `AS` operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.
- **is_comparison** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like ==, >, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

  New in version 0.9.2: - added the *`Operators.op.is_comparison`* flag.

See also:

Redefining and Creating New Operators

Using custom operators in join conditions

operate(*op*, **other*, ***kwargs*)

inherited from the `operate()` method of `Operators`

Operate on an argument.

This is the lowest level of operation, raises `NotImplementedError` by default.

Overriding this on a subclass can allow common behavior to be applied to all operations. For example, overriding *`ColumnOperators`* to apply `func.lower()` to the left and right side:

```
class MyComparator(ColumnOperators):
    def operate(self, op, other):
        return op(func.lower(self), func.lower(other))
```

Parameters

- **op** – Operator callable.
- ***other** – the 'other' side of the operation. Will be a single scalar for most operations.
- ****kwargs** – modifiers. These may be passed by special operators such as `ColumnOperators.contains()`.

reverse_operate(*op*, *other*, ***kwargs*)
inherited from the `reverse_operate()` *method of* `Operators`

Reverse operate on an argument.

Usage is the same as `operate()`.

startswith(*other*, ***kwargs*)
Implement the `startwith` operator.

In a column context, produces the clause `LIKE '<other>%'`

timetuple = None
Hack, allows datetime objects to be compared on the LHS.

class sqlalchemy.sql.base.**DialectKWArgs**
Establish the ability for a class to have dialect-specific arguments with defaults and constructor validation.

The `DialectKWArgs` interacts with the `DefaultDialect.construct_arguments` present on a dialect.

See also:

`DefaultDialect.construct_arguments`

classmethod argument_for(*dialect_name*, *argument_name*, *default*)
Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

dialect_kwargs
A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the *DialectKWArgs.dialect_options* collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The *DialectKWArgs.dialect_kwargs* collection is now writable.

See also:

DialectKWArgs.dialect_options - nested dictionary form

dialect_options

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

DialectKWArgs.dialect_kwargs - flat dictionary form

kwargs

A synonym for *DialectKWArgs.dialect_kwargs*.

class sqlalchemy.sql.expression.**Extract**(*field*, *expr*, ***kwargs*)

Bases: *sqlalchemy.sql.expression.ColumnElement*

Represent a SQL EXTRACT clause, `extract(field FROM expr)`.

__init__(*field*, *expr*, ***kwargs*)

Construct a new *Extract* object.

This constructor is mirrored as a public API function; see *extract()* for a full usage and argument description.

class sqlalchemy.sql.elements.**False_**

Bases: *sqlalchemy.sql.expression.ColumnElement*

Represent the `false` keyword, or equivalent, in a SQL statement.

*False_* is accessed as a constant via the *false()* function.

class sqlalchemy.sql.expression.**FunctionFilter**(*func*, **criterion*)

Bases: *sqlalchemy.sql.expression.ColumnElement*

Represent a function FILTER clause.

This is a special operator against aggregate and window functions, which controls which rows are passed to it. It's supported only by certain database backends.

Invocation of *FunctionFilter* is via *FunctionElement.filter()*:

```
func.count(1).filter(True)
```

New in version 1.0.0.

See also:

FunctionElement.filter()

__init__(*func*, **criterion*)

Construct a new `FunctionFilter` object.

This constructor is mirrored as a public API function; see `funcfilter()` for a full usage and argument description.

filter(**criterion*)

Produce an additional FILTER against the function.

This method adds additional criteria to the initial criteria set up by `FunctionElement.filter()`.

Multiple criteria are joined together at SQL render time via `AND`.

over(*partition_by=None*, *order_by=None*)

Produce an OVER clause against this filtered function.

Used against aggregate or so-called "window" functions, for database backends that support window functions.

The expression:

```
func.rank().filter(MyClass.y > 5).over(order_by='x')
```

is shorthand for:

```
from sqlalchemy import over, funcfilter
over(funcfilter(func.rank(), MyClass.y > 5), order_by='x')
```

See `over()` for a full description.

class sqlalchemy.sql.expression.**Label**(*name*, *element*, *type_=None*)

Bases: `sqlalchemy.sql.expression.ColumnElement`

Represents a column label (AS).

Represent a label, as typically applied to any column-level element using the `AS` sql keyword.

__init__(*name*, *element*, *type_=None*)

Construct a new `Label` object.

This constructor is mirrored as a public API function; see `label()` for a full usage and argument description.

class sqlalchemy.sql.elements.**Null**

Bases: `sqlalchemy.sql.expression.ColumnElement`

Represent the NULL keyword in a SQL statement.

`Null` is accessed as a constant via the `null()` function.

class sqlalchemy.sql.expression.**Over**(*element*, *partition_by=None*, *order_by=None*)

Bases: `sqlalchemy.sql.expression.ColumnElement`

Represent an OVER clause.

This is a special operator against a so-called "window" function, as well as any aggregate function, which produces results relative to the result set itself. It's supported only by certain database backends.

__init__(*element*, *partition_by=None*, *order_by=None*)

Construct a new `Over` object.

This constructor is mirrored as a public API function; see `over()` for a full usage and argument description.

func

the element referred to by this *Over* clause.

Deprecated since version 1.1: the `func` element has been renamed to `.element`. The two attributes are synonymous though `.func` is read-only.

class `sqlalchemy.sql.expression.`**`TextClause`**(*text*, *bind=None*)

Bases: *sqlalchemy.sql.expression.Executable*, *sqlalchemy.sql.expression.ClauseElement*

Represent a literal SQL text fragment.

E.g.:

```
from sqlalchemy import text

t = text("SELECT * FROM users")
result = connection.execute(t)
```

The *Text* construct is produced using the *text()* function; see that function for full documentation.

See also:

text()

bindparams(**binds*, ***names_to_values*)

Establish the values and/or types of bound parameters within this *TextClause* construct.

Given a text construct such as:

```
from sqlalchemy import text
stmt = text("SELECT id, name FROM user WHERE name=:name "
            "AND timestamp=:timestamp")
```

the *TextClause.bindparams()* method can be used to establish the initial value of `:name` and `:timestamp`, using simple keyword arguments:

```
stmt = stmt.bindparams(name='jack',
            timestamp=datetime.datetime(2012, 10, 8, 15, 12, 5))
```

Where above, new *BindParameter* objects will be generated with the names `name` and `timestamp`, and values of `jack` and `datetime.datetime(2012, 10, 8, 15, 12, 5)`, respectively. The types will be inferred from the values given, in this case *String* and *DateTime*.

When specific typing behavior is needed, the positional `*binds` argument can be used in which to specify *bindparam()* constructs directly. These constructs must include at least the `key` argument, then an optional value and type:

```
from sqlalchemy import bindparam
stmt = stmt.bindparams(
                bindparam('name', value='jack', type_=String),
                bindparam('timestamp', type_=DateTime)
            )
```

Above, we specified the type of *DateTime* for the `timestamp` bind, and the type of *String* for the `name` bind. In the case of `name` we also set the default value of `"jack"`.

Additional bound parameters can be supplied at statement execution time, e.g.:

```
result = connection.execute(stmt,
            timestamp=datetime.datetime(2012, 10, 8, 15, 12, 5))
```

The *TextClause.bindparams()* method can be called repeatedly, where it will re-use existing *BindParameter* objects to add new information. For example, we can call

TextClause.bindparams() first with typing information, and a second time with value information, and it will be combined:

```
stmt = text("SELECT id, name FROM user WHERE name=:name "
            "AND timestamp=:timestamp")
stmt = stmt.bindparams(
    bindparam('name', type_=String),
    bindparam('timestamp', type_=DateTime)
)
stmt = stmt.bindparams(
    name='jack',
    timestamp=datetime.datetime(2012, 10, 8, 15, 12, 5)
)
```

New in version 0.9.0: The *TextClause.bindparams()* method supersedes the argument `bindparams` passed to *text()*.

columns(**cols*, ***types*)

Turn this *TextClause* object into a *TextAsFrom* object that can be embedded into another statement.

This function essentially bridges the gap between an entirely textual SELECT statement and the SQL expression language concept of a "selectable":

```
from sqlalchemy.sql import column, text

stmt = text("SELECT id, name FROM some_table")
stmt = stmt.columns(column('id'), column('name')).alias('st')

stmt = select([mytable]).\
        select_from(
            mytable.join(stmt, mytable.c.name == stmt.c.name)
        ).where(stmt.c.id > 5)
```

Above, we used untyped *column()* elements. These can also have types specified, which will impact how the column behaves in expressions as well as determining result set behavior:

```
stmt = text("SELECT id, name, timestamp FROM some_table")
stmt = stmt.columns(
            column('id', Integer),
            column('name', Unicode),
            column('timestamp', DateTime)
        )

for id, name, timestamp in connection.execute(stmt):
    print(id, name, timestamp)
```

Keyword arguments allow just the names and types of columns to be specified, where the *column()* elements will be generated automatically:

```
stmt = text("SELECT id, name, timestamp FROM some_table")
stmt = stmt.columns(
            id=Integer,
            name=Unicode,
            timestamp=DateTime
        )

for id, name, timestamp in connection.execute(stmt):
    print(id, name, timestamp)
```

The *TextClause.columns()* method provides a direct route to calling *FromClause.alias()*

as well as *SelectBase.cte()* against a textual SELECT statement:

```
stmt = stmt.columns(id=Integer, name=String).cte('st')

stmt = select([sometable]).where(sometable.c.id == stmt.c.id)
```

New in version 0.9.0: *text()* can now be converted into a fully featured "selectable" construct using the *TextClause.columns()* method. This method supersedes the `typemap` argument to *text()*.

class sqlalchemy.sql.expression.**Tuple**(**clauses*, ***kw*)

Bases: *sqlalchemy.sql.expression.ClauseList*, *sqlalchemy.sql.expression.ColumnElement*

Represent a SQL tuple.

__init__(**clauses*, ***kw*)

Construct a new *Tuple* object.

This constructor is mirrored as a public API function; see *tuple_()* for a full usage and argument description.

class sqlalchemy.sql.expression.**WithinGroup**(*element*, **order_by*)

Bases: *sqlalchemy.sql.expression.ColumnElement*

Represent a WITHIN GROUP (ORDER BY) clause.

This is a special operator against so-called so-called "ordered set aggregate" and "hypothetical set aggregate" functions, including `percentile_cont()`, `rank()`, `dense_rank()`, etc.

It's supported only by certain database backends, such as PostgreSQL, Oracle and MS SQL Server.

The *WithinGroup* consturct extracts its type from the method *FunctionElement.within_group_type()*. If this returns `None`, the function's `.type` is used.

__init__(*element*, **order_by*)

Construct a new *WithinGroup* object.

This constructor is mirrored as a public API function; see *within_group()* for a full usage and argument description.

over(*partition_by=None*, *order_by=None*)

Produce an OVER clause against this *WithinGroup* construct.

This function has the same signature as that of *FunctionElement.over()*.

class sqlalchemy.sql.elements.**True_**

Bases: *sqlalchemy.sql.expression.ColumnElement*

Represent the `true` keyword, or equivalent, in a SQL statement.

*True_* is accessed as a constant via the *true()* function.

class sqlalchemy.sql.expression.**TypeCoerce**(*expression*, *type_*)

Bases: *sqlalchemy.sql.expression.ColumnElement*

Represent a Python-side type-coercion wrapper.

TypeCoerce supplies the *expression.type_coerce()* function; see that function for usage details.

Changed in version 1.1: The *type_coerce()* function now produces a persistent *TypeCoerce* wrapper object rather than translating the given object in place.

See also:

expression.type_coerce()

__init__(*expression*, *type_*)

Construct a new `TypeCoerce` object.

This constructor is mirrored as a public API function; see `type_coerce()` for a full usage and argument description.

class sqlalchemy.sql.operators.**custom_op**(*opstring*, *precedence=0*, *is_comparison=False*, *natural_self_precedent=False*)

Represent a 'custom' operator.

`custom_op` is normally instantitated when the `ColumnOperators.op()` method is used to create a custom operator callable. The class can also be used directly when programmatically constructing expressions. E.g. to represent the "factorial" operation:

```
from sqlalchemy.sql import UnaryExpression
from sqlalchemy.sql import operators
from sqlalchemy import Numeric

unary = UnaryExpression(table.c.somecolumn,
        modifier=operators.custom_op("!"),
        type_=Numeric)
```

class sqlalchemy.sql.operators.**Operators**

Base of comparison and logical operators.

Implements base methods `operate()` and `reverse_operate()`, as well as `__and__()`, `__or__()`, `__invert__()`.

Usually is used via its most common subclass `ColumnOperators`.

__and__(*other*)

Implement the `&` operator.

When used with SQL expressions, results in an AND operation, equivalent to `and_()`, that is:

```
a & b
```

is equivalent to:

```
from sqlalchemy import and_
and_(a, b)
```

Care should be taken when using `&` regarding operator precedence; the `&` operator has the highest precedence. The operands should be enclosed in parenthesis if they contain further sub expressions:

```
(a == 2) & (b == 4)
```

__invert__()

Implement the ~ operator.

When used with SQL expressions, results in a NOT operation, equivalent to `not_()`, that is:

```
~a
```

is equivalent to:

```
from sqlalchemy import not_
not_(a)
```

__or__(*other*)

Implement the `|` operator.

When used with SQL expressions, results in an OR operation, equivalent to `or_()`, that is:

```
a | b
```

is equivalent to:

```
from sqlalchemy import or_
or_(a, b)
```

Care should be taken when using `|` regarding operator precedence; the `|` operator has the highest precedence. The operands should be enclosed in parenthesis if they contain further sub expressions:

```
(a == 2) | (b == 4)
```

op(*opstring*, *precedence=0*, *is_comparison=False*)

produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

This function can also be used to make bitwise operators explicit. For example:

```
somecolumn.op('&')(0xff)
```

is a bitwise AND of the value in `somecolumn`.

Parameters

- **operator** – a string which will be output as the infix operator between this element and the expression passed to the generated function.
- **precedence** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of `0` is lower than all operators except for the comma (`,`) and `AS` operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.
- **is_comparison** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like ==, >, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

  New in version 0.9.2: - added the *`Operators.op.is_comparison`* flag.

See also:

Redefining and Creating New Operators

Using custom operators in join conditions

operate(*op*, **other*, ***kwargs*)

Operate on an argument.

This is the lowest level of operation, raises `NotImplementedError` by default.

Overriding this on a subclass can allow common behavior to be applied to all operations. For example, overriding *`ColumnOperators`* to apply `func.lower()` to the left and right side:

```
class MyComparator(ColumnOperators):
    def operate(self, op, other):
        return op(func.lower(self), func.lower(other))
```

Parameters

- **op** – Operator callable.
- ***other** – the 'other' side of the operation. Will be a single scalar for most operations.
- ****kwargs** – modifiers. These may be passed by special operators such as `ColumnOperators.contains()`.

reverse_operate(*op*, *other*, ***kwargs*)

Reverse operate on an argument.

Usage is the same as `operate()`.

class sqlalchemy.sql.elements.**quoted_name**

Bases: sqlalchemy.util.langhelpers.MemoizedSlots, __builtin__.unicode

Represent a SQL identifier combined with quoting preferences.

quoted_name is a Python unicode/str subclass which represents a particular identifier name along with a `quote` flag. This `quote` flag, when set to `True` or `False`, overrides automatic quoting behavior for this identifier in order to either unconditionally quote or to not quote the name. If left at its default of `None`, quoting behavior is applied to the identifier on a per-backend basis based on an examination of the token itself.

A *quoted_name* object with `quote=True` is also prevented from being modified in the case of a so-called "name normalize" option. Certain database backends, such as Oracle, Firebird, and DB2 "normalize" case-insensitive names as uppercase. The SQLAlchemy dialects for these backends convert from SQLAlchemy's lower-case-means-insensitive convention to the upper-case-means-insensitive conventions of those backends. The `quote=True` flag here will prevent this conversion from occurring to support an identifier that's quoted as all lower case against such a backend.

The *quoted_name* object is normally created automatically when specifying the name for key schema constructs such as *Table*, *Column*, and others. The class can also be passed explicitly as the name to any function that receives a name which can be quoted. Such as to use the *Engine.has_table()* method with an unconditionally quoted name:

```
from sqlaclchemy import create_engine
from sqlalchemy.sql.elements import quoted_name

engine = create_engine("oracle+cx_oracle://some_dsn")
engine.has_table(quoted_name("some_table", True))
```

The above logic will run the "has table" logic against the Oracle backend, passing the name exactly as `"some_table"` without converting to upper case.

New in version 0.9.0.

class sqlalchemy.sql.expression.**UnaryExpression**(*element*, *operator=None*, *modifier=None*, *type_=None*, *negate=None*, *wraps_column_expression=False*)

Bases: *sqlalchemy.sql.expression.ColumnElement*

Define a 'unary' expression.

A unary expression has a single column expression and an operator. The operator can be placed on the left (where it is called the 'operator') or right (where it is called the 'modifier') of the column expression.

`UnaryExpression` is the basis for several unary operators including those used by `desc()`, `asc()`, `distinct()`, `nullsfirst()` and `nullslast()`.

compare(*other*, ***kw*)
Compare this `UnaryExpression` against the given `ClauseElement`.

3.2.2 Selectables, Tables, FROM objects

The term "selectable" refers to any object that rows can be selected from; in SQLAlchemy, these objects descend from `FromClause` and their distinguishing feature is their `FromClause.c` attribute, which is a namespace of all the columns contained within the FROM clause (these elements are themselves `ColumnElement` subclasses).

sqlalchemy.sql.expression.**alias**(*selectable*, *name=None*, *flat=False*)
Return an `Alias` object.

An `Alias` represents any `FromClause` with an alternate name assigned within SQL, typically using the `AS` clause when generated, e.g. `SELECT * FROM table AS aliasname`.

Similar functionality is available via the `alias()` method available on all `FromClause` subclasses.

When an `Alias` is created from a `Table` object, this has the effect of the table being rendered as `tablename AS aliasname` in a SELECT statement.

For `select()` objects, the effect is that of creating a named subquery, i.e. `(select ...) AS aliasname`.

The `name` parameter is optional, and provides the name to use in the rendered SQL. If blank, an "anonymous" name will be deterministically generated at compile time. Deterministic means the name is guaranteed to be unique against other constructs used in the same statement, and will also be the same name for each successive compilation of the same statement object.

Parameters

- **selectable** – any `FromClause` subclass, such as a table, select statement, etc.
- **name** – string name to be assigned as the alias. If `None`, a name will be deterministically generated at compile time.
- **flat** – Will be passed through to if the given selectable is an instance of `Join` - see `Join.alias()` for details.

  New in version 0.9.0.

sqlalchemy.sql.expression.**except_**(**selects*, ***kwargs*)
Return an `EXCEPT` of multiple selectables.

The returned object is an instance of `CompoundSelect`.

***selects** a list of `Select` instances.

****kwargs** available keyword arguments are the same as those of `select()`.

sqlalchemy.sql.expression.**except_all**(**selects*, ***kwargs*)
Return an `EXCEPT ALL` of multiple selectables.

The returned object is an instance of `CompoundSelect`.

***selects** a list of `Select` instances.

****kwargs** available keyword arguments are the same as those of `select()`.

sqlalchemy.sql.expression.**exists**(**args*, ***kwargs*)
Construct a new `Exists` against an existing `Select` object.

Calling styles are of the following forms:

```
# use on an existing select()
s = select([table.c.col1]).where(table.c.col2==5)
s = exists(s)

# construct a select() at once
exists(['*'], **select_arguments).where(criterion)

# columns argument is optional, generates "EXISTS (SELECT *)"
# by default.
exists().where(table.c.col2==5)
```

sqlalchemy.sql.expression.**intersect**(**selects*, ***kwargs*)

Return an INTERSECT of multiple selectables.

The returned object is an instance of *CompoundSelect*.

***selects** a list of *Select* instances.

****kwargs** available keyword arguments are the same as those of *select()*.

sqlalchemy.sql.expression.**intersect_all**(**selects*, ***kwargs*)

Return an INTERSECT ALL of multiple selectables.

The returned object is an instance of *CompoundSelect*.

***selects** a list of *Select* instances.

****kwargs** available keyword arguments are the same as those of *select()*.

sqlalchemy.sql.expression.**join**(*left*, *right*, *onclause=None*, *isouter=False*)

Produce a *Join* object, given two *FromClause* expressions.

E.g.:

```
j = join(user_table, address_table,
         user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Similar functionality is available given any *FromClause* object (e.g. such as a *Table*) using the *FromClause.join()* method.

Parameters

- **left** – The left side of the join.
- **right** – the right side of the join; this is any *FromClause* object such as a *Table* object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at None, *FromClause.join()* will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.

See also:

FromClause.join() - method form, based on a given left side

Join - the type of object produced

sqlalchemy.sql.expression.**outerjoin**(*left*, *right*, *onclause=None*)

Return an OUTER JOIN clause element.

The returned object is an instance of *Join*.

Similar functionality is also available via the *outerjoin()* method on any *FromClause*.

Parameters

- **left** – The left side of the join.
- **right** – The right side of the join.
- **onclause** – Optional criterion for the ON clause, is derived from foreign key relationships established between left and right otherwise.

To chain joins together, use the *FromClause.join()* or *FromClause.outerjoin()* methods on the resulting *Join* object.

sqlalchemy.sql.expression.**select**(*columns=None*, *whereclause=None*, *from_obj=None*, *distinct=False*, *having=None*, *correlate=True*, *prefixes=None*, *suffixes=None*, ***kwargs*)

Construct a new *Select*.

Similar functionality is also available via the *FromClause.select()* method on any *FromClause*.

All arguments which accept *ClauseElement* arguments also accept string arguments, which will be converted as appropriate into either *text()* or *literal_column()* constructs.

See also:

Selecting - Core Tutorial description of *select()*.

Parameters

- **columns** – A list of *ColumnElement* or *FromClause* objects which will form the columns clause of the resulting statement. For those objects that are instances of *FromClause* (typically *Table* or *Alias* objects), the *FromClause.c* collection is extracted to form a collection of *ColumnElement* objects.

  This parameter will also accept *Text* constructs as given, as well as ORM-mapped classes.

  Note: The *select.columns* parameter is not available in the method form of *select()*, e.g. *FromClause.select()*.

  See also:

  Select.column()

  Select.with_only_columns()

- **whereclause** – A *ClauseElement* expression which will be used to form the WHERE clause. It is typically preferable to add WHERE criterion to an existing *Select* using method chaining with *Select.where()*.

  See also:

  Select.where()

- **from_obj** – A list of *ClauseElement* objects which will be added to the FROM clause of the resulting statement. This is equivalent to calling *Select.select_from()* using method chaining on an existing *Select* object.

  See also:

  Select.select_from() - full description of explicit FROM clause specification.

- **autocommit** – Deprecated. Use `.execution_options(autocommit=<True|False>)` to set the autocommit option.

  See also:

  Executable.execution_options()

- **bind=None** – an *Engine* or *Connection* instance to which the resulting *Select* object will be bound. The *Select* object will otherwise automatically bind to whatever Connectable instances can be located within its contained *ClauseElement* members.

- **correlate=True** – indicates that this *Select* object should have its contained *FromClause* elements "correlated" to an enclosing *Select* object. It is typically preferable to specify correlations on an existing *Select* construct using *Select.correlate()*.

  See also:

  Select.correlate() - full description of correlation.

- **distinct=False** – when True, applies a DISTINCT qualifier to the columns clause of the resulting statement.

  The boolean argument may also be a column expression or list of column expressions - this is a special calling form which is understood by the Postgresql dialect to render the DISTINCT ON (<columns>) syntax.

  distinct is also available on an existing *Select* object via the *distinct()* method.

  See also:

  Select.distinct()

- **for_update=False** –

  when True, applies FOR UPDATE to the end of the resulting statement.

  Deprecated since version 0.9.0: - use *Select.with_for_update()* to specify the structure of the FOR UPDATE clause.

  for_update accepts various string values interpreted by specific backends, including:

  – "read" - on MySQL, translates to LOCK IN SHARE MODE; on Postgresql, translates to FOR SHARE.

  – "nowait" - on Postgresql and Oracle, translates to FOR UPDATE NOWAIT.

  – "read_nowait" - on Postgresql, translates to FOR SHARE NOWAIT.

  See also:

  Select.with_for_update() - improved API for specifying the FOR UPDATE clause.

- **group_by** – a list of *ClauseElement* objects which will comprise the GROUP BY clause of the resulting select. This parameter is typically specified more naturally using the *Select.group_by()* method on an existing *Select*.

  See also:

  Select.group_by()

- **having** – a *ClauseElement* that will comprise the HAVING clause of the resulting select when GROUP BY is used. This parameter is typically specified more naturally using the *Select.having()* method on an existing *Select*.

  See also:

  Select.having()

- **limit=None** – a numerical value which usually renders as a LIMIT expression in the resulting select. Backends that don't support LIMIT will attempt to provide similar functionality. This parameter is typically specified more naturally using the *Select.limit()* method on an existing *Select*.

  See also:

  Select.limit()

- **offset=None** – a numeric value which usually renders as an OFFSET expression in the resulting select. Backends that don't support OFFSET will attempt to provide similar functionality. This parameter is typically specified more naturally using the *Select.offset()* method on an existing *Select*.

  See also:

  Select.offset()

- **order_by** – a scalar or list of *ClauseElement* objects which will comprise the ORDER BY clause of the resulting select. This parameter is typically specified more naturally using the *Select.order_by()* method on an existing *Select*.

  See also:

  Select.order_by()

- **use_labels=False** – when True, the statement will be generated using labels for each column in the columns clause, which qualify each column with its parent table's (or aliases) name so that name conflicts between columns in different tables don't occur. The format of the label is <tablename>_<column>. The "c" collection of the resulting *Select* object will use these names as well for targeting column members.

  This parameter can also be specified on an existing *Select* object using the *Select.apply_labels()* method.

  See also:

  Select.apply_labels()

sqlalchemy.sql.expression.**subquery**(*alias*, **args*, ***kwargs*)

Return an *Alias* object derived from a *Select*.

name alias name

*args, **kwargs

all other arguments are delivered to the *select()* function.

`sqlalchemy.sql.expression.table`(*name*, **columns*)

Produce a new `TableClause`.

The object returned is an instance of `TableClause`, which represents the "syntactical" portion of the schema-level `Table` object. It may be used to construct lightweight table constructs.

Changed in version 1.0.0: `expression.table()` can now be imported from the plain `sqlalchemy` namespace like any other SQL element.

Parameters

- **name** – Name of the table.
- **columns** – A collection of `expression.column()` constructs.

`sqlalchemy.sql.expression.union`(**selects*, ***kwargs*)

Return a `UNION` of multiple selectables.

The returned object is an instance of `CompoundSelect`.

A similar `union()` method is available on all `FromClause` subclasses.

***selects** a list of `Select` instances.

****kwargs** available keyword arguments are the same as those of `select()`.

`sqlalchemy.sql.expression.union_all`(**selects*, ***kwargs*)

Return a `UNION ALL` of multiple selectables.

The returned object is an instance of `CompoundSelect`.

A similar `union_all()` method is available on all `FromClause` subclasses.

***selects** a list of `Select` instances.

****kwargs** available keyword arguments are the same as those of `select()`.

class `sqlalchemy.sql.expression.Alias`(*selectable*, *name=None*)

Bases: `sqlalchemy.sql.expression.FromClause`

Represents an table or selectable alias (AS).

Represents an alias, as typically applied to any table or sub-select within a SQL statement using the `AS` keyword (or without the keyword on certain databases such as Oracle).

This object is constructed from the `alias()` module level function as well as the `FromClause.alias()` method available on all `FromClause` subclasses.

alias(*name=None*, *flat=False*)

inherited from the `alias()` *method of* `FromClause`

return an alias of this `FromClause`.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

c

inherited from the `c` *attribute of* `FromClause`

An alias for the `columns` attribute.

columns

inherited from the `columns` *attribute of* `FromClause`

A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

compare(*other*, ***kw*)

inherited from the `compare()` *method of* `ClauseElement`

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

**kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see `ColumnElement`)

compile(*bind=None*, *dialect=None*, ***kw*)

inherited from the `compile()` *method of* `ClauseElement`

Compile this SQL expression.

The return value is a `Compiled` object. Calling `str()` or `unicode()` on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the `params` accessor.

Parameters

- **bind** – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.
- **column_keys** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- **dialect** – A `Dialect` instance from which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- **inline** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

How do I render SQL expressions as strings, possibly with bound parameters inlined?

correspond_on_equivalents(*column*, *equivalents*)
inherited from the `correspond_on_equivalents()` *method of* `FromClause`

Return corresponding_column for the given column, or if None search for a match in the given dictionary.

corresponding_column(*column*, *require_embedded=False*)
inherited from the `corresponding_column()` *method of* `FromClause`

Given a `ColumnElement`, return the exported `ColumnElement` object from this `Selectable` which corresponds to that original `Column` via a common ancestor column.

Parameters

- **column** – the target `ColumnElement` to be matched
- **require_embedded** – only return corresponding columns for the given `ColumnElement`, if the given `ColumnElement` is actually present within a sub-element of this `FromClause`. Normally the column will match if it merely shares a common ancestor with one of the exported columns of this `FromClause`.

count(*whereclause=None*, ***params*)
inherited from the `count()` *method of* `FromClause`

return a SELECT COUNT generated against this `FromClause`.

foreign_keys
inherited from the `foreign_keys` *attribute of* `FromClause`

Return the collection of ForeignKey objects which this FromClause references.

join(*right*, *onclause=None*, *isouter=False*)
inherited from the `join()` *method of* `FromClause`

Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                    user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.

See also:

join() - standalone function

Join - the type of object produced

outerjoin(*right*, *onclause=None*)
inherited from the *outerjoin()* *method of* *FromClause*

Return a *Join* from this *FromClause* to another *FromClause*, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                         user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any *FromClause* object such as a *Table* object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at None, *FromClause.join()* will attempt to join the two tables based on a foreign key relationship.

See also:

FromClause.join()

Join

params(**optionaldict*, ***kwargs*)
inherited from the *params()* *method of* *ClauseElement*

Return a copy with *bindparam()* elements replaced.

Returns a copy of this ClauseElement with *bindparam()* elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

primary_key
inherited from the *primary_key* *attribute of* *FromClause*

Return the collection of Column objects which comprise the primary key of this FromClause.

replace_selectable(*old*, *alias*)
inherited from the *replace_selectable()* *method of* *FromClause*

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this *FromClause*.

select(*whereclause=None*, ***params*)
inherited from the `select()` *method of* `FromClause`

return a SELECT of this `FromClause`.

See also:

`select()` - general purpose method which allows for arbitrary column lists.

unique_params(**optionaldict*, ***kwargs*)
inherited from the `unique_params()` *method of* `ClauseElement`

Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

class sqlalchemy.sql.expression.**CompoundSelect**(*keyword*, **selects*, ***kwargs*)
Bases: `sqlalchemy.sql.expression.GenerativeSelect`

Forms the basis of `UNION`, `UNION ALL`, and other SELECT-based set operations.

See also:

`union()`

`union_all()`

`intersect()`

`intersect_all()`

`except()`

`except_all()`

alias(*name=None*, *flat=False*)
inherited from the `alias()` *method of* `FromClause`

return an alias of this `FromClause`.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

append_group_by(**clauses*)
inherited from the `append_group_by()` *method of* `GenerativeSelect`

Append the given GROUP BY criterion applied to this selectable.

The criterion will be appended to any pre-existing GROUP BY criterion.

This is an **in-place** mutation method; the `group_by()` method is preferred, as it provides standard method chaining.

append_order_by(**clauses*)
inherited from the `append_order_by()` *method of* `GenerativeSelect`

Append the given ORDER BY criterion applied to this selectable.

The criterion will be appended to any pre-existing ORDER BY criterion.

This is an **in-place** mutation method; the `order_by()` method is preferred, as it provides standard method chaining.

apply_labels()
inherited from the `apply_labels()` *method of* `GenerativeSelect`

return a new selectable with the 'use_labels' flag set to True.

This will result in column expressions being generated using labels against their table name, such as "SELECT somecolumn AS tablename_somecolumn". This allows selectables which contain multiple FROM clauses to produce a unique set of column names regardless of name conflicts among the individual FROM clauses.

as_scalar()
inherited from the `as_scalar()` *method of* `SelectBase`

return a 'scalar' representation of this selectable, which can be used as a column expression.

Typically, a select statement which has only one column in its columns clause is eligible to be used as a scalar expression.

The returned object is an instance of `ScalarSelect`.

autocommit()
inherited from the `autocommit()` *method of* `SelectBase`

return a new selectable with the 'autocommit' flag set to True.

Deprecated since version 0.6: `autocommit()` is deprecated. Use `Executable.execution_options()` with the 'autocommit' flag.

c
inherited from the `c` *attribute of* `FromClause`

An alias for the `columns` attribute.

columns
inherited from the `columns` *attribute of* `FromClause`

A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

compare(*other*, ***kw*)
inherited from the `compare()` *method of* `ClauseElement`

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

**kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see `ColumnElement`)

compile(*bind=None*, *dialect=None*, ***kw*)
inherited from the `compile()` *method of* `ClauseElement`

Compile this SQL expression.

The return value is a `Compiled` object. Calling `str()` or `unicode()` on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the `params` accessor.

Parameters

- **bind** – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.

- **column_keys** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- **dialect** – A `Dialect` instance from which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- **inline** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

How do I render SQL expressions as strings, possibly with bound parameters inlined?

correspond_on_equivalents(*column*, *equivalents*)
inherited from the `correspond_on_equivalents()` *method of* `FromClause`

Return corresponding_column for the given column, or if None search for a match in the given dictionary.

corresponding_column(*column*, *require_embedded=False*)
inherited from the `corresponding_column()` *method of* `FromClause`

Given a `ColumnElement`, return the exported `ColumnElement` object from this `Selectable` which corresponds to that original `Column` via a common ancestor column.

Parameters

- **column** – the target `ColumnElement` to be matched
- **require_embedded** – only return corresponding columns for the given `ColumnElement`, if the given `ColumnElement` is actually present within a sub-element of this `FromClause`. Normally the column will match if it merely shares a common ancestor with one of the exported columns of this `FromClause`.

count(*whereclause=None*, ***params*)
inherited from the `count()` *method of* `FromClause`

return a SELECT COUNT generated against this `FromClause`.

cte(*name=None*, *recursive=False*)
inherited from the `cte()` *method of* `SelectBase`

Return a new `CTE`, or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

SQLAlchemy detects *CTE* objects, which are treated similarly to *Alias* objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

New in version 0.7.6.

Parameters

- **name** – name given to the common table expression. Like `_FromClause.alias()`, the name can be left as `None` in which case an anonymous symbol will be used at query compile time.
- **recursive** – if `True`, will render `WITH RECURSIVE`. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.

The following examples illustrate two examples from Postgresql's documentation at http://www.postgresql.org/docs/8.4/static/queries-with.html.

Example 1, non recursive:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,
    Column('region', String),
    Column('amount', Integer),
    Column('product', String),
    Column('quantity', Integer)
)

regional_sales = select([
                    orders.c.region,
                    func.sum(orders.c.amount).label('total_sales')
                ]).group_by(orders.c.region).cte("regional_sales")


top_regions = select([regional_sales.c.region]).\
        where(
            regional_sales.c.total_sales >
            select([
                func.sum(regional_sales.c.total_sales)/10
            ])
        ).cte("top_regions")

statement = select([
            orders.c.region,
            orders.c.product,
            func.sum(orders.c.quantity).label("product_units"),
            func.sum(orders.c.amount).label("product_sales")
    ]).where(orders.c.region.in_(
        select([top_regions.c.region])
    )).group_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()
```

Example 2, WITH RECURSIVE:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,
    Column('part', String),
    Column('sub_part', String),
    Column('quantity', Integer),
)

included_parts = select([
                    parts.c.sub_part,
                    parts.c.part,
                    parts.c.quantity]).\
                    where(parts.c.part=='our part').\
                    cte(recursive=True)


incl_alias = included_parts.alias()
parts_alias = parts.alias()
included_parts = included_parts.union_all(
    select([
        parts_alias.c.sub_part,
        parts_alias.c.part,
        parts_alias.c.quantity
    ]).
        where(parts_alias.c.part==incl_alias.c.sub_part)
)

statement = select([
            included_parts.c.sub_part,
            func.sum(included_parts.c.quantity).
              label('total_quantity')
        ]).\
        group_by(included_parts.c.sub_part)

result = conn.execute(statement).fetchall()
```

See also:

`orm.query.Query.cte()` - ORM version of `SelectBase.cte()`.

description

inherited from the `description` *attribute of* `FromClause`

a brief description of this FromClause.

Used primarily for error message formatting.

execute(**multiparams*, ***params*)

inherited from the `execute()` *method of* `Executable`

Compile and execute this `Executable`.

execution_options(***kw*)

inherited from the `execution_options()` *method of* `Executable`

Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per `Connection` basis. Additionally, the `Engine` and ORM `Query` objects provide access to execution options which they in turn configure upon connections.

The `execution_options()` method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See `Connection.execution_options()` for a full list of possible options.

See also:

`Connection.execution_options()`

`Query.execution_options()`

for_update

inherited from the `for_update` *attribute of* `GenerativeSelect`

Provide legacy dialect support for the `for_update` attribute.

foreign_keys

inherited from the `foreign_keys` *attribute of* `FromClause`

Return the collection of ForeignKey objects which this FromClause references.

group_by(**clauses*)

inherited from the `group_by()` *method of* `GenerativeSelect`

return a new selectable with the given list of GROUP BY criterion applied.

The criterion will be appended to any pre-existing GROUP BY criterion.

join(*right*, *onclause=None*, *isouter=False*)

inherited from the `join()` *method of* `FromClause`

Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                    user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.

- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.

See also:

`join()` - standalone function

`Join` - the type of object produced

label(*name*)

inherited from the `label()` *method of* `SelectBase`

return a 'scalar' representation of this selectable, embedded as a subquery with a label.

See also:

`as_scalar()`.

limit(*limit*)

inherited from the `limit()` *method of* `GenerativeSelect`

return a new selectable with the given LIMIT criterion applied.

This is a numerical value which usually renders as a `LIMIT` expression in the resulting select. Backends that don't support `LIMIT` will attempt to provide similar functionality.

Changed in version 1.0.0: - `Select.limit()` can now accept arbitrary SQL expressions as well as integer values.

Parameters **limit** – an integer LIMIT parameter, or a SQL expression that provides an integer result.

offset(*offset*)

inherited from the `offset()` *method of* `GenerativeSelect`

return a new selectable with the given OFFSET criterion applied.

This is a numeric value which usually renders as an `OFFSET` expression in the resulting select. Backends that don't support `OFFSET` will attempt to provide similar functionality.

Changed in version 1.0.0: - `Select.offset()` can now accept arbitrary SQL expressions as well as integer values.

Parameters **offset** – an integer OFFSET parameter, or a SQL expression that provides an integer result.

order_by(**clauses*)

inherited from the `order_by()` *method of* `GenerativeSelect`

return a new selectable with the given list of ORDER BY criterion applied.

The criterion will be appended to any pre-existing ORDER BY criterion.

outerjoin(*right*, *onclause=None*)

inherited from the `outerjoin()` *method of* `FromClause`

Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                         user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.

See also:

`FromClause.join()`

`Join`

params(**optionaldict*, ***kwargs*)

inherited from the `params()` *method of* `ClauseElement`

Return a copy with `bindparam()` elements replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

primary_key

inherited from the `primary_key` *attribute of* `FromClause`

Return the collection of Column objects which comprise the primary key of this FromClause.

replace_selectable(*old*, *alias*)

inherited from the `replace_selectable()` *method of* `FromClause`

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

scalar(**multiparams*, ***params*)

inherited from the `scalar()` *method of* `Executable`

Compile and execute this `Executable`, returning the result's scalar representation.

select(*whereclause=None*, ***params*)

inherited from the `select()` *method of* `FromClause`

return a SELECT of this `FromClause`.

See also:

`select()` - general purpose method which allows for arbitrary column lists.

unique_params(**optionaldict*, ***kwargs*)

inherited from the `unique_params()` *method of* `ClauseElement`

Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

with_for_update(*nowait=False*, *read=False*, *of=None*)

inherited from the `with_for_update()` *method of* `GenerativeSelect`

Specify a `FOR UPDATE` clause for this `GenerativeSelect`.

E.g.:

```
stmt = select([table]).with_for_update(nowait=True)
```

On a database like Postgresql or Oracle, the above would render a statement like:

```
SELECT table.a, table.b FROM table FOR UPDATE NOWAIT
```

on other backends, the `nowait` option is ignored and instead would produce:

```
SELECT table.a, table.b FROM table FOR UPDATE
```

When called with no arguments, the statement will render with the suffix `FOR UPDATE`. Additional arguments can then be provided which allow for common database-specific variants.

Parameters

- **nowait** – boolean; will render `FOR UPDATE NOWAIT` on Oracle and Postgresql dialects.
- **read** – boolean; will render `LOCK IN SHARE MODE` on MySQL, `FOR SHARE` on Postgresql. On Postgresql, when combined with `nowait`, will render `FOR SHARE NOWAIT`.
- **of** – SQL expression or list of SQL expression elements (typically `Column` objects or a compatible expression) which will render into a `FOR UPDATE OF` clause; supported by PostgreSQL and Oracle. May render as a table or as a column depending on backend.

New in version 0.9.0.

class `sqlalchemy.sql.expression.`**CTE**(*selectable*, *name=None*, *recursive=False*, *_cte_alias=None*, *_restates=frozenset([])*, *_suffixes=None*)

Bases: `sqlalchemy.sql.expression.Generative`, `sqlalchemy.sql.expression.HasSuffixes`, `sqlalchemy.sql.expression.Alias`

Represent a Common Table Expression.

The `CTE` object is obtained using the `SelectBase.cte()` method from any selectable. See that method for complete examples.

New in version 0.7.6.

c

inherited from the `c` *attribute of* `FromClause`

An alias for the `columns` attribute.

columns

inherited from the `columns` *attribute of* `FromClause`

A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

compare(*other*, ***kw*)

inherited from the `compare()` *method of* `ClauseElement`

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

**kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see `ColumnElement`)

compile(*bind=None*, *dialect=None*, ***kw*)

inherited from the `compile()` *method of* `ClauseElement`

Compile this SQL expression.

The return value is a `Compiled` object. Calling `str()` or `unicode()` on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the `params` accessor.

Parameters

- **bind** – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.
- **column_keys** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- **dialect** – A `Dialect` instance from which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- **inline** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

How do I render SQL expressions as strings, possibly with bound parameters inlined?

correspond_on_equivalents(*column*, *equivalents*)
inherited from the `correspond_on_equivalents()` *method of* `FromClause`

Return corresponding_column for the given column, or if None search for a match in the given dictionary.

corresponding_column(*column*, *require_embedded=False*)
inherited from the `corresponding_column()` *method of* `FromClause`

Given a `ColumnElement`, return the exported `ColumnElement` object from this `Selectable` which corresponds to that original `Column` via a common ancestor column.

Parameters

- **column** – the target `ColumnElement` to be matched
- **require_embedded** – only return corresponding columns for the given `ColumnElement`, if the given `ColumnElement` is actually present within a sub-element of this `FromClause`. Normally the column will match if it merely shares a common ancestor with one of the exported columns of this `FromClause`.

count(*whereclause=None*, ***params*)
inherited from the `count()` *method of* `FromClause`

return a SELECT COUNT generated against this `FromClause`.

foreign_keys
inherited from the `foreign_keys` *attribute of* `FromClause`

Return the collection of ForeignKey objects which this FromClause references.

join(*right*, *onclause=None*, *isouter=False*)
inherited from the `join()` *method of* `FromClause`

Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                    user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.

See also:

`join()` - standalone function

`Join` - the type of object produced

outerjoin(*right*, *onclause=None*)

inherited from the `outerjoin()` *method of* `FromClause`

Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                         user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.

See also:

`FromClause.join()`

`Join`

params(**optionaldict*, ***kwargs*)

inherited from the `params()` *method of* `ClauseElement`

Return a copy with `bindparam()` elements replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

primary_key

inherited from the `primary_key` *attribute of* `FromClause`

Return the collection of Column objects which comprise the primary key of this FromClause.

replace_selectable(*old*, *alias*)

inherited from the `replace_selectable()` *method of* `FromClause`

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

select(*whereclause=None*, ***params*)

inherited from the `select()` *method of* `FromClause`

return a SELECT of this `FromClause`.

See also:

select() - general purpose method which allows for arbitrary column lists.

suffix_with(**expr*, ***kw*)

inherited from the *suffix_with()* *method of* *HasSuffixes*

Add one or more expressions following the statement as a whole.

This is used to support backend-specific suffix keywords on certain constructs.

E.g.:

```
stmt = select([col1, col2]).cte().suffix_with(
    "cycle empno set y_cycle to 1 default 0", dialect="oracle")
```

Multiple suffixes can be specified by multiple calls to *suffix_with()*.

Parameters

- ***expr** – textual or *ClauseElement* construct which will be rendered following the target clause.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this suffix to only that dialect.

unique_params(**optionaldict*, ***kwargs*)

inherited from the *unique_params()* *method of* *ClauseElement*

Return a copy with *bindparam()* elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

class sqlalchemy.sql.expression.**Executable**

Bases: sqlalchemy.sql.expression.Generative

Mark a ClauseElement as supporting execution.

Executable is a superclass for all "statement" types of objects, including *select()*, *delete()*, *update()*, *insert()*, *text()*.

bind

Returns the *Engine* or *Connection* to which this *Executable* is bound, or None if none found.

This is a traversal which checks locally, then checks among the "from" clauses of associated objects until a bound engine or connection is found.

execute(**multiparams*, ***params*)

Compile and execute this *Executable*.

execution_options(***kw*)

Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per *Connection* basis. Additionally, the *Engine* and ORM *Query* objects provide access to execution options which they in turn configure upon connections.

The *execution_options()* method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See `Connection.execution_options()` for a full list of possible options.

See also:

`Connection.execution_options()`

`Query.execution_options()`

scalar(**multiparams*, ***params*)
Compile and execute this `Executable`, returning the result's scalar representation.

class sqlalchemy.sql.expression.**FromClause**
Bases: `sqlalchemy.sql.expression.Selectable`

Represent an element that can be used within the `FROM` clause of a `SELECT` statement.

The most common forms of `FromClause` are the `Table` and the `select()` constructs. Key features common to all `FromClause` objects include:

- a `c` collection, which provides per-name access to a collection of `ColumnElement` objects.
- a `primary_key` attribute, which is a collection of all those `ColumnElement` objects that indicate the `primary_key` flag.
- Methods to generate various derivations of a "from" clause, including `FromClause.alias()`, `FromClause.join()`, `FromClause.select()`.

alias(*name=None*, *flat=False*)
return an alias of this `FromClause`.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

c
An alias for the `columns` attribute.

columns
A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

correspond_on_equivalents(*column*, *equivalents*)
Return corresponding_column for the given column, or if None search for a match in the given dictionary.

corresponding_column(*column*, *require_embedded=False*)
Given a `ColumnElement`, return the exported `ColumnElement` object from this `Selectable` which corresponds to that original `Column` via a common ancestor column.

Parameters

- **column** – the target `ColumnElement` to be matched
- **require_embedded** – only return corresponding columns for the given `ColumnElement`, if the given `ColumnElement` is actually present within a sub-element of this `FromClause`. Normally the column will match if it merely shares a common ancestor with one of the exported columns of this `FromClause`.

count(*whereclause=None*, ***params*)
return a SELECT COUNT generated against this `FromClause`.

description
a brief description of this FromClause.

Used primarily for error message formatting.

foreign_keys
Return the collection of ForeignKey objects which this FromClause references.

is_derived_from(*fromclause*)
Return True if this FromClause is 'derived' from the given FromClause.

An example would be an Alias of a Table is derived from that Table.

join(*right*, *onclause=None*, *isouter=False*)
Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                    user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.

See also:

`join()` - standalone function

`Join` - the type of object produced

outerjoin(*right*, *onclause=None*)
Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                         user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any *FromClause* object such as a *Table* object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at None, *FromClause.join()* will attempt to join the two tables based on a foreign key relationship.

See also:

FromClause.join()

Join

primary_key

Return the collection of Column objects which comprise the primary key of this FromClause.

replace_selectable(*old*, *alias*)

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this *FromClause*.

schema = None

Define the 'schema' attribute for this *FromClause*.

This is typically None for most objects except that of *Table*, where it is taken as the value of the *Table.schema* argument.

select(*whereclause=None*, ***params*)

return a SELECT of this *FromClause*.

See also:

select() - general purpose method which allows for arbitrary column lists.

class sqlalchemy.sql.expression.**GenerativeSelect**(*use_labels=False*, *for_update=False*, *limit=None*, *offset=None*, *order_by=None*, *group_by=None*, *bind=None*, *autocommit=None*)

Bases: *sqlalchemy.sql.expression.SelectBase*

Base class for SELECT statements where additional elements can be added.

This serves as the base for *Select* and *CompoundSelect* where elements such as ORDER BY, GROUP BY can be added and column rendering can be controlled. Compare to *TextAsFrom*, which, while it subclasses *SelectBase* and is also a SELECT construct, represents a fixed textual string which cannot be altered at this level, only wrapped as a subquery.

New in version 0.9.0: *GenerativeSelect* was added to provide functionality specific to *Select* and *CompoundSelect* while allowing *SelectBase* to be used for other SELECT-like objects, e.g. *TextAsFrom*.

alias(*name=None*, *flat=False*)

inherited from the alias() method of FromClause

return an alias of this *FromClause*.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

append_group_by(**clauses*)

Append the given GROUP BY criterion applied to this selectable.

The criterion will be appended to any pre-existing GROUP BY criterion.

This is an **in-place** mutation method; the `group_by()` method is preferred, as it provides standard method chaining.

append_order_by(**clauses*)

Append the given ORDER BY criterion applied to this selectable.

The criterion will be appended to any pre-existing ORDER BY criterion.

This is an **in-place** mutation method; the `order_by()` method is preferred, as it provides standard method chaining.

apply_labels()

return a new selectable with the 'use_labels' flag set to True.

This will result in column expressions being generated using labels against their table name, such as "SELECT somecolumn AS tablename_somecolumn". This allows selectables which contain multiple FROM clauses to produce a unique set of column names regardless of name conflicts among the individual FROM clauses.

as_scalar()

inherited from the `as_scalar()` *method of* `SelectBase`

return a 'scalar' representation of this selectable, which can be used as a column expression.

Typically, a select statement which has only one column in its columns clause is eligible to be used as a scalar expression.

The returned object is an instance of `ScalarSelect`.

autocommit()

inherited from the `autocommit()` *method of* `SelectBase`

return a new selectable with the 'autocommit' flag set to True.

Deprecated since version 0.6: `autocommit()` is deprecated. Use `Executable.execution_options()` with the 'autocommit' flag.

bind

inherited from the `bind` *attribute of* `Executable`

Returns the `Engine` or `Connection` to which this `Executable` is bound, or None if none found.

This is a traversal which checks locally, then checks among the "from" clauses of associated objects until a bound engine or connection is found.

c

inherited from the `c` *attribute of* `FromClause`

An alias for the `columns` attribute.

columns

inherited from the `columns` *attribute of* `FromClause`

A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

compare(*other*, ***kw*)

inherited from the `compare()` *method of* `ClauseElement`

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

**kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see `ColumnElement`)

compile(*bind=None*, *dialect=None*, ***kw*)

inherited from the `compile()` *method of* `ClauseElement`

Compile this SQL expression.

The return value is a `Compiled` object. Calling `str()` or `unicode()` on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the `params` accessor.

Parameters

- **bind** – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.
- **column_keys** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- **dialect** – A `Dialect` instance from which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- **inline** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

How do I render SQL expressions as strings, possibly with bound parameters inlined?

correspond_on_equivalents(*column*, *equivalents*)
inherited from the `correspond_on_equivalents()` *method of* `FromClause`

Return corresponding_column for the given column, or if None search for a match in the given dictionary.

corresponding_column(*column*, *require_embedded=False*)
inherited from the `corresponding_column()` *method of* `FromClause`

Given a `ColumnElement`, return the exported `ColumnElement` object from this `Selectable` which corresponds to that original `Column` via a common ancestor column.

Parameters

- **column** – the target `ColumnElement` to be matched
- **require_embedded** – only return corresponding columns for the given `ColumnElement`, if the given `ColumnElement` is actually present within a sub-element of this `FromClause`. Normally the column will match if it merely shares a common ancestor with one of the exported columns of this `FromClause`.

count(*whereclause=None*, ***params*)
inherited from the `count()` *method of* `FromClause`

return a SELECT COUNT generated against this `FromClause`.

cte(*name=None*, *recursive=False*)
inherited from the `cte()` *method of* `SelectBase`

Return a new `CTE`, or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

SQLAlchemy detects `CTE` objects, which are treated similarly to `Alias` objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

New in version 0.7.6.

Parameters

- **name** – name given to the common table expression. Like `_FromClause.alias()`, the name can be left as `None` in which case an anonymous symbol will be used at query compile time.
- **recursive** – if `True`, will render `WITH RECURSIVE`. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.

The following examples illustrate two examples from Postgresql's documentation at http://www.postgresql.org/docs/8.4/static/queries-with.html.

Example 1, non recursive:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,
    Column('region', String),
    Column('amount', Integer),
    Column('product', String),
```

```
        Column('quantity', Integer)
    )

    regional_sales = select([
                        orders.c.region,
                        func.sum(orders.c.amount).label('total_sales')
                    ]).group_by(orders.c.region).cte("regional_sales")


    top_regions = select([regional_sales.c.region]).\
            where(
                regional_sales.c.total_sales >
                select([
                    func.sum(regional_sales.c.total_sales)/10
                ])
            ).cte("top_regions")

    statement = select([
                orders.c.region,
                orders.c.product,
                func.sum(orders.c.quantity).label("product_units"),
                func.sum(orders.c.amount).label("product_sales")
        ]).where(orders.c.region.in_(
            select([top_regions.c.region])
        )).group_by(orders.c.region, orders.c.product)

    result = conn.execute(statement).fetchall()
```

Example 2, WITH RECURSIVE:

```
    from sqlalchemy import (Table, Column, String, Integer,
                            MetaData, select, func)

    metadata = MetaData()

    parts = Table('parts', metadata,
        Column('part', String),
        Column('sub_part', String),
        Column('quantity', Integer),
    )

    included_parts = select([
                        parts.c.sub_part,
                        parts.c.part,
                        parts.c.quantity]).\
                        where(parts.c.part=='our part').\
                        cte(recursive=True)


    incl_alias = included_parts.alias()
    parts_alias = parts.alias()
    included_parts = included_parts.union_all(
        select([
            parts_alias.c.sub_part,
            parts_alias.c.part,
            parts_alias.c.quantity
        ]).
            where(parts_alias.c.part==incl_alias.c.sub_part)
```

```
        )

        statement = select([
                    included_parts.c.sub_part,
                    func.sum(included_parts.c.quantity).
                      label('total_quantity')
                ]).\
                group_by(included_parts.c.sub_part)

        result = conn.execute(statement).fetchall()
```

See also:

orm.query.Query.cte() - ORM version of *SelectBase.cte()*.

description

inherited from the *description* *attribute of* *FromClause*

a brief description of this FromClause.

Used primarily for error message formatting.

execute(**multiparams*, ***params*)

inherited from the *execute()* *method of* *Executable*

Compile and execute this *Executable*.

execution_options(***kw*)

inherited from the *execution_options()* *method of* *Executable*

Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per *Connection* basis. Additionally, the *Engine* and ORM *Query* objects provide access to execution options which they in turn configure upon connections.

The *execution_options()* method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See *Connection.execution_options()* for a full list of possible options.

See also:

Connection.execution_options()

Query.execution_options()

for_update

Provide legacy dialect support for the `for_update` attribute.

foreign_keys

inherited from the *foreign_keys* *attribute of* *FromClause*

Return the collection of ForeignKey objects which this FromClause references.

get_children(***kwargs*)

inherited from the *get_children()* *method of* *ClauseElement*

Return immediate child elements of this *ClauseElement*.

This is used for visit traversal.

**kwargs may contain flags that change the collection that is returned, for example to return a subset of items in order to cut down on larger traversals, or to return child items from a different context (such as schema-level collections instead of clause-level).

group_by(**clauses*)
return a new selectable with the given list of GROUP BY criterion applied.

The criterion will be appended to any pre-existing GROUP BY criterion.

is_derived_from(*fromclause*)
inherited from the `is_derived_from()` *method of* `FromClause`

Return True if this FromClause is 'derived' from the given FromClause.

An example would be an Alias of a Table is derived from that Table.

join(*right*, *onclause=None*, *isouter=False*)
inherited from the `join()` *method of* `FromClause`

Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                    user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.

See also:

`join()` - standalone function

`Join` - the type of object produced

label(*name*)
inherited from the `label()` *method of* `SelectBase`

return a 'scalar' representation of this selectable, embedded as a subquery with a label.

See also:

`as_scalar()`.

limit(*limit*)
return a new selectable with the given LIMIT criterion applied.

This is a numerical value which usually renders as a `LIMIT` expression in the resulting select. Backends that don't support `LIMIT` will attempt to provide similar functionality.

Changed in version 1.0.0: - *`Select.limit()`* can now accept arbitrary SQL expressions as well as integer values.

Parameters **`limit`** – an integer LIMIT parameter, or a SQL expression that provides an integer result.

`offset`(*offset*)

return a new selectable with the given OFFSET criterion applied.

This is a numeric value which usually renders as an `OFFSET` expression in the resulting select. Backends that don't support `OFFSET` will attempt to provide similar functionality.

Changed in version 1.0.0: - *`Select.offset()`* can now accept arbitrary SQL expressions as well as integer values.

Parameters **`offset`** – an integer OFFSET parameter, or a SQL expression that provides an integer result.

`order_by`(**clauses*)

return a new selectable with the given list of ORDER BY criterion applied.

The criterion will be appended to any pre-existing ORDER BY criterion.

`outerjoin`(*right*, *onclause=None*)

inherited from the `outerjoin()` method of `FromClause`

Return a *`Join`* from this *`FromClause`* to another *`FromClause`*, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                         user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **`right`** – the right side of the join; this is any *`FromClause`* object such as a *`Table`* object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **`onclause`** – a SQL expression representing the ON clause of the join. If left at `None`, *`FromClause.join()`* will attempt to join the two tables based on a foreign key relationship.

See also:

`FromClause.join()`

`Join`

`params`(**optionaldict*, ***kwargs*)

inherited from the `params()` method of `ClauseElement`

Return a copy with *`bindparam()`* elements replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

primary_key

inherited from the `primary_key` *attribute of* `FromClause`

Return the collection of Column objects which comprise the primary key of this FromClause.

replace_selectable(*old*, *alias*)

inherited from the `replace_selectable()` *method of* `FromClause`

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

scalar(**multiparams*, ***params*)

inherited from the `scalar()` *method of* `Executable`

Compile and execute this `Executable`, returning the result's scalar representation.

select(*whereclause=None*, ***params*)

inherited from the `select()` *method of* `FromClause`

return a SELECT of this `FromClause`.

See also:

`select()` - general purpose method which allows for arbitrary column lists.

self_group(*against=None*)

inherited from the `self_group()` *method of* `ClauseElement`

Apply a 'grouping' to this `ClauseElement`.

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by `select()` constructs when placed into the FROM clause of another `select()`. (Note that subqueries should be normally created using the `Select.alias()` method, as many platforms require nested SELECT statements to be named).

As expressions are composed together, the application of `self_group()` is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like `x OR (y AND z)` - AND takes precedence over OR.

The base `self_group()` method of `ClauseElement` just returns self.

unique_params(**optionaldict*, ***kwargs*)

inherited from the `unique_params()` *method of* `ClauseElement`

Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

with_for_update(*nowait=False*, *read=False*, *of=None*)

Specify a `FOR UPDATE` clause for this `GenerativeSelect`.

E.g.:

```
stmt = select([table]).with_for_update(nowait=True)
```

On a database like Postgresql or Oracle, the above would render a statement like:

```
SELECT table.a, table.b FROM table FOR UPDATE NOWAIT
```

on other backends, the `nowait` option is ignored and instead would produce:

```
SELECT table.a, table.b FROM table FOR UPDATE
```

When called with no arguments, the statement will render with the suffix `FOR UPDATE`. Additional arguments can then be provided which allow for common database-specific variants.

Parameters

- **nowait** – boolean; will render `FOR UPDATE NOWAIT` on Oracle and Postgresql dialects.
- **read** – boolean; will render `LOCK IN SHARE MODE` on MySQL, `FOR SHARE` on Postgresql. On Postgresql, when combined with `nowait`, will render `FOR SHARE NOWAIT`.
- **of** – SQL expression or list of SQL expression elements (typically *Column* objects or a compatible expression) which will render into a `FOR UPDATE OF` clause; supported by PostgreSQL and Oracle. May render as a table or as a column depending on backend.

New in version 0.9.0.

class `sqlalchemy.sql.expression.`**`HasPrefixes`**

`prefix_with`(**expr*, ***kw*)

Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

```
stmt = table.insert().prefix_with("LOW_PRIORITY", dialect="mysql")
```

Multiple prefixes can be specified by multiple calls to *prefix_with()*.

Parameters

- ***expr** – textual or *ClauseElement* construct which will be rendered following the INSERT, UPDATE, or DELETE keyword.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect.

class `sqlalchemy.sql.expression.`**`HasSuffixes`**

`suffix_with`(**expr*, ***kw*)

Add one or more expressions following the statement as a whole.

This is used to support backend-specific suffix keywords on certain constructs.

E.g.:

```
stmt = select([col1, col2]).cte().suffix_with(
    "cycle empno set y_cycle to 1 default 0", dialect="oracle")
```

Multiple suffixes can be specified by multiple calls to *suffix_with()*.

Parameters

- ***expr** – textual or *ClauseElement* construct which will be rendered following the target clause.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this suffix to only that dialect.

class sqlalchemy.sql.expression.**Join**(*left*, *right*, *onclause=None*, *isouter=False*)

Bases: *sqlalchemy.sql.expression.FromClause*

represent a JOIN construct between two *FromClause* elements.

The public constructor function for *Join* is the module-level *join()* function, as well as the *FromClause.join()* method of any *FromClause* (e.g. such as *Table*).

See also:

join()

FromClause.join()

__init__(*left*, *right*, *onclause=None*, *isouter=False*)

Construct a new *Join*.

The usual entrypoint here is the *join()* function or the *FromClause.join()* method of any *FromClause* object.

alias(*name=None*, *flat=False*)

return an alias of this *Join*.

The default behavior here is to first produce a SELECT construct from this *Join*, then to produce an *Alias* from that. So given a join of the form:

```
j = table_a.join(table_b, table_a.c.id == table_b.c.a_id)
```

The JOIN by itself would look like:

```
table_a JOIN table_b ON table_a.id = table_b.a_id
```

Whereas the alias of the above, `j.alias()`, would in a SELECT context look like:

```
(SELECT table_a.id AS table_a_id, table_b.id AS table_b_id,
    table_b.a_id AS table_b_a_id
    FROM table_a
    JOIN table_b ON table_a.id = table_b.a_id) AS anon_1
```

The equivalent long-hand form, given a *Join* object `j`, is:

```
from sqlalchemy import select, alias
j = alias(
    select([j.left, j.right]).\
        select_from(j).\
        with_labels(True).\
        correlate(False),
    name=name
)
```

The selectable produced by *Join.alias()* features the same columns as that of the two individual selectables presented under a single name - the individual columns are "auto-labeled", meaning the `.c.` collection of the resulting *Alias* represents the names of the individual columns using a `<tablename>_<columname>` scheme:

```
j.c.table_a_id
j.c.table_b_a_id
```

Join.alias() also features an alternate option for aliasing joins which produces no enclosing SELECT and does not normally apply labels to the column names. The `flat=True` option will call *FromClause.alias()* against the left and right sides individually. Using this option, no new `SELECT` is produced; we instead, from a construct as below:

```
j = table_a.join(table_b, table_a.c.id == table_b.c.a_id)
j = j.alias(flat=True)
```

we get a result like this:

```
table_a AS table_a_1 JOIN table_b AS table_b_1 ON
table_a_1.id = table_b_1.a_id
```

The `flat=True` argument is also propagated to the contained selectables, so that a composite join such as:

```
j = table_a.join(
        table_b.join(table_c,
                table_b.c.id == table_c.c.b_id),
        table_b.c.a_id == table_a.c.id
    ).alias(flat=True)
```

Will produce an expression like:

```
table_a AS table_a_1 JOIN (
        table_b AS table_b_1 JOIN table_c AS table_c_1
        ON table_b_1.id = table_c_1.b_id
) ON table_a_1.id = table_b_1.a_id
```

The standalone *alias()* function as well as the base *FromClause.alias()* method also support the `flat=True` argument as a no-op, so that the argument can be passed to the `alias()` method of any selectable.

New in version 0.9.0: Added the `flat=True` option to create "aliases" of joins without enclosing inside of a SELECT subquery.

Parameters

- **name** – name given to the alias.
- **flat** – if True, produce an alias of the left and right sides of this *Join* and return the join of those two selectables. This produces join expression that does not include an enclosing SELECT.

  New in version 0.9.0.

See also:

alias()

c

inherited from the c attribute of FromClause

An alias for the *columns* attribute.

columns

inherited from the columns attribute of FromClause

A named-based collection of *ColumnElement* objects maintained by this *FromClause*.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

compare(*other*, ***kw*)

inherited from the `compare()` *method of* `ClauseElement`

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

**kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see `ColumnElement`)

compile(*bind=None*, *dialect=None*, ***kw*)

inherited from the `compile()` *method of* `ClauseElement`

Compile this SQL expression.

The return value is a `Compiled` object. Calling `str()` or `unicode()` on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the `params` accessor.

Parameters

- **bind** – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.
- **column_keys** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- **dialect** – A `Dialect` instance from which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- **inline** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

How do I render SQL expressions as strings, possibly with bound parameters inlined?

correspond_on_equivalents(*column*, *equivalents*)
inherited from the `correspond_on_equivalents()` *method of* `FromClause`

Return corresponding_column for the given column, or if None search for a match in the given dictionary.

corresponding_column(*column*, *require_embedded=False*)
inherited from the `corresponding_column()` *method of* `FromClause`

Given a `ColumnElement`, return the exported `ColumnElement` object from this `Selectable` which corresponds to that original `Column` via a common ancestor column.

Parameters

- **column** – the target `ColumnElement` to be matched
- **require_embedded** – only return corresponding columns for the given `ColumnElement`, if the given `ColumnElement` is actually present within a sub-element of this `FromClause`. Normally the column will match if it merely shares a common ancestor with one of the exported columns of this `FromClause`.

count(*whereclause=None*, ***params*)
inherited from the `count()` *method of* `FromClause`

return a SELECT COUNT generated against this `FromClause`.

foreign_keys
inherited from the `foreign_keys` *attribute of* `FromClause`

Return the collection of ForeignKey objects which this FromClause references.

join(*right*, *onclause=None*, *isouter=False*)
inherited from the `join()` *method of* `FromClause`

Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                    user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.

See also:

`join()` - standalone function

`Join` - the type of object produced

outerjoin(*right*, *onclause=None*)

inherited from the `outerjoin()` *method of* `FromClause`

Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                         user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.

See also:

`FromClause.join()`

`Join`

params(**optionaldict*, ***kwargs*)

inherited from the `params()` *method of* `ClauseElement`

Return a copy with `bindparam()` elements replaced.

Returns a copy of this ClauseElement with `bindparam()` elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

primary_key

inherited from the `primary_key` *attribute of* `FromClause`

Return the collection of Column objects which comprise the primary key of this FromClause.

replace_selectable(*old*, *alias*)

inherited from the `replace_selectable()` *method of* `FromClause`

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

select(*whereclause=None*, ***kwargs*)

Create a `Select` from this `Join`.

The equivalent long-hand form, given a `Join` object `j`, is:

```
from sqlalchemy import select
j = select([j.left, j.right], **kw).\
            where(whereclause).\
            select_from(j)
```

Parameters

- **whereclause** – the WHERE criterion that will be sent to the `select()` function
- ****kwargs** – all other kwargs are sent to the underlying `select()` function.

unique_params(**optionaldict*, ***kwargs*)

inherited from the `unique_params()` *method of* `ClauseElement`

Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

class sqlalchemy.sql.expression.**ScalarSelect**(*element*)

Bases: `sqlalchemy.sql.expression.Generative`, `sqlalchemy.sql.expression.Grouping`

where(*crit*)

Apply a WHERE clause to the SELECT statement referred to by this `ScalarSelect`.

class sqlalchemy.sql.expression.**Select**(*columns=None*, *whereclause=None*, *from_obj=None*, *distinct=False*, *having=None*, *correlate=True*, *prefixes=None*, *suffixes=None*, ***kwargs*)

Bases: `sqlalchemy.sql.expression.HasPrefixes`, `sqlalchemy.sql.expression.HasSuffixes`, `sqlalchemy.sql.expression.GenerativeSelect`

Represents a `SELECT` statement.

__init__(*columns=None*, *whereclause=None*, *from_obj=None*, *distinct=False*, *having=None*, *correlate=True*, *prefixes=None*, *suffixes=None*, ***kwargs*)

Construct a new `Select` object.

This constructor is mirrored as a public API function; see `select()` for a full usage and argument description.

alias(*name=None*, *flat=False*)

inherited from the `alias()` *method of* `FromClause`

return an alias of this `FromClause`.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

append_column(*column*)

append the given column expression to the columns clause of this select() construct.

This is an **in-place** mutation method; the `column()` method is preferred, as it provides standard method chaining.

append_correlation(*fromclause*)

append the given correlation expression to this select() construct.

This is an **in-place** mutation method; the `correlate()` method is preferred, as it provides standard method chaining.

append_from(*fromclause*)
append the given FromClause expression to this select() construct's FROM clause.

This is an **in-place** mutation method; the `select_from()` method is preferred, as it provides standard method chaining.

append_group_by(**clauses*)
inherited from the `append_group_by()` *method of* `GenerativeSelect`

Append the given GROUP BY criterion applied to this selectable.

The criterion will be appended to any pre-existing GROUP BY criterion.

This is an **in-place** mutation method; the `group_by()` method is preferred, as it provides standard method chaining.

append_having(*having*)
append the given expression to this select() construct's HAVING criterion.

The expression will be joined to existing HAVING criterion via AND.

This is an **in-place** mutation method; the `having()` method is preferred, as it provides standard method chaining.

append_order_by(**clauses*)
inherited from the `append_order_by()` *method of* `GenerativeSelect`

Append the given ORDER BY criterion applied to this selectable.

The criterion will be appended to any pre-existing ORDER BY criterion.

This is an **in-place** mutation method; the `order_by()` method is preferred, as it provides standard method chaining.

append_prefix(*clause*)
append the given columns clause prefix expression to this select() construct.

This is an **in-place** mutation method; the `prefix_with()` method is preferred, as it provides standard method chaining.

append_whereclause(*whereclause*)
append the given expression to this select() construct's WHERE criterion.

The expression will be joined to existing WHERE criterion via AND.

This is an **in-place** mutation method; the `where()` method is preferred, as it provides standard method chaining.

apply_labels()
inherited from the `apply_labels()` *method of* `GenerativeSelect`

return a new selectable with the 'use_labels' flag set to True.

This will result in column expressions being generated using labels against their table name, such as "SELECT somecolumn AS tablename_somecolumn". This allows selectables which contain multiple FROM clauses to produce a unique set of column names regardless of name conflicts among the individual FROM clauses.

as_scalar()
inherited from the `as_scalar()` *method of* `SelectBase`

return a 'scalar' representation of this selectable, which can be used as a column expression.

Typically, a select statement which has only one column in its columns clause is eligible to be used as a scalar expression.

The returned object is an instance of `ScalarSelect`.

autocommit()

inherited from the `autocommit()` *method of* `SelectBase`

return a new selectable with the 'autocommit' flag set to True.

Deprecated since version 0.6: `autocommit()` is deprecated. Use `Executable.execution_options()` with the 'autocommit' flag.

c

inherited from the `c` *attribute of* `FromClause`

An alias for the `columns` attribute.

column(*column*)

return a new select() construct with the given column expression added to its columns clause.

columns

inherited from the `columns` *attribute of* `FromClause`

A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

compare(*other*, ***kw*)

inherited from the `compare()` *method of* `ClauseElement`

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

**kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see `ColumnElement`)

compile(*bind=None*, *dialect=None*, ***kw*)

inherited from the `compile()` *method of* `ClauseElement`

Compile this SQL expression.

The return value is a `Compiled` object. Calling `str()` or `unicode()` on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the `params` accessor.

Parameters

- **bind** – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.
- **column_keys** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- **dialect** – A `Dialect` instance from which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- **inline** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to

create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.

- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

How do I render SQL expressions as strings, possibly with bound parameters inlined?

correlate(**fromclauses*)

return a new `Select` which will correlate the given FROM clauses to that of an enclosing `Select`.

Calling this method turns off the `Select` object's default behavior of "auto-correlation". Normally, FROM elements which appear in a `Select` that encloses this one via its WHERE clause, ORDER BY, HAVING or columns clause will be omitted from this `Select` object's FROM clause. Setting an explicit correlation collection using the `Select.correlate()` method provides a fixed list of FROM objects that can potentially take place in this process.

When `Select.correlate()` is used to apply specific FROM clauses for correlation, the FROM elements become candidates for correlation regardless of how deeply nested this `Select` object is, relative to an enclosing `Select` which refers to the same FROM object. This is in contrast to the behavior of "auto-correlation" which only correlates to an immediate enclosing `Select`. Multi-level correlation ensures that the link between enclosed and enclosing `Select` is always via at least one WHERE/ORDER BY/HAVING/columns clause in order for correlation to take place.

If `None` is passed, the `Select` object will correlate none of its FROM entries, and all will render unconditionally in the local FROM clause.

Parameters ***fromclauses** – a list of one or more `FromClause` constructs, or other compatible constructs (i.e. ORM-mapped classes) to become part of the correlate collection.

Changed in version 0.8.0: ORM-mapped classes are accepted by `Select.correlate()`.

Changed in version 0.8.0: The `Select.correlate()` method no longer unconditionally removes entries from the FROM clause; instead, the candidate FROM entries must also be matched by a FROM entry located in an enclosing `Select`, which ultimately encloses this one as present in the WHERE clause, ORDER BY clause, HAVING clause, or columns clause of an enclosing `Select()`.

Changed in version 0.8.2: explicit correlation takes place via any level of nesting of `Select` objects; in previous 0.8 versions, correlation would only occur relative to the immediate enclosing `Select` construct.

See also:

`Select.correlate_except()`

Correlated Subqueries

correlate_except(**fromclauses*)

return a new `Select` which will omit the given FROM clauses from the auto-correlation process.

Calling `Select.correlate_except()` turns off the `Select` object's default behavior of "auto-correlation" for the given FROM elements. An element specified here will unconditionally appear in the FROM list, while all other FROM elements remain subject to normal auto-correlation behaviors.

Changed in version 0.8.2: The `Select.correlate_except()` method was improved to fully prevent FROM clauses specified here from being omitted from the immediate FROM clause of this `Select`.

If `None` is passed, the `Select` object will correlate all of its FROM entries.

Changed in version 0.8.2: calling `correlate_except(None)` will correctly auto-correlate all FROM clauses.

Parameters ***fromclauses** – a list of one or more `FromClause` constructs, or other compatible constructs (i.e. ORM-mapped classes) to become part of the correlate-exception collection.

See also:

`Select.correlate()`

Correlated Subqueries

correspond_on_equivalents(*column*, *equivalents*)

inherited from the `correspond_on_equivalents()` *method of* `FromClause`

Return corresponding_column for the given column, or if None search for a match in the given dictionary.

corresponding_column(*column*, *require_embedded=False*)

inherited from the `corresponding_column()` *method of* `FromClause`

Given a `ColumnElement`, return the exported `ColumnElement` object from this `Selectable` which corresponds to that original `Column` via a common ancestor column.

Parameters

- **column** – the target `ColumnElement` to be matched
- **require_embedded** – only return corresponding columns for the given `ColumnElement`, if the given `ColumnElement` is actually present within a sub-element of this `FromClause`. Normally the column will match if it merely shares a common ancestor with one of the exported columns of this `FromClause`.

count(*whereclause=None*, ***params*)

inherited from the `count()` *method of* `FromClause`

return a SELECT COUNT generated against this `FromClause`.

cte(*name=None*, *recursive=False*)

inherited from the `cte()` *method of* `SelectBase`

Return a new `CTE`, or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

SQLAlchemy detects `CTE` objects, which are treated similarly to `Alias` objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

New in version 0.7.6.

Parameters

- **name** – name given to the common table expression. Like `_FromClause.alias()`, the name can be left as `None` in which case an anonymous symbol will be used at query compile time.
- **recursive** – if `True`, will render `WITH RECURSIVE`. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.

The following examples illustrate two examples from Postgresql's documentation at http://www.postgresql.org/docs/8.4/static/queries-with.html.

Example 1, non recursive:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,
    Column('region', String),
    Column('amount', Integer),
    Column('product', String),
    Column('quantity', Integer)
)

regional_sales = select([
                    orders.c.region,
                    func.sum(orders.c.amount).label('total_sales')
                ]).group_by(orders.c.region).cte("regional_sales")


top_regions = select([regional_sales.c.region]).\
        where(
            regional_sales.c.total_sales >
            select([
                func.sum(regional_sales.c.total_sales)/10
            ])
        ).cte("top_regions")

statement = select([
            orders.c.region,
            orders.c.product,
            func.sum(orders.c.quantity).label("product_units"),
            func.sum(orders.c.amount).label("product_sales")
    ]).where(orders.c.region.in_(
        select([top_regions.c.region])
    )).group_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()
```

Example 2, WITH RECURSIVE:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,
```

```
            Column('part', String),
            Column('sub_part', String),
            Column('quantity', Integer),
        )

        included_parts = select([
                            parts.c.sub_part,
                            parts.c.part,
                            parts.c.quantity]).\
                            where(parts.c.part=='our part').\
                            cte(recursive=True)


        incl_alias = included_parts.alias()
        parts_alias = parts.alias()
        included_parts = included_parts.union_all(
            select([
                parts_alias.c.sub_part,
                parts_alias.c.part,
                parts_alias.c.quantity
            ]).
                where(parts_alias.c.part==incl_alias.c.sub_part)
        )

        statement = select([
                    included_parts.c.sub_part,
                    func.sum(included_parts.c.quantity).
                      label('total_quantity')
                ]).\
                group_by(included_parts.c.sub_part)

        result = conn.execute(statement).fetchall()
```

See also:

orm.query.Query.cte() - ORM version of *SelectBase.cte()*.

description

inherited from the `description` *attribute of* `FromClause`

a brief description of this FromClause.

Used primarily for error message formatting.

distinct(**expr*)

Return a new select() construct which will apply DISTINCT to its columns clause.

Parameters ***expr** – optional column expressions. When present, the Postgresql dialect will render a `DISTINCT ON (<expressions>>)` construct.

except_(*other*, ***kwargs*)

return a SQL EXCEPT of this select() construct against the given selectable.

except_all(*other*, ***kwargs*)

return a SQL EXCEPT ALL of this select() construct against the given selectable.

execute(**multiparams*, ***params*)

inherited from the `execute()` *method of* `Executable`

Compile and execute this `Executable`.

execution_options(***kw*)

inherited from the `execution_options()` *method of* `Executable`

Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per `Connection` basis. Additionally, the `Engine` and ORM `Query` objects provide access to execution options which they in turn configure upon connections.

The `execution_options()` method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See `Connection.execution_options()` for a full list of possible options.

See also:

`Connection.execution_options()`

`Query.execution_options()`

for_update

inherited from the `for_update` *attribute of* `GenerativeSelect`

Provide legacy dialect support for the `for_update` attribute.

foreign_keys

inherited from the `foreign_keys` *attribute of* `FromClause`

Return the collection of ForeignKey objects which this FromClause references.

froms

Return the displayed list of FromClause elements.

get_children(*column_collections=True*, ***kwargs*)

return child elements as per the ClauseElement specification.

group_by(**clauses*)

inherited from the `group_by()` *method of* `GenerativeSelect`

return a new selectable with the given list of GROUP BY criterion applied.

The criterion will be appended to any pre-existing GROUP BY criterion.

having(*having*)

return a new select() construct with the given expression added to its HAVING clause, joined to the existing clause via AND, if any.

inner_columns

an iterator of all ColumnElement expressions which would be rendered into the columns clause of the resulting SELECT statement.

intersect(*other*, ***kwargs*)

return a SQL INTERSECT of this select() construct against the given selectable.

intersect_all(*other*, ***kwargs*)

return a SQL INTERSECT ALL of this select() construct against the given selectable.

join(*right*, *onclause=None*, *isouter=False*)

inherited from the `join()` *method of* `FromClause`

Return a *Join* from this *FromClause* to another *FromClause*.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                    user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any *FromClause* object such as a *Table* object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, *FromClause.join()* will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.

See also:

join() - standalone function

Join - the type of object produced

label(*name*)

inherited from the *label()* *method of* *SelectBase*

return a 'scalar' representation of this selectable, embedded as a subquery with a label.

See also:

as_scalar().

limit(*limit*)

inherited from the *limit()* *method of* *GenerativeSelect*

return a new selectable with the given LIMIT criterion applied.

This is a numerical value which usually renders as a `LIMIT` expression in the resulting select. Backends that don't support `LIMIT` will attempt to provide similar functionality.

Changed in version 1.0.0: - *Select.limit()* can now accept arbitrary SQL expressions as well as integer values.

Parameters **limit** – an integer LIMIT parameter, or a SQL expression that provides an integer result.

locate_all_froms(**args*, ***kw*)

return a Set of all FromClause elements referenced by this Select.

This set is a superset of that returned by the `froms` property, which is specifically for those FromClause elements that would actually be rendered.

offset(*offset*)

inherited from the *offset()* *method of* *GenerativeSelect*

return a new selectable with the given OFFSET criterion applied.

This is a numeric value which usually renders as an `OFFSET` expression in the resulting select. Backends that don't support `OFFSET` will attempt to provide similar functionality.

Changed in version 1.0.0: - *`Select.offset()`* can now accept arbitrary SQL expressions as well as integer values.

Parameters **`offset`** – an integer OFFSET parameter, or a SQL expression that provides an integer result.

`order_by`(**clauses*)
inherited from the `order_by()` method of `GenerativeSelect`

return a new selectable with the given list of ORDER BY criterion applied.

The criterion will be appended to any pre-existing ORDER BY criterion.

`outerjoin`(*right*, *onclause=None*)
inherited from the `outerjoin()` method of `FromClause`

Return a *`Join`* from this *`FromClause`* to another *`FromClause`*, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                         user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **`right`** – the right side of the join; this is any *`FromClause`* object such as a *`Table`* object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **`onclause`** – a SQL expression representing the ON clause of the join. If left at `None`, *`FromClause.join()`* will attempt to join the two tables based on a foreign key relationship.

See also:

`FromClause.join()`

`Join`

`params`(**optionaldict*, ***kwargs*)
inherited from the `params()` method of `ClauseElement`

Return a copy with *`bindparam()`* elements replaced.

Returns a copy of this ClauseElement with *`bindparam()`* elements replaced with values taken from the given dictionary:

```
>>> clause = column('x') + bindparam('foo')
>>> print clause.compile().params
{'foo':None}
>>> print clause.params({'foo':7}).compile().params
{'foo':7}
```

prefix_with(**expr*, ***kw*)

inherited from the `prefix_with()` *method of* `HasPrefixes`

Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

```
stmt = table.insert().prefix_with("LOW_PRIORITY", dialect="mysql")
```

Multiple prefixes can be specified by multiple calls to `prefix_with()`.

Parameters

- ***expr** – textual or `ClauseElement` construct which will be rendered following the INSERT, UPDATE, or DELETE keyword.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect.

primary_key

inherited from the `primary_key` *attribute of* `FromClause`

Return the collection of Column objects which comprise the primary key of this FromClause.

reduce_columns(*only_synonyms=True*)

Return a new :func'.select' construct with redundantly named, equivalently-valued columns removed from the columns clause.

"Redundant" here means two columns where one refers to the other either based on foreign key, or via a simple equality comparison in the WHERE clause of the statement. The primary purpose of this method is to automatically construct a select statement with all uniquely-named columns, without the need to use table-qualified labels as `apply_labels()` does.

When columns are omitted based on foreign key, the referred-to column is the one that's kept. When columns are omitted based on WHERE eqivalence, the first column in the columns clause is the one that's kept.

Parameters **only_synonyms** – when True, limit the removal of columns to those which have the same name as the equivalent. Otherwise, all columns that are equivalent to another are removed.

New in version 0.8.

replace_selectable(*old*, *alias*)

inherited from the `replace_selectable()` *method of* `FromClause`

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

scalar(**multiparams*, ***params*)

inherited from the `scalar()` *method of* `Executable`

Compile and execute this `Executable`, returning the result's scalar representation.

select(*whereclause=None*, ***params*)

inherited from the `select()` *method of* `FromClause`

return a SELECT of this `FromClause`.

See also:

`select()` - general purpose method which allows for arbitrary column lists.

select_from(*fromclause*)

return a new `select()` construct with the given FROM expression merged into its list of FROM objects.

E.g.:

```
table1 = table('t1', column('a'))
table2 = table('t2', column('b'))
s = select([table1.c.a]).\
    select_from(
        table1.join(table2, table1.c.a==table2.c.b)
    )
```

The "from" list is a unique set on the identity of each element, so adding an already present `Table` or other selectable will have no effect. Passing a `Join` that refers to an already present `Table` or other selectable will have the effect of concealing the presence of that selectable as an individual element in the rendered FROM list, instead rendering it into a JOIN clause.

While the typical purpose of `Select.select_from()` is to replace the default, derived FROM clause with a join, it can also be called with individual table elements, multiple times if desired, in the case that the FROM clause cannot be fully derived from the columns clause:

```
select([func.count('*')]).select_from(table1)
```

self_group(*against=None*)

return a 'grouping' construct as per the ClauseElement specification.

This produces an element that can be embedded in an expression. Note that this method is called automatically as needed when constructing expressions and should not require explicit use.

suffix_with(**expr*, ***kw*)

inherited from the `suffix_with()` *method of* `HasSuffixes`

Add one or more expressions following the statement as a whole.

This is used to support backend-specific suffix keywords on certain constructs.

E.g.:

```
stmt = select([col1, col2]).cte().suffix_with(
    "cycle empno set y_cycle to 1 default 0", dialect="oracle")
```

Multiple suffixes can be specified by multiple calls to `suffix_with()`.

Parameters

- ***expr** – textual or `ClauseElement` construct which will be rendered following the target clause.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this suffix to only that dialect.

union(*other*, ***kwargs*)

return a SQL UNION of this select() construct against the given selectable.

union_all(*other*, ***kwargs*)

return a SQL UNION ALL of this select() construct against the given selectable.

unique_params(**optionaldict*, ***kwargs*)

inherited from the `unique_params()` *method of* `ClauseElement`

Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

where(*whereclause*)

return a new select() construct with the given expression added to its WHERE clause, joined to the existing clause via AND, if any.

with_for_update(*nowait=False*, *read=False*, *of=None*)

inherited from the `with_for_update()` *method of* `GenerativeSelect`

Specify a `FOR UPDATE` clause for this `GenerativeSelect`.

E.g.:

```
stmt = select([table]).with_for_update(nowait=True)
```

On a database like Postgresql or Oracle, the above would render a statement like:

```
SELECT table.a, table.b FROM table FOR UPDATE NOWAIT
```

on other backends, the `nowait` option is ignored and instead would produce:

```
SELECT table.a, table.b FROM table FOR UPDATE
```

When called with no arguments, the statement will render with the suffix `FOR UPDATE`. Additional arguments can then be provided which allow for common database-specific variants.

Parameters

- **nowait** – boolean; will render `FOR UPDATE NOWAIT` on Oracle and Postgresql dialects.
- **read** – boolean; will render `LOCK IN SHARE MODE` on MySQL, `FOR SHARE` on Postgresql. On Postgresql, when combined with `nowait`, will render `FOR SHARE NOWAIT`.
- **of** – SQL expression or list of SQL expression elements (typically `Column` objects or a compatible expression) which will render into a `FOR UPDATE OF` clause; supported by PostgreSQL and Oracle. May render as a table or as a column depending on backend.

New in version 0.9.0.

with_hint(*selectable*, *text*, *dialect_name='*')

Add an indexing or other executional context hint for the given selectable to this `Select`.

The text of the hint is rendered in the appropriate location for the database backend in use, relative to the given `Table` or `Alias` passed as the `selectable` argument. The dialect implementation typically uses Python string substitution syntax with the token `%(name)s` to render the name of the table or alias. E.g. when using Oracle, the following:

```
select([mytable]).\
    with_hint(mytable, "index(%(name)s ix_mytable)")
```

Would render SQL as:

```
select /*+ index(mytable ix_mytable) */ ... from mytable
```

The `dialect_name` option will limit the rendering of a particular hint to a particular backend. Such as, to add hints for both Oracle and Sybase simultaneously:

```
select([mytable]).\
    with_hint(mytable, "index(%(name)s ix_mytable)", 'oracle').\
    with_hint(mytable, "WITH INDEX ix_mytable", 'sybase')
```

See also:

Select.with_statement_hint()

with_only_columns(*columns*)

Return a new *select()* construct with its columns clause replaced with the given columns.

Changed in version 0.7.3: Due to a bug fix, this method has a slight behavioral change as of version 0.7.3. Prior to version 0.7.3, the FROM clause of a *select()* was calculated upfront and as new columns were added; in 0.7.3 and later it's calculated at compile time, fixing an issue regarding late binding of columns to parent tables. This changes the behavior of *Select.with_only_columns()* in that FROM clauses no longer represented in the new list are dropped, but this behavior is more consistent in that the FROM clauses are consistently derived from the current columns clause. The original intent of this method is to allow trimming of the existing columns list to be fewer columns than originally present; the use case of replacing the columns list with an entirely different one hadn't been anticipated until 0.7.3 was released; the usage guidelines below illustrate how this should be done.

This method is exactly equivalent to as if the original *select()* had been called with the given columns clause. I.e. a statement:

```
s = select([table1.c.a, table1.c.b])
s = s.with_only_columns([table1.c.b])
```

should be exactly equivalent to:

```
s = select([table1.c.b])
```

This means that FROM clauses which are only derived from the column list will be discarded if the new column list no longer contains that FROM:

```
>>> table1 = table('t1', column('a'), column('b'))
>>> table2 = table('t2', column('a'), column('b'))
>>> s1 = select([table1.c.a, table2.c.b])
>>> print s1
SELECT t1.a, t2.b FROM t1, t2
>>> s2 = s1.with_only_columns([table2.c.b])
>>> print s2
SELECT t2.b FROM t1
```

The preferred way to maintain a specific FROM clause in the construct, assuming it won't be represented anywhere else (i.e. not in the WHERE clause, etc.) is to set it using *Select.select_from()*:

```
>>> s1 = select([table1.c.a, table2.c.b]).\
...         select_from(table1.join(table2,
...                 table1.c.a==table2.c.a))
>>> s2 = s1.with_only_columns([table2.c.b])
>>> print s2
SELECT t2.b FROM t1 JOIN t2 ON t1.a=t2.a
```

Care should also be taken to use the correct set of column objects passed to *Select.with_only_columns()*. Since the method is essentially equivalent to calling the *select()* construct in the first place with the given columns, the columns passed to *Select.with_only_columns()* should usually be a subset of those which were passed to the *select()* construct, not those which are available from the `.c` collection of that *select()*. That is:

```
s = select([table1.c.a, table1.c.b]).select_from(table1)
s = s.with_only_columns([table1.c.b])
```

and **not**:

```
# usually incorrect
s = s.with_only_columns([s.c.b])
```

The latter would produce the SQL:

```
SELECT b
FROM (SELECT t1.a AS a, t1.b AS b
FROM t1), t1
```

Since the *select()* construct is essentially being asked to select both from `table1` as well as itself.

with_statement_hint(*text*, *dialect_name='*'*)

add a statement hint to this *Select*.

This method is similar to *Select.with_hint()* except that it does not require an individual table, and instead applies to the statement as a whole.

Hints here are specific to the backend database and may include directives such as isolation levels, file directives, fetch directives, etc.

New in version 1.0.0.

See also:

Select.with_hint()

class sqlalchemy.sql.expression.**Selectable**

Bases: *sqlalchemy.sql.expression.ClauseElement*

mark a class as being selectable

class sqlalchemy.sql.expression.**SelectBase**

Bases: *sqlalchemy.sql.expression.Executable*, *sqlalchemy.sql.expression.FromClause*

Base class for SELECT statements.

This includes *Select*, *CompoundSelect* and *TextAsFrom*.

as_scalar()

return a 'scalar' representation of this selectable, which can be used as a column expression.

Typically, a select statement which has only one column in its columns clause is eligible to be used as a scalar expression.

The returned object is an instance of *ScalarSelect*.

autocommit()

return a new selectable with the 'autocommit' flag set to True.

Deprecated since version 0.6: `autocommit()` is deprecated. Use *Executable.execution_options()* with the 'autocommit' flag.

cte(*name=None*, *recursive=False*)

Return a new *CTE*, or Common Table Expression instance.

Common table expressions are a SQL standard whereby SELECT statements can draw upon secondary statements specified along with the primary statement, using a clause called "WITH". Special semantics regarding UNION can also be employed to allow "recursive" queries, where a SELECT statement can draw upon the set of rows that have previously been selected.

SQLAlchemy detects *CTE* objects, which are treated similarly to *Alias* objects, as special elements to be delivered to the FROM clause of the statement as well as to a WITH clause at the top of the statement.

New in version 0.7.6.

Parameters

- **name** – name given to the common table expression. Like `_FromClause.alias()`, the name can be left as `None` in which case an anonymous symbol will be used at query compile time.
- **recursive** – if `True`, will render `WITH RECURSIVE`. A recursive common table expression is intended to be used in conjunction with UNION ALL in order to derive rows from those already selected.

The following examples illustrate two examples from Postgresql's documentation at http://www.postgresql.org/docs/8.4/static/queries-with.html.

Example 1, non recursive:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

orders = Table('orders', metadata,
    Column('region', String),
    Column('amount', Integer),
    Column('product', String),
    Column('quantity', Integer)
)

regional_sales = select([
                    orders.c.region,
                    func.sum(orders.c.amount).label('total_sales')
                ]).group_by(orders.c.region).cte("regional_sales")


top_regions = select([regional_sales.c.region]).\
        where(
            regional_sales.c.total_sales >
            select([
                func.sum(regional_sales.c.total_sales)/10
            ])
        ).cte("top_regions")

statement = select([
            orders.c.region,
            orders.c.product,
            func.sum(orders.c.quantity).label("product_units"),
            func.sum(orders.c.amount).label("product_sales")
    ]).where(orders.c.region.in_(
        select([top_regions.c.region])
    )).group_by(orders.c.region, orders.c.product)

result = conn.execute(statement).fetchall()
```

Example 2, WITH RECURSIVE:

```
from sqlalchemy import (Table, Column, String, Integer,
                        MetaData, select, func)

metadata = MetaData()

parts = Table('parts', metadata,
```

```
            Column('part', String),
            Column('sub_part', String),
            Column('quantity', Integer),
        )

        included_parts = select([
                            parts.c.sub_part,
                            parts.c.part,
                            parts.c.quantity]).\
                            where(parts.c.part=='our part').\
                            cte(recursive=True)


        incl_alias = included_parts.alias()
        parts_alias = parts.alias()
        included_parts = included_parts.union_all(
            select([
                parts_alias.c.sub_part,
                parts_alias.c.part,
                parts_alias.c.quantity
            ]).
                where(parts_alias.c.part==incl_alias.c.sub_part)
        )

        statement = select([
                    included_parts.c.sub_part,
                    func.sum(included_parts.c.quantity).
                      label('total_quantity')
                ]).\
                group_by(included_parts.c.sub_part)

        result = conn.execute(statement).fetchall()
```

See also:

orm.query.Query.cte() - ORM version of *SelectBase.cte()*.

label(*name*)

return a 'scalar' representation of this selectable, embedded as a subquery with a label.

See also:

as_scalar().

class sqlalchemy.sql.expression.**TableClause**(*name*, **columns*)

Bases: sqlalchemy.sql.expression.Immutable, *sqlalchemy.sql.expression.FromClause*

Represents a minimal "table" construct.

This is a lightweight table object that has only a name and a collection of columns, which are typically produced by the *expression.column()* function:

```
from sqlalchemy import table, column

user = table("user",
        column("id"),
        column("name"),
        column("description"),
)
```

The `TableClause` construct serves as the base for the more commonly used `Table` object, providing the usual set of `FromClause` services including the `.c.` collection and statement generation methods.

It does **not** provide all the additional schema-level services of `Table`, including constraints, references to other tables, or support for `MetaData`-level services. It's useful on its own as an ad-hoc construct used to generate quick SQL statements when a more fully fledged `Table` is not on hand.

__init__(*name*, **columns*)

Construct a new `TableClause` object.

This constructor is mirrored as a public API function; see `table()` for a full usage and argument description.

alias(*name=None*, *flat=False*)

inherited from the `alias()` *method of* `FromClause`

return an alias of this `FromClause`.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

c

inherited from the `c` *attribute of* `FromClause`

An alias for the `columns` attribute.

columns

inherited from the `columns` *attribute of* `FromClause`

A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

compare(*other*, ***kw*)

inherited from the `compare()` *method of* `ClauseElement`

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

**kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see `ColumnElement`)

compile(*bind=None*, *dialect=None*, ***kw*)

inherited from the `compile()` *method of* `ClauseElement`

Compile this SQL expression.

The return value is a `Compiled` object. Calling `str()` or `unicode()` on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the `params` accessor.

Parameters

- **bind** – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.

- **column_keys** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- **dialect** – A `Dialect` instance from which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- **inline** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

How do I render SQL expressions as strings, possibly with bound parameters inlined?

correspond_on_equivalents(*column*, *equivalents*)
inherited from the `correspond_on_equivalents()` *method of* `FromClause`

Return corresponding_column for the given column, or if None search for a match in the given dictionary.

corresponding_column(*column*, *require_embedded=False*)
inherited from the `corresponding_column()` *method of* `FromClause`

Given a `ColumnElement`, return the exported `ColumnElement` object from this `Selectable` which corresponds to that original `Column` via a common ancestor column.

Parameters

- **column** – the target `ColumnElement` to be matched
- **require_embedded** – only return corresponding columns for the given `ColumnElement`, if the given `ColumnElement` is actually present within a sub-element of this `FromClause`. Normally the column will match if it merely shares a common ancestor with one of the exported columns of this `FromClause`.

count(*whereclause=None*, ***params*)
return a SELECT COUNT generated against this `TableClause`.

delete(*whereclause=None*, ***kwargs*)
Generate a `delete()` construct against this `TableClause`.

E.g.:

```
table.delete().where(table.c.id==7)
```

See `delete()` for argument and usage information.

foreign_keys
inherited from the `foreign_keys` *attribute of* `FromClause`

Return the collection of ForeignKey objects which this FromClause references.

implicit_returning = **False**
`TableClause` doesn't support having a primary key or column -level defaults, so implicit returning doesn't apply.

insert(*values=None*, *inline=False*, ***kwargs*)
Generate an `insert()` construct against this `TableClause`.

E.g.:

```
table.insert().values(name='foo')
```

See `insert()` for argument and usage information.

is_derived_from(*fromclause*)
inherited from the `is_derived_from()` *method of* `FromClause`

Return True if this FromClause is 'derived' from the given FromClause.

An example would be an Alias of a Table is derived from that Table.

join(*right*, *onclause=None*, *isouter=False*)
inherited from the `join()` *method of* `FromClause`

Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                    user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.

See also:

`join()` - standalone function

`Join` - the type of object produced

outerjoin(*right*, *onclause=None*)

inherited from the `outerjoin()` *method of* `FromClause`

Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                         user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.

See also:

`FromClause.join()`

`Join`

primary_key

inherited from the `primary_key` *attribute of* `FromClause`

Return the collection of Column objects which comprise the primary key of this FromClause.

replace_selectable(*old*, *alias*)

inherited from the `replace_selectable()` *method of* `FromClause`

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this `FromClause`.

select(*whereclause=None*, ***params*)

inherited from the `select()` *method of* `FromClause`

return a SELECT of this `FromClause`.

See also:

`select()` - general purpose method which allows for arbitrary column lists.

self_group(*against=None*)

inherited from the `self_group()` *method of* `ClauseElement`

Apply a 'grouping' to this `ClauseElement`.

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by `select()` constructs when placed into the FROM clause of another `select()`. (Note that subqueries should be normally created using the `Select.alias()` method, as many platforms require nested SELECT statements to be named).

As expressions are composed together, the application of `self_group()` is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like `x OR (y AND z)` - AND takes precedence over OR.

The base `self_group()` method of `ClauseElement` just returns self.

update(*whereclause=None*, *values=None*, *inline=False*, ***kwargs*)

Generate an `update()` construct against this `TableClause`.

E.g.:

```
table.update().where(table.c.id==7).values(name='foo')
```

See `update()` for argument and usage information.

class sqlalchemy.sql.expression.**TextAsFrom**(*text*, *columns*)

Bases: `sqlalchemy.sql.expression.SelectBase`

Wrap a `TextClause` construct within a `SelectBase` interface.

This allows the `TextClause` object to gain a `.c` collection and other FROM-like capabilities such as `FromClause.alias()`, `SelectBase.cte()`, etc.

The `TextAsFrom` construct is produced via the `TextClause.columns()` method - see that method for details.

New in version 0.9.0.

See also:

`text()`

`TextClause.columns()`

3.2.3 Insert, Updates, Deletes

INSERT, UPDATE and DELETE statements build on a hierarchy starting with `UpdateBase`. The `Insert` and `Update` constructs build on the intermediary `ValuesBase`.

sqlalchemy.sql.expression.**delete**(*table*, *whereclause=None*, *bind=None*, *returning=None*, *prefixes=None*, ***dialect_kw*)

Construct `Delete` object.

Similar functionality is available via the `delete()` method on `Table`.

Parameters

- **table** – The table to delete rows from.
- **whereclause** – A `ClauseElement` describing the `WHERE` condition of the `DELETE` statement. Note that the `where()` generative method may be used instead.

See also:

Deletes - SQL Expression Tutorial

sqlalchemy.sql.expression.**insert**(*table*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return_defaults=False*, ***dialect_kw*)

Construct an `Insert` object.

Similar functionality is available via the `insert()` method on `Table`.

Parameters

- **table** – *TableClause* which is the subject of the insert.
- **values** – collection of values to be inserted; see *Insert.values()* for a description of allowed formats here. Can be omitted entirely; a *Insert* construct will also dynamically render the VALUES clause at execution time based on the parameters passed to *Connection.execute()*.
- **inline** – if True, no attempt will be made to retrieve the SQL-generated default values to be provided within the statement; in particular, this allows SQL expressions to be rendered 'inline' within the statement without the need to pre-execute them beforehand; for backends that support "returning", this turns off the "implicit returning" feature for the statement.

If both *values* and compile-time bind parameters are present, the compile-time bind parameters override the information specified within *values* on a per-key basis.

The keys within *values* can be either *Column* objects or their string identifiers. Each key may reference one of:

•a literal data value (i.e. string, number, etc.);

•a Column object;

•a SELECT statement.

If a SELECT statement is specified which references this INSERT statement's table, the statement will be correlated against the INSERT statement.

See also:

Insert Expressions - SQL Expression Tutorial

Inserts, Updates and Deletes - SQL Expression Tutorial

sqlalchemy.sql.expression.**update**(*table*, *whereclause=None*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return_defaults=False*, *preserve_parameter_order=False*, ***dialect_kw*)

Construct an *Update* object.

E.g.:

```
from sqlalchemy import update

stmt = update(users).where(users.c.id==5).\
        values(name='user #5')
```

Similar functionality is available via the *update()* method on *Table*:

```
stmt = users.update().\
            where(users.c.id==5).\
            values(name='user #5')
```

Parameters

- **table** – A *Table* object representing the database table to be updated.
- **whereclause** – Optional SQL expression describing the WHERE condition of the UPDATE statement. Modern applications may prefer to use the generative *where()* method to specify the WHERE clause.

  The WHERE clause can refer to multiple tables. For databases which support this, an UPDATE FROM clause will be generated, or on MySQL, a multi-table update. The statement will fail on databases that don't have support for multi-table update statements. A

SQL-standard method of referring to additional tables in the WHERE clause is to use a correlated subquery:

```
users.update().values(name='ed').where(
        users.c.name==select([addresses.c.email_address]).\
                    where(addresses.c.user_id==users.c.id).\
                    as_scalar()
        )
```

Changed in version 0.7.4: The WHERE clause can refer to multiple tables.

- **values** – Optional dictionary which specifies the `SET` conditions of the `UPDATE`. If left as `None`, the `SET` conditions are determined from those parameters passed to the statement during the execution and/or compilation of the statement. When compiled standalone without any parameters, the `SET` clause generates for all columns.

  Modern applications may prefer to use the generative *`Update.values()`* method to set the values of the UPDATE statement.

- **inline** – if True, SQL defaults present on *`Column`* objects via the `default` keyword will be compiled 'inline' into the statement and not pre-executed. This means that their values will not be available in the dictionary returned from *`ResultProxy.last_updated_params()`*.

- **preserve_parameter_order** – if True, the update statement is expected to receive parameters **only** via the *`Update.values()`* method, and they must be passed as a Python `list` of 2-tuples. The rendered UPDATE statement will emit the SET clause for each referenced column maintaining this order.

  New in version 1.0.10.

  See also:

  Parameter-Ordered Updates - full example of the *`preserve_parameter_order`* flag

If both `values` and compile-time bind parameters are present, the compile-time bind parameters override the information specified within `values` on a per-key basis.

The keys within `values` can be either *`Column`* objects or their string identifiers (specifically the "key" of the *`Column`*, normally but not necessarily equivalent to its "name"). Normally, the *`Column`* objects used here are expected to be part of the target *`Table`* that is the table to be updated. However when using MySQL, a multiple-table UPDATE statement can refer to columns from any of the tables referred to in the WHERE clause.

The values referred to in `values` are typically:

- a literal data value (i.e. string, number, etc.)
- a SQL expression, such as a related *`Column`*, a scalar-returning *`select()`* construct, etc.

When combining *`select()`* constructs within the values clause of an *`update()`* construct, the subquery represented by the *`select()`* should be *correlated* to the parent table, that is, providing criterion which links the table inside the subquery to the outer table being updated:

```
users.update().values(
        name=select([addresses.c.email_address]).\
                where(addresses.c.user_id==users.c.id).\
                as_scalar()
    )
```

See also:

Inserts, Updates and Deletes - SQL Expression Language Tutorial

class `sqlalchemy.sql.expression.`**`Delete`**(*table*, *whereclause=None*, *bind=None*, *returning=None*, *prefixes=None*, ***dialect_kw*)

Bases: `sqlalchemy.sql.expression.UpdateBase`

Represent a DELETE construct.

The `Delete` object is created using the `delete()` function.

`__init__`(*table*, *whereclause=None*, *bind=None*, *returning=None*, *prefixes=None*, ***dialect_kw*)

Construct a new `Delete` object.

This constructor is mirrored as a public API function; see `delete()` for a full usage and argument description.

`argument_for`(*dialect_name*, *argument_name*, *default*)

inherited from the `argument_for()` *method of* `DialectKWArgs`

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **`dialect_name`** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **`argument_name`** – name of the parameter.
- **`default`** – default value of the parameter.

New in version 0.9.4.

`bind`

inherited from the `bind` *attribute of* `UpdateBase`

Return a 'bind' linked to this `UpdateBase` or a `Table` associated with it.

`compare`(*other*, ***kw*)

inherited from the `compare()` *method of* `ClauseElement`

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

**kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see `ColumnElement`)

compile(*bind=None*, *dialect=None*, ***kw*)

inherited from the `compile()` method of `ClauseElement`

Compile this SQL expression.

The return value is a `Compiled` object. Calling `str()` or `unicode()` on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the `params` accessor.

Parameters

- **bind** – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.
- **column_keys** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- **dialect** – A `Dialect` instance from which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- **inline** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

How do I render SQL expressions as strings, possibly with bound parameters inlined?

dialect_kwargs

inherited from the `dialect_kwargs` attribute of `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options

inherited from the `dialect_options` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

execute(**multiparams*, ***params*)

inherited from the `execute()` *method of* `Executable`

Compile and execute this `Executable`.

execution_options(***kw*)

inherited from the `execution_options()` *method of* `Executable`

Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per `Connection` basis. Additionally, the `Engine` and ORM `Query` objects provide access to execution options which they in turn configure upon connections.

The `execution_options()` method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See `Connection.execution_options()` for a full list of possible options.

See also:

`Connection.execution_options()`

`Query.execution_options()`

kwargs

inherited from the `kwargs` *attribute of* `DialectKWArgs`

A synonym for `DialectKWArgs.dialect_kwargs`.

params(**arg*, ***kw*)

inherited from the `params()` *method of* `UpdateBase`

Set the parameters for the statement.

This method raises `NotImplementedError` on the base class, and is overridden by `ValuesBase` to provide the SET/VALUES clause of UPDATE and INSERT.

prefix_with(**expr*, ***kw*)

inherited from the `prefix_with()` *method of* `HasPrefixes`

Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

```
stmt = table.insert().prefix_with("LOW_PRIORITY", dialect="mysql")
```

Multiple prefixes can be specified by multiple calls to `prefix_with()`.

Parameters

- ***expr** – textual or `ClauseElement` construct which will be rendered following the INSERT, UPDATE, or DELETE keyword.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect.

returning(**cols*)

inherited from the `returning()` *method of* `UpdateBase`

Add a RETURNING or equivalent clause to this statement.

e.g.:

```
stmt = table.update().\
          where(table.c.data == 'value').\
          values(status='X').\
          returning(table.c.server_flag,
                    table.c.updated_timestamp)

for server_flag, updated_timestamp in connection.execute(stmt):
    print(server_flag, updated_timestamp)
```

The given collection of column expressions should be derived from the table that is the target of the INSERT, UPDATE, or DELETE. While `Column` objects are typical, the elements can also be expressions:

```
stmt = table.insert().returning(
    (table.c.first_name + " " + table.c.last_name).
    label('fullname'))
```

Upon compilation, a RETURNING clause, or database equivalent, will be rendered within the statement. For INSERT and UPDATE, the values are the newly inserted/updated values. For DELETE, the values are those of the rows which were deleted.

Upon execution, the values of the columns to be returned are made available via the result set and can be iterated using `ResultProxy.fetchone()` and similar. For DBAPIs which do not natively support returning values (i.e. cx_oracle), SQLAlchemy will approximate this behavior at the result level so that a reasonable amount of behavioral neutrality is provided.

Note that not all databases/DBAPIs support RETURNING. For those backends with no support, an exception is raised upon compilation and/or execution. For those who do support it, the functionality across backends varies greatly, including restrictions on executemany() and other statements which return multiple rows. Please read the documentation notes for the database in use in order to determine the availability of RETURNING.

See also:

`ValuesBase.return_defaults()` - an alternative method tailored towards efficient fetching of server-side defaults and triggers for single-row INSERTs or UPDATEs.

scalar(**multiparams*, ***params*)
inherited from the `scalar()` *method of* `Executable`

Compile and execute this `Executable`, returning the result's scalar representation.

self_group(*against=None*)
inherited from the `self_group()` *method of* `ClauseElement`

Apply a 'grouping' to this `ClauseElement`.

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by `select()` constructs when placed into the FROM clause of another `select()`. (Note that subqueries should be normally created using the `Select.alias()` method, as many platforms require nested SELECT statements to be named).

As expressions are composed together, the application of `self_group()` is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like `x OR (y AND z)` - AND takes precedence over OR.

The base `self_group()` method of `ClauseElement` just returns self.

unique_params(**optionaldict*, ***kwargs*)
inherited from the `unique_params()` *method of* `ClauseElement`

Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

where(*whereclause*)
Add the given WHERE clause to a newly returned delete construct.

with_hint(*text*, *selectable=None*, *dialect_name='*'*)
inherited from the `with_hint()` *method of* `UpdateBase`

Add a table hint for a single table to this INSERT/UPDATE/DELETE statement.

Note: `UpdateBase.with_hint()` currently applies only to Microsoft SQL Server. For MySQL INSERT/UPDATE/DELETE hints, use `UpdateBase.prefix_with()`.

The text of the hint is rendered in the appropriate location for the database backend in use, relative to the `Table` that is the subject of this statement, or optionally to that of the given `Table` passed as the `selectable` argument.

The `dialect_name` option will limit the rendering of a particular hint to a particular backend. Such as, to add a hint that only takes effect for SQL Server:

```
mytable.insert().with_hint("WITH (PAGLOCK)", dialect_name="mssql")
```

New in version 0.7.6.

Parameters

- **text** – Text of the hint.
- **selectable** – optional `Table` that specifies an element of the FROM clause within an UPDATE or DELETE to be the subject of the hint - applies only to certain backends.

- **dialect_name** – defaults to *, if specified as the name of a particular dialect, will apply these hints only when that dialect is in use.

class sqlalchemy.sql.expression.**Insert**(*table*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return_defaults=False*, ***dialect_kw*)

Bases: sqlalchemy.sql.expression.ValuesBase

Represent an INSERT construct.

The Insert object is created using the insert() function.

See also:

Insert Expressions

__init__(*table*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return_defaults=False*, ***dialect_kw*)

Construct a new Insert object.

This constructor is mirrored as a public API function; see insert() for a full usage and argument description.

argument_for(*dialect_name*, *argument_name*, *default*)

inherited from the argument_for() *method of* DialectKWArgs

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The DialectKWArgs.argument_for() method is a per-argument way adding extra arguments to the DefaultDialect.construct_arguments dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a NoSuchModuleError is raised. The dialect must also include an existing DefaultDialect.construct_arguments collection, indicating that it participates in the keyword-argument validation and default system, else ArgumentError is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

bind

inherited from the bind *attribute of* UpdateBase

Return a 'bind' linked to this UpdateBase or a Table associated with it.

compare(*other*, ***kw*)

inherited from the `compare()` *method of* `ClauseElement`

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

**kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see `ColumnElement`)

compile(*bind=None*, *dialect=None*, ***kw*)

inherited from the `compile()` *method of* `ClauseElement`

Compile this SQL expression.

The return value is a `Compiled` object. Calling `str()` or `unicode()` on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the `params` accessor.

Parameters

- **bind** – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.
- **column_keys** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- **dialect** – A `Dialect` instance from which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- **inline** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

How do I render SQL expressions as strings, possibly with bound parameters inlined?

dialect_kwargs

inherited from the `dialect_kwargs` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the *DialectKWArgs.dialect_options* collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The *DialectKWArgs.dialect_kwargs* collection is now writable.

See also:

DialectKWArgs.dialect_options - nested dictionary form

dialect_options

inherited from the dialect_options attribute of DialectKWArgs

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

DialectKWArgs.dialect_kwargs - flat dictionary form

execute(**multiparams*, ***params*)

inherited from the execute() method of Executable

Compile and execute this *Executable*.

execution_options(***kw*)

inherited from the execution_options() method of Executable

Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per *Connection* basis. Additionally, the *Engine* and ORM *Query* objects provide access to execution options which they in turn configure upon connections.

The *execution_options()* method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See *Connection.execution_options()* for a full list of possible options.

See also:

Connection.execution_options()

Query.execution_options()

from_select(*names*, *select*, *include_defaults=True*)

Return a new *Insert* construct which represents an `INSERT...FROM SELECT` statement.

e.g.:

```
sel = select([table1.c.a, table1.c.b]).where(table1.c.c > 5)
ins = table2.insert().from_select(['a', 'b'], sel)
```

Parameters

- **names** – a sequence of string column names or *Column* objects representing the target columns.
- **select** – a *select()* construct, *FromClause* or other construct which resolves into a *FromClause*, such as an ORM *Query* object, etc. The order of columns returned from this FROM clause should correspond to the order of columns sent as the `names` parameter; while this is not checked before passing along to the database, the database would normally raise an exception if these column lists don't correspond.
- **include_defaults** – if True, non-server default values and SQL expressions as specified on *Column* objects (as documented in *Column Insert/Update Defaults*) not otherwise specified in the list of names will be rendered into the INSERT and SELECT statements, so that these values are also included in the data to be inserted.

  Note: A Python-side default that uses a Python callable function will only be invoked **once** for the whole statement, and **not per row**.

  New in version 1.0.0: - *Insert.from_select()* now renders Python-side and SQL expression column defaults into the SELECT statement for columns otherwise not included in the list of column names.

Changed in version 1.0.0: an INSERT that uses FROM SELECT implies that the *insert.inline* flag is set to True, indicating that the statement will not attempt to fetch the "last inserted primary key" or other defaults. The statement deals with an arbitrary number of rows, so the *ResultProxy.inserted_primary_key* accessor does not apply.

New in version 0.8.3.

kwargs

inherited from the *kwargs* *attribute of* *DialectKWArgs*

A synonym for *DialectKWArgs.dialect_kwargs*.

params(**arg*, ***kw*)

inherited from the *params()* *method of* *UpdateBase*

Set the parameters for the statement.

This method raises `NotImplementedError` on the base class, and is overridden by *ValuesBase* to provide the SET/VALUES clause of UPDATE and INSERT.

prefix_with(**expr*, ***kw*)

inherited from the *prefix_with()* *method of* *HasPrefixes*

Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

```
stmt = table.insert().prefix_with("LOW_PRIORITY", dialect="mysql")
```

Multiple prefixes can be specified by multiple calls to *prefix_with()*.

Parameters

- ***expr** – textual or *ClauseElement* construct which will be rendered following the INSERT, UPDATE, or DELETE keyword.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect.

return_defaults(**cols*)

inherited from the *return_defaults()* *method of* *ValuesBase*

Make use of a RETURNING clause for the purpose of fetching server-side expressions and defaults.

E.g.:

```
stmt = table.insert().values(data='newdata').return_defaults()

result = connection.execute(stmt)

server_created_at = result.returned_defaults['created_at']
```

When used against a backend that supports RETURNING, all column values generated by SQL expression or server-side-default will be added to any existing RETURNING clause, provided that *UpdateBase.returning()* is not used simultaneously. The column values will then be available on the result using the *ResultProxy.returned_defaults* accessor as a dictionary, referring to values keyed to the *Column* object as well as its `.key`.

This method differs from *UpdateBase.returning()* in these ways:

1. *ValuesBase.return_defaults()* is only intended for use with an INSERT or an UPDATE statement that matches exactly one row. While the RETURNING construct in the general sense supports multiple rows for a multi-row UPDATE or DELETE statement, or for special cases of INSERT that return multiple rows (e.g. INSERT from SELECT, multi-valued VALUES clause), *ValuesBase.return_defaults()* is intended only for an "ORM-style" single-row INSERT/UPDATE statement. The row returned by the statement is also consumed implicitly when *ValuesBase.return_defaults()* is used. By contrast, *UpdateBase.returning()* leaves the RETURNING result-set intact with a collection of any number of rows.
2. It is compatible with the existing logic to fetch auto-generated primary key values, also known as "implicit returning". Backends that support RETURNING will automatically make use of RETURNING in order to fetch the value of newly generated primary keys; while the *UpdateBase.returning()* method circumvents this behavior, *ValuesBase.return_defaults()* leaves it intact.
3. It can be called against any backend. Backends that don't support RETURNING will skip the usage of the feature, rather than raising an exception. The return value of *ResultProxy.returned_defaults* will be `None`

ValuesBase.return_defaults() is used by the ORM to provide an efficient implementation for the `eager_defaults` feature of *mapper()*.

Parameters **cols** – optional list of column key names or *Column* objects. If omitted, all column expressions evaluated on the server are added to the returning list.

New in version 0.9.0.

See also:

UpdateBase.returning()

ResultProxy.returned_defaults

returning(**cols*)

inherited from the *returning()* *method of* *UpdateBase*

Add a RETURNING or equivalent clause to this statement.

e.g.:

```
stmt = table.update().\
          where(table.c.data == 'value').\
          values(status='X').\
          returning(table.c.server_flag,
                    table.c.updated_timestamp)

for server_flag, updated_timestamp in connection.execute(stmt):
    print(server_flag, updated_timestamp)
```

The given collection of column expressions should be derived from the table that is the target of the INSERT, UPDATE, or DELETE. While `Column` objects are typical, the elements can also be expressions:

```
stmt = table.insert().returning(
    (table.c.first_name + " " + table.c.last_name).
    label('fullname'))
```

Upon compilation, a RETURNING clause, or database equivalent, will be rendered within the statement. For INSERT and UPDATE, the values are the newly inserted/updated values. For DELETE, the values are those of the rows which were deleted.

Upon execution, the values of the columns to be returned are made available via the result set and can be iterated using `ResultProxy.fetchone()` and similar. For DBAPIs which do not natively support returning values (i.e. cx_oracle), SQLAlchemy will approximate this behavior at the result level so that a reasonable amount of behavioral neutrality is provided.

Note that not all databases/DBAPIs support RETURNING. For those backends with no support, an exception is raised upon compilation and/or execution. For those who do support it, the functionality across backends varies greatly, including restrictions on executemany() and other statements which return multiple rows. Please read the documentation notes for the database in use in order to determine the availability of RETURNING.

See also:

`ValuesBase.return_defaults()` - an alternative method tailored towards efficient fetching of server-side defaults and triggers for single-row INSERTs or UPDATEs.

scalar(**multiparams*, ***params*)

inherited from the `scalar()` *method of* `Executable`

Compile and execute this `Executable`, returning the result's scalar representation.

self_group(*against=None*)

inherited from the `self_group()` *method of* `ClauseElement`

Apply a 'grouping' to this `ClauseElement`.

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by `select()` constructs when placed into the FROM clause of another `select()`. (Note that subqueries should be normally created using the `Select.alias()` method, as many platforms require nested SELECT statements to be named).

As expressions are composed together, the application of `self_group()` is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like `x OR (y AND z)` - AND takes precedence over OR.

The base `self_group()` method of `ClauseElement` just returns self.

unique_params(**optionaldict*, ***kwargs*)

inherited from the `unique_params()` *method of* `ClauseElement`

Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

values(**args*, ***kwargs*)

inherited from the `values()` *method of* `ValuesBase`

specify a fixed VALUES clause for an INSERT statement, or the SET clause for an UPDATE.

Note that the `Insert` and `Update` constructs support per-execution time formatting of the VALUES and/or SET clauses, based on the arguments passed to `Connection.execute()`. However, the `ValuesBase.values()` method can be used to "fix" a particular set of parameters into the statement.

Multiple calls to `ValuesBase.values()` will produce a new construct, each one with the parameter list modified to include the new parameters sent. In the typical case of a single dictionary of parameters, the newly passed keys will replace the same keys in the previous construct. In the case of a list-based "multiple values" construct, each new list of values is extended onto the existing list of values.

Parameters

- ****kwargs** – key value pairs representing the string key of a `Column` mapped to the value to be rendered into the VALUES or SET clause:

```
users.insert().values(name="some name")

users.update().where(users.c.id==5).values(name="some name")
```

- ***args** – As an alternative to passing key/value parameters, a dictionary, tuple, or list of dictionaries or tuples can be passed as a single positional argument in order to form the VALUES or SET clause of the statement. The forms that are accepted vary based on whether this is an `Insert` or an `Update` construct.

  For either an `Insert` or `Update` construct, a single dictionary can be passed, which works the same as that of the kwargs form:

```
users.insert().values({"name": "some name"})

users.update().values({"name": "some new name"})
```

  Also for either form but more typically for the `Insert` construct, a tuple that contains an entry for every column in the table is also accepted:

```
users.insert().values((5, "some name"))
```

  The `Insert` construct also supports being passed a list of dictionaries or full-table-tuples, which on the server will render the less common SQL syntax of "multiple values" - this syntax is supported on backends such as SQLite, Postgresql, MySQL, but not necessarily others:

```
users.insert().values([
                      {"name": "some name"},
                      {"name": "some other name"},
                      {"name": "yet another name"},
                  ])
```

The above form would render a multiple VALUES statement similar to:

```
INSERT INTO users (name) VALUES
                (:name_1),
                (:name_2),
                (:name_3)
```

It is essential to note that **passing multiple values is NOT the same as using traditional executemany() form**. The above syntax is a **special** syntax not typically used. To emit an INSERT statement against multiple rows, the normal method is to pass a multiple values list to the `Connection.execute()` method, which is supported by all database backends and is generally more efficient for a very large number of parameters.

See also:

Executing Multiple Statements - an introduction to the traditional Core method of multiple parameter set invocation for INSERTs and other statements.

Changed in version 1.0.0: an INSERT that uses a multiple-VALUES clause, even a list of length one, implies that the `Insert.inline` flag is set to True, indicating that the statement will not attempt to fetch the "last inserted primary key" or other defaults. The statement deals with an arbitrary number of rows, so the `ResultProxy.inserted_primary_key` accessor does not apply.

Changed in version 1.0.0: A multiple-VALUES INSERT now supports columns with Python side default values and callables in the same way as that of an "executemany" style of invocation; the callable is invoked for each row. See *Python-side defaults invoked for each row invidually when using a multivalued insert* for other details.

The `Update` construct supports a special form which is a list of 2-tuples, which when provided must be passed in conjunction with the `preserve_parameter_order` parameter. This form causes the UPDATE statement to render the SET clauses using the order of parameters given to `Update.values()`, rather than the ordering of columns given in the `Table`.

New in version 1.0.10: - added support for parameter-ordered UPDATE statements via the `preserve_parameter_order` flag.

See also:

Parameter-Ordered Updates - full example of the `preserve_parameter_order` flag

See also:

Inserts, Updates and Deletes - SQL Expression Language Tutorial

`insert()` - produce an `INSERT` statement

`update()` - produce an `UPDATE` statement

with_hint(*text*, *selectable=None*, *dialect_name='*')

inherited from the `with_hint()` *method of* `UpdateBase`

Add a table hint for a single table to this INSERT/UPDATE/DELETE statement.

Note: `UpdateBase.with_hint()` currently applies only to Microsoft SQL Server. For MySQL INSERT/UPDATE/DELETE hints, use `UpdateBase.prefix_with()`.

The text of the hint is rendered in the appropriate location for the database backend in use, relative to the `Table` that is the subject of this statement, or optionally to that of the given `Table` passed as the `selectable` argument.

The `dialect_name` option will limit the rendering of a particular hint to a particular backend. Such as, to add a hint that only takes effect for SQL Server:

```
mytable.insert().with_hint("WITH (PAGLOCK)", dialect_name="mssql")
```

New in version 0.7.6.

Parameters

- **text** – Text of the hint.
- **selectable** – optional `Table` that specifies an element of the FROM clause within an UPDATE or DELETE to be the subject of the hint - applies only to certain backends.
- **dialect_name** – defaults to *, if specified as the name of a particular dialect, will apply these hints only when that dialect is in use.

class `sqlalchemy.sql.expression.`**`Update`**(*table*, *whereclause=None*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return_defaults=False*, *preserve_parameter_order=False*, ***dialect_kw*)

Bases: `sqlalchemy.sql.expression.ValuesBase`

Represent an Update construct.

The `Update` object is created using the `update()` function.

`__init__`(*table*, *whereclause=None*, *values=None*, *inline=False*, *bind=None*, *prefixes=None*, *returning=None*, *return_defaults=False*, *preserve_parameter_order=False*, ***dialect_kw*)

Construct a new `Update` object.

This constructor is mirrored as a public API function; see `update()` for a full usage and argument description.

`argument_for`(*dialect_name*, *argument_name*, *default*)

inherited from the `argument_for()` *method of* `DialectKWArgs`

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else

ArgumentError is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.

- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

bind

inherited from the bind attribute of UpdateBase

Return a 'bind' linked to this UpdateBase or a Table associated with it.

compare(*other*, ***kw*)

inherited from the compare() method of ClauseElement

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

**kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see ColumnElement)

compile(*bind=None*, *dialect=None*, ***kw*)

inherited from the compile() method of ClauseElement

Compile this SQL expression.

The return value is a Compiled object. Calling str() or unicode() on the returned value will yield a string representation of the result. The Compiled object also can return a dictionary of bind parameter names and values using the params accessor.

Parameters

- **bind** – An Engine or Connection from which a Compiled will be acquired. This argument takes precedence over this ClauseElement's bound engine, if any.
- **column_keys** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If None, all columns from the target table object are rendered.
- **dialect** – A Dialect instance from which a Compiled will be acquired. This argument takes precedence over the *bind* argument as well as this ClauseElement's bound engine, if any.
- **inline** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the literal_binds flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))
```

```
s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

How do I render SQL expressions as strings, possibly with bound parameters inlined?

dialect_kwargs

inherited from the `dialect_kwargs` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options

inherited from the `dialect_options` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

execute(**multiparams*, ***params*)

inherited from the `execute()` *method of* `Executable`

Compile and execute this `Executable`.

execution_options(***kw*)

inherited from the `execution_options()` *method of* `Executable`

Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per `Connection` basis. Additionally, the `Engine` and ORM `Query` objects provide access to execution options which they in turn configure upon connections.

The `execution_options()` method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See `Connection.execution_options()` for a full list of possible options.

See also:

`Connection.execution_options()`

`Query.execution_options()`

kwargs

inherited from the `kwargs` *attribute of* `DialectKWArgs`

A synonym for `DialectKWArgs.dialect_kwargs`.

params(**arg*, ***kw*)

inherited from the `params()` *method of* `UpdateBase`

Set the parameters for the statement.

This method raises `NotImplementedError` on the base class, and is overridden by `ValuesBase` to provide the SET/VALUES clause of UPDATE and INSERT.

prefix_with(**expr*, ***kw*)

inherited from the `prefix_with()` *method of* `HasPrefixes`

Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

```
stmt = table.insert().prefix_with("LOW_PRIORITY", dialect="mysql")
```

Multiple prefixes can be specified by multiple calls to `prefix_with()`.

Parameters

- ***expr** – textual or `ClauseElement` construct which will be rendered following the INSERT, UPDATE, or DELETE keyword.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect.

return_defaults(**cols*)

inherited from the `return_defaults()` *method of* `ValuesBase`

Make use of a RETURNING clause for the purpose of fetching server-side expressions and defaults.

E.g.:

```
stmt = table.insert().values(data='newdata').return_defaults()

result = connection.execute(stmt)

server_created_at = result.returned_defaults['created_at']
```

When used against a backend that supports RETURNING, all column values generated by SQL expression or server-side-default will be added to any existing RETURNING clause, provided that `UpdateBase.returning()` is not used simultaneously. The column values will then be available on the result using the `ResultProxy.returned_defaults` accessor as a dictionary, referring to values keyed to the `Column` object as well as its `.key`.

This method differs from `UpdateBase.returning()` in these ways:

1. *ValuesBase.return_defaults()* is only intended for use with an INSERT or an UPDATE statement that matches exactly one row. While the RETURNING construct in the general sense supports multiple rows for a multi-row UPDATE or DELETE statement, or for special cases of INSERT that return multiple rows (e.g. INSERT from SELECT, multi-valued VALUES clause), *ValuesBase.return_defaults()* is intended only for an "ORM-style" single-row INSERT/UPDATE statement. The row returned by the statement is also consumed implicitly when *ValuesBase.return_defaults()* is used. By contrast, *UpdateBase.returning()* leaves the RETURNING result-set intact with a collection of any number of rows.
2. It is compatible with the existing logic to fetch auto-generated primary key values, also known as "implicit returning". Backends that support RETURNING will automatically make use of RETURNING in order to fetch the value of newly generated primary keys; while the *UpdateBase.returning()* method circumvents this behavior, *ValuesBase.return_defaults()* leaves it intact.
3. It can be called against any backend. Backends that don't support RETURNING will skip the usage of the feature, rather than raising an exception. The return value of *ResultProxy.returned_defaults* will be `None`

ValuesBase.return_defaults() is used by the ORM to provide an efficient implementation for the `eager_defaults` feature of *mapper()*.

Parameters **cols** – optional list of column key names or *Column* objects. If omitted, all column expressions evaluated on the server are added to the returning list.

New in version 0.9.0.

See also:

UpdateBase.returning()

ResultProxy.returned_defaults

returning(**cols*)

inherited from the *returning()* *method of* *UpdateBase*

Add a RETURNING or equivalent clause to this statement.

e.g.:

```
stmt = table.update().\
          where(table.c.data == 'value').\
          values(status='X').\
          returning(table.c.server_flag,
                    table.c.updated_timestamp)

for server_flag, updated_timestamp in connection.execute(stmt):
    print(server_flag, updated_timestamp)
```

The given collection of column expressions should be derived from the table that is the target of the INSERT, UPDATE, or DELETE. While *Column* objects are typical, the elements can also be expressions:

```
stmt = table.insert().returning(
    (table.c.first_name + " " + table.c.last_name).
    label('fullname'))
```

Upon compilation, a RETURNING clause, or database equivalent, will be rendered within the statement. For INSERT and UPDATE, the values are the newly inserted/updated values. For DELETE, the values are those of the rows which were deleted.

Upon execution, the values of the columns to be returned are made available via the result set and can be iterated using *ResultProxy.fetchone()* and similar. For DBAPIs which do not natively support

returning values (i.e. cx_oracle), SQLAlchemy will approximate this behavior at the result level so that a reasonable amount of behavioral neutrality is provided.

Note that not all databases/DBAPIs support RETURNING. For those backends with no support, an exception is raised upon compilation and/or execution. For those who do support it, the functionality across backends varies greatly, including restrictions on executemany() and other statements which return multiple rows. Please read the documentation notes for the database in use in order to determine the availability of RETURNING.

See also:

`ValuesBase.return_defaults()` - an alternative method tailored towards efficient fetching of server-side defaults and triggers for single-row INSERTs or UPDATEs.

scalar(**multiparams*, ***params*)
inherited from the `scalar()` *method of* `Executable`

Compile and execute this `Executable`, returning the result's scalar representation.

self_group(*against=None*)
inherited from the `self_group()` *method of* `ClauseElement`

Apply a 'grouping' to this `ClauseElement`.

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by `select()` constructs when placed into the FROM clause of another `select()`. (Note that subqueries should be normally created using the `Select.alias()` method, as many platforms require nested SELECT statements to be named).

As expressions are composed together, the application of `self_group()` is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like `x OR (y AND z)` - AND takes precedence over OR.

The base `self_group()` method of `ClauseElement` just returns self.

unique_params(**optionaldict*, ***kwargs*)
inherited from the `unique_params()` *method of* `ClauseElement`

Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

values(**args*, ***kwargs*)
inherited from the `values()` *method of* `ValuesBase`

specify a fixed VALUES clause for an INSERT statement, or the SET clause for an UPDATE.

Note that the `Insert` and `Update` constructs support per-execution time formatting of the VALUES and/or SET clauses, based on the arguments passed to `Connection.execute()`. However, the `ValuesBase.values()` method can be used to "fix" a particular set of parameters into the statement.

Multiple calls to `ValuesBase.values()` will produce a new construct, each one with the parameter list modified to include the new parameters sent. In the typical case of a single dictionary of parameters, the newly passed keys will replace the same keys in the previous construct. In the case of a list-based "multiple values" construct, each new list of values is extended onto the existing list of values.

Parameters

- ****kwargs** – key value pairs representing the string key of a `Column` mapped to the value to be rendered into the VALUES or SET clause:

```
users.insert().values(name="some name")

users.update().where(users.c.id==5).values(name="some name")
```

- ***args** – As an alternative to passing key/value parameters, a dictionary, tuple, or list of dictionaries or tuples can be passed as a single positional argument in order to form the VALUES or SET clause of the statement. The forms that are accepted vary based on whether this is an *Insert* or an *Update* construct.

  For either an *Insert* or *Update* construct, a single dictionary can be passed, which works the same as that of the kwargs form:

```
users.insert().values({"name": "some name"})

users.update().values({"name": "some new name"})
```

  Also for either form but more typically for the *Insert* construct, a tuple that contains an entry for every column in the table is also accepted:

```
users.insert().values((5, "some name"))
```

  The *Insert* construct also supports being passed a list of dictionaries or full-table-tuples, which on the server will render the less common SQL syntax of "multiple values" - this syntax is supported on backends such as SQLite, Postgresql, MySQL, but not necessarily others:

```
users.insert().values([
                      {"name": "some name"},
                      {"name": "some other name"},
                      {"name": "yet another name"},
                  ])
```

  The above form would render a multiple VALUES statement similar to:

```
INSERT INTO users (name) VALUES
                 (:name_1),
                 (:name_2),
                 (:name_3)
```

  It is essential to note that **passing multiple values is NOT the same as using traditional executemany() form**. The above syntax is a **special** syntax not typically used. To emit an INSERT statement against multiple rows, the normal method is to pass a multiple values list to the *Connection.execute()* method, which is supported by all database backends and is generally more efficient for a very large number of parameters.

  See also:

  Executing Multiple Statements - an introduction to the traditional Core method of multiple parameter set invocation for INSERTs and other statements.

  Changed in version 1.0.0: an INSERT that uses a multiple-VALUES clause, even a list of length one, implies that the *Insert.inline* flag is set to True, indicating that the statement will not attempt to fetch the "last inserted primary key" or other defaults. The statement deals with an arbitrary number of rows, so the *ResultProxy.inserted_primary_key* accessor does not apply.

Changed in version 1.0.0: A multiple-VALUES INSERT now supports columns with Python side default values and callables in the same way as that of an "executemany" style of invocation; the callable is invoked for each row. See *Python-side defaults invoked for each row invidually when using a multivalued insert* for other details.

The `Update` construct supports a special form which is a list of 2-tuples, which when provided must be passed in conjunction with the `preserve_parameter_order` parameter. This form causes the UPDATE statement to render the SET clauses using the order of parameters given to `Update.values()`, rather than the ordering of columns given in the `Table`.

New in version 1.0.10: - added support for parameter-ordered UPDATE statements via the `preserve_parameter_order` flag.

See also:

Parameter-Ordered Updates - full example of the `preserve_parameter_order` flag

See also:

Inserts, Updates and Deletes - SQL Expression Language Tutorial

`insert()` - produce an `INSERT` statement

`update()` - produce an `UPDATE` statement

where(*whereclause*)

return a new update() construct with the given expression added to its WHERE clause, joined to the existing clause via AND, if any.

with_hint(*text*, *selectable=None*, *dialect_name='*'*)

inherited from the `with_hint()` method of `UpdateBase`

Add a table hint for a single table to this INSERT/UPDATE/DELETE statement.

Note: `UpdateBase.with_hint()` currently applies only to Microsoft SQL Server. For MySQL INSERT/UPDATE/DELETE hints, use `UpdateBase.prefix_with()`.

The text of the hint is rendered in the appropriate location for the database backend in use, relative to the `Table` that is the subject of this statement, or optionally to that of the given `Table` passed as the `selectable` argument.

The `dialect_name` option will limit the rendering of a particular hint to a particular backend. Such as, to add a hint that only takes effect for SQL Server:

```
mytable.insert().with_hint("WITH (PAGLOCK)", dialect_name="mssql")
```

New in version 0.7.6.

Parameters

- **text** – Text of the hint.
- **selectable** – optional `Table` that specifies an element of the FROM clause within an UPDATE or DELETE to be the subject of the hint - applies only to certain backends.
- **dialect_name** – defaults to *, if specified as the name of a particular dialect, will apply these hints only when that dialect is in use.

class sqlalchemy.sql.expression.**UpdateBase**

Bases: *sqlalchemy.sql.base.DialectKWArgs*, *sqlalchemy.sql.expression.HasPrefixes*, *sqlalchemy.sql.expression.Executable*, *sqlalchemy.sql.expression.ClauseElement*

Form the base for INSERT, UPDATE, and DELETE statements.

__init__

inherited from the __init__ *attribute of* object

x.__init__(...) initializes x; see help(type(x)) for signature

argument_for(*dialect_name*, *argument_name*, *default*)

inherited from the argument_for() *method of* DialectKWArgs

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The *DialectKWArgs.argument_for()* method is a per-argument way adding extra arguments to the *DefaultDialect.construct_arguments* dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a *NoSuchModuleError* is raised. The dialect must also include an existing *DefaultDialect.construct_arguments* collection, indicating that it participates in the keyword-argument validation and default system, else *ArgumentError* is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

bind

Return a 'bind' linked to this *UpdateBase* or a *Table* associated with it.

compare(*other*, ***kw*)

inherited from the compare() *method of* ClauseElement

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

**kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see *ColumnElement*)

compile(*bind=None*, *dialect=None*, ***kw*)

inherited from the compile() *method of* ClauseElement

Compile this SQL expression.

The return value is a *Compiled* object. Calling `str()` or `unicode()` on the returned value will yield a string representation of the result. The *Compiled* object also can return a dictionary of bind parameter names and values using the `params` accessor.

Parameters

- **bind** – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this *ClauseElement*'s bound engine, if any.
- **column_keys** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- **dialect** – A `Dialect` instance from which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this *ClauseElement*'s bound engine, if any.
- **inline** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

How do I render SQL expressions as strings, possibly with bound parameters inlined?

dialect_kwargs

inherited from the *dialect_kwargs* *attribute of* *DialectKWArgs*

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the *DialectKWArgs.dialect_options* collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The *DialectKWArgs.dialect_kwargs* collection is now writable.

See also:

DialectKWArgs.dialect_options - nested dictionary form

dialect_options
inherited from the `dialect_options` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

execute(**multiparams*, ***params*)
inherited from the `execute()` *method of* `Executable`

Compile and execute this `Executable`.

execution_options(***kw*)
inherited from the `execution_options()` *method of* `Executable`

Set non-SQL options for the statement which take effect during execution.

Execution options can be set on a per-statement or per `Connection` basis. Additionally, the `Engine` and ORM `Query` objects provide access to execution options which they in turn configure upon connections.

The `execution_options()` method is generative. A new instance of this statement is returned that contains the options:

```
statement = select([table.c.x, table.c.y])
statement = statement.execution_options(autocommit=True)
```

Note that only a subset of possible execution options can be applied to a statement - these include "autocommit" and "stream_results", but not "isolation_level" or "compiled_cache". See `Connection.execution_options()` for a full list of possible options.

See also:

`Connection.execution_options()`

`Query.execution_options()`

get_children(***kwargs*)
inherited from the `get_children()` *method of* `ClauseElement`

Return immediate child elements of this `ClauseElement`.

This is used for visit traversal.

**kwargs may contain flags that change the collection that is returned, for example to return a subset of items in order to cut down on larger traversals, or to return child items from a different context (such as schema-level collections instead of clause-level).

kwargs
inherited from the `kwargs` *attribute of* `DialectKWArgs`

A synonym for `DialectKWArgs.dialect_kwargs`.

params(**arg*, ***kw*)
Set the parameters for the statement.

This method raises `NotImplementedError` on the base class, and is overridden by `ValuesBase` to provide the SET/VALUES clause of UPDATE and INSERT.

prefix_with(**expr*, ***kw*)

inherited from the `prefix_with()` *method of* `HasPrefixes`

Add one or more expressions following the statement keyword, i.e. SELECT, INSERT, UPDATE, or DELETE. Generative.

This is used to support backend-specific prefix keywords such as those provided by MySQL.

E.g.:

```
stmt = table.insert().prefix_with("LOW_PRIORITY", dialect="mysql")
```

Multiple prefixes can be specified by multiple calls to `prefix_with()`.

Parameters

- ***expr** – textual or `ClauseElement` construct which will be rendered following the INSERT, UPDATE, or DELETE keyword.
- ****kw** – A single keyword 'dialect' is accepted. This is an optional string dialect name which will limit rendering of this prefix to only that dialect.

returning(**cols*)

Add a RETURNING or equivalent clause to this statement.

e.g.:

```
stmt = table.update().\
          where(table.c.data == 'value').\
          values(status='X').\
          returning(table.c.server_flag,
                    table.c.updated_timestamp)

for server_flag, updated_timestamp in connection.execute(stmt):
    print(server_flag, updated_timestamp)
```

The given collection of column expressions should be derived from the table that is the target of the INSERT, UPDATE, or DELETE. While `Column` objects are typical, the elements can also be expressions:

```
stmt = table.insert().returning(
    (table.c.first_name + " " + table.c.last_name).
    label('fullname'))
```

Upon compilation, a RETURNING clause, or database equivalent, will be rendered within the statement. For INSERT and UPDATE, the values are the newly inserted/updated values. For DELETE, the values are those of the rows which were deleted.

Upon execution, the values of the columns to be returned are made available via the result set and can be iterated using `ResultProxy.fetchone()` and similar. For DBAPIs which do not natively support returning values (i.e. cx_oracle), SQLAlchemy will approximate this behavior at the result level so that a reasonable amount of behavioral neutrality is provided.

Note that not all databases/DBAPIs support RETURNING. For those backends with no support, an exception is raised upon compilation and/or execution. For those who do support it, the functionality across backends varies greatly, including restrictions on executemany() and other statements which return multiple rows. Please read the documentation notes for the database in use in order to determine the availability of RETURNING.

See also:

`ValuesBase.return_defaults()` - an alternative method tailored towards efficient fetching of server-side defaults and triggers for single-row INSERTs or UPDATEs.

scalar(**multiparams*, ***params*)

inherited from the `scalar()` *method of* `Executable`

Compile and execute this `Executable`, returning the result's scalar representation.

self_group(*against=None*)

inherited from the `self_group()` *method of* `ClauseElement`

Apply a 'grouping' to this `ClauseElement`.

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by `select()` constructs when placed into the FROM clause of another `select()`. (Note that subqueries should be normally created using the `Select.alias()` method, as many platforms require nested SELECT statements to be named).

As expressions are composed together, the application of `self_group()` is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like `x OR (y AND z)` - AND takes precedence over OR.

The base `self_group()` method of `ClauseElement` just returns self.

unique_params(**optionaldict*, ***kwargs*)

inherited from the `unique_params()` *method of* `ClauseElement`

Return a copy with `bindparam()` elements replaced.

Same functionality as `params()`, except adds *unique=True* to affected bind parameters so that multiple statements can be used.

with_hint(*text*, *selectable=None*, *dialect_name='*'*)

Add a table hint for a single table to this INSERT/UPDATE/DELETE statement.

Note: `UpdateBase.with_hint()` currently applies only to Microsoft SQL Server. For MySQL INSERT/UPDATE/DELETE hints, use `UpdateBase.prefix_with()`.

The text of the hint is rendered in the appropriate location for the database backend in use, relative to the `Table` that is the subject of this statement, or optionally to that of the given `Table` passed as the `selectable` argument.

The `dialect_name` option will limit the rendering of a particular hint to a particular backend. Such as, to add a hint that only takes effect for SQL Server:

```
mytable.insert().with_hint("WITH (PAGLOCK)", dialect_name="mssql")
```

New in version 0.7.6.

Parameters

- **text** – Text of the hint.
- **selectable** – optional `Table` that specifies an element of the FROM clause within an UPDATE or DELETE to be the subject of the hint - applies only to certain backends.
- **dialect_name** – defaults to `*`, if specified as the name of a particular dialect, will apply these hints only when that dialect is in use.

class sqlalchemy.sql.expression.**ValuesBase**(*table*, *values*, *prefixes*)

Bases: *sqlalchemy.sql.expression.UpdateBase*

Supplies support for *ValuesBase.values()* to INSERT and UPDATE constructs.

return_defaults(**cols*)

Make use of a RETURNING clause for the purpose of fetching server-side expressions and defaults.

E.g.:

```
stmt = table.insert().values(data='newdata').return_defaults()

result = connection.execute(stmt)

server_created_at = result.returned_defaults['created_at']
```

When used against a backend that supports RETURNING, all column values generated by SQL expression or server-side-default will be added to any existing RETURNING clause, provided that *UpdateBase.returning()* is not used simultaneously. The column values will then be available on the result using the *ResultProxy.returned_defaults* accessor as a dictionary, referring to values keyed to the *Column* object as well as its `.key`.

This method differs from *UpdateBase.returning()* in these ways:

1. *ValuesBase.return_defaults()* is only intended for use with an INSERT or an UPDATE statement that matches exactly one row. While the RETURNING construct in the general sense supports multiple rows for a multi-row UPDATE or DELETE statement, or for special cases of INSERT that return multiple rows (e.g. INSERT from SELECT, multi-valued VALUES clause), *ValuesBase.return_defaults()* is intended only for an "ORM-style" single-row INSERT/UPDATE statement. The row returned by the statement is also consumed implicitly when *ValuesBase.return_defaults()* is used. By contrast, *UpdateBase.returning()* leaves the RETURNING result-set intact with a collection of any number of rows.
2. It is compatible with the existing logic to fetch auto-generated primary key values, also known as "implicit returning". Backends that support RETURNING will automatically make use of RETURNING in order to fetch the value of newly generated primary keys; while the *UpdateBase.returning()* method circumvents this behavior, *ValuesBase.return_defaults()* leaves it intact.
3. It can be called against any backend. Backends that don't support RETURNING will skip the usage of the feature, rather than raising an exception. The return value of *ResultProxy.returned_defaults* will be `None`

ValuesBase.return_defaults() is used by the ORM to provide an efficient implementation for the `eager_defaults` feature of *mapper()*.

Parameters **cols** – optional list of column key names or *Column* objects. If omitted, all column expressions evaluated on the server are added to the returning list.

New in version 0.9.0.

See also:

UpdateBase.returning()

ResultProxy.returned_defaults

values(**args*, ***kwargs*)

specify a fixed VALUES clause for an INSERT statement, or the SET clause for an UPDATE.

Note that the *Insert* and *Update* constructs support per-execution time formatting of the VALUES and/or SET clauses, based on the arguments passed to *Connection.execute()*. However, the

`ValuesBase.values()` method can be used to "fix" a particular set of parameters into the statement.

Multiple calls to `ValuesBase.values()` will produce a new construct, each one with the parameter list modified to include the new parameters sent. In the typical case of a single dictionary of parameters, the newly passed keys will replace the same keys in the previous construct. In the case of a list-based "multiple values" construct, each new list of values is extended onto the existing list of values.

Parameters

- **`**kwargs`** – key value pairs representing the string key of a `Column` mapped to the value to be rendered into the VALUES or SET clause:

```
users.insert().values(name="some name")

users.update().where(users.c.id==5).values(name="some name")
```

- **`*args`** – As an alternative to passing key/value parameters, a dictionary, tuple, or list of dictionaries or tuples can be passed as a single positional argument in order to form the VALUES or SET clause of the statement. The forms that are accepted vary based on whether this is an `Insert` or an `Update` construct.

  For either an `Insert` or `Update` construct, a single dictionary can be passed, which works the same as that of the kwargs form:

```
users.insert().values({"name": "some name"})

users.update().values({"name": "some new name"})
```

  Also for either form but more typically for the `Insert` construct, a tuple that contains an entry for every column in the table is also accepted:

```
users.insert().values((5, "some name"))
```

  The `Insert` construct also supports being passed a list of dictionaries or full-table-tuples, which on the server will render the less common SQL syntax of "multiple values" - this syntax is supported on backends such as SQLite, Postgresql, MySQL, but not necessarily others:

```
users.insert().values([
                    {"name": "some name"},
                    {"name": "some other name"},
                    {"name": "yet another name"},
                ])
```

  The above form would render a multiple VALUES statement similar to:

```
INSERT INTO users (name) VALUES
                (:name_1),
                (:name_2),
                (:name_3)
```

  It is essential to note that **passing multiple values is NOT the same as using traditional executemany() form**. The above syntax is a **special** syntax not typically used. To emit an INSERT statement against multiple rows, the normal method is to pass a multiple values list to the `Connection.execute()` method, which is supported by all database backends and is generally more efficient for a very large number of parameters.

See also:

Executing Multiple Statements - an introduction to the traditional Core method of multiple parameter set invocation for INSERTs and other statements.

Changed in version 1.0.0: an INSERT that uses a multiple-VALUES clause, even a list of length one, implies that the `Insert.inline` flag is set to True, indicating that the statement will not attempt to fetch the "last inserted primary key" or other defaults. The statement deals with an arbitrary number of rows, so the `ResultProxy.inserted_primary_key` accessor does not apply.

Changed in version 1.0.0: A multiple-VALUES INSERT now supports columns with Python side default values and callables in the same way as that of an "executemany" style of invocation; the callable is invoked for each row. See *Python-side defaults invoked for each row invidually when using a multivalued insert* for other details.

The `Update` construct supports a special form which is a list of 2-tuples, which when provided must be passed in conjunction with the `preserve_parameter_order` parameter. This form causes the UPDATE statement to render the SET clauses using the order of parameters given to `Update.values()`, rather than the ordering of columns given in the `Table`.

New in version 1.0.10: - added support for parameter-ordered UPDATE statements via the `preserve_parameter_order` flag.

See also:

Parameter-Ordered Updates - full example of the `preserve_parameter_order` flag

See also:

Inserts, Updates and Deletes - SQL Expression Language Tutorial

`insert()` - produce an `INSERT` statement

`update()` - produce an `UPDATE` statement

3.2.4 SQL and Generic Functions

SQL functions which are known to SQLAlchemy with regards to database-specific rendering, return types and argument behavior. Generic functions are invoked like all SQL functions, using the `func` attribute:

```
select([func.count()]).select_from(sometable)
```

Note that any name not known to `func` generates the function name as is - there is no restriction on what SQL functions can be called, known or unknown to SQLAlchemy, built-in or user defined. The section here only describes those functions where SQLAlchemy already knows what argument and return types are in use. SQL function API, factories, and built-in functions.

class `sqlalchemy.sql.functions.`**`AnsiFunction`**(***kwargs*)

Bases: `sqlalchemy.sql.functions.GenericFunction`

`identifier` = **'AnsiFunction'**

`name` = **'AnsiFunction'**

class `sqlalchemy.sql.functions.`**`Function`**(*name*, **clauses*, ***kw*)

Bases: `sqlalchemy.sql.functions.FunctionElement`

Describe a named SQL function.

See the superclass `FunctionElement` for a description of public methods.

See also:

`func` - namespace which produces registered or ad-hoc `Function` instances.

`GenericFunction` - allows creation of registered function types.

__init__(*name*, **clauses*, ***kw*)

Construct a `Function`.

The `func` construct is normally used to construct new `Function` instances.

class sqlalchemy.sql.functions.**FunctionElement**(**clauses*, ***kwargs*)

Bases: `sqlalchemy.sql.expression.Executable`, `sqlalchemy.sql.expression.ColumnElement`, `sqlalchemy.sql.expression.FromClause`

Base for SQL function-oriented constructs.

See also:

`Function` - named SQL function.

`func` - namespace which produces registered or ad-hoc `Function` instances.

`GenericFunction` - allows creation of registered function types.

__init__(**clauses*, ***kwargs*)

Construct a `FunctionElement`.

alias(*name=None*, *flat=False*)

Produce a `Alias` construct against this `FunctionElement`.

This construct wraps the function in a named alias which is suitable for the FROM clause.

e.g.:

```
from sqlalchemy.sql import column

stmt = select([column('data')]).select_from(
    func.unnest(Table.data).alias('data_view')
)
```

Would produce:

```
SELECT data
FROM unnest(sometable.data) AS data_view
```

New in version 0.9.8: The `FunctionElement.alias()` method is now supported. Previously, this method's behavior was undefined and did not behave consistently across versions.

clauses

Return the underlying `ClauseList` which contains the arguments for this `FunctionElement`.

columns

The set of columns exported by this `FunctionElement`.

Function objects currently have no result column names built in; this method returns a single-element column collection with an anonymously named column.

An interim approach to providing named columns for a function as a FROM clause is to build a `select()` with the desired columns:

```
from sqlalchemy.sql import column

stmt = select([column('x'), column('y')]).                select_from(func.myfunction())
```

execute()

Execute this *FunctionElement* against an embedded 'bind'.

This first calls *select()* to produce a SELECT construct.

Note that *FunctionElement* can be passed to the *Connectable.execute()* method of *Connection* or *Engine*.

filter(**criterion*)

Produce a FILTER clause against this function.

Used against aggregate and window functions, for database backends that support the "FILTER" clause.

The expression:

```
func.count(1).filter(True)
```

is shorthand for:

```
from sqlalchemy import funcfilter
funcfilter(func.count(1), True)
```

New in version 1.0.0.

See also:

FunctionFilter

funcfilter()

get_children(***kwargs*)

over(*partition_by=None*, *order_by=None*)

Produce an OVER clause against this function.

Used against aggregate or so-called "window" functions, for database backends that support window functions.

The expression:

```
func.row_number().over(order_by='x')
```

is shorthand for:

```
from sqlalchemy import over
over(func.row_number(), order_by='x')
```

See *over()* for a full description.

New in version 0.7.

packagenames = ()

scalar()

Execute this *FunctionElement* against an embedded 'bind' and return a scalar value.

This first calls *select()* to produce a SELECT construct.

Note that *FunctionElement* can be passed to the *Connectable.scalar()* method of *Connection* or *Engine*.

select()

Produce a *select()* construct against this *FunctionElement*.

This is shorthand for:

```
s = select([function_element])
```

self_group(*against=None*)

within_group(**order_by*)

Produce a WITHIN GROUP (ORDER BY expr) clause against this function.

Used against so-called "ordered set aggregate" and "hypothetical set aggregate" functions, including *percentile_cont*, *rank*, *dense_rank*, etc.

See *within_group()* for a full description.

New in version 1.1.

within_group_type(*within_group*)

For types that define their return type as based on the criteria within a WITHIN GROUP (ORDER BY) expression, called by the *WithinGroup* construct.

Returns None by default, in which case the function's normal `.type` is used.

class sqlalchemy.sql.functions.**GenericFunction**(**args*, ***kwargs*)

Bases: *sqlalchemy.sql.functions.Function*

Define a 'generic' function.

A generic function is a pre-established *Function* class that is instantiated automatically when called by name from the *func* attribute. Note that calling any name from *func* has the effect that a new *Function* instance is created automatically, given that name. The primary use case for defining a *GenericFunction* class is so that a function of a particular name may be given a fixed return type. It can also include custom argument parsing schemes as well as additional methods.

Subclasses of *GenericFunction* are automatically registered under the name of the class. For example, a user-defined function `as_utc()` would be available immediately:

```
from sqlalchemy.sql.functions import GenericFunction
from sqlalchemy.types import DateTime

class as_utc(GenericFunction):
    type = DateTime

print select([func.as_utc()])
```

User-defined generic functions can be organized into packages by specifying the "package" attribute when defining *GenericFunction*. Third party libraries containing many functions may want to use this in order to avoid name conflicts with other systems. For example, if our `as_utc()` function were part of a package "time":

```
class as_utc(GenericFunction):
    type = DateTime
    package = "time"
```

The above function would be available from *func* using the package name `time`:

```
print select([func.time.as_utc()])
```

A final option is to allow the function to be accessed from one name in *func* but to render as a different name. The `identifier` attribute will override the name used to access the function as loaded from *func*, but will retain the usage of `name` as the rendered name:

```
class GeoBuffer(GenericFunction):
    type = Geometry
    package = "geo"
    name = "ST_Buffer"
    identifier = "buffer"
```

The above function will render as follows:

```
>>> print func.geo.buffer()
ST_Buffer()
```

New in version 0.8: *GenericFunction* now supports automatic registration of new functions as well as package and custom naming support.

Changed in version 0.8: The attribute name `type` is used to specify the function's return type at the class level. Previously, the name `__return_type__` was used. This name is still recognized for backwards-compatibility.

coerce_arguments = True

identifier = 'GenericFunction'

name = 'GenericFunction'

class sqlalchemy.sql.functions.**OrderedSetAgg**(**args*, ***kwargs*)
Bases: *sqlalchemy.sql.functions.GenericFunction*

Define a function where the return type is based on the sort expression type as defined by the expression passed to the *FunctionElement.within_group()* method.

array_for_multi_clause = False

identifier = 'OrderedSetAgg'

name = 'OrderedSetAgg'

within_group_type(*within_group*)

class sqlalchemy.sql.functions.**ReturnTypeFromArgs**(**args*, ***kwargs*)
Bases: *sqlalchemy.sql.functions.GenericFunction*

Define a function whose return type is the same as its arguments.

identifier = 'ReturnTypeFromArgs'

name = 'ReturnTypeFromArgs'

class sqlalchemy.sql.functions.**array_agg**(**args*, ***kwargs*)
Bases: *sqlalchemy.sql.functions.GenericFunction*

support for the ARRAY_AGG function.

The `func.array_agg(expr)` construct returns an expression of type *types.ARRAY*.

e.g.:

```
stmt = select([func.array_agg(table.c.values)[2:5]])
```

New in version 1.1.

See also:

postgresql.array_agg() - PostgreSQL-specific version that returns *postgresql.ARRAY*, which has PG-specific operators added.

identifier = 'array_agg'

name = 'array_agg'

type
alias of `ARRAY`

class `sqlalchemy.sql.functions.`**`char_length`**(*arg*, ***kwargs*)
Bases: *sqlalchemy.sql.functions.GenericFunction*

identifier = 'char_length'

name = 'char_length'

type
alias of `Integer`

class `sqlalchemy.sql.functions.`**`coalesce`**(**args*, ***kwargs*)
Bases: *sqlalchemy.sql.functions.ReturnTypeFromArgs*

identifier = 'coalesce'

name = 'coalesce'

class `sqlalchemy.sql.functions.`**`concat`**(**args*, ***kwargs*)
Bases: *sqlalchemy.sql.functions.GenericFunction*

identifier = 'concat'

name = 'concat'

type
alias of `String`

class `sqlalchemy.sql.functions.`**`count`**(*expression=None*, ***kwargs*)
Bases: *sqlalchemy.sql.functions.GenericFunction*

The ANSI COUNT aggregate function. With no arguments, emits COUNT *.

identifier = 'count'

name = 'count'

type
alias of `Integer`

class `sqlalchemy.sql.functions.`**`cume_dist`**(**args*, ***kwargs*)
Bases: *sqlalchemy.sql.functions.GenericFunction*

Implement the `cume_dist` hypothetical-set aggregate function.

This function must be used with the *FunctionElement.within_group()* modifier to supply a sort expression to operate upon.

The return type of this function is *Numeric*.

New in version 1.1.

identifier = 'cume_dist'

name = 'cume_dist'

type = Numeric()

class `sqlalchemy.sql.functions.`**`current_date`**(***kwargs*)
Bases: *sqlalchemy.sql.functions.AnsiFunction*

identifier = 'current_date'

name = 'current_date'

type
alias of `Date`

class `sqlalchemy.sql.functions.`**`current_time`**`(`***kwargs*`)`
Bases: *sqlalchemy.sql.functions.AnsiFunction*

identifier = 'current_time'

name = 'current_time'

type
alias of `Time`

class `sqlalchemy.sql.functions.`**`current_timestamp`**`(`***kwargs*`)`
Bases: *sqlalchemy.sql.functions.AnsiFunction*

identifier = 'current_timestamp'

name = 'current_timestamp'

type
alias of `DateTime`

class `sqlalchemy.sql.functions.`**`current_user`**`(`***kwargs*`)`
Bases: *sqlalchemy.sql.functions.AnsiFunction*

identifier = 'current_user'

name = 'current_user'

type
alias of `String`

class `sqlalchemy.sql.functions.`**`dense_rank`**`(`**args*, ***kwargs*`)`
Bases: *sqlalchemy.sql.functions.GenericFunction*

Implement the `dense_rank` hypothetical-set aggregate function.

This function must be used with the *FunctionElement.within_group()* modifier to supply a sort expression to operate upon.

The return type of this function is *Integer*.

New in version 1.1.

identifier = 'dense_rank'

name = 'dense_rank'

type = Integer()

class `sqlalchemy.sql.functions.`**`localtime`**`(`***kwargs*`)`
Bases: *sqlalchemy.sql.functions.AnsiFunction*

identifier = 'localtime'

name = 'localtime'

type
alias of `DateTime`

class `sqlalchemy.sql.functions.`**`localtimestamp`**`(`***kwargs*`)`
Bases: *sqlalchemy.sql.functions.AnsiFunction*

identifier = 'localtimestamp'

name = 'localtimestamp'

type
alias of `DateTime`

class `sqlalchemy.sql.functions.`**`max`**(**args*, ***kwargs*)
Bases: *sqlalchemy.sql.functions.ReturnTypeFromArgs*

identifier = 'max'

name = 'max'

class `sqlalchemy.sql.functions.`**`min`**(**args*, ***kwargs*)
Bases: *sqlalchemy.sql.functions.ReturnTypeFromArgs*

identifier = 'min'

name = 'min'

class `sqlalchemy.sql.functions.`**`mode`**(**args*, ***kwargs*)
Bases: *sqlalchemy.sql.functions.OrderedSetAgg*

implement the `mode` ordered-set aggregate function.

This function must be used with the *FunctionElement.within_group()* modifier to supply a sort expression to operate upon.

The return type of this function is the same as the sort expression.

New in version 1.1.

identifier = 'mode'

name = 'mode'

class `sqlalchemy.sql.functions.`**`next_value`**(*seq*, ***kw*)
Bases: *sqlalchemy.sql.functions.GenericFunction*

Represent the 'next value', given a *Sequence* as its single argument.

Compiles into the appropriate function on each backend, or will raise NotImplementedError if used on a backend that does not provide support for sequences.

identifier = 'next_value'

name = 'next_value'

type = Integer()

class `sqlalchemy.sql.functions.`**`now`**(**args*, ***kwargs*)
Bases: *sqlalchemy.sql.functions.GenericFunction*

identifier = 'now'

name = 'now'

type
alias of `DateTime`

class `sqlalchemy.sql.functions.`**`percent_rank`**(**args*, ***kwargs*)
Bases: *sqlalchemy.sql.functions.GenericFunction*

Implement the `percent_rank` hypothetical-set aggregate function.

This function must be used with the *FunctionElement.within_group()* modifier to supply a sort expression to operate upon.

The return type of this function is *Numeric*.

New in version 1.1.

identifier = 'percent_rank'

name = 'percent_rank'

type = Numeric()

class sqlalchemy.sql.functions.**percentile_cont**(**args*, ***kwargs*)

Bases: *sqlalchemy.sql.functions.OrderedSetAgg*

implement the `percentile_cont` ordered-set aggregate function.

This function must be used with the *FunctionElement.within_group()* modifier to supply a sort expression to operate upon.

The return type of this function is the same as the sort expression, or if the arguments are an array, an *types.ARRAY* of the sort expression's type.

New in version 1.1.

array_for_multi_clause = True

identifier = 'percentile_cont'

name = 'percentile_cont'

class sqlalchemy.sql.functions.**percentile_disc**(**args*, ***kwargs*)

Bases: *sqlalchemy.sql.functions.OrderedSetAgg*

implement the `percentile_disc` ordered-set aggregate function.

This function must be used with the *FunctionElement.within_group()* modifier to supply a sort expression to operate upon.

The return type of this function is the same as the sort expression, or if the arguments are an array, an *types.ARRAY* of the sort expression's type.

New in version 1.1.

array_for_multi_clause = True

identifier = 'percentile_disc'

name = 'percentile_disc'

class sqlalchemy.sql.functions.**random**(**args*, ***kwargs*)

Bases: *sqlalchemy.sql.functions.GenericFunction*

identifier = 'random'

name = 'random'

class sqlalchemy.sql.functions.**rank**(**args*, ***kwargs*)

Bases: *sqlalchemy.sql.functions.GenericFunction*

Implement the `rank` hypothetical-set aggregate function.

This function must be used with the *FunctionElement.within_group()* modifier to supply a sort expression to operate upon.

The return type of this function is *Integer*.

New in version 1.1.

identifier = 'rank'

name = 'rank'

type = Integer()

sqlalchemy.sql.functions.**register_function**(*identifier*, *fn*, *package='_default'*)

Associate a callable with a particular func. name.

This is normally called by _GenericMeta, but is also available by itself so that a non-Function construct can be associated with the *func* accessor (i.e. CAST, EXTRACT).

class sqlalchemy.sql.functions.**session_user**(***kwargs*)

Bases: *sqlalchemy.sql.functions.AnsiFunction*

identifier = 'session_user'

name = 'session_user'

type

alias of `String`

class sqlalchemy.sql.functions.**sum**(**args*, ***kwargs*)

Bases: *sqlalchemy.sql.functions.ReturnTypeFromArgs*

identifier = 'sum'

name = 'sum'

class sqlalchemy.sql.functions.**sysdate**(***kwargs*)

Bases: *sqlalchemy.sql.functions.AnsiFunction*

identifier = 'sysdate'

name = 'sysdate'

type

alias of `DateTime`

class sqlalchemy.sql.functions.**user**(***kwargs*)

Bases: *sqlalchemy.sql.functions.AnsiFunction*

identifier = 'user'

name = 'user'

type

alias of `String`

3.2.5 Custom SQL Constructs and Compilation Extension

Provides an API for creation of custom ClauseElements and compilers.

Synopsis

Usage involves the creation of one or more *ClauseElement* subclasses and one or more callables defining its compilation:

```
from sqlalchemy.ext.compiler import compiles
from sqlalchemy.sql.expression import ColumnClause

class MyColumn(ColumnClause):
    pass

@compiles(MyColumn)
def compile_mycolumn(element, compiler, **kw):
    return "[%s]" % element.name
```

Above, `MyColumn` extends *ColumnClause*, the base expression element for named column objects. The `compiles` decorator registers itself with the `MyColumn` class so that it is invoked when the object is compiled to a string:

```
from sqlalchemy import select

s = select([MyColumn('x'), MyColumn('y')])
print str(s)
```

Produces:

```
SELECT [x], [y]
```

Dialect-specific compilation rules

Compilers can also be made dialect-specific. The appropriate compiler will be invoked for the dialect in use:

```
from sqlalchemy.schema import DDLElement

class AlterColumn(DDLElement):

    def __init__(self, column, cmd):
        self.column = column
        self.cmd = cmd

@compiles(AlterColumn)
def visit_alter_column(element, compiler, **kw):
    return "ALTER COLUMN %s ..." % element.column.name

@compiles(AlterColumn, 'postgresql')
def visit_alter_column(element, compiler, **kw):
    return "ALTER TABLE %s ALTER COLUMN %s ..." % (element.table.name,
                                                   element.column.name)
```

The second `visit_alter_table` will be invoked when any `postgresql` dialect is used.

Compiling sub-elements of a custom expression construct

The `compiler` argument is the *Compiled* object in use. This object can be inspected for any information about the in-progress compilation, including `compiler.dialect`, `compiler.statement` etc. The *SQLCompiler* and *DDLCompiler* both include a `process()` method which can be used for compilation of embedded attributes:

```
from sqlalchemy.sql.expression import Executable, ClauseElement

class InsertFromSelect(Executable, ClauseElement):
    def __init__(self, table, select):
        self.table = table
        self.select = select

@compiles(InsertFromSelect)
def visit_insert_from_select(element, compiler, **kw):
    return "INSERT INTO %s (%s)" % (
        compiler.process(element.table, asfrom=True),
```

```
        compiler.process(element.select)
    )

insert = InsertFromSelect(t1, select([t1]).where(t1.c.x>5))
print insert
```

Produces:

```
"INSERT INTO mytable (SELECT mytable.x, mytable.y, mytable.z
                      FROM mytable WHERE mytable.x > :x_1)"
```

Note: The above `InsertFromSelect` construct is only an example, this actual functionality is already available using the *`Insert.from_select()`* method.

Note: The above `InsertFromSelect` construct probably wants to have "autocommit" enabled. See *Enabling Autocommit on a Construct* for this step.

Cross Compiling between SQL and DDL compilers

SQL and DDL constructs are each compiled using different base compilers - `SQLCompiler` and `DDLCompiler`. A common need is to access the compilation rules of SQL expressions from within a DDL expression. The `DDLCompiler` includes an accessor `sql_compiler` for this reason, such as below where we generate a CHECK constraint that embeds a SQL expression:

```
@compiles(MyConstraint)
def compile_my_constraint(constraint, ddlcompiler, **kw):
    return "CONSTRAINT %s CHECK (%s)" % (
        constraint.name,
        ddlcompiler.sql_compiler.process(constraint.expression)
    )
```

Enabling Autocommit on a Construct

Recall from the section *Understanding Autocommit* that the *`Engine`*, when asked to execute a construct in the absence of a user-defined transaction, detects if the given construct represents DML or DDL, that is, a data modification or data definition statement, which requires (or may require, in the case of DDL) that the transaction generated by the DBAPI be committed (recall that DBAPI always has a transaction going on regardless of what SQLAlchemy does). Checking for this is actually accomplished by checking for the "autocommit" execution option on the construct. When building a construct like an INSERT derivation, a new DDL type, or perhaps a stored procedure that alters data, the "autocommit" option needs to be set in order for the statement to function with "connectionless" execution (as described in *Connectionless Execution, Implicit Execution*).

Currently a quick way to do this is to subclass *`Executable`*, then add the "autocommit" flag to the `_execution_options` dictionary (note this is a "frozen" dictionary which supplies a generative `union()` method):

```
from sqlalchemy.sql.expression import Executable, ClauseElement

class MyInsertThing(Executable, ClauseElement):
    _execution_options = \
        Executable._execution_options.union({'autocommit': True})
```

More succinctly, if the construct is truly similar to an INSERT, UPDATE, or DELETE, *UpdateBase* can be used, which already is a subclass of *Executable*, *ClauseElement* and includes the `autocommit` flag:

```
from sqlalchemy.sql.expression import UpdateBase

class MyInsertThing(UpdateBase):
    def __init__(self, ...):
        ...
```

DDL elements that subclass *DDLElement* already have the "autocommit" flag turned on.

Changing the default compilation of existing constructs

The compiler extension applies just as well to the existing constructs. When overriding the compilation of a built in SQL construct, the @compiles decorator is invoked upon the appropriate class (be sure to use the class, i.e. `Insert` or `Select`, instead of the creation function such as `insert()` or `select()`).

Within the new compilation function, to get at the "original" compilation routine, use the appropriate visit_XXX method - this because compiler.process() will call upon the overriding routine and cause an endless loop. Such as, to add "prefix" to all insert statements:

```
from sqlalchemy.sql.expression import Insert

@compiles(Insert)
def prefix_inserts(insert, compiler, **kw):
    return compiler.visit_insert(insert.prefix_with("some prefix"), **kw)
```

The above compiler will prefix all INSERT statements with "some prefix" when compiled.

Changing Compilation of Types

`compiler` works for types, too, such as below where we implement the MS-SQL specific 'max' keyword for `String`/VARCHAR:

```
@compiles(String, 'mssql')
@compiles(VARCHAR, 'mssql')
def compile_varchar(element, compiler, **kw):
    if element.length == 'max':
        return "VARCHAR('max')"
    else:
        return compiler.visit_VARCHAR(element, **kw)

foo = Table('foo', metadata,
    Column('data', VARCHAR('max'))
)
```

Subclassing Guidelines

A big part of using the compiler extension is subclassing SQLAlchemy expression constructs. To make this easier, the expression and schema packages feature a set of "bases" intended for common tasks. A synopsis is as follows:

- *ClauseElement* - This is the root expression class. Any SQL expression can be derived from this base, and is probably the best choice for longer constructs such as specialized INSERT statements.
- *ColumnElement* - The root of all "column-like" elements. Anything that you'd place in the "columns" clause of a SELECT statement (as well as order by and group by) can derive from this - the object will automatically have Python "comparison" behavior.

  ColumnElement classes want to have a `type` member which is expression's return type. This can be established at the instance level in the constructor, or at the class level if its generally constant:

```
class timestamp(ColumnElement):
    type = TIMESTAMP()
```

- *FunctionElement* - This is a hybrid of a `ColumnElement` and a "from clause" like object, and represents a SQL function or stored procedure type of call. Since most databases support statements along the line of "SELECT FROM <some function>" `FunctionElement` adds in the ability to be used in the FROM clause of a `select()` construct:

```
from sqlalchemy.sql.expression import FunctionElement

class coalesce(FunctionElement):
    name = 'coalesce'

@compiles(coalesce)
def compile(element, compiler, **kw):
    return "coalesce(%s)" % compiler.process(element.clauses)

@compiles(coalesce, 'oracle')
def compile(element, compiler, **kw):
    if len(element.clauses) > 2:
        raise TypeError("coalesce only supports two arguments on Oracle")
    return "nvl(%s)" % compiler.process(element.clauses)
```

- *DDLElement* - The root of all DDL expressions, like CREATE TABLE, ALTER TABLE, etc. Compilation of `DDLElement` subclasses is issued by a `DDLCompiler` instead of a `SQLCompiler`. `DDLElement` also features `Table` and `MetaData` event hooks via the `execute_at()` method, allowing the construct to be invoked during CREATE TABLE and DROP TABLE sequences.
- *Executable* - This is a mixin which should be used with any expression class that represents a "standalone" SQL statement that can be passed directly to an `execute()` method. It is already implicit within `DDLElement` and `FunctionElement`.

Further Examples

"UTC timestamp" function

A function that works like "CURRENT_TIMESTAMP" except applies the appropriate conversions so that the time is in UTC time. Timestamps are best stored in relational databases as UTC, without time zones. UTC so that your database doesn't think time has gone backwards in the hour when daylight savings ends, without timezones because timezones are like character encodings - they're best applied only at the endpoints of an application (i.e. convert to UTC upon user input, re-apply desired timezone upon display).

For Postgresql and Microsoft SQL Server:

```
from sqlalchemy.sql import expression
from sqlalchemy.ext.compiler import compiles
from sqlalchemy.types import DateTime
```

```
class utcnow(expression.FunctionElement):
    type = DateTime()

@compiles(utcnow, 'postgresql')
def pg_utcnow(element, compiler, **kw):
    return "TIMEZONE('utc', CURRENT_TIMESTAMP)"

@compiles(utcnow, 'mssql')
def ms_utcnow(element, compiler, **kw):
    return "GETUTCDATE()"
```

Example usage:

```
from sqlalchemy import (
            Table, Column, Integer, String, DateTime, MetaData
        )
metadata = MetaData()
event = Table("event", metadata,
    Column("id", Integer, primary_key=True),
    Column("description", String(50), nullable=False),
    Column("timestamp", DateTime, server_default=utcnow())
)
```

"GREATEST" function

The "GREATEST" function is given any number of arguments and returns the one that is of the highest value - its equivalent to Python's `max` function. A SQL standard version versus a CASE based version which only accommodates two arguments:

```
from sqlalchemy.sql import expression
from sqlalchemy.ext.compiler import compiles
from sqlalchemy.types import Numeric

class greatest(expression.FunctionElement):
    type = Numeric()
    name = 'greatest'

@compiles(greatest)
def default_greatest(element, compiler, **kw):
    return compiler.visit_function(element)

@compiles(greatest, 'sqlite')
@compiles(greatest, 'mssql')
@compiles(greatest, 'oracle')
def case_greatest(element, compiler, **kw):
    arg1, arg2 = list(element.clauses)
    return "CASE WHEN %s > %s THEN %s ELSE %s END" % (
        compiler.process(arg1),
        compiler.process(arg2),
        compiler.process(arg1),
        compiler.process(arg2),
    )
```

Example usage:

```
Session.query(Account).\
        filter(
            greatest(
                Account.checking_balance,
                Account.savings_balance) > 10000
        )
```

"false" expression

Render a "false" constant expression, rendering as "0" on platforms that don't have a "false" constant:

```
from sqlalchemy.sql import expression
from sqlalchemy.ext.compiler import compiles

class sql_false(expression.ColumnElement):
    pass

@compiles(sql_false)
def default_false(element, compiler, **kw):
    return "false"

@compiles(sql_false, 'mssql')
@compiles(sql_false, 'mysql')
@compiles(sql_false, 'oracle')
def int_false(element, compiler, **kw):
    return "0"
```

Example usage:

```
from sqlalchemy import select, union_all

exp = union_all(
    select([users.c.name, sql_false().label("enrolled")]),
    select([customers.c.name, customers.c.enrolled])
)
```

`sqlalchemy.ext.compiler.`**`compiles`**(*class_*, **specs*)

Register a function as a compiler for a given `ClauseElement` type.

`sqlalchemy.ext.compiler.`**`deregister`**(*class_*)

Remove all custom compilers associated with a given `ClauseElement` type.

3.2.6 Expression Serializer Extension

Serializer/Deserializer objects for usage with SQLAlchemy query structures, allowing "contextual" deserialization.

Any SQLAlchemy query structure, either based on sqlalchemy.sql.* or sqlalchemy.orm.* can be used. The mappers, Tables, Columns, Session etc. which are referenced by the structure are not persisted in serialized form, but are instead re-associated with the query structure when it is deserialized.

Usage is nearly the same as that of the standard Python pickle module:

```
from sqlalchemy.ext.serializer import loads, dumps
metadata = MetaData(bind=some_engine)
Session = scoped_session(sessionmaker())

# ... define mappers

query = Session.query(MyClass).
    filter(MyClass.somedata=='foo').order_by(MyClass.sortkey)

# pickle the query
serialized = dumps(query)

# unpickle.  Pass in metadata + scoped_session
query2 = loads(serialized, metadata, Session)

print query2.all()
```

Similar restrictions as when using raw pickle apply; mapped classes must be themselves be pickleable, meaning they are importable from a module-level namespace.

The serializer module is only appropriate for query structures. It is not needed for:

- instances of user-defined classes. These contain no references to engines, sessions or expression constructs in the typical case and can be serialized directly.
- Table metadata that is to be loaded entirely from the serialized structure (i.e. is not already declared in the application). Regular pickle.loads()/dumps() can be used to fully dump any `MetaData` object, typically one which was reflected from an existing database at some previous point in time. The serializer module is specifically for the opposite case, where the Table metadata is already present in memory.

`sqlalchemy.ext.serializer.`**`Serializer`**(**args*, ***kw*)

`sqlalchemy.ext.serializer.`**`Deserializer`**(*file*, *metadata=None*, *scoped_session=None*, *engine=None*)

`sqlalchemy.ext.serializer.`**`dumps`**(*obj*, *protocol=0*)

`sqlalchemy.ext.serializer.`**`loads`**(*data*, *metadata=None*, *scoped_session=None*, *engine=None*)

3.3 Schema Definition Language

This section references SQLAlchemy **schema metadata**, a comprehensive system of describing and inspecting database schemas.

The core of SQLAlchemy's query and object mapping operations are supported by *database metadata*, which is comprised of Python objects that describe tables and other schema-level objects. These objects are at the core of three major types of operations - issuing CREATE and DROP statements (known as *DDL*), constructing SQL queries, and expressing information about structures that already exist within the database.

Database metadata can be expressed by explicitly naming the various components and their properties, using constructs such as `Table`, `Column`, `ForeignKey` and `Sequence`, all of which are imported from the `sqlalchemy.schema` package. It can also be generated by SQLAlchemy using a process called *reflection*, which means you start with a single object such as `Table`, assign it a name, and then instruct SQLAlchemy to load all the additional information related to that name from a particular engine source.

A key feature of SQLAlchemy's database metadata constructs is that they are designed to be used in a *declarative* style which closely resembles that of real DDL. They are therefore most intuitive to those who have some background in creating real schema generation scripts.

3.3.1 Describing Databases with MetaData

This section discusses the fundamental *Table*, *Column* and *MetaData* objects.

A collection of metadata entities is stored in an object aptly named *MetaData*:

```
from sqlalchemy import *

metadata = MetaData()
```

MetaData is a container object that keeps together many different features of a database (or multiple databases) being described.

To represent a table, use the *Table* class. Its two primary arguments are the table name, then the *MetaData* object which it will be associated with. The remaining positional arguments are mostly *Column* objects describing each column:

```
user = Table('user', metadata,
    Column('user_id', Integer, primary_key=True),
    Column('user_name', String(16), nullable=False),
    Column('email_address', String(60)),
    Column('password', String(20), nullable=False)
)
```

Above, a table called `user` is described, which contains four columns. The primary key of the table consists of the `user_id` column. Multiple columns may be assigned the `primary_key=True` flag which denotes a multi-column primary key, known as a *composite* primary key.

Note also that each column describes its datatype using objects corresponding to genericized types, such as *Integer* and *String*. SQLAlchemy features dozens of types of varying levels of specificity as well as the ability to create custom types. Documentation on the type system can be found at *Column and Data Types*.

Accessing Tables and Columns

The *MetaData* object contains all of the schema constructs we've associated with it. It supports a few methods of accessing these table objects, such as the `sorted_tables` accessor which returns a list of each *Table* object in order of foreign key dependency (that is, each table is preceded by all tables which it references):

```
>>> for t in metadata.sorted_tables:
...     print t.name
user
user_preference
invoice
invoice_item
```

In most cases, individual *Table* objects have been explicitly declared, and these objects are typically accessed directly as module-level variables in an application. Once a *Table* has been defined, it has a full set of accessors which allow inspection of its properties. Given the following *Table* definition:

```
employees = Table('employees', metadata,
    Column('employee_id', Integer, primary_key=True),
    Column('employee_name', String(60), nullable=False),
    Column('employee_dept', Integer, ForeignKey("departments.department_id"))
)
```

Note the `ForeignKey` object used in this table - this construct defines a reference to a remote table, and is fully described in *Defining Foreign Keys*. Methods of accessing information about this table include:

```
# access the column "EMPLOYEE_ID":
employees.columns.employee_id

# or just
employees.c.employee_id

# via string
employees.c['employee_id']

# iterate through all columns
for c in employees.c:
    print c

# get the table's primary key columns
for primary_key in employees.primary_key:
    print primary_key

# get the table's foreign key objects:
for fkey in employees.foreign_keys:
    print fkey

# access the table's MetaData:
employees.metadata

# access the table's bound Engine or Connection, if its MetaData is bound:
employees.bind

# access a column's name, type, nullable, primary key, foreign key
employees.c.employee_id.name
employees.c.employee_id.type
employees.c.employee_id.nullable
employees.c.employee_id.primary_key
employees.c.employee_dept.foreign_keys

# get the "key" of a column, which defaults to its name, but can
# be any user-defined string:
employees.c.employee_name.key

# access a column's table:
employees.c.employee_id.table is employees

# get the table related by a foreign key
list(employees.c.employee_dept.foreign_keys)[0].column.table
```

Creating and Dropping Database Tables

Once you've defined some `Table` objects, assuming you're working with a brand new database one thing you might want to do is issue CREATE statements for those tables and their related constructs (as an aside, it's also quite possible that you *don't* want to do this, if you already have some preferred methodology such as tools included with your database or an existing scripting system - if that's the case, feel free to skip this section - SQLAlchemy has no requirement that it be used to create your tables).

The usual way to issue CREATE is to use `create_all()` on the `MetaData` object. This method will issue queries that first check for the existence of each individual table, and if not found will issue the CREATE statements:

```
engine = create_engine('sqlite:///:memory:')

metadata = MetaData()

user = Table('user', metadata,
    Column('user_id', Integer, primary_key=True),
    Column('user_name', String(16), nullable=False),
    Column('email_address', String(60), key='email'),
    Column('password', String(20), nullable=False)
)

user_prefs = Table('user_prefs', metadata,
    Column('pref_id', Integer, primary_key=True),
    Column('user_id', Integer, ForeignKey("user.user_id"), nullable=False),
    Column('pref_name', String(40), nullable=False),
    Column('pref_value', String(100))
)

metadata.create_all(engine)
PRAGMA table_info(user){}
CREATE TABLE user(
        user_id INTEGER NOT NULL PRIMARY KEY,
        user_name VARCHAR(16) NOT NULL,
        email_address VARCHAR(60),
        password VARCHAR(20) NOT NULL
)
PRAGMA table_info(user_prefs){}
CREATE TABLE user_prefs(
        pref_id INTEGER NOT NULL PRIMARY KEY,
        user_id INTEGER NOT NULL REFERENCES user(user_id),
        pref_name VARCHAR(40) NOT NULL,
        pref_value VARCHAR(100)
)
```

create_all() creates foreign key constraints between tables usually inline with the table definition itself, and for this reason it also generates the tables in order of their dependency. There are options to change this behavior such that `ALTER TABLE` is used instead.

Dropping all tables is similarly achieved using the *drop_all()* method. This method does the exact opposite of *create_all()* - the presence of each table is checked first, and tables are dropped in reverse order of dependency.

Creating and dropping individual tables can be done via the `create()` and `drop()` methods of *Table*. These methods by default issue the CREATE or DROP regardless of the table being present:

```
engine = create_engine('sqlite:///:memory:')

meta = MetaData()

employees = Table('employees', meta,
    Column('employee_id', Integer, primary_key=True),
    Column('employee_name', String(60), nullable=False, key='name'),
    Column('employee_dept', Integer, ForeignKey("departments.department_id"))
)
employees.create(engine)
CREATE TABLE employees(
employee_id SERIAL NOT NULL PRIMARY KEY,
employee_name VARCHAR(60) NOT NULL,
employee_dept INTEGER REFERENCES departments(department_id)
```

```
)
{}
```

`drop()` method:

```
employees.drop(engine)
DROP TABLE employees
{}
```

To enable the "check first for the table existing" logic, add the `checkfirst=True` argument to `create()` or `drop()`:

```
employees.create(engine, checkfirst=True)
employees.drop(engine, checkfirst=False)
```

Altering Schemas through Migrations

While SQLAlchemy directly supports emitting CREATE and DROP statements for schema constructs, the ability to alter those constructs, usually via the ALTER statement as well as other database-specific constructs, is outside of the scope of SQLAlchemy itself. While it's easy enough to emit ALTER statements and similar by hand, such as by passing a string to *`Connection.execute()`* or by using the *`DDL`* construct, it's a common practice to automate the maintenance of database schemas in relation to application code using schema migration tools.

There are two major migration tools available for SQLAlchemy:

- Alembic - Written by the author of SQLAlchemy, Alembic features a highly customizable environment and a minimalistic usage pattern, supporting such features as transactional DDL, automatic generation of "candidate" migrations, an "offline" mode which generates SQL scripts, and support for branch resolution.
- SQLAlchemy-Migrate - The original migration tool for SQLAlchemy, SQLAlchemy-Migrate is widely used and continues under active development. SQLAlchemy-Migrate includes features such as SQL script generation, ORM class generation, ORM model comparison, and extensive support for SQLite migrations.

Specifying the Schema Name

Some databases support the concept of multiple schemas. A *`Table`* can reference this by specifying the `schema` keyword argument:

```
financial_info = Table('financial_info', meta,
    Column('id', Integer, primary_key=True),
    Column('value', String(100), nullable=False),
    schema='remote_banks'
)
```

Within the *`MetaData`* collection, this table will be identified by the combination of `financial_info` and `remote_banks`. If another table called `financial_info` is referenced without the `remote_banks` schema, it will refer to a different *`Table`*. *`ForeignKey`* objects can specify references to columns in this table using the form `remote_banks.financial_info.id`.

The `schema` argument should be used for any name qualifiers required, including Oracle's "owner" attribute and similar. It also can accommodate a dotted name for longer schemes:

```
schema="dbo.scott"
```

Backend-Specific Options

Table supports database-specific options. For example, MySQL has different table backend types, including "MyISAM" and "InnoDB". This can be expressed with *Table* using `mysql_engine`:

```
addresses = Table('engine_email_addresses', meta,
    Column('address_id', Integer, primary_key=True),
    Column('remote_user_id', Integer, ForeignKey(users.c.user_id)),
    Column('email_address', String(20)),
    mysql_engine='InnoDB'
)
```

Other backends may support table-level options as well - these would be described in the individual documentation sections for each dialect.

Column, Table, MetaData API

class `sqlalchemy.schema.`**`Column`**(**args*, ***kwargs*)

Bases: *sqlalchemy.schema.SchemaItem*, *sqlalchemy.sql.expression.ColumnClause*

Represents a column in a database table.

`__eq__`(*other*)

inherited from the *__eq__()* *method of* *ColumnOperators*

Implement the == operator.

In a column context, produces the clause `a = b`. If the target is `None`, produces `a IS NULL`.

`__init__`(**args*, ***kwargs*)

Construct a new `Column` object.

Parameters

- **`name`** – The name of this column as represented in the database. This argument may be the first positional argument, or specified via keyword.

  Names which contain no upper case characters will be treated as case insensitive names, and will not be quoted unless they are a reserved word. Names with any number of upper case characters will be quoted and sent exactly. Note that this behavior applies even for databases which standardize upper case names as case insensitive such as Oracle.

  The name field may be omitted at construction time and applied later, at any time before the Column is associated with a *Table*. This is to support convenient usage within the *declarative* extension.

- **`type_`** – The column's type, indicated using an instance which subclasses *TypeEngine*. If no arguments are required for the type, the class of the type can be sent as well, e.g.:

```
# use a type with arguments
Column('data', String(50))

# use no arguments
Column('level', Integer)
```

The `type` argument may be the second positional argument or specified by keyword.

If the `type` is `None` or is omitted, it will first default to the special type *NullType*. If and when this *Column* is made to refer to another column using *ForeignKey* and/or *ForeignKeyConstraint*, the type of the remote-referenced column will be copied to this column as well, at the moment that the foreign key is resolved against that remote *Column* object.

Changed in version 0.9.0: Support for propagation of type to a *Column* from its *ForeignKey* object has been improved and should be more reliable and timely.

- ***args** – Additional positional arguments include various *SchemaItem* derived constructs which will be applied as options to the column. These include instances of *Constraint*, *ForeignKey*, *ColumnDefault*, and *Sequence*. In some cases an equivalent keyword argument is available such as `server_default`, `default` and `unique`.
- **autoincrement** – Set up "auto increment" semantics for an integer primary key column. The default value is the string `"auto"` which indicates that a single-column primary key that is of an INTEGER type with no stated client-side or python-side defaults should receive auto increment semantics automatically; all other varieties of primary key columns will not. This includes that DDL such as Postgresql SERIAL or MySQL AUTO_INCREMENT will be emitted for this column during a table create, as well as that the column is assumed to generate new integer primary key values when an INSERT statement invokes which will be retrieved by the dialect.

  The flag may be set to `True` to indicate that a column which is part of a composite (e.g. multi-column) primary key should have autoincrement semantics, though note that only one column within a primary key may have this setting. It can also be set to `True` to indicate autoincrement semantics on a column that has a client-side or server-side default configured, however note that not all dialects can accommodate all styles of default as an "autoincrement". It can also be set to `False` on a single-column primary key that has a datatype of INTEGER in order to disable auto increment semantics for that column.

  Changed in version 1.1: The autoincrement flag now defaults to `"auto"` which indicates autoincrement semantics by default for single-column integer primary keys only; for composite (multi-column) primary keys, autoincrement is never implicitly enabled; as always, `autoincrement=True` will allow for at most one of those columns to be an "autoincrement" column. `autoincrement=True` may also be set on a *Column* that has an explicit client-side or server-side default, subject to limitations of the backend database and dialect.

  The setting *only* has an effect for columns which are:

  - Integer derived (i.e. INT, SMALLINT, BIGINT).
  - Part of the primary key
  - Not refering to another column via *ForeignKey*, unless the value is specified as `'ignore_fk'`:

```
# turn on autoincrement for this column despite
# the ForeignKey()
Column('id', ForeignKey('other.id'),
            primary_key=True, autoincrement='ignore_fk')
```

  It is typically not desirable to have "autoincrement" enabled on a column that refers to another via foreign key, as such a column is required to refer to a value that

originates from elsewhere.

The setting has these two effects on columns that meet the above criteria:

– DDL issued for the column will include database-specific keywords intended to signify this column as an "autoincrement" column, such as AUTO INCREMENT on MySQL, SERIAL on Postgresql, and IDENTITY on MS-SQL. It does *not* issue AUTOINCREMENT for SQLite since this is a special SQLite flag that is not required for autoincrementing behavior.

See also:

SQLite Auto Incrementing Behavior

– The column will be considered to be available using an "autoincrement" method specific to the backend database, such as calling upon `cursor.lastrowid`, using RETURNING in an INSERT statement to get at a sequence-generated value, or using special functions such as "SELECT scope_identity()". These methods are highly specific to the DBAPIs and databases in use and vary greatly, so care should be taken when associating `autoincrement=True` with a custom default generation function.

- **default** – A scalar, Python callable, or *ColumnElement* expression representing the *default value* for this column, which will be invoked upon insert if this column is otherwise not specified in the VALUES clause of the insert. This is a shortcut to using *ColumnDefault* as a positional argument; see that class for full detail on the structure of the argument.

  Contrast this argument to *Column.server_default* which creates a default generator on the database side.

  See also:

  Column Insert/Update Defaults

- **doc** – optional String that can be used by the ORM or similar to document attributes. This attribute does not render SQL comments (a future attribute 'comment' will achieve that).

- **key** – An optional string identifier which will identify this `Column` object on the *Table*. When a key is provided, this is the only identifier referencing the `Column` within the application, including ORM attribute mapping; the `name` field is used only when rendering SQL.

- **index** – When `True`, indicates that the column is indexed. This is a shortcut for using a *Index* construct on the table. To specify indexes with explicit names or indexes that contain multiple columns, use the *Index* construct instead.

- **info** – Optional data dictionary which will be populated into the *SchemaItem.info* attribute of this object.

- **nullable** – If set to the default of `True`, indicates the column will be rendered as allowing NULL, else it's rendered as NOT NULL. This parameter is only used when issuing CREATE TABLE statements.

- **onupdate** – A scalar, Python callable, or *ClauseElement* representing a default value to be applied to the column within UPDATE statements, which wil be invoked upon update if this column is not present in the SET clause of the update. This is a shortcut to using *ColumnDefault* as a positional argument with `for_update=True`.

- **primary_key** – If `True`, marks this column as a primary key column. Multiple columns can have this flag set to specify composite primary keys. As an alternative, the primary key of a *Table* can be specified via an explicit *PrimaryKeyConstraint* object.
- **server_default** – A *FetchedValue* instance, str, Unicode or *text()* construct representing the DDL DEFAULT value for the column.

  String types will be emitted as-is, surrounded by single quotes:

```
Column('x', Text, server_default="val")

x TEXT DEFAULT 'val'
```

  A *text()* expression will be rendered as-is, without quotes:

```
Column('y', DateTime, server_default=text('NOW()'))

y DATETIME DEFAULT NOW()
```

  Strings and text() will be converted into a *DefaultClause* object upon initialization.

  Use *FetchedValue* to indicate that an already-existing column will generate a default value on the database side which will be available to SQLAlchemy for post-fetch after inserts. This construct does not specify any DDL and the implementation is left to the database, such as via a trigger.

  See also:

  Server Side Defaults

- **server_onupdate** – A *FetchedValue* instance representing a database-side default generation function. This indicates to SQLAlchemy that a newly generated value will be available after updates. This construct does not specify any DDL and the implementation is left to the database, such as via a trigger.
- **quote** – Force quoting of this column's name on or off, corresponding to `True` or `False`. When left at its default of `None`, the column identifier will be quoted according to whether the name is case sensitive (identifiers with at least one upper case character are treated as case sensitive), or if it's a reserved word. This flag is only needed to force quoting of a reserved word which is not known by the SQLAlchemy dialect.
- **unique** – When `True`, indicates that this column contains a unique constraint, or if `index` is `True` as well, indicates that the *Index* should be created with the unique flag. To specify multiple columns in the constraint/index or to specify an explicit name, use the *UniqueConstraint* or *Index* constructs explicitly.
- **system** – When `True`, indicates this is a "system" column, that is a column which is automatically made available by the database, and should not be included in the columns list for a `CREATE TABLE` statement.

  For more elaborate scenarios where columns should be conditionally rendered differently on different backends, consider custom compilation rules for *CreateColumn*.

  New in version 0.8.3: Added the `system=True` parameter to *Column*.

__le__(*other*)

inherited from the __le__() method of ColumnOperators

Implement the <= operator.

In a column context, produces the clause `a <= b`.

__lt__(*other*)
inherited from the `__lt__()` *method of* `ColumnOperators`

Implement the < operator.

In a column context, produces the clause `a < b`.

__ne__(*other*)
inherited from the `__ne__()` *method of* `ColumnOperators`

Implement the != operator.

In a column context, produces the clause `a != b`. If the target is `None`, produces `a IS NOT NULL`.

all_()
inherited from the `all_()` *method of* `ColumnOperators`

Produce a `all_()` clause against the parent object.

New in version 1.1.

anon_label
inherited from the `anon_label` *attribute of* `ColumnElement`

provides a constant 'anonymous label' for this ColumnElement.

This is a label() expression which will be named at compile time. The same label() is returned each time anon_label is called so that expressions can reference anon_label multiple times, producing the same label name at compile time.

the compiler uses this function automatically at compile time for expressions that are known to be 'unnamed' like binary expressions and function calls.

any_()
inherited from the `any_()` *method of* `ColumnOperators`

Produce a `any_()` clause against the parent object.

New in version 1.1.

asc()
inherited from the `asc()` *method of* `ColumnOperators`

Produce a `asc()` clause against the parent object.

between(*cleft*, *cright*, *symmetric=False*)
inherited from the `between()` *method of* `ColumnOperators`

Produce a `between()` clause against the parent object, given the lower and upper range.

cast(*type_*)
inherited from the `cast()` *method of* `ColumnElement`

Produce a type cast, i.e. `CAST(<expression> AS <type>)`.

This is a shortcut to the `cast()` function.

New in version 1.0.7.

collate(*collation*)
inherited from the `collate()` *method of* `ColumnOperators`

Produce a `collate()` clause against the parent object, given the collation string.

compare(*other*, *use_proxies=False*, *equivalents=None*, ***kw*)
inherited from the `compare()` *method of* `ColumnElement`

Compare this ColumnElement to another.

Special arguments understood:

Parameters

- **use_proxies** – when True, consider two columns that share a common base column as equivalent (i.e. shares_lineage())
- **equivalents** – a dictionary of columns as keys mapped to sets of columns. If the given "other" column is present in this dictionary, if any of the columns in the corresponding set() pass the comparison test, the result is True. This is used to expand the comparison to other columns that may be known to be equivalent to this one via foreign key or other criterion.

compile(*bind=None*, *dialect=None*, ***kw*)
inherited from the `compile()` *method of* `ClauseElement`

Compile this SQL expression.

The return value is a `Compiled` object. Calling `str()` or `unicode()` on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the `params` accessor.

Parameters

- **bind** – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.
- **column_keys** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- **dialect** – A `Dialect` instance from which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- **inline** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

How do I render SQL expressions as strings, possibly with bound parameters inlined?

concat(*other*)
inherited from the `concat()` *method of* `ColumnOperators`

Implement the 'concat' operator.

In a column context, produces the clause `a || b`, or uses the `concat()` operator on MySQL.

contains(*other*, ***kwargs*)
inherited from the `contains()` *method of* `ColumnOperators`

Implement the 'contains' operator.

In a column context, produces the clause `LIKE '%<other>%'`

copy(***kw*)
Create a copy of this `Column`, unitialized.

This is used in `Table.tometadata`.

desc()
inherited from the `desc()` *method of* `ColumnOperators`

Produce a `desc()` clause against the parent object.

distinct()
inherited from the `distinct()` *method of* `ColumnOperators`

Produce a `distinct()` clause against the parent object.

endswith(*other*, ***kwargs*)
inherited from the `endswith()` *method of* `ColumnOperators`

Implement the 'endswith' operator.

In a column context, produces the clause `LIKE '%<other>'`

expression
inherited from the `expression` *attribute of* `ColumnElement`

Return a column expression.

Part of the inspection interface; returns self.

ilike(*other*, *escape=None*)
inherited from the `ilike()` *method of* `ColumnOperators`

Implement the `ilike` operator.

In a column context, produces the clause `a ILIKE other`.

E.g.:

```
select([sometable]).where(sometable.c.column.ilike("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the `ESCAPE` keyword, e.g.:

```
somecolumn.ilike("foo/%bar", escape="/")
```

See also:

ColumnOperators.like()

in_(*other*)

inherited from the in_() method of ColumnOperators

Implement the `in` operator.

In a column context, produces the clause `a IN other`. "other" may be a tuple/list of column expressions, or a *select()* construct.

info

inherited from the info attribute of SchemaItem

Info dictionary associated with the object, allowing user-defined data to be associated with this *SchemaItem*.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as *Table* and *Column*.

is_(*other*)

inherited from the is_() method of ColumnOperators

Implement the `IS` operator.

Normally, `IS` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.isnot()

isnot(*other*)

inherited from the isnot() method of ColumnOperators

Implement the `IS NOT` operator.

Normally, `IS NOT` is generated automatically when comparing to a value of `None`, which resolves to `NULL`. However, explicit usage of `IS NOT` may be desirable if comparing to boolean values on certain platforms.

New in version 0.7.9.

See also:

ColumnOperators.is_()

label(*name*)

inherited from the label() method of ColumnElement

Produce a column label, i.e. `<columnname> AS <name>`.

This is a shortcut to the *label()* function.

if 'name' is None, an anonymous label name will be generated.

like(*other*, *escape=None*)

inherited from the like() method of ColumnOperators

Implement the `like` operator.

In a column context, produces the clause `a LIKE other`.

E.g.:

```
select([sometable]).where(sometable.c.column.like("%foobar%"))
```

Parameters

- **other** – expression to be compared
- **escape** – optional escape character, renders the ESCAPE keyword, e.g.:

```
somecolumn.like("foo/%bar", escape="/")
```

See also:

ColumnOperators.ilike()

match(*other*, ***kwargs*)
inherited from the match() method of ColumnOperators

Implements a database-specific 'match' operator.

match() attempts to resolve to a MATCH-like function or operator provided by the backend. Examples include:

- Postgresql - renders `x @@ to_tsquery(y)`
- MySQL - renders `MATCH (x) AGAINST (y IN BOOLEAN MODE)`
- Oracle - renders `CONTAINS(x, y)`
- other backends may provide special implementations.
- Backends without any special implementation will emit the operator as "MATCH". This is compatible with SQlite, for example.

notilike(*other*, *escape=None*)
inherited from the notilike() method of ColumnOperators

implement the `NOT ILIKE` operator.

This is equivalent to using negation with *ColumnOperators.ilike()*, i.e. `~x.ilike(y)`.

New in version 0.8.

See also:

ColumnOperators.ilike()

notin_(*other*)
inherited from the notin_() method of ColumnOperators

implement the `NOT IN` operator.

This is equivalent to using negation with *ColumnOperators.in_()*, i.e. `~x.in_(y)`.

New in version 0.8.

See also:

ColumnOperators.in_()

notlike(*other*, *escape=None*)
inherited from the notlike() method of ColumnOperators

implement the `NOT LIKE` operator.

This is equivalent to using negation with *ColumnOperators.like()*, i.e. `~x.like(y)`.

New in version 0.8.

See also:

`ColumnOperators.like()`

nullsfirst()

inherited from the `nullsfirst()` *method of* `ColumnOperators`

Produce a `nullsfirst()` clause against the parent object.

nullslast()

inherited from the `nullslast()` *method of* `ColumnOperators`

Produce a `nullslast()` clause against the parent object.

op(*opstring*, *precedence=0*, *is_comparison=False*)

inherited from the `op()` *method of* `Operators`

produce a generic operator function.

e.g.:

```
somecolumn.op("*")(5)
```

produces:

```
somecolumn * 5
```

This function can also be used to make bitwise operators explicit. For example:

```
somecolumn.op('&')(0xff)
```

is a bitwise AND of the value in `somecolumn`.

Parameters

- **operator** – a string which will be output as the infix operator between this element and the expression passed to the generated function.
- **precedence** – precedence to apply to the operator, when parenthesizing expressions. A lower number will cause the expression to be parenthesized when applied against another operator with higher precedence. The default value of `0` is lower than all operators except for the comma (`,`) and `AS` operators. A value of 100 will be higher or equal to all operators, and -100 will be lower than or equal to all operators.

  New in version 0.8: - added the 'precedence' argument.
- **is_comparison** – if True, the operator will be considered as a "comparison" operator, that is which evaluates to a boolean true/false value, like ==, >, etc. This flag should be set so that ORM relationships can establish that the operator is a comparison operator when used in a custom join condition.

  New in version 0.9.2: - added the `Operators.op.is_comparison` flag.

See also:

Redefining and Creating New Operators

Using custom operators in join conditions

quote

inherited from the `quote` *attribute of* `SchemaItem`

Return the value of the `quote` flag passed to this schema object, for those schema items which have a `name` field.

Deprecated since version 0.9: Use `<obj>.name.quote`

references(*column*)
Return True if this Column references the given column via foreign key.

shares_lineage(*othercolumn*)
inherited from the `shares_lineage()` *method of* `ColumnElement`

Return True if the given `ColumnElement` has a common ancestor to this `ColumnElement`.

startswith(*other*, ***kwargs*)
inherited from the `startswith()` *method of* `ColumnOperators`

Implement the `startwith` operator.

In a column context, produces the clause `LIKE '<other>%'`

class `sqlalchemy.schema.`**MetaData**(*bind=None*, *reflect=False*, *schema=None*, *quote_schema=None*, *naming_convention=immutabledict({'ix': 'ix_%(column_0_label)s'})*, *info=None*)
Bases: `sqlalchemy.schema.SchemaItem`

A collection of `Table` objects and their associated schema constructs.

Holds a collection of `Table` objects as well as an optional binding to an `Engine` or `Connection`. If bound, the `Table` objects in the collection and their columns may participate in implicit SQL execution.

The `Table` objects themselves are stored in the `MetaData.tables` dictionary.

`MetaData` is a thread-safe object for read operations. Construction of new tables within a single `MetaData` object, either explicitly or via reflection, may not be completely thread-safe.

See also:

Describing Databases with MetaData - Introduction to database metadata

__init__(*bind=None*, *reflect=False*, *schema=None*, *quote_schema=None*, *naming_convention=immutabledict({'ix': 'ix_%(column_0_label)s'})*, *info=None*)
Create a new MetaData object.

Parameters

- **bind** – An Engine or Connection to bind to. May also be a string or URL instance, these are passed to create_engine() and this MetaData will be bound to the resulting engine.
- **reflect** – Optional, automatically load all tables from the bound database. Defaults to False. `bind` is required when this option is set.

  Deprecated since version 0.8: Please use the `MetaData.reflect()` method.
- **schema** – The default schema to use for the `Table`, `Sequence`, and other objects associated with this `MetaData`. Defaults to `None`.
- **quote_schema** – Sets the `quote_schema` flag for those `Table`, `Sequence`, and other objects which make usage of the local `schema` name.
- **info** – Optional data dictionary which will be populated into the `SchemaItem.info` attribute of this object.

  New in version 1.0.0.
- **naming_convention** – a dictionary referring to values which will establish default naming conventions for `Constraint` and `Index` objects, for those objects which are not given a name explicitly.

The keys of this dictionary may be:

- a constraint or Index class, e.g. the *UniqueConstraint*, *ForeignKeyConstraint* class, the *Index* class
- a string mnemonic for one of the known constraint classes; `"fk"`, `"pk"`, `"ix"`, `"ck"`, `"uq"` for foreign key, primary key, index, check, and unique constraint, respectively.
- the string name of a user-defined "token" that can be used to define new naming tokens.

The values associated with each "constraint class" or "constraint mnemonic" key are string naming templates, such as `"uq_%(table_name)s_%(column_0_name)s"`, which describe how the name should be composed. The values associated with user-defined "token" keys should be callables of the form `fn(constraint, table)`, which accepts the constraint/index object and *Table* as arguments, returning a string result.

The built-in names are as follows, some of which may only be available for certain types of constraint:

- `%(table_name)s` - the name of the *Table* object associated with the constraint.
- `%(referred_table_name)s` - the name of the *Table* object associated with the referencing target of a *ForeignKeyConstraint*.
- `%(column_0_name)s` - the name of the *Column* at index position "0" within the constraint.
- `%(column_0_label)s` - the label of the *Column* at index position "0", e.g. `Column.label`
- `%(column_0_key)s` - the key of the *Column* at index position "0", e.g. `Column.key`
- `%(referred_column_0_name)s` - the name of a *Column* at index position "0" referenced by a *ForeignKeyConstraint*.
- `%(constraint_name)s` - a special key that refers to the existing name given to the constraint. When this key is present, the *Constraint* object's existing name will be replaced with one that is composed from template string that uses this token. When this token is present, it is required that the *Constraint* is given an expicit name ahead of time.
- user-defined: any additional token may be implemented by passing it along with a `fn(constraint, table)` callable to the naming_convention dictionary.

New in version 0.9.2.

See also:

Configuring Constraint Naming Conventions - for detailed usage examples.

append_ddl_listener(*event_name*, *listener*)

Append a DDL event listener to this `MetaData`.

Deprecated since version 0.7: See *DDLEvents*.

bind

An *Engine* or *Connection* to which this *MetaData* is bound.

Typically, a *Engine* is assigned to this attribute so that "implicit execution" may be used, or alternatively as a means of providing engine binding information to an ORM *Session* object:

```
engine = create_engine("someurl://")
metadata.bind = engine
```

See also:

Connectionless Execution, Implicit Execution - background on "bound metadata"

clear()

Clear all Table objects from this MetaData.

create_all(*bind=None*, *tables=None*, *checkfirst=True*)

Create all tables stored in this metadata.

Conditional by default, will not attempt to recreate tables already present in the target database.

Parameters

- **bind** – A *Connectable* used to access the database; if None, uses the existing bind on this `MetaData`, if any.
- **tables** – Optional list of `Table` objects, which is a subset of the total tables in the `MetaData` (others are ignored).
- **checkfirst** – Defaults to True, don't issue CREATEs for tables already present in the target database.

drop_all(*bind=None*, *tables=None*, *checkfirst=True*)

Drop all tables stored in this metadata.

Conditional by default, will not attempt to drop tables not present in the target database.

Parameters

- **bind** – A *Connectable* used to access the database; if None, uses the existing bind on this `MetaData`, if any.
- **tables** – Optional list of `Table` objects, which is a subset of the total tables in the `MetaData` (others are ignored).
- **checkfirst** – Defaults to True, only issue DROPs for tables confirmed to be present in the target database.

is_bound()

True if this MetaData is bound to an Engine or Connection.

reflect(*bind=None*, *schema=None*, *views=False*, *only=None*, *extend_existing=False*, *autoload_replace=True*, ***dialect_kwargs*)

Load all available table definitions from the database.

Automatically creates `Table` entries in this `MetaData` for any table available in the database but not yet present in the `MetaData`. May be called multiple times to pick up tables recently added to the database, however no special action is taken if a table in this `MetaData` no longer exists in the database.

Parameters

- **bind** – A *Connectable* used to access the database; if None, uses the existing bind on this `MetaData`, if any.
- **schema** – Optional, query and reflect tables from an alterate schema. If None, the schema associated with this *MetaData* is used, if any.
- **views** – If True, also reflect views.

- **only** – Optional. Load only a sub-set of available named tables. May be specified as a sequence of names or a callable.

  If a sequence of names is provided, only those tables will be reflected. An error is raised if a table is requested but not available. Named tables already present in this `MetaData` are ignored.

  If a callable is provided, it will be used as a boolean predicate to filter the list of potential table names. The callable is called with a table name and this `MetaData` instance as positional arguments and should return a true value for any table to reflect.

- **extend_existing** – Passed along to each *Table* as *Table.extend_existing*.

  New in version 0.9.1.

- **autoload_replace** – Passed along to each *Table* as *Table.autoload_replace*.

  New in version 0.9.1.

- ****dialect_kwargs** – Additional keyword arguments not mentioned above are dialect specific, and passed in the form `<dialectname>_<argname>`. See the documentation regarding an individual dialect at *Dialects* for detail on documented arguments.

  New in version 0.9.2: - Added *MetaData.reflect.**dialect_kwargs* to support dialect-level reflection options for all *Table* objects reflected.

remove(*table*)

Remove the given Table object from this MetaData.

sorted_tables

Returns a list of *Table* objects sorted in order of foreign key dependency.

The sorting will place *Table* objects that have dependencies first, before the dependencies themselves, representing the order in which they can be created. To get the order in which the tables would be dropped, use the `reversed()` Python built-in.

> **Warning:** The *sorted_tables* accessor cannot by itself accommodate automatic resolution of dependency cycles between tables, which are usually caused by mutually dependent foreign key constraints. To resolve these cycles, either the *ForeignKeyConstraint.use_alter* parameter may be appled to those constraints, or use the *schema.sort_tables_and_constraints()* function which will break out foreign key constraints involved in cycles separately.

See also:

schema.sort_tables()

schema.sort_tables_and_constraints()

MetaData.tables

Inspector.get_table_names()

Inspector.get_sorted_table_and_fkc_names()

tables = None

A dictionary of *Table* objects keyed to their name or "table key".

The exact key is that determined by the *Table.key* attribute; for a table with no `Table.schema` attribute, this is the same as `Table.name`. For a table with a schema, it is typically of the form `schemaname.tablename`.

See also:

MetaData.sorted_tables

class sqlalchemy.schema.**SchemaItem**

Bases: sqlalchemy.sql.expression.SchemaEventTarget, sqlalchemy.sql.visitors.Visitable

Base class for items that define a database schema.

get_children(***kwargs*)

used to allow SchemaVisitor access

info

Info dictionary associated with the object, allowing user-defined data to be associated with this *SchemaItem*.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as *Table* and *Column*.

quote

Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

Deprecated since version 0.9: Use <obj>.name.quote

class sqlalchemy.schema.**Table**(**args*, ***kw*)

Bases: *sqlalchemy.sql.base.DialectKWArgs*, *sqlalchemy.schema.SchemaItem*, *sqlalchemy.sql.expression.TableClause*

Represent a table in a database.

e.g.:

```
mytable = Table("mytable", metadata,
                Column('mytable_id', Integer, primary_key=True),
                Column('value', String(50))
           )
```

The *Table* object constructs a unique instance of itself based on its name and optional schema name within the given *MetaData* object. Calling the *Table* constructor with the same name and same *MetaData* argument a second time will return the *same* *Table* object - in this way the *Table* constructor acts as a registry function.

See also:

Describing Databases with MetaData - Introduction to database metadata

Constructor arguments are as follows:

Parameters

- **name** – The name of this table as represented in the database.

  The table name, along with the value of the schema parameter, forms a key which uniquely identifies this *Table* within the owning *MetaData* collection. Additional calls to *Table* with the same name, metadata, and schema name will return the same *Table* object.

  Names which contain no upper case characters will be treated as case insensitive names, and will not be quoted unless they are a reserved word or contain special characters. A name with any number of upper case characters is considered to be case sensitive, and will be sent as quoted.

  To enable unconditional quoting for the table name, specify the flag quote=True to the constructor, or use the *quoted_name* construct to specify the name.

- **metadata** – a *MetaData* object which will contain this table. The metadata is used as a point of association of this table with other tables which are referenced via foreign key. It also may be used to associate this table with a particular *Connectable*.
- ***args** – Additional positional arguments are used primarily to add the list of *Column* objects contained within this table. Similar to the style of a CREATE TABLE statement, other *SchemaItem* constructs may be added here, including *PrimaryKeyConstraint*, and *ForeignKeyConstraint*.
- **autoload** – Defaults to False, unless *Table.autoload_with* is set in which case it defaults to True; *Column* objects for this table should be reflected from the database, possibly augmenting or replacing existing *Column* objects that were expicitly specified.

  Changed in version 1.0.0: setting the *Table.autoload_with* parameter implies that *Table.autoload* will default to True.

  See also:

  Reflecting Database Objects
- **autoload_replace** – Defaults to `True`; when using *Table.autoload* in conjunction with *Table.extend_existing*, indicates that *Column* objects present in the already-existing *Table* object should be replaced with columns of the same name retrieved from the autoload process. When `False`, columns already present under existing names will be omitted from the reflection process.

  Note that this setting does not impact *Column* objects specified programmatically within the call to *Table* that also is autoloading; those *Column* objects will always replace existing columns of the same name when *Table.extend_existing* is `True`.

  New in version 0.7.5.

  See also:

  Table.autoload

  Table.extend_existing
- **autoload_with** – An *Engine* or *Connection* object with which this *Table* object will be reflected; when set to a non-None value, it implies that *Table.autoload* is `True`. If left unset, but *Table.autoload* is explicitly set to `True`, an autoload operation will attempt to proceed by locating an *Engine* or *Connection* bound to the underlying *MetaData* object.

  See also:

  Table.autoload
- **extend_existing** – When `True`, indicates that if this *Table* is already present in the given *MetaData*, apply further arguments within the constructor to the existing *Table*.

  If *Table.extend_existing* or *Table.keep_existing* are not set, and the given name of the new *Table* refers to a *Table* that is already present in the target *MetaData* collection, and this *Table* specifies additional columns or other constructs or flags that modify the table's state, an error is raised. The purpose of these two mutually-exclusive flags is to specify what action should be taken when a *Table* is specified that matches an existing *Table*, yet specifies additional constructs.

  Table.extend_existing will also work in conjunction with *Table.autoload* to run a new reflection operation against the database, even if a *Table* of the same name is already present in the target *MetaData*; newly reflected *Column* objects and

other options will be added into the state of the *Table*, potentially overwriting existing columns and options of the same name.

Changed in version 0.7.4: *Table.extend_existing* will invoke a new reflection operation when combined with *Table.autoload* set to True.

As is always the case with *Table.autoload*, *Column* objects can be specified in the same *Table* constructor, which will take precedence. Below, the existing table `mytable` will be augmented with *Column* objects both reflected from the database, as well as the given *Column* named "y":

```
Table("mytable", metadata,
            Column('y', Integer),
            extend_existing=True,
            autoload=True,
            autoload_with=engine
        )
```

See also:

Table.autoload

Table.autoload_replace

Table.keep_existing

- **implicit_returning** – True by default - indicates that RETURNING can be used by default to fetch newly inserted primary key values, for backends which support this. Note that create_engine() also provides an implicit_returning flag.
- **include_columns** – A list of strings indicating a subset of columns to be loaded via the `autoload` operation; table columns who aren't present in this list will not be represented on the resulting `Table` object. Defaults to `None` which indicates all columns should be reflected.
- **info** – Optional data dictionary which will be populated into the *SchemaItem.info* attribute of this object.
- **keep_existing** – When `True`, indicates that if this Table is already present in the given *MetaData*, ignore further arguments within the constructor to the existing *Table*, and return the *Table* object as originally created. This is to allow a function that wishes to define a new *Table* on first call, but on subsequent calls will return the same *Table*, without any of the declarations (particularly constraints) being applied a second time.

  If *Table.extend_existing* or *Table.keep_existing* are not set, and the given name of the new *Table* refers to a *Table* that is already present in the target *MetaData* collection, and this *Table* specifies additional columns or other constructs or flags that modify the table's state, an error is raised. The purpose of these two mutually-exclusive flags is to specify what action should be taken when a *Table* is specified that matches an existing *Table*, yet specifies additional constructs.

  See also:

  Table.extend_existing

- **listeners** – A list of tuples of the form `(<eventname>, <fn>)` which will be passed to *event.listen()* upon construction. This alternate hook to *event.listen()* allows the establishment of a listener function specific to this *Table* before the "autoload" process begins. Particularly useful for the *DDLEvents.column_reflect()* event:

```
def listen_for_reflect(table, column_info):
    "handle the column reflection event"
    # ...

t = Table(
    'sometable',
    autoload=True,
    listeners=[
        ('column_reflect', listen_for_reflect)
    ])
```

- **mustexist** – When `True`, indicates that this Table must already be present in the given *`MetaData`* collection, else an exception is raised.
- **prefixes** – A list of strings to insert after CREATE in the CREATE TABLE statement. They will be separated by spaces.
- **quote** – Force quoting of this table's name on or off, corresponding to `True` or `False`. When left at its default of `None`, the column identifier will be quoted according to whether the name is case sensitive (identifiers with at least one upper case character are treated as case sensitive), or if it's a reserved word. This flag is only needed to force quoting of a reserved word which is not known by the SQLAlchemy dialect.
- **quote_schema** – same as 'quote' but applies to the schema identifier.
- **schema** – The schema name for this table, which is required if the table resides in a schema other than the default selected schema for the engine's database connection. Defaults to `None`.

  The quoting rules for the schema name are the same as those for the `name` parameter, in that quoting is applied for reserved words or case-sensitive names; to enable unconditional quoting for the schema name, specify the flag `quote_schema=True` to the constructor, or use the *`quoted_name`* construct to specify the name.
- **useexisting** – Deprecated. Use *`Table.extend_existing`*.
- ****kw** – Additional keyword arguments not mentioned above are dialect specific, and passed in the form `<dialectname>_<argname>`. See the documentation regarding an individual dialect at *Dialects* for detail on documented arguments.

__init__(**args*, ***kw*)

Constructor for *`Table`*.

This method is a no-op. See the top-level documentation for *`Table`* for constructor arguments.

add_is_dependent_on(*table*)

Add a 'dependency' for this Table.

This is another Table object which must be created first before this one can, or dropped after this one.

Usually, dependencies between tables are determined via ForeignKey objects. However, for other situations that create dependencies outside of foreign keys (rules, inheriting), this method can manually establish such a link.

alias(*name=None*, *flat=False*)

inherited from the `alias()` method of `FromClause`

return an alias of this *`FromClause`*.

This is shorthand for calling:

```
from sqlalchemy import alias
a = alias(self, name=name)
```

See `alias()` for details.

append_column(*column*)

Append a `Column` to this `Table`.

The "key" of the newly added `Column`, i.e. the value of its `.key` attribute, will then be available in the `.c` collection of this `Table`, and the column definition will be included in any CREATE TABLE, SELECT, UPDATE, etc. statements generated from this `Table` construct.

Note that this does **not** change the definition of the table as it exists within any underlying database, assuming that table has already been created in the database. Relational databases support the addition of columns to existing tables using the SQL ALTER command, which would need to be emitted for an already-existing table that doesn't contain the newly added column.

append_constraint(*constraint*)

Append a `Constraint` to this `Table`.

This has the effect of the constraint being included in any future CREATE TABLE statement, assuming specific DDL creation events have not been associated with the given `Constraint` object.

Note that this does **not** produce the constraint within the relational database automatically, for a table that already exists in the database. To add a constraint to an existing relational database table, the SQL ALTER command must be used. SQLAlchemy also provides the `AddConstraint` construct which can produce this SQL when invoked as an executable clause.

append_ddl_listener(*event_name*, *listener*)

Append a DDL event listener to this `Table`.

Deprecated since version 0.7: See `DDLEvents`.

argument_for(*dialect_name*, *argument_name*, *default*)

inherited from the `argument_for()` *method of* `DialectKWArgs`

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.

- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

bind

Return the connectable associated with this Table.

c

inherited from the `c` *attribute of* `FromClause`

An alias for the `columns` attribute.

columns

inherited from the `columns` *attribute of* `FromClause`

A named-based collection of `ColumnElement` objects maintained by this `FromClause`.

The `columns`, or `c` collection, is the gateway to the construction of SQL expressions using table-bound or other selectable-bound columns:

```
select([mytable]).where(mytable.c.somecolumn == 5)
```

compare(*other*, ***kw*)

inherited from the `compare()` *method of* `ClauseElement`

Compare this ClauseElement to the given ClauseElement.

Subclasses should override the default behavior, which is a straight identity comparison.

**kw are arguments consumed by subclass compare() methods and may be used to modify the criteria for comparison. (see `ColumnElement`)

compile(*bind=None*, *dialect=None*, ***kw*)

inherited from the `compile()` *method of* `ClauseElement`

Compile this SQL expression.

The return value is a `Compiled` object. Calling `str()` or `unicode()` on the returned value will yield a string representation of the result. The `Compiled` object also can return a dictionary of bind parameter names and values using the `params` accessor.

Parameters

- **bind** – An `Engine` or `Connection` from which a `Compiled` will be acquired. This argument takes precedence over this `ClauseElement`'s bound engine, if any.
- **column_keys** – Used for INSERT and UPDATE statements, a list of column names which should be present in the VALUES clause of the compiled statement. If `None`, all columns from the target table object are rendered.
- **dialect** – A `Dialect` instance from which a `Compiled` will be acquired. This argument takes precedence over the *bind* argument as well as this `ClauseElement`'s bound engine, if any.
- **inline** – Used for INSERT statements, for a dialect which does not support inline retrieval of newly generated primary key columns, will force the expression used to create the new primary key value to be rendered inline within the INSERT statement's VALUES clause. This typically refers to Sequence execution but may also refer to any server-side default generation function associated with a primary key *Column*.
- **compile_kwargs** – optional dictionary of additional parameters that will be passed through to the compiler within all "visit" methods. This allows any custom

flag to be passed through to a custom compilation construct, for example. It is also used for the case of passing the `literal_binds` flag through:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print s.compile(compile_kwargs={"literal_binds": True})
```

New in version 0.9.0.

See also:

How do I render SQL expressions as strings, possibly with bound parameters inlined?

correspond_on_equivalents(*column*, *equivalents*)
inherited from the `correspond_on_equivalents()` *method of* `FromClause`

Return corresponding_column for the given column, or if None search for a match in the given dictionary.

corresponding_column(*column*, *require_embedded=False*)
inherited from the `corresponding_column()` *method of* `FromClause`

Given a `ColumnElement`, return the exported `ColumnElement` object from this `Selectable` which corresponds to that original `Column` via a common ancestor column.

Parameters

- **column** – the target `ColumnElement` to be matched
- **require_embedded** – only return corresponding columns for the given `ColumnElement`, if the given `ColumnElement` is actually present within a sub-element of this `FromClause`. Normally the column will match if it merely shares a common ancestor with one of the exported columns of this `FromClause`.

count(*whereclause=None*, ***params*)
inherited from the `count()` *method of* `TableClause`

return a SELECT COUNT generated against this `TableClause`.

create(*bind=None*, *checkfirst=False*)
Issue a `CREATE` statement for this `Table`, using the given `Connectable` for connectivity.

See also:

`MetaData.create_all()`.

delete(*whereclause=None*, ***kwargs*)
inherited from the `delete()` *method of* `TableClause`

Generate a `delete()` construct against this `TableClause`.

E.g.:

```
table.delete().where(table.c.id==7)
```

See `delete()` for argument and usage information.

dialect_kwargs
inherited from the `dialect_kwargs` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options

inherited from the `dialect_options` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

drop(*bind=None*, *checkfirst=False*)

Issue a `DROP` statement for this `Table`, using the given `Connectable` for connectivity.

See also:

`MetaData.drop_all()`.

exists(*bind=None*)

Return True if this table exists.

foreign_key_constraints

`ForeignKeyConstraint` objects referred to by this `Table`.

This list is produced from the collection of `ForeignKey` objects currently associated.

New in version 1.0.0.

foreign_keys

inherited from the `foreign_keys` *attribute of* `FromClause`

Return the collection of ForeignKey objects which this FromClause references.

info

inherited from the `info` *attribute of* `SchemaItem`

Info dictionary associated with the object, allowing user-defined data to be associated with this `SchemaItem`.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as `Table` and `Column`.

insert(*values=None*, *inline=False*, ***kwargs*)

inherited from the `insert()` *method of* `TableClause`

Generate an `insert()` construct against this `TableClause`.

E.g.:

```
table.insert().values(name='foo')
```

See `insert()` for argument and usage information.

is_derived_from(*fromclause*)

inherited from the `is_derived_from()` *method of* `FromClause`

Return True if this FromClause is 'derived' from the given FromClause.

An example would be an Alias of a Table is derived from that Table.

join(*right*, *onclause=None*, *isouter=False*)

inherited from the `join()` *method of* `FromClause`

Return a `Join` from this `FromClause` to another `FromClause`.

E.g.:

```
from sqlalchemy import join

j = user_table.join(address_table,
                    user_table.c.id == address_table.c.user_id)
stmt = select([user_table]).select_from(j)
```

would emit SQL along the lines of:

```
SELECT user.id, user.name FROM user
JOIN address ON user.id = address.user_id
```

Parameters

- **right** – the right side of the join; this is any `FromClause` object such as a `Table` object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at `None`, `FromClause.join()` will attempt to join the two tables based on a foreign key relationship.
- **isouter** – if True, render a LEFT OUTER JOIN, instead of JOIN.

See also:

`join()` - standalone function

`Join` - the type of object produced

key

Return the 'key' for this `Table`.

This value is used as the dictionary key within the `MetaData.tables` collection. It is typically the same as that of `Table.name` for a table with no `Table.schema` set; otherwise it is typically of the form `schemaname.tablename`.

kwargs

inherited from the `kwargs` *attribute of* `DialectKWArgs`

A synonym for `DialectKWArgs.dialect_kwargs`.

outerjoin(*right*, *onclause=None*)

inherited from the `outerjoin()` *method of* `FromClause`

Return a `Join` from this `FromClause` to another `FromClause`, with the "isouter" flag set to True.

E.g.:

```
from sqlalchemy import outerjoin

j = user_table.outerjoin(address_table,
                         user_table.c.id == address_table.c.user_id)
```

The above is equivalent to:

```
j = user_table.join(
    address_table,
    user_table.c.id == address_table.c.user_id,
    isouter=True)
```

Parameters

- **right** – the right side of the join; this is any *FromClause* object such as a *Table* object, and may also be a selectable-compatible object such as an ORM-mapped class.
- **onclause** – a SQL expression representing the ON clause of the join. If left at None, *FromClause.join()* will attempt to join the two tables based on a foreign key relationship.

See also:

FromClause.join()

Join

primary_key

inherited from the primary_key attribute of FromClause

Return the collection of Column objects which comprise the primary key of this FromClause.

quote

inherited from the quote attribute of SchemaItem

Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

Deprecated since version 0.9: Use <obj>.name.quote

quote_schema

Return the value of the quote_schema flag passed to this *Table*.

Deprecated since version 0.9: Use table.schema.quote

replace_selectable(*old*, *alias*)

inherited from the replace_selectable() method of FromClause

replace all occurrences of FromClause 'old' with the given Alias object, returning a copy of this *FromClause*.

select(*whereclause=None*, ***params*)

inherited from the select() method of FromClause

return a SELECT of this *FromClause*.

See also:

select() - general purpose method which allows for arbitrary column lists.

self_group(*against=None*)

inherited from the `self_group()` *method of* `ClauseElement`

Apply a 'grouping' to this `ClauseElement`.

This method is overridden by subclasses to return a "grouping" construct, i.e. parenthesis. In particular it's used by "binary" expressions to provide a grouping around themselves when placed into a larger expression, as well as by `select()` constructs when placed into the FROM clause of another `select()`. (Note that subqueries should be normally created using the `Select.alias()` method, as many platforms require nested SELECT statements to be named).

As expressions are composed together, the application of `self_group()` is automatic - end-user code should never need to use this method directly. Note that SQLAlchemy's clause constructs take operator precedence into account - so parenthesis might not be needed, for example, in an expression like `x OR (y AND z)` - AND takes precedence over OR.

The base `self_group()` method of `ClauseElement` just returns self.

tometadata(*metadata*, *schema=symbol('retain_schema')*, *referred_schema_fn=None*, *name=None*)

Return a copy of this `Table` associated with a different `MetaData`.

E.g.:

```
m1 = MetaData()

user = Table('user', m1, Column('id', Integer, priamry_key=True))

m2 = MetaData()
user_copy = user.tometadata(m2)
```

Parameters

- **metadata** – Target `MetaData` object, into which the new `Table` object will be created.
- **schema** – optional string name indicating the target schema. Defaults to the special symbol `RETAIN_SCHEMA` which indicates that no change to the schema name should be made in the new `Table`. If set to a string name, the new `Table` will have this new name as the `.schema`. If set to `None`, the schema will be set to that of the schema set on the target `MetaData`, which is typically `None` as well, unless set explicitly:

```
m2 = MetaData(schema='newschema')

# user_copy_one will have "newschema" as the schema name
user_copy_one = user.tometadata(m2, schema=None)

m3 = MetaData()  # schema defaults to None

# user_copy_two will have None as the schema name
user_copy_two = user.tometadata(m3, schema=None)
```

- **referred_schema_fn** – optional callable which can be supplied in order to provide for the schema name that should be assigned to the referenced table of a `ForeignKeyConstraint`. The callable accepts this parent `Table`, the target schema that we are changing to, the `ForeignKeyConstraint` object, and the existing "target schema" of that constraint. The function should return the string schema name that should be applied. E.g.:

```
def referred_schema_fn(table, to_schema,
                                constraint, referred_schema):
    if referred_schema == 'base_tables':
        return referred_schema
    else:
        return to_schema

new_table = table.tometadata(m2, schema="alt_schema",
                        referred_schema_fn=referred_schema_fn)
```

New in version 0.9.2.

- **name** – optional string name indicating the target table name. If not specified or None, the table name is retained. This allows a `Table` to be copied to the same `MetaData` target with a new name.

  New in version 1.0.0.

update(*whereclause=None*, *values=None*, *inline=False*, ***kwargs*)
inherited from the `update()` *method of* `TableClause`

Generate an `update()` construct against this `TableClause`.

E.g.:

```
table.update().where(table.c.id==7).values(name='foo')
```

See `update()` for argument and usage information.

class sqlalchemy.schema.**ThreadLocalMetaData**
Bases: `sqlalchemy.schema.MetaData`

A MetaData variant that presents a different `bind` in every thread.

Makes the `bind` property of the MetaData a thread-local value, allowing this collection of tables to be bound to different `Engine` implementations or connections in each thread.

The ThreadLocalMetaData starts off bound to None in each thread. Binds must be made explicitly by assigning to the `bind` property or using `connect()`. You can also re-bind dynamically multiple times per thread, just like a regular `MetaData`.

__init__()
Construct a ThreadLocalMetaData.

bind
The bound Engine or Connection for this thread.

This property may be assigned an Engine or Connection, or assigned a string or URL to automatically create a basic Engine for this bind with `create_engine()`.

dispose()
Dispose all bound engines, in all thread contexts.

is_bound()
True if there is a bind for this thread.

3.3.2 Reflecting Database Objects

A `Table` object can be instructed to load information about itself from the corresponding database schema object already existing within the database. This process is called *reflection*. In the most simple case you need only specify

the table name, a *MetaData* object, and the `autoload=True` flag. If the *MetaData* is not persistently bound, also add the `autoload_with` argument:

```
>>> messages = Table('messages', meta, autoload=True, autoload_with=engine)
>>> [c.name for c in messages.columns]
['message_id', 'message_name', 'date']
```

The above operation will use the given engine to query the database for information about the `messages` table, and will then generate *Column*, *ForeignKey*, and other objects corresponding to this information as though the *Table* object were hand-constructed in Python.

When tables are reflected, if a given table references another one via foreign key, a second *Table* object is created within the *MetaData* object representing the connection. Below, assume the table `shopping_cart_items` references a table named `shopping_carts`. Reflecting the `shopping_cart_items` table has the effect such that the `shopping_carts` table will also be loaded:

```
>>> shopping_cart_items = Table('shopping_cart_items', meta, autoload=True, autoload_with=engine)
>>> 'shopping_carts' in meta.tables:
True
```

The *MetaData* has an interesting "singleton-like" behavior such that if you requested both tables individually, *MetaData* will ensure that exactly one *Table* object is created for each distinct table name. The *Table* constructor actually returns to you the already-existing *Table* object if one already exists with the given name. Such as below, we can access the already generated `shopping_carts` table just by naming it:

```
shopping_carts = Table('shopping_carts', meta)
```

Of course, it's a good idea to use `autoload=True` with the above table regardless. This is so that the table's attributes will be loaded if they have not been already. The autoload operation only occurs for the table if it hasn't already been loaded; once loaded, new calls to *Table* with the same name will not re-issue any reflection queries.

Overriding Reflected Columns

Individual columns can be overridden with explicit values when reflecting tables; this is handy for specifying custom datatypes, constraints such as primary keys that may not be configured within the database, etc.:

```
>>> mytable = Table('mytable', meta,
... Column('id', Integer, primary_key=True),   # override reflected 'id' to have primary key
... Column('mydata', Unicode(50)),    # override reflected 'mydata' to be Unicode
... autoload=True)
```

Reflecting Views

The reflection system can also reflect views. Basic usage is the same as that of a table:

```
my_view = Table("some_view", metadata, autoload=True)
```

Above, `my_view` is a *Table* object with *Column* objects representing the names and types of each column within the view "some_view".

Usually, it's desired to have at least a primary key constraint when reflecting a view, if not foreign keys as well. View reflection doesn't extrapolate these constraints.

Use the "override" technique for this, specifying explicitly those columns which are part of the primary key or have foreign key constraints:

```
my_view = Table("some_view", metadata,
                Column("view_id", Integer, primary_key=True),
                Column("related_thing", Integer, ForeignKey("othertable.thing_id")),
                autoload=True
)
```

Reflecting All Tables at Once

The *MetaData* object can also get a listing of tables and reflect the full set. This is achieved by using the *reflect()* method. After calling it, all located tables are present within the *MetaData* object's dictionary of tables:

```
meta = MetaData()
meta.reflect(bind=someengine)
users_table = meta.tables['users']
addresses_table = meta.tables['addresses']
```

`metadata.reflect()` also provides a handy way to clear or delete all the rows in a database:

```
meta = MetaData()
meta.reflect(bind=someengine)
for table in reversed(meta.sorted_tables):
    someengine.execute(table.delete())
```

Fine Grained Reflection with Inspector

A low level interface which provides a backend-agnostic system of loading lists of schema, table, column, and constraint descriptions from a given database is also available. This is known as the "Inspector":

```
from sqlalchemy import create_engine
from sqlalchemy.engine import reflection
engine = create_engine('...')
insp = reflection.Inspector.from_engine(engine)
print insp.get_table_names()
```

class `sqlalchemy.engine.reflection.`**`Inspector`**(*bind*)

Performs database schema inspection.

The Inspector acts as a proxy to the reflection methods of the *Dialect*, providing a consistent interface as well as caching support for previously fetched metadata.

A *Inspector* object is usually created via the *inspect()* function:

```
from sqlalchemy import inspect, create_engine
engine = create_engine('...')
insp = inspect(engine)
```

The inspection method above is equivalent to using the *Inspector.from_engine()* method, i.e.:

```
engine = create_engine('...')
insp = Inspector.from_engine(engine)
```

Where above, the *Dialect* may opt to return an *Inspector* subclass that provides additional methods specific to the dialect's target database.

__init__(*bind*)

Initialize a new *Inspector*.

Parameters **bind** – a *Connectable*, which is typically an instance of *Engine* or *Connection*.

For a dialect-specific instance of *Inspector*, see *Inspector.from_engine()*

default_schema_name

Return the default schema name presented by the dialect for the current engine's database user.

E.g. this is typically `public` for Postgresql and `dbo` for SQL Server.

classmethod from_engine(*bind*)

Construct a new dialect-specific Inspector object from the given engine or connection.

Parameters **bind** – a *Connectable*, which is typically an instance of *Engine* or *Connection*.

This method differs from direct a direct constructor call of *Inspector* in that the *Dialect* is given a chance to provide a dialect-specific *Inspector* instance, which may provide additional methods.

See the example at *Inspector*.

get_columns(*table_name*, *schema=None*, ***kw*)

Return information about columns in *table_name*.

Given a string *table_name* and an optional string *schema*, return column information as a list of dicts with these keys:

name the column's name

type *TypeEngine*

nullable boolean

default the column's default value

attrs dict containing optional column attributes

Parameters

- **table_name** – string name of the table. For special quoting, use *quoted_name*.
- **schema** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use *quoted_name*.

get_foreign_keys(*table_name*, *schema=None*, ***kw*)

Return information about foreign_keys in *table_name*.

Given a string *table_name*, and an optional string *schema*, return foreign key information as a list of dicts with these keys:

constrained_columns a list of column names that make up the foreign key

referred_schema the name of the referred schema

referred_table the name of the referred table

referred_columns a list of column names in the referred table that correspond to constrained_columns

name optional name of the foreign key constraint.

Parameters

- **table_name** – string name of the table. For special quoting, use `quoted_name`.
- **schema** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use `quoted_name`.

get_indexes(*table_name*, *schema=None*, ***kw*)

Return information about indexes in *table_name*.

Given a string *table_name* and an optional string *schema*, return index information as a list of dicts with these keys:

name the index's name

column_names list of column names in order

unique boolean

dialect_options dict of dialect-specific index options. May not be present for all dialects.

New in version 1.0.0.

Parameters

- **table_name** – string name of the table. For special quoting, use `quoted_name`.
- **schema** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use `quoted_name`.

get_pk_constraint(*table_name*, *schema=None*, ***kw*)

Return information about primary key constraint on *table_name*.

Given a string *table_name*, and an optional string *schema*, return primary key information as a dictionary with these keys:

constrained_columns a list of column names that make up the primary key

name optional name of the primary key constraint.

Parameters

- **table_name** – string name of the table. For special quoting, use `quoted_name`.
- **schema** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use `quoted_name`.

get_primary_keys(*table_name*, *schema=None*, ***kw*)

Return information about primary keys in *table_name*.

Deprecated since version 0.7: Call to deprecated method get_primary_keys. Use get_pk_constraint instead.

Given a string *table_name*, and an optional string *schema*, return primary key information as a list of column names.

get_schema_names()

Return all schema names.

get_sorted_table_and_fkc_names(*schema=None*)

Return dependency-sorted table and foreign key constraint names in referred to within a particular schema.

This will yield 2-tuples of `(tablename, [(tname, fkname), (tname, fkname), ...])` consisting of table names in CREATE order grouped with the foreign key constraint names that are not detected as belonging to a cycle. The final element will be `(None, [(tname, fkname), (tname, fkname), ..])` which will consist of remaining foreign key constraint names that would require a separate CREATE step after-the-fact, based on dependencies between tables.

New in version 1.0.-.

See also:

Inspector.get_table_names()

sort_tables_and_constraints() **- similar method which works** with an already-given *MetaData*.

get_table_names(*schema=None*, *order_by=None*)

Return all table names in referred to within a particular schema.

The names are expected to be real tables only, not views. Views are instead returned using the *Inspector.get_view_names()* method.

Parameters

- **schema** – Schema name. If `schema` is left at `None`, the database's default schema is used, else the named schema is searched. If the database does not support named schemas, behavior is undefined if `schema` is not passed as `None`. For special quoting, use *quoted_name*.
- **order_by** – Optional, may be the string "foreign_key" to sort the result on foreign key dependencies. Does not automatically resolve cycles, and will raise *CircularDependencyError* if cycles exist.

  Deprecated since version 1.0.0: - see *Inspector.get_sorted_table_and_fkc_names()* for a version of this which resolves foreign key cycles between tables automatically.

  Changed in version 0.8: the "foreign_key" sorting sorts tables in order of dependee to dependent; that is, in creation order, rather than in drop order. This is to maintain consistency with similar features such as *MetaData.sorted_tables* and `util.sort_tables()`.

See also:

Inspector.get_sorted_table_and_fkc_names()

MetaData.sorted_tables

get_table_options(*table_name*, *schema=None*, ***kw*)

Return a dictionary of options specified when the table of the given name was created.

This currently includes some options that apply to MySQL tables.

Parameters

- **table_name** – string name of the table. For special quoting, use *quoted_name*.
- **schema** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use *quoted_name*.

get_temp_table_names()

return a list of temporary table names for the current bind.

This method is unsupported by most dialects; currently only SQLite implements it.

New in version 1.0.0.

get_temp_view_names()

return a list of temporary view names for the current bind.

This method is unsupported by most dialects; currently only SQLite implements it.

New in version 1.0.0.

get_unique_constraints(*table_name*, *schema=None*, ***kw*)

Return information about unique constraints in *table_name*.

Given a string *table_name* and an optional string *schema*, return unique constraint information as a list of dicts with these keys:

name the unique constraint's name

column_names list of column names in order

Parameters

- **table_name** – string name of the table. For special quoting, use `quoted_name`.
- **schema** – string schema name; if omitted, uses the default schema of the database connection. For special quoting, use `quoted_name`.

New in version 0.8.4.

get_view_definition(*view_name*, *schema=None*)

Return definition for *view_name*.

Parameters **schema** – Optional, retrieve names from a non-default schema. For special quoting, use `quoted_name`.

get_view_names(*schema=None*)

Return all view names in *schema*.

Parameters **schema** – Optional, retrieve names from a non-default schema. For special quoting, use `quoted_name`.

reflecttable(*table*, *include_columns*, *exclude_columns=()*)

Given a Table object, load its internal constructs based on introspection.

This is the underlying method used by most dialects to produce table reflection. Direct usage is like:

```
from sqlalchemy import create_engine, MetaData, Table
from sqlalchemy.engine import reflection

engine = create_engine('...')
meta = MetaData()
user_table = Table('user', meta)
insp = Inspector.from_engine(engine)
insp.reflecttable(user_table, None)
```

Parameters

- **table** – a `Table` instance.
- **include_columns** – a list of string column names to include in the reflection process. If `None`, all columns are reflected.

Limitations of Reflection

It's important to note that the reflection process recreates *Table* metadata using only information which is represented in the relational database. This process by definition cannot restore aspects of a schema that aren't actually stored in the database. State which is not available from reflection includes but is not limited to:

- Client side defaults, either Python functions or SQL expressions defined using the `default` keyword of *Column* (note this is separate from `server_default`, which specifically is what's available via reflection).
- Column information, e.g. data that might have been placed into the *Column.info* dictionary
- The value of the `.quote` setting for *Column* or *Table*
- The association of a particular *Sequence* with a given *Column*

The relational database also in many cases reports on table metadata in a different format than what was specified in SQLAlchemy. The *Table* objects returned from reflection cannot be always relied upon to produce the identical DDL as the original Python-defined *Table* objects. Areas where this occurs includes server defaults, column-associated sequences and various idosyncrasies regarding constraints and datatypes. Server side defaults may be returned with cast directives (typically Postgresql will include a `::<type>` cast) or different quoting patterns than originally specified.

Another category of limitation includes schema structures for which reflection is only partially or not yet defined. Recent improvements to reflection allow things like views, indexes and foreign key options to be reflected. As of this writing, structures like CHECK constraints, table comments, and triggers are not reflected.

3.3.3 Column Insert/Update Defaults

SQLAlchemy provides a very rich featureset regarding column level events which take place during INSERT and UPDATE statements. Options include:

- Scalar values used as defaults during INSERT and UPDATE operations
- Python functions which execute upon INSERT and UPDATE operations
- SQL expressions which are embedded in INSERT statements (or in some cases execute beforehand)
- SQL expressions which are embedded in UPDATE statements
- Server side default values used during INSERT
- Markers for server-side triggers used during UPDATE

The general rule for all insert/update defaults is that they only take effect if no value for a particular column is passed as an `execute()` parameter; otherwise, the given value is used.

Scalar Defaults

The simplest kind of default is a scalar value used as the default value of a column:

```
Table("mytable", meta,
    Column("somecolumn", Integer, default=12)
)
```

Above, the value "12" will be bound as the column value during an INSERT if no other value is supplied.

A scalar value may also be associated with an UPDATE statement, though this is not very common (as UPDATE statements are usually looking for dynamic defaults):

```
Table("mytable", meta,
    Column("somecolumn", Integer, onupdate=25)
)
```

Python-Executed Functions

The `Column.default` and `Column.onupdate` keyword arguments also accept Python functions. These functions are invoked at the time of insert or update if no other value for that column is supplied, and the value returned is used for the column's value. Below illustrates a crude "sequence" that assigns an incrementing counter to a primary key column:

```
# a function which counts upwards
i = 0
def mydefault():
    global i
    i += 1
    return i

t = Table("mytable", meta,
    Column('id', Integer, primary_key=True, default=mydefault),
)
```

It should be noted that for real "incrementing sequence" behavior, the built-in capabilities of the database should normally be used, which may include sequence objects or other autoincrementing capabilities. For primary key columns, SQLAlchemy will in most cases use these capabilities automatically. See the API documentation for `Column` including the `Column.autoincrement` flag, as well as the section on `Sequence` later in this chapter for background on standard primary key generation techniques.

To illustrate onupdate, we assign the Python `datetime` function `now` to the `Column.onupdate` attribute:

```
import datetime

t = Table("mytable", meta,
    Column('id', Integer, primary_key=True),

    # define 'last_updated' to be populated with datetime.now()
    Column('last_updated', DateTime, onupdate=datetime.datetime.now),
)
```

When an update statement executes and no value is passed for `last_updated`, the `datetime.datetime.now()` Python function is executed and its return value used as the value for `last_updated`. Notice that we provide `now` as the function itself without calling it (i.e. there are no parenthesis following) - SQLAlchemy will execute the function at the time the statement executes.

Context-Sensitive Default Functions

The Python functions used by `Column.default` and `Column.onupdate` may also make use of the current statement's context in order to determine a value. The *context* of a statement is an internal SQLAlchemy object which contains all information about the statement being executed, including its source expression, the parameters associated with it and the cursor. The typical use case for this context with regards to default generation is to have access to the other values being inserted or updated on the row. To access the context, provide a function that accepts a single `context` argument:

```
def mydefault(context):
    return context.current_parameters['counter'] + 12

t = Table('mytable', meta,
    Column('counter', Integer),
    Column('counter_plus_twelve', Integer, default=mydefault, onupdate=mydefault)
)
```

Above we illustrate a default function which will execute for all INSERT and UPDATE statements where a value for `counter_plus_twelve` was otherwise not provided, and the value will be that of whatever value is present in the execution for the `counter` column, plus the number 12.

While the context object passed to the default function has many attributes, the `current_parameters` member is a special member provided only during the execution of a default function for the purposes of deriving defaults from its existing values. For a single statement that is executing many sets of bind parameters, the user-defined function is called for each set of parameters, and `current_parameters` will be provided with each individual parameter set for each execution.

SQL Expressions

The "default" and "onupdate" keywords may also be passed SQL expressions, including select statements or direct function calls:

```
t = Table("mytable", meta,
    Column('id', Integer, primary_key=True),

    # define 'create_date' to default to now()
    Column('create_date', DateTime, default=func.now()),

    # define 'key' to pull its default from the 'keyvalues' table
    Column('key', String(20), default=keyvalues.select(keyvalues.c.type='type1', limit=1)),

    # define 'last_modified' to use the current_timestamp SQL function on update
    Column('last_modified', DateTime, onupdate=func.utc_timestamp())
    )
```

Above, the `create_date` column will be populated with the result of the `now()` SQL function (which, depending on backend, compiles into `NOW()` or `CURRENT_TIMESTAMP` in most cases) during an INSERT statement, and the `key` column with the result of a SELECT subquery from another table. The `last_modified` column will be populated with the value of `UTC_TIMESTAMP()`, a function specific to MySQL, when an UPDATE statement is emitted for this table.

Note that when using `func` functions, unlike when using Python *datetime* functions we *do* call the function, i.e. with parenthesis "()" - this is because what we want in this case is the return value of the function, which is the SQL expression construct that will be rendered into the INSERT or UPDATE statement.

The above SQL functions are usually executed "inline" with the INSERT or UPDATE statement being executed, meaning, a single statement is executed which embeds the given expressions or subqueries within the VALUES or SET clause of the statement. Although in some cases, the function is "pre-executed" in a SELECT statement of its own beforehand. This happens when all of the following is true:

- the column is a primary key column
- the database dialect does not support a usable `cursor.lastrowid` accessor (or equivalent); this currently includes PostgreSQL, Oracle, and Firebird, as well as some MySQL dialects.

- the dialect does not support the "RETURNING" clause or similar, or the `implicit_returning` flag is set to `False` for the dialect. Dialects which support RETURNING currently include Postgresql, Oracle, Firebird, and MS-SQL.
- the statement is a single execution, i.e. only supplies one set of parameters and doesn't use "executemany" behavior
- the `inline=True` flag is not set on the *Insert()* or *Update()* construct, and the statement has not defined an explicit *returning()* clause.

Whether or not the default generation clause "pre-executes" is not something that normally needs to be considered, unless it is being addressed for performance reasons.

When the statement is executed with a single set of parameters (that is, it is not an "executemany" style execution), the returned *ResultProxy* will contain a collection accessible via *ResultProxy.postfetch_cols()* which contains a list of all *Column* objects which had an inline-executed default. Similarly, all parameters which were bound to the statement, including all Python and SQL expressions which were pre-executed, are present in the *ResultProxy.last_inserted_params()* or *ResultProxy.last_updated_params()* collections on *ResultProxy*. The *ResultProxy.inserted_primary_key* collection contains a list of primary key values for the row inserted (a list so that single-column and composite-column primary keys are represented in the same format).

Server Side Defaults

A variant on the SQL expression default is the *Column.server_default*, which gets placed in the CREATE TABLE statement during a *Table.create()* operation:

```
t = Table('test', meta,
    Column('abc', String(20), server_default='abc'),
    Column('created_at', DateTime, server_default=text("sysdate"))
)
```

A create call for the above table will produce:

```
CREATE TABLE test (
    abc varchar(20) default 'abc',
    created_at datetime default sysdate
)
```

The behavior of *Column.server_default* is similar to that of a regular SQL default; if it's placed on a primary key column for a database which doesn't have a way to "postfetch" the ID, and the statement is not "inlined", the SQL expression is pre-executed; otherwise, SQLAlchemy lets the default fire off on the database side normally.

Triggered Columns

Columns with values set by a database trigger or other external process may be called out using *FetchedValue* as a marker:

```
t = Table('test', meta,
    Column('abc', String(20), server_default=FetchedValue()),
    Column('def', String(20), server_onupdate=FetchedValue())
)
```

Changed in version 0.8.0b2,0.7.10: The `for_update` argument on *FetchedValue* is set automatically when specified as the `server_onupdate` argument. If using an older version, specify the onupdate above as `server_onupdate=FetchedValue(for_update=True)`.

These markers do not emit a "default" clause when the table is created, however they do set the same internal flags as a static `server_default` clause, providing hints to higher-level tools that a "post-fetch" of these rows should be performed after an insert or update.

Note: It's generally not appropriate to use *FetchedValue* in conjunction with a primary key column, particularly when using the ORM or any other scenario where the *ResultProxy.inserted_primary_key* attribute is required. This is becaue the "post-fetch" operation requires that the primary key value already be available, so that the row can be selected on its primary key.

For a server-generated primary key value, all databases provide special accessors or other techniques in order to acquire the "last inserted primary key" column of a table. These mechanisms aren't affected by the presence of *FetchedValue*. For special situations where triggers are used to generate primary key values, and the database in use does not support the `RETURNING` clause, it may be necessary to forego the usage of the trigger and instead apply the SQL expression or function as a "pre execute" expression:

```
t = Table('test', meta,
        Column('abc', MyType, default=func.generate_new_value(), primary_key=True)
)
```

Where above, when *Table.insert()* is used, the `func.generate_new_value()` expression will be pre-executed in the context of a scalar `SELECT` statement, and the new value will be applied to the subsequent `INSERT`, while at the same time being made available to the *ResultProxy.inserted_primary_key* attribute.

Defining Sequences

SQLAlchemy represents database sequences using the *Sequence* object, which is considered to be a special case of "column default". It only has an effect on databases which have explicit support for sequences, which currently includes Postgresql, Oracle, and Firebird. The *Sequence* object is otherwise ignored.

The *Sequence* may be placed on any column as a "default" generator to be used during INSERT operations, and can also be configured to fire off during UPDATE operations if desired. It is most commonly used in conjunction with a single integer primary key column:

```
table = Table("cartitems", meta,
    Column("cart_id", Integer, Sequence('cart_id_seq'), primary_key=True),
    Column("description", String(40)),
    Column("createdate", DateTime())
)
```

Where above, the table "cartitems" is associated with a sequence named "cart_id_seq". When INSERT statements take place for "cartitems", and no value is passed for the "cart_id" column, the "cart_id_seq" sequence will be used to generate a value.

When the *Sequence* is associated with a table, CREATE and DROP statements issued for that table will also issue CREATE/DROP for the sequence object as well, thus "bundling" the sequence object with its parent table.

The *Sequence* object also implements special functionality to accommodate Postgresql's SERIAL datatype. The SERIAL type in PG automatically generates a sequence that is used implicitly during inserts. This means that if a *Table* object defines a *Sequence* on its primary key column so that it works with Oracle and Firebird, the *Sequence* would get in the way of the "implicit" sequence that PG would normally use. For this use case, add

the flag `optional=True` to the *Sequence* object - this indicates that the *Sequence* should only be used if the database provides no other option for generating primary key identifiers.

The *Sequence* object also has the ability to be executed standalone like a SQL expression, which has the effect of calling its "next value" function:

```
seq = Sequence('some_sequence')
nextid = connection.execute(seq)
```

Associating a Sequence as the Server Side Default

When we associate a *Sequence* with a *Column* as above, this association is an **in-Python only** association. The CREATE TABLE that would be generated for our *Table* would not refer to this sequence. If we want the sequence to be used as a server-side default, meaning it takes place even if we emit INSERT commands to the table from the SQL commandline, we can use the *Column.server_default* parameter in conjunction with the value-generation function of the sequence, available from the *Sequence.next_value()* method:

```
cart_id_seq = Sequence('cart_id_seq')
table = Table("cartitems", meta,
    Column(
        "cart_id", Integer, cart_id_seq,
        server_default=cart_id_seq.next_value(), primary_key=True),
    Column("description", String(40)),
    Column("createdate", DateTime())
)
```

The above metadata will generate a CREATE TABLE statement on Postgresql as:

```
CREATE TABLE cartitems (
    cart_id INTEGER DEFAULT nextval('cart_id_seq') NOT NULL,
    description VARCHAR(40),
    createdate TIMESTAMP WITHOUT TIME ZONE,
    PRIMARY KEY (cart_id)
)
```

We place the *Sequence* also as a Python-side default above, that is, it is mentioned twice in the *Column* definition. Depending on the backend in use, this may not be strictly necessary, for example on the Postgresql backend the Core will use `RETURNING` to access the newly generated primary key value in any case. However, for the best compatibility, *Sequence* was originally intended to be a Python-side directive first and foremost so it's probably a good idea to specify it in this way as well.

Default Objects API

class `sqlalchemy.schema.`**`ColumnDefault`**(*arg*, ***kwargs*)

Bases: *sqlalchemy.schema.DefaultGenerator*

A plain default value on a column.

This could correspond to a constant, a callable function, or a SQL clause.

ColumnDefault is generated automatically whenever the `default`, `onupdate` arguments of *Column* are used. A *ColumnDefault* can be passed positionally as well.

For example, the following:

```
Column('foo', Integer, default=50)
```

Is equivalent to:

```
Column('foo', Integer, ColumnDefault(50))
```

class sqlalchemy.schema.**DefaultClause**(*arg*, *for_update=False*, *_reflected=False*)

Bases: *sqlalchemy.schema.FetchedValue*

A DDL-specified DEFAULT column value.

DefaultClause is a *FetchedValue* that also generates a "DEFAULT" clause when "CREATE TABLE" is emitted.

DefaultClause is generated automatically whenever the server_default, server_onupdate arguments of *Column* are used. A *DefaultClause* can be passed positionally as well.

For example, the following:

```
Column('foo', Integer, server_default="50")
```

Is equivalent to:

```
Column('foo', Integer, DefaultClause("50"))
```

class sqlalchemy.schema.**DefaultGenerator**(*for_update=False*)

Bases: sqlalchemy.schema._NotAColumnExpr, *sqlalchemy.schema.SchemaItem*

Base class for column *default* values.

class sqlalchemy.schema.**FetchedValue**(*for_update=False*)

Bases: sqlalchemy.schema._NotAColumnExpr, sqlalchemy.sql.expression.SchemaEventTarget

A marker for a transparent database-side default.

Use *FetchedValue* when the database is configured to provide some automatic default for a column.

E.g.:

```
Column('foo', Integer, FetchedValue())
```

Would indicate that some trigger or default generator will create a new value for the foo column during an INSERT.

See also:

Triggered Columns

class sqlalchemy.schema.**PassiveDefault**(**arg*, ***kw*)

Bases: *sqlalchemy.schema.DefaultClause*

A DDL-specified DEFAULT column value.

Deprecated since version 0.6: *PassiveDefault* is deprecated. Use *DefaultClause*.

class sqlalchemy.schema.**Sequence**(*name*, *start=None*, *increment=None*, *minvalue=None*, *maxvalue=None*, *nominvalue=None*, *nomaxvalue=None*, *cycle=None*, *schema=None*, *optional=False*, *quote=None*, *metadata=None*, *quote_schema=None*, *for_update=False*)

Bases: *sqlalchemy.schema.DefaultGenerator*

Represents a named database sequence.

The *Sequence* object represents the name and configurational parameters of a database sequence. It also represents a construct that can be "executed" by a SQLAlchemy *Engine* or *Connection*, rendering the appropriate "next value" function for the target database and returning a result.

The *Sequence* is typically associated with a primary key column:

```
some_table = Table(
    'some_table', metadata,
    Column('id', Integer, Sequence('some_table_seq'),
    primary_key=True)
)
```

When CREATE TABLE is emitted for the above *Table*, if the target platform supports sequences, a CREATE SEQUENCE statement will be emitted as well. For platforms that don't support sequences, the *Sequence* construct is ignored.

See also:

CreateSequence

DropSequence

__init__(*name*, *start=None*, *increment=None*, *minvalue=None*, *maxvalue=None*, *nominvalue=None*, *nomaxvalue=None*, *cycle=None*, *schema=None*, *optional=False*, *quote=None*, *metadata=None*, *quote_schema=None*, *for_update=False*)

Construct a *Sequence* object.

Parameters

- **name** – The name of the sequence.
- **start** – the starting index of the sequence. This value is used when the CREATE SEQUENCE command is emitted to the database as the value of the "START WITH" clause. If None, the clause is omitted, which on most platforms indicates a starting value of 1.
- **increment** – the increment value of the sequence. This value is used when the CREATE SEQUENCE command is emitted to the database as the value of the "INCREMENT BY" clause. If None, the clause is omitted, which on most platforms indicates an increment of 1.
- **minvalue** – the minimum value of the sequence. This value is used when the CREATE SEQUENCE command is emitted to the database as the value of the "MINVALUE" clause. If None, the clause is omitted, which on most platforms indicates a minvalue of 1 and -2^63-1 for ascending and descending sequences, respectively.

  New in version 1.0.7.
- **maxvalue** – the maximum value of the sequence. This value is used when the CREATE SEQUENCE command is emitted to the database as the value of the "MAXVALUE" clause. If None, the clause is omitted, which on most platforms indicates a maxvalue of 2^63-1 and -1 for ascending and descending sequences, respectively.

  New in version 1.0.7.
- **nominvalue** – no minimum value of the sequence. This value is used when the CREATE SEQUENCE command is emitted to the database as the value of the "NO MINVALUE" clause. If None, the clause is omitted, which on most platforms indicates a minvalue of 1 and -2^63-1 for ascending and descending sequences, respectively.

  New in version 1.0.7.

- **nomaxvalue** – no maximum value of the sequence. This value is used when the CREATE SEQUENCE command is emitted to the database as the value of the "NO MAXVALUE" clause. If `None`, the clause is omitted, which on most platforms indicates a maxvalue of 2^63-1 and -1 for ascending and descending sequences, respectively.

  New in version 1.0.7.

- **cycle** – allows the sequence to wrap around when the maxvalue or minvalue has been reached by an ascending or descending sequence respectively. This value is used when the CREATE SEQUENCE command is emitted to the database as the "CYCLE" clause. If the limit is reached, the next number generated will be the minvalue or maxvalue, respectively. If cycle=False (the default) any calls to nextval after the sequence has reached its maximum value will return an error.

  New in version 1.0.7.

- **schema** – Optional schema name for the sequence, if located in a schema other than the default.

- **optional** – boolean value, when `True`, indicates that this *Sequence* object only needs to be explicitly generated on backends that don't provide another way to generate primary key identifiers. Currently, it essentially means, "don't create this sequence on the Postgresql backend, where the SERIAL keyword creates a sequence for us automatically".

- **quote** – boolean value, when `True` or `False`, explicitly forces quoting of the schema name on or off. When left at its default of `None`, normal quoting rules based on casing and reserved words take place.

- **quote_schema** – set the quoting preferences for the `schema` name.

- **metadata** – optional *MetaData* object which will be associated with this *Sequence*. A *Sequence* that is associated with a *MetaData* gains access to the `bind` of that *MetaData*, meaning the *Sequence.create()* and *Sequence.drop()* methods will make usage of that engine automatically.

  Changed in version 0.7: Additionally, the appropriate CREATE SEQUENCE/ DROP SEQUENCE DDL commands will be emitted corresponding to this *Sequence* when *MetaData.create_all()* and *MetaData.drop_all()* are invoked.

  Note that when a *Sequence* is applied to a *Column*, the *Sequence* is automatically associated with the *MetaData* object of that column's parent *Table*, when that association is made. The *Sequence* will then be subject to automatic CREATE SEQUENCE/DROP SEQUENCE corresponding to when the *Table* object itself is created or dropped, rather than that of the *MetaData* object overall.

- **for_update** – Indicates this *Sequence*, when associated with a *Column*, should be invoked for UPDATE statements on that column's table, rather than for INSERT statements, when no value is otherwise present for that column in the statement.

create(*bind=None*, *checkfirst=True*)
Creates this sequence in the database.

drop(*bind=None*, *checkfirst=True*)
Drops this sequence from the database.

next_value()
Return a *next_value* function element which will render the appropriate increment function for this *Sequence* within any SQL expression.

3.3.4 Defining Constraints and Indexes

This section will discuss SQL constraints and indexes. In SQLAlchemy the key classes include `ForeignKeyConstraint` and `Index`.

Defining Foreign Keys

A *foreign key* in SQL is a table-level construct that constrains one or more columns in that table to only allow values that are present in a different set of columns, typically but not always located on a different table. We call the columns which are constrained the *foreign key* columns and the columns which they are constrained towards the *referenced* columns. The referenced columns almost always define the primary key for their owning table, though there are exceptions to this. The foreign key is the "joint" that connects together pairs of rows which have a relationship with each other, and SQLAlchemy assigns very deep importance to this concept in virtually every area of its operation.

In SQLAlchemy as well as in DDL, foreign key constraints can be defined as additional attributes within the table clause, or for single-column foreign keys they may optionally be specified within the definition of a single column. The single column foreign key is more common, and at the column level is specified by constructing a `ForeignKey` object as an argument to a `Column` object:

```
user_preference = Table('user_preference', metadata,
    Column('pref_id', Integer, primary_key=True),
    Column('user_id', Integer, ForeignKey("user.user_id"), nullable=False),
    Column('pref_name', String(40), nullable=False),
    Column('pref_value', String(100))
)
```

Above, we define a new table `user_preference` for which each row must contain a value in the `user_id` column that also exists in the `user` table's `user_id` column.

The argument to `ForeignKey` is most commonly a string of the form *<tablename>.<columnname>*, or for a table in a remote schema or "owner" of the form *<schemaname>.<tablename>.<columnname>*. It may also be an actual `Column` object, which as we'll see later is accessed from an existing `Table` object via its `c` collection:

```
ForeignKey(user.c.user_id)
```

The advantage to using a string is that the in-python linkage between `user` and `user_preference` is resolved only when first needed, so that table objects can be easily spread across multiple modules and defined in any order.

Foreign keys may also be defined at the table level, using the `ForeignKeyConstraint` object. This object can describe a single- or multi-column foreign key. A multi-column foreign key is known as a *composite* foreign key, and almost always references a table that has a composite primary key. Below we define a table `invoice` which has a composite primary key:

```
invoice = Table('invoice', metadata,
    Column('invoice_id', Integer, primary_key=True),
    Column('ref_num', Integer, primary_key=True),
    Column('description', String(60), nullable=False)
)
```

And then a table `invoice_item` with a composite foreign key referencing `invoice`:

```
invoice_item = Table('invoice_item', metadata,
    Column('item_id', Integer, primary_key=True),
    Column('item_name', String(60), nullable=False),
    Column('invoice_id', Integer, nullable=False),
```

```
    Column('ref_num', Integer, nullable=False),
    ForeignKeyConstraint(['invoice_id', 'ref_num'], ['invoice.invoice_id', 'invoice.ref_num'])
)
```

It's important to note that the *ForeignKeyConstraint* is the only way to define a composite foreign key. While we could also have placed individual *ForeignKey* objects on both the `invoice_item.invoice_id` and `invoice_item.ref_num` columns, SQLAlchemy would not be aware that these two values should be paired together - it would be two individual foreign key constraints instead of a single composite foreign key referencing two columns.

Creating/Dropping Foreign Key Constraints via ALTER

The behavior we've seen in tutorials and elsewhere involving foreign keys with DDL illustrates that the constraints are typically rendered "inline" within the CREATE TABLE statement, such as:

```
CREATE TABLE addresses (
    id INTEGER NOT NULL,
    user_id INTEGER,
    email_address VARCHAR NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT user_id_fk FOREIGN KEY(user_id) REFERENCES users (id)
)
```

The `CONSTRAINT .. FOREIGN KEY` directive is used to create the constraint in an "inline" fashion within the CREATE TABLE definition. The *MetaData.create_all()* and *MetaData.drop_all()* methods do this by default, using a topological sort of all the *Table* objects involved such that tables are created and dropped in order of their foreign key dependency (this sort is also available via the *MetaData.sorted_tables* accessor).

This approach can't work when two or more foreign key constraints are involved in a "dependency cycle", where a set of tables are mutually dependent on each other, assuming the backend enforces foreign keys (always the case except on SQLite, MySQL/MyISAM). The methods will therefore break out constraints in such a cycle into separate ALTER statements, on all backends other than SQLite which does not support most forms of ALTER. Given a schema like:

```
node = Table(
    'node', metadata,
    Column('node_id', Integer, primary_key=True),
    Column(
        'primary_element', Integer,
        ForeignKey('element.element_id')
    )
)

element = Table(
    'element', metadata,
    Column('element_id', Integer, primary_key=True),
    Column('parent_node_id', Integer),
    ForeignKeyConstraint(
        ['parent_node_id'], ['node.node_id'],
        name='fk_element_parent_node_id'
    )
)
```

When we call upon *MetaData.create_all()* on a backend such as the Postgresql backend, the cycle between these two tables is resolved and the constraints are created separately:

```
>>> with engine.connect() as conn:
...     metadata.create_all(conn, checkfirst=False)
CREATE TABLE element (
    element_id SERIAL NOT NULL,
    parent_node_id INTEGER,
    PRIMARY KEY (element_id)
)

CREATE TABLE node (
    node_id SERIAL NOT NULL,
    primary_element INTEGER,
    PRIMARY KEY (node_id)
)

ALTER TABLE element ADD CONSTRAINT fk_element_parent_node_id
    FOREIGN KEY(parent_node_id) REFERENCES node (node_id)
ALTER TABLE node ADD FOREIGN KEY(primary_element)
    REFERENCES element (element_id)
```

In order to emit DROP for these tables, the same logic applies, however note here that in SQL, to emit DROP CONSTRAINT requires that the constraint has a name. In the case of the `'node'` table above, we haven't named this constraint; the system will therefore attempt to emit DROP for only those constraints that are named:

```
>>> with engine.connect() as conn:
...     metadata.drop_all(conn, checkfirst=False)
ALTER TABLE element DROP CONSTRAINT fk_element_parent_node_id
DROP TABLE node
DROP TABLE element
```

In the case where the cycle cannot be resolved, such as if we hadn't applied a name to either constraint here, we will receive the following error:

```
sqlalchemy.exc.CircularDependencyError: Can't sort tables for DROP;
an unresolvable foreign key dependency exists between tables:
element, node.  Please ensure that the ForeignKey and ForeignKeyConstraint
objects involved in the cycle have names so that they can be dropped
using DROP CONSTRAINT.
```

This error only applies to the DROP case as we can emit "ADD CONSTRAINT" in the CREATE case without a name; the database typically assigns one automatically.

The *`ForeignKeyConstraint.use_alter`* and *`ForeignKey.use_alter`* keyword arguments can be used to manually resolve dependency cycles. We can add this flag only to the `'element'` table as follows:

```
element = Table(
    'element', metadata,
    Column('element_id', Integer, primary_key=True),
    Column('parent_node_id', Integer),
    ForeignKeyConstraint(
        ['parent_node_id'], ['node.node_id'],
        use_alter=True, name='fk_element_parent_node_id'
    )
)
```

in our CREATE DDL we will see the ALTER statement only for this constraint, and not the other one:

```
>>> with engine.connect() as conn:
...     metadata.create_all(conn, checkfirst=False)
CREATE TABLE element (
    element_id SERIAL NOT NULL,
    parent_node_id INTEGER,
    PRIMARY KEY (element_id)
)

CREATE TABLE node (
    node_id SERIAL NOT NULL,
    primary_element INTEGER,
    PRIMARY KEY (node_id),
    FOREIGN KEY(primary_element) REFERENCES element (element_id)
)

ALTER TABLE element ADD CONSTRAINT fk_element_parent_node_id
FOREIGN KEY(parent_node_id) REFERENCES node (node_id)
```

ForeignKeyConstraint.use_alter and *ForeignKey.use_alter*, when used in conjunction with a drop operation, will require that the constraint is named, else an error like the following is generated:

```
sqlalchemy.exc.CompileError: Can't emit DROP CONSTRAINT for constraint
ForeignKeyConstraint(...); it has no name
```

Changed in version 1.0.0: - The DDL system invoked by *MetaData.create_all()* and *MetaData.drop_all()* will now automatically resolve mutually depdendent foreign keys between tables declared by *ForeignKeyConstraint* and *ForeignKey* objects, without the need to explicitly set the *ForeignKeyConstraint.use_alter* flag.

Changed in version 1.0.0: - The *ForeignKeyConstraint.use_alter* flag can be used with an un-named constraint; only the DROP operation will emit a specific error when actually called upon.

See also:

Configuring Constraint Naming Conventions

sort_tables_and_constraints()

ON UPDATE and ON DELETE

Most databases support *cascading* of foreign key values, that is the when a parent row is updated the new value is placed in child rows, or when the parent row is deleted all corresponding child rows are set to null or deleted. In data definition language these are specified using phrases like "ON UPDATE CASCADE", "ON DELETE CASCADE", and "ON DELETE SET NULL", corresponding to foreign key constraints. The phrase after "ON UPDATE" or "ON DELETE" may also other allow other phrases that are specific to the database in use. The *ForeignKey* and *ForeignKeyConstraint* objects support the generation of this clause via the `onupdate` and `ondelete` keyword arguments. The value is any string which will be output after the appropriate "ON UPDATE" or "ON DELETE" phrase:

```
child = Table('child', meta,
    Column('id', Integer,
            ForeignKey('parent.id', onupdate="CASCADE", ondelete="CASCADE"),
            primary_key=True
    )
)
```

```
composite = Table('composite', meta,
    Column('id', Integer, primary_key=True),
    Column('rev_id', Integer),
    Column('note_id', Integer),
    ForeignKeyConstraint(
                ['rev_id', 'note_id'],
                ['revisions.id', 'revisions.note_id'],
                onupdate="CASCADE", ondelete="SET NULL"
    )
)
```

Note that these clauses are not supported on SQLite, and require `InnoDB` tables when used with MySQL. They may also not be supported on other databases.

UNIQUE Constraint

Unique constraints can be created anonymously on a single column using the `unique` keyword on *Column*. Explicitly named unique constraints and/or those with multiple columns are created via the *UniqueConstraint* table-level construct.

```
from sqlalchemy import UniqueConstraint

meta = MetaData()
mytable = Table('mytable', meta,

    # per-column anonymous unique constraint
    Column('col1', Integer, unique=True),

    Column('col2', Integer),
    Column('col3', Integer),

    # explicit/composite unique constraint.  'name' is optional.
    UniqueConstraint('col2', 'col3', name='uix_1')
    )
```

CHECK Constraint

Check constraints can be named or unnamed and can be created at the Column or Table level, using the *CheckConstraint* construct. The text of the check constraint is passed directly through to the database, so there is limited "database independent" behavior. Column level check constraints generally should only refer to the column to which they are placed, while table level constraints can refer to any columns in the table.

Note that some databases do not actively support check constraints such as MySQL.

```
from sqlalchemy import CheckConstraint

meta = MetaData()
mytable = Table('mytable', meta,

    # per-column CHECK constraint
    Column('col1', Integer, CheckConstraint('col1>5')),

    Column('col2', Integer),
    Column('col3', Integer),
```

```
    # table level CHECK constraint.  'name' is optional.
    CheckConstraint('col2 > col3 + 5', name='check1')
    )

mytable.create(engine)
CREATE TABLE mytable (
    col1 INTEGER  CHECK (col1>5),
    col2 INTEGER,
    col3 INTEGER,
    CONSTRAINT check1  CHECK (col2 > col3 + 5)
)
```

PRIMARY KEY Constraint

The primary key constraint of any `Table` object is implicitly present, based on the `Column` objects that are marked with the `Column.primary_key` flag. The `PrimaryKeyConstraint` object provides explicit access to this constraint, which includes the option of being configured directly:

```
from sqlalchemy import PrimaryKeyConstraint

my_table = Table('mytable', metadata,
            Column('id', Integer),
            Column('version_id', Integer),
            Column('data', String(50)),
            PrimaryKeyConstraint('id', 'version_id', name='mytable_pk')
        )
```

See also:

`PrimaryKeyConstraint` - detailed API documentation.

Setting up Constraints when using the Declarative ORM Extension

The `Table` is the SQLAlchemy Core construct that allows one to define table metadata, which among other things can be used by the SQLAlchemy ORM as a target to map a class. The *Declarative* extension allows the `Table` object to be created automatically, given the contents of the table primarily as a mapping of `Column` objects.

To apply table-level constraint objects such as `ForeignKeyConstraint` to a table defined using Declarative, use the `__table_args__` attribute, described at *Table Configuration*.

Configuring Constraint Naming Conventions

Relational databases typically assign explicit names to all constraints and indexes. In the common case that a table is created using `CREATE TABLE` where constraints such as CHECK, UNIQUE, and PRIMARY KEY constraints are produced inline with the table definition, the database usually has a system in place in which names are automatically assigned to these constraints, if a name is not otherwise specified. When an existing database table is altered in a database using a command such as `ALTER TABLE`, this command typically needs to specify expicit names for new constraints as well as be able to specify the name of an existing constraint that is to be dropped or modified.

Constraints can be named explicitly using the `Constraint.name` parameter, and for indexes the `Index.name` parameter. However, in the case of constraints this parameter is optional. There are also the use cases of using the `Column.unique` and `Column.index` parameters which create `UniqueConstraint` and `Index` objects without an explicit name being specified.

The use case of alteration of existing tables and constraints can be handled by schema migration tools such as Alembic. However, neither Alembic nor SQLAlchemy currently create names for constraint objects where the name is otherwise unspecified, leading to the case where being able to alter existing constraints means that one must reverse-engineer the naming system used by the relational database to auto-assign names, or that care must be taken to ensure that all constraints are named.

In contrast to having to assign explicit names to all *Constraint* and *Index* objects, automated naming schemes can be constructed using events. This approach has the advantage that constraints will get a consistent naming scheme without the need for explicit name parameters throughout the code, and also that the convention takes place just as well for those constraints and indexes produced by the *Column.unique* and *Column.index* parameters. As of SQLAlchemy 0.9.2 this event-based approach is included, and can be configured using the argument *MetaData.naming_convention*.

MetaData.naming_convention refers to a dictionary which accepts the *Index* class or individual *Constraint* classes as keys, and Python string templates as values. It also accepts a series of string-codes as alternative keys, `"fk"`, `"pk"`, `"ix"`, `"ck"`, `"uq"` for foreign key, primary key, index, check, and unique constraint, respectively. The string templates in this dictionary are used whenever a constraint or index is associated with this *MetaData* object that does not have an existing name given (including one exception case where an existing name can be further embellished).

An example naming convention that suits basic cases is as follows:

```
convention = {
  "ix": 'ix_%(column_0_label)s',
  "uq": "uq_%(table_name)s_%(column_0_name)s",
  "ck": "ck_%(table_name)s_%(constraint_name)s",
  "fk": "fk_%(table_name)s_%(column_0_name)s_%(referred_table_name)s",
  "pk": "pk_%(table_name)s"
}

metadata = MetaData(naming_convention=convention)
```

The above convention will establish names for all constraints within the target *MetaData* collection. For example, we can observe the name produced when we create an unnamed *UniqueConstraint*:

```
>>> user_table = Table('user', metadata,
...                 Column('id', Integer, primary_key=True),
...                 Column('name', String(30), nullable=False),
...                 UniqueConstraint('name')
... )
>>> list(user_table.constraints)[1].name
'uq_user_name'
```

This same feature takes effect even if we just use the *Column.unique* flag:

```
>>> user_table = Table('user', metadata,
...                  Column('id', Integer, primary_key=True),
...                  Column('name', String(30), nullable=False, unique=True)
...     )
>>> list(user_table.constraints)[1].name
'uq_user_name'
```

A key advantage to the naming convention approach is that the names are established at Python construction time, rather than at DDL emit time. The effect this has when using Alembic's `--autogenerate` feature is that the naming convention will be explicit when a new migration script is generated:

```
def upgrade():
    op.create_unique_constraint("uq_user_name", "user", ["name"])
```

The above "uq_user_name" string was copied from the *UniqueConstraint* object that --autogenerate located in our metadata.

The default value for *MetaData.naming_convention* handles the long-standing SQLAlchemy behavior of assigning a name to a *Index* object that is created using the *Column.index* parameter:

```
>>> from sqlalchemy.sql.schema import DEFAULT_NAMING_CONVENTION
>>> DEFAULT_NAMING_CONVENTION
immutabledict({'ix': 'ix_%(column_0_label)s'})
```

The tokens available include %(table_name)s, %(referred_table_name)s, %(column_0_name)s, %(column_0_label)s, %(column_0_key)s, %(referred_column_0_name)s, and %(constraint_name)s; the documentation for *MetaData.naming_convention* describes each individually. New tokens can also be added, by specifying an additional token and a callable within the naming_convention dictionary. For example, if we wanted to name our foreign key constraints using a GUID scheme, we could do that as follows:

```
import uuid

def fk_guid(constraint, table):
    str_tokens = [
        table.name,
    ] + [
        element.parent.name for element in constraint.elements
    ] + [
        element.target_fullname for element in constraint.elements
    ]
    guid = uuid.uuid5(uuid.NAMESPACE_OID, "_".join(str_tokens).encode('ascii'))
    return str(guid)

convention = {
    "fk_guid": fk_guid,
    "ix": 'ix_%(column_0_label)s',
    "fk": "fk_%(fk_guid)s",
}
```

Above, when we create a new *ForeignKeyConstraint*, we will get a name as follows:

```
>>> metadata = MetaData(naming_convention=convention)

>>> user_table = Table('user', metadata,
...         Column('id', Integer, primary_key=True),
...         Column('version', Integer, primary_key=True),
...         Column('data', String(30))
...     )
>>> address_table = Table('address', metadata,
...        Column('id', Integer, primary_key=True),
...        Column('user_id', Integer),
...        Column('user_version_id', Integer)
...    )
>>> fk = ForeignKeyConstraint(['user_id', 'user_version_id'],
...                 ['user.id', 'user.version'])
```

```
>>> address_table.append_constraint(fk)
>>> fk.name
fk_0cd51ab5-8d70-56e8-a83c-86661737766d
```

See also:

MetaData.naming_convention - for additional usage details as well as a listing of all available naming components.

The Importance of Naming Constraints - in the Alembic documentation.

New in version 0.9.2: Added the *MetaData.naming_convention* argument.

Naming CHECK Constraints

The *CheckConstraint* object is configured against an arbitrary SQL expression, which can have any number of columns present, and additionally is often configured using a raw SQL string. Therefore a common convention to use with *CheckConstraint* is one where we expect the object to have a name already, and we then enhance it with other convention elements. A typical convention is `"ck_%(table_name)s_%(constraint_name)s"`:

```
metadata = MetaData(
    naming_convention={"ck": "ck_%(table_name)s_%(constraint_name)s"}
)

Table('foo', metadata,
    Column('value', Integer),
    CheckConstraint('value > 5', name='value_gt_5')
)
```

The above table will produce the name `ck_foo_value_gt_5`:

```
CREATE TABLE foo (
    value INTEGER,
    CONSTRAINT ck_foo_value_gt_5 CHECK (value > 5)
)
```

CheckConstraint also supports the `%(columns_0_name)s` token; we can make use of this by ensuring we use a *Column* or *sql.expression.column()* element within the constraint's expression, either by declaring the constraint separate from the table:

```
metadata = MetaData(
    naming_convention={"ck": "ck_%(table_name)s_%(column_0_name)s"}
)

foo = Table('foo', metadata,
    Column('value', Integer)
)

CheckConstraint(foo.c.value > 5)
```

or by using a *sql.expression.column()* inline:

```
from sqlalchemy import column

metadata = MetaData(
```

```
    naming_convention={"ck": "ck_%(table_name)s_%(column_0_name)s"}
)

foo = Table('foo', metadata,
    Column('value', Integer),
    CheckConstraint(column('value') > 5)
)
```

Both will produce the name `ck_foo_value`:

```
CREATE TABLE foo (
    value INTEGER,
    CONSTRAINT ck_foo_value CHECK (value > 5)
)
```

The determination of the name of "column zero" is performed by scanning the given expression for column objects. If the expression has more than one column present, the scan does use a deterministic search, however the structure of the expression will determine which column is noted as "column zero".

New in version 1.0.0: The *CheckConstraint* object now supports the `column_0_name` naming convention token.

Configuring Naming for Boolean, Enum, and other schema types

The *SchemaType* class refers to type objects such as *Boolean* and *Enum* which generate a CHECK constraint accompanying the type. The name for the constraint here is most directly set up by sending the "name" parameter, e.g. *Boolean.name*:

```
Table('foo', metadata,
    Column('flag', Boolean(name='ck_foo_flag'))
)
```

The naming convention feature may be combined with these types as well, normally by using a convention which includes `%(constraint_name)s` and then applying a name to the type:

```
metadata = MetaData(
    naming_convention={"ck": "ck_%(table_name)s_%(constraint_name)s"}
)

Table('foo', metadata,
    Column('flag', Boolean(name='flag_bool'))
)
```

The above table will produce the constraint name `ck_foo_flag_bool`:

```
CREATE TABLE foo (
    flag BOOL,
    CONSTRAINT ck_foo_flag_bool CHECK (flag IN (0, 1))
)
```

The *SchemaType* classes use special internal symbols so that the naming convention is only determined at DDL compile time. On Postgresql, there's a native BOOLEAN type, so the CHECK constraint of *Boolean* is not needed; we are safe to set up a *Boolean* type without a name, even though a naming convention is in place for check

constraints. This convention will only be consulted for the CHECK constraint if we run against a database without a native BOOLEAN type like SQLite or MySQL.

The CHECK constraint may also make use of the `column_0_name` token, which works nicely with *SchemaType* since these constraints have only one column:

```
metadata = MetaData(
    naming_convention={"ck": "ck_%(table_name)s_%(column_0_name)s"}
)

Table('foo', metadata,
    Column('flag', Boolean())
)
```

The above schema will produce:

```
CREATE TABLE foo (
    flag BOOL,
    CONSTRAINT ck_foo_flag CHECK (flag IN (0, 1))
)
```

Changed in version 1.0: Constraint naming conventions that don't include `%(constraint_name)s` again work with *SchemaType* constraints.

Constraints API

class `sqlalchemy.schema.`**`Constraint`**(*name=None*, *deferrable=None*, *initially=None*, *_create_rule=None*, *info=None*, *_type_bound=False*, ***dialect_kw*)

Bases: *sqlalchemy.sql.base.DialectKWArgs*, *sqlalchemy.schema.SchemaItem*

A table-level SQL constraint.

`__init__`(*name=None*, *deferrable=None*, *initially=None*, *_create_rule=None*, *info=None*, *_type_bound=False*, ***dialect_kw*)

Create a SQL constraint.

Parameters

- **`name`** – Optional, the in-database name of this `Constraint`.
- **`deferrable`** – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint.
- **`initially`** – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint.
- **`info`** – Optional data dictionary which will be populated into the *SchemaItem.info* attribute of this object.

  New in version 1.0.0.
- **`_create_rule`** – a callable which is passed the DDLCompiler object during compilation. Returns True or False to signal inline generation of this Constraint.

  The AddConstraint and DropConstraint DDL constructs provide DDLElement's more comprehensive "conditional DDL" approach that is passed a database connection when DDL is being issued. _create_rule is instead called during any CREATE TABLE compilation, where there may not be any transaction/connection in progress.

However, it allows conditional compilation of the constraint even for backends which do not support addition of constraints through ALTER TABLE, which currently includes SQLite.

_create_rule is used by some types to create constraints. Currently, its call signature is subject to change at any time.

- ****dialect_kw** – Additional keyword arguments are dialect specific, and passed in the form `<dialectname>_<argname>`. See the documentation regarding an individual dialect at *Dialects* for detail on documented arguments.

class `sqlalchemy.schema.`**`ColumnCollectionMixin`**(**columns*, ***kw*)

columns = None

A `ColumnCollection` of `Column` objects.

This collection represents the columns which are referred to by this object.

class `sqlalchemy.schema.`**`ColumnCollectionConstraint`**(**columns*, ***kw*)

Bases: `sqlalchemy.schema.ColumnCollectionMixin`, `sqlalchemy.schema.Constraint`

A constraint that proxies a ColumnCollection.

`__init__`(**columns*, ***kw*)

Parameters

- ***columns** – A sequence of column names or Column objects.
- **name** – Optional, the in-database name of this constraint.
- **deferrable** – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint.
- **initially** – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint.
- ****kw** – other keyword arguments including dialect-specific arguments are propagated to the `Constraint` superclass.

argument_for(*dialect_name*, *argument_name*, *default*)

inherited from the `argument_for()` method of `DialectKWArgs`

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that

it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.

- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

contains_column(*col*)

Return True if this constraint contains the given column.

Note that this object also contains an attribute `.columns` which is a `ColumnCollection` of `Column` objects.

dialect_kwargs

inherited from the `dialect_kwargs` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options

inherited from the `dialect_options` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

get_children(***kwargs*)

inherited from the `get_children()` *method of* `SchemaItem`

used to allow SchemaVisitor access

info

inherited from the `info` *attribute of* `SchemaItem`

Info dictionary associated with the object, allowing user-defined data to be associated with this `SchemaItem`.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as `Table` and `Column`.

kwargs

inherited from the `kwargs` *attribute of* `DialectKWArgs`

A synonym for `DialectKWArgs.dialect_kwargs`.

quote

inherited from the `quote` *attribute of* `SchemaItem`

Return the value of the `quote` flag passed to this schema object, for those schema items which have a `name` field.

Deprecated since version 0.9: Use `<obj>.name.quote`

class `sqlalchemy.schema.`**`CheckConstraint`**(*sqltext*, *name=None*, *deferrable=None*, *initially=None*, *table=None*, *info=None*, *_create_rule=None*, *_autoattach=True*, *_type_bound=False*)

Bases: `sqlalchemy.schema.ColumnCollectionConstraint`

A table- or column-level CHECK constraint.

Can be included in the definition of a Table or Column.

`__init__`(*sqltext*, *name=None*, *deferrable=None*, *initially=None*, *table=None*, *info=None*, *_create_rule=None*, *_autoattach=True*, *_type_bound=False*)

Construct a CHECK constraint.

Parameters

- **sqltext** – A string containing the constraint definition, which will be used verbatim, or a SQL expression construct. If given as a string, the object is converted to a `Text` object. If the textual string includes a colon character, escape this using a backslash:

```
CheckConstraint(r"foo ~ E'a(?\:b|c)d")
```

- **name** – Optional, the in-database name of the constraint.
- **deferrable** – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint.
- **initially** – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint.
- **info** – Optional data dictionary which will be populated into the `SchemaItem.info` attribute of this object.

  New in version 1.0.0.

argument_for(*dialect_name*, *argument_name*, *default*)

inherited from the `argument_for()` *method of* `DialectKWArgs`

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

contains_column(*col*)

inherited from the `contains_column()` *method of* `ColumnCollectionConstraint`

Return True if this constraint contains the given column.

Note that this object also contains an attribute `.columns` which is a `ColumnCollection` of `Column` objects.

dialect_kwargs

inherited from the `dialect_kwargs` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options

inherited from the `dialect_options` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

get_children(***kwargs*)

inherited from the `get_children()` *method of* `SchemaItem`

used to allow SchemaVisitor access

info

inherited from the `info` *attribute of* `SchemaItem`

Info dictionary associated with the object, allowing user-defined data to be associated with this `SchemaItem`.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as `Table` and `Column`.

kwargs

inherited from the `kwargs` *attribute of* `DialectKWArgs`

A synonym for `DialectKWArgs.dialect_kwargs`.

quote

inherited from the `quote` *attribute of* `SchemaItem`

Return the value of the `quote` flag passed to this schema object, for those schema items which have a `name` field.

Deprecated since version 0.9: Use `<obj>.name.quote`

class `sqlalchemy.schema.`**ForeignKey**(*column*, *_constraint=None*, *use_alter=False*, *name=None*, *onupdate=None*, *ondelete=None*, *deferrable=None*, *initially=None*, *link_to_name=False*, *match=None*, *info=None*, ***dialect_kw*)

Bases: `sqlalchemy.sql.base.DialectKWArgs`, `sqlalchemy.schema.SchemaItem`

Defines a dependency between two columns.

`ForeignKey` is specified as an argument to a `Column` object, e.g.:

```
t = Table("remote_table", metadata,
    Column("remote_id", ForeignKey("main_table.id"))
)
```

Note that `ForeignKey` is only a marker object that defines a dependency between two columns. The actual constraint is in all cases represented by the `ForeignKeyConstraint` object. This object will be generated automatically when a `ForeignKey` is associated with a `Column` which in turn is associated with a `Table`. Conversely, when `ForeignKeyConstraint` is applied to a `Table`, `ForeignKey` markers are automatically generated to be present on each associated `Column`, which are also associated with the constraint object.

Note that you cannot define a "composite" foreign key constraint, that is a constraint between a grouping of multiple parent/child columns, using `ForeignKey` objects. To define this grouping, the `ForeignKeyConstraint` object must be used, and applied to the `Table`. The associated `ForeignKey` objects are created automatically.

The `ForeignKey` objects associated with an individual `Column` object are available in the *foreign_keys* collection of that column.

Further examples of foreign key configuration are in *Defining Foreign Keys*.

__init__(*column*, *_constraint=None*, *use_alter=False*, *name=None*, *onupdate=None*, *ondelete=None*, *deferrable=None*, *initially=None*, *link_to_name=False*, *match=None*, *info=None*, ***dialect_kw*)

Construct a column-level FOREIGN KEY.

The `ForeignKey` object when constructed generates a `ForeignKeyConstraint` which is associated with the parent `Table` object's collection of constraints.

Parameters

- **column** – A single target column for the key relationship. A `Column` object or a column name as a string: `tablename.columnkey` or `schema.tablename.columnkey`. `columnkey` is the `key` which has been assigned to the column (defaults to the column name itself), unless `link_to_name` is `True` in which case the rendered name of the column is used.

  New in version 0.7.4: Note that if the schema name is not included, and the underlying `MetaData` has a "schema", that value will be used.

- **name** – Optional string. An in-database name for the key if *constraint* is not provided.
- **onupdate** – Optional string. If set, emit ON UPDATE <value> when issuing DDL for this constraint. Typical values include CASCADE, DELETE and RESTRICT.
- **ondelete** – Optional string. If set, emit ON DELETE <value> when issuing DDL for this constraint. Typical values include CASCADE, DELETE and RESTRICT.
- **deferrable** – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint.
- **initially** – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint.
- **link_to_name** – if True, the string name given in `column` is the rendered name of the referenced column, not its locally assigned `key`.
- **use_alter** – passed to the underlying `ForeignKeyConstraint` to indicate the constraint should be generated/dropped externally from the CREATE TABLE/ DROP TABLE statement. See `ForeignKeyConstraint.use_alter` for further description.

  See also:

  `ForeignKeyConstraint.use_alter`

  Creating/Dropping Foreign Key Constraints via ALTER

- **match** – Optional string. If set, emit MATCH <value> when issuing DDL for this constraint. Typical values include SIMPLE, PARTIAL and FULL.
- **info** – Optional data dictionary which will be populated into the `SchemaItem.info` attribute of this object.

  New in version 1.0.0.

- ****dialect_kw** – Additional keyword arguments are dialect specific, and passed in the form `<dialectname>_<argname>`. The arguments are ultimately handled by a corresponding `ForeignKeyConstraint`. See the documentation regarding an individual dialect at *Dialects* for detail on documented arguments.

  New in version 0.9.2.

argument_for(*dialect_name*, *argument_name*, *default*)

inherited from the `argument_for()` *method of* `DialectKWArgs`

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The *DialectKWArgs.argument_for()* method is a per-argument way adding extra arguments to the *DefaultDialect.construct_arguments* dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a *NoSuchModuleError* is raised. The dialect must also include an existing *DefaultDialect.construct_arguments* collection, indicating that it participates in the keyword-argument validation and default system, else *ArgumentError* is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

column

Return the target *Column* referenced by this *ForeignKey*.

If no target column has been established, an exception is raised.

Changed in version 0.9.0: Foreign key target column resolution now occurs as soon as both the ForeignKey object and the remote Column to which it refers are both associated with the same MetaData object.

copy(*schema=None*)

Produce a copy of this *ForeignKey* object.

The new *ForeignKey* will not be bound to any *Column*.

This method is usually used by the internal copy procedures of *Column*, *Table*, and *MetaData*.

Parameters **schema** – The returned *ForeignKey* will reference the original table and column name, qualified by the given string schema name.

dialect_kwargs

inherited from the *dialect_kwargs* *attribute of* *DialectKWArgs*

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the *DialectKWArgs.dialect_options* collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options

inherited from the `dialect_options` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

get_children(***kwargs*)

inherited from the `get_children()` *method of* `SchemaItem`

used to allow SchemaVisitor access

get_referent(*table*)

Return the `Column` in the given `Table` referenced by this `ForeignKey`.

Returns None if this `ForeignKey` does not reference the given `Table`.

info

inherited from the `info` *attribute of* `SchemaItem`

Info dictionary associated with the object, allowing user-defined data to be associated with this `SchemaItem`.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as `Table` and `Column`.

kwargs

inherited from the `kwargs` *attribute of* `DialectKWArgs`

A synonym for `DialectKWArgs.dialect_kwargs`.

quote

inherited from the `quote` *attribute of* `SchemaItem`

Return the value of the `quote` flag passed to this schema object, for those schema items which have a `name` field.

Deprecated since version 0.9: Use `<obj>.name.quote`

references(*table*)

Return True if the given `Table` is referenced by this `ForeignKey`.

target_fullname

Return a string based 'column specification' for this `ForeignKey`.

This is usually the equivalent of the string-based "tablename.colname" argument first passed to the object's constructor.

class sqlalchemy.schema.**ForeignKeyConstraint**(*columns*, *refcolumns*, *name=None*, *onupdate=None*, *ondelete=None*, *deferrable=None*, *initially=None*, *use_alter=False*, *link_to_name=False*, *match=None*, *table=None*, *info=None*, ***dialect_kw*)

Bases: *sqlalchemy.schema.ColumnCollectionConstraint*

A table-level FOREIGN KEY constraint.

Defines a single column or composite FOREIGN KEY ... REFERENCES constraint. For a no-frills, single column foreign key, adding a *ForeignKey* to the definition of a *Column* is a shorthand equivalent for an unnamed, single column *ForeignKeyConstraint*.

Examples of foreign key configuration are in *Defining Foreign Keys*.

__init__(*columns*, *refcolumns*, *name=None*, *onupdate=None*, *ondelete=None*, *deferrable=None*, *initially=None*, *use_alter=False*, *link_to_name=False*, *match=None*, *table=None*, *info=None*, ***dialect_kw*)

Construct a composite-capable FOREIGN KEY.

Parameters

- **columns** – A sequence of local column names. The named columns must be defined and present in the parent Table. The names should match the `key` given to each column (defaults to the name) unless `link_to_name` is True.
- **refcolumns** – A sequence of foreign column names or Column objects. The columns must all be located within the same Table.
- **name** – Optional, the in-database name of the key.
- **onupdate** – Optional string. If set, emit ON UPDATE <value> when issuing DDL for this constraint. Typical values include CASCADE, DELETE and RESTRICT.
- **ondelete** – Optional string. If set, emit ON DELETE <value> when issuing DDL for this constraint. Typical values include CASCADE, DELETE and RESTRICT.
- **deferrable** – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint.
- **initially** – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint.
- **link_to_name** – if True, the string name given in `column` is the rendered name of the referenced column, not its locally assigned `key`.
- **use_alter** – If True, do not emit the DDL for this constraint as part of the CREATE TABLE definition. Instead, generate it via an ALTER TABLE statement issued after the full collection of tables have been created, and drop it via an ALTER TABLE statement before the full collection of tables are dropped.

  The use of *ForeignKeyConstraint.use_alter* is particularly geared towards the case where two or more tables are established within a mutually-dependent foreign key constraint relationship; however, the *MetaData.create_all()* and *MetaData.drop_all()* methods will perform this resolution automatically, so the flag is normally not needed.

  Changed in version 1.0.0: Automatic resolution of foreign key cycles has been added, removing the need to use the *ForeignKeyConstraint.use_alter* in typical use cases.

  See also:

Creating/Dropping Foreign Key Constraints via ALTER

- **match** – Optional string. If set, emit MATCH <value> when issuing DDL for this constraint. Typical values include SIMPLE, PARTIAL and FULL.
- **info** – Optional data dictionary which will be populated into the `SchemaItem.info` attribute of this object.

  New in version 1.0.0.
- ****dialect_kw** – Additional keyword arguments are dialect specific, and passed in the form `<dialectname>_<argname>`. See the documentation regarding an individual dialect at *Dialects* for detail on documented arguments.

  New in version 0.9.2.

argument_for(*dialect_name*, *argument_name*, *default*)

inherited from the `argument_for()` *method of* `DialectKWArgs`

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

column_keys

Return a list of string keys representing the local columns in this `ForeignKeyConstraint`.

This list is either the original string arguments sent to the constructor of the `ForeignKeyConstraint`, or if the constraint has been initialized with `Column` objects, is the string .key of each element.

New in version 1.0.0.

contains_column(*col*)

inherited from the `contains_column()` *method of* `ColumnCollectionConstraint`

Return True if this constraint contains the given column.

Note that this object also contains an attribute `.columns` which is a *ColumnCollection* of *Column* objects.

dialect_kwargs

inherited from the dialect_kwargs attribute of DialectKWArgs

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the *DialectKWArgs.dialect_options* collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The *DialectKWArgs.dialect_kwargs* collection is now writable.

See also:

DialectKWArgs.dialect_options - nested dictionary form

dialect_options

inherited from the dialect_options attribute of DialectKWArgs

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

DialectKWArgs.dialect_kwargs - flat dictionary form

get_children(***kwargs*)

inherited from the get_children() method of SchemaItem

used to allow SchemaVisitor access

info

inherited from the info attribute of SchemaItem

Info dictionary associated with the object, allowing user-defined data to be associated with this *SchemaItem*.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as *Table* and *Column*.

kwargs

inherited from the kwargs attribute of DialectKWArgs

A synonym for *DialectKWArgs.dialect_kwargs*.

quote

inherited from the quote attribute of SchemaItem

Return the value of the `quote` flag passed to this schema object, for those schema items which have a `name` field.

Deprecated since version 0.9: Use `<obj>.name.quote`

referred_table

The *Table* object to which this *ForeignKeyConstraint* references.

This is a dynamically calculated attribute which may not be available if the constraint and/or parent table is not yet associated with a metadata collection that contains the referred table.

New in version 1.0.0.

class sqlalchemy.schema.**PrimaryKeyConstraint**(**columns*, ***kw*)

Bases: *sqlalchemy.schema.ColumnCollectionConstraint*

A table-level PRIMARY KEY constraint.

The *PrimaryKeyConstraint* object is present automatically on any *Table* object; it is assigned a set of *Column* objects corresponding to those marked with the *Column.primary_key* flag:

```
>>> my_table = Table('mytable', metadata,
...                  Column('id', Integer, primary_key=True),
...                  Column('version_id', Integer, primary_key=True),
...                  Column('data', String(50))
...      )
>>> my_table.primary_key
PrimaryKeyConstraint(
    Column('id', Integer(), table=<mytable>,
           primary_key=True, nullable=False),
    Column('version_id', Integer(), table=<mytable>,
           primary_key=True, nullable=False)
)
```

The primary key of a *Table* can also be specified by using a *PrimaryKeyConstraint* object explicitly; in this mode of usage, the "name" of the constraint can also be specified, as well as other options which may be recognized by dialects:

```
my_table = Table('mytable', metadata,
            Column('id', Integer),
            Column('version_id', Integer),
            Column('data', String(50)),
            PrimaryKeyConstraint('id', 'version_id',
                                 name='mytable_pk')
        )
```

The two styles of column-specification should generally not be mixed. An warning is emitted if the columns present in the *PrimaryKeyConstraint* don't match the columns that were marked as `primary_key=True`, if both are present; in this case, the columns are taken strictly from the *PrimaryKeyConstraint* declaration, and those columns otherwise marked as `primary_key=True` are ignored. This behavior is intended to be backwards compatible with previous behavior.

Changed in version 0.9.2: Using a mixture of columns within a *PrimaryKeyConstraint* in addition to columns marked as `primary_key=True` now emits a warning if the lists don't match. The ultimate behavior of ignoring those columns marked with the flag only is currently maintained for backwards compatibility; this warning may raise an exception in a future release.

For the use case where specific options are to be specified on the *PrimaryKeyConstraint*, but the usual style of using `primary_key=True` flags is still desirable, an empty *PrimaryKeyConstraint* may be specified, which will take on the primary key column collection from the *Table* based on the flags:

```
my_table = Table('mytable', metadata,
            Column('id', Integer, primary_key=True),
            Column('version_id', Integer, primary_key=True),
            Column('data', String(50)),
            PrimaryKeyConstraint(name='mytable_pk',
```

```
                                        mssql_clustered=True)
            )
```

New in version 0.9.2: an empty `PrimaryKeyConstraint` may now be specified for the purposes of establishing keyword arguments with the constraint, independently of the specification of "primary key" columns within the `Table` itself; columns marked as `primary_key=True` will be gathered into the empty constraint's column collection.

__init__(**columns*, ***kw*)

inherited from the `__init__()` method of `ColumnCollectionConstraint`

Parameters

- ***columns** – A sequence of column names or Column objects.
- **name** – Optional, the in-database name of this constraint.
- **deferrable** – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint.
- **initially** – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint.
- ****kw** – other keyword arguments including dialect-specific arguments are propagated to the `Constraint` superclass.

argument_for(*dialect_name*, *argument_name*, *default*)

inherited from the `argument_for()` method of `DialectKWArgs`

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

contains_column(*col*)

inherited from the `contains_column()` *method of* `ColumnCollectionConstraint`

Return True if this constraint contains the given column.

Note that this object also contains an attribute `.columns` which is a `ColumnCollection` of `Column` objects.

dialect_kwargs

inherited from the `dialect_kwargs` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options

inherited from the `dialect_options` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

get_children(***kwargs*)

inherited from the `get_children()` *method of* `SchemaItem`

used to allow SchemaVisitor access

info

inherited from the `info` *attribute of* `SchemaItem`

Info dictionary associated with the object, allowing user-defined data to be associated with this `SchemaItem`.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as `Table` and `Column`.

kwargs

inherited from the `kwargs` *attribute of* `DialectKWArgs`

A synonym for `DialectKWArgs.dialect_kwargs`.

quote

inherited from the `quote` *attribute of* `SchemaItem`

Return the value of the `quote` flag passed to this schema object, for those schema items which have a `name` field.

Deprecated since version 0.9: Use `<obj>.name.quote`

class `sqlalchemy.schema.`**`UniqueConstraint`**(**columns*, ***kw*)

Bases: *sqlalchemy.schema.ColumnCollectionConstraint*

A table-level UNIQUE constraint.

Defines a single column or composite UNIQUE constraint. For a no-frills, single column constraint, adding `unique=True` to the `Column` definition is a shorthand equivalent for an unnamed, single column UniqueConstraint.

`__init__`(**columns*, ***kw*)

inherited from the `__init__()` method of `ColumnCollectionConstraint`

Parameters

- **`*columns`** – A sequence of column names or Column objects.
- **`name`** – Optional, the in-database name of this constraint.
- **`deferrable`** – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint.
- **`initially`** – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint.
- **`**kw`** – other keyword arguments including dialect-specific arguments are propagated to the `Constraint` superclass.

`argument_for`(*dialect_name*, *argument_name*, *default*)

inherited from the `argument_for()` method of `DialectKWArgs`

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **`dialect_name`** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **`argument_name`** – name of the parameter.

- **default** – default value of the parameter.

New in version 0.9.4.

contains_column(*col*)

inherited from the `contains_column()` *method of* `ColumnCollectionConstraint`

Return True if this constraint contains the given column.

Note that this object also contains an attribute `.columns` which is a `ColumnCollection` of `Column` objects.

dialect_kwargs

inherited from the `dialect_kwargs` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options

inherited from the `dialect_options` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

`DialectKWArgs.dialect_kwargs` - flat dictionary form

get_children(***kwargs*)

inherited from the `get_children()` *method of* `SchemaItem`

used to allow SchemaVisitor access

info

inherited from the `info` *attribute of* `SchemaItem`

Info dictionary associated with the object, allowing user-defined data to be associated with this `SchemaItem`.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as `Table` and `Column`.

kwargs

inherited from the `kwargs` *attribute of* `DialectKWArgs`

A synonym for `DialectKWArgs.dialect_kwargs`.

quote

inherited from the `quote` *attribute of* `SchemaItem`

Return the value of the `quote` flag passed to this schema object, for those schema items which have a `name` field.

Deprecated since version 0.9: Use `<obj>.name.quote`

`sqlalchemy.schema.`**`conv`**(*cls*, *value*, *quote=None*)

Mark a string indicating that a name has already been converted by a naming convention.

This is a string subclass that indicates a name that should not be subject to any further naming conventions.

E.g. when we create a `Constraint` using a naming convention as follows:

```
m = MetaData(naming_convention={
    "ck": "ck_%(table_name)s_%(constraint_name)s"
})
t = Table('t', m, Column('x', Integer),
                  CheckConstraint('x > 5', name='x5'))
```

The name of the above constraint will be rendered as `"ck_t_x5"`. That is, the existing name `x5` is used in the naming convention as the `constraint_name` token.

In some situations, such as in migration scripts, we may be rendering the above `CheckConstraint` with a name that's already been converted. In order to make sure the name isn't double-modified, the new name is applied using the `schema.conv()` marker. We can use this explicitly as follows:

```
m = MetaData(naming_convention={
    "ck": "ck_%(table_name)s_%(constraint_name)s"
})
t = Table('t', m, Column('x', Integer),
                  CheckConstraint('x > 5', name=conv('ck_t_x5')))
```

Where above, the `schema.conv()` marker indicates that the constraint name here is final, and the name will render as `"ck_t_x5"` and not `"ck_t_ck_t_x5"`

New in version 0.9.4.

See also:

Configuring Constraint Naming Conventions

Indexes

Indexes can be created anonymously (using an auto-generated name `ix_<column label>`) for a single column using the inline `index` keyword on `Column`, which also modifies the usage of `unique` to apply the uniqueness to the index itself, instead of adding a separate UNIQUE constraint. For indexes with specific names or which encompass more than one column, use the `Index` construct, which requires a name.

Below we illustrate a `Table` with several `Index` objects associated. The DDL for "CREATE INDEX" is issued right after the create statements for the table:

```
meta = MetaData()
mytable = Table('mytable', meta,
    # an indexed column, with index "ix_mytable_col1"
    Column('col1', Integer, index=True),

    # a uniquely indexed column with index "ix_mytable_col2"
    Column('col2', Integer, index=True, unique=True),
```

```
    Column('col3', Integer),
    Column('col4', Integer),

    Column('col5', Integer),
    Column('col6', Integer),
    )

# place an index on col3, col4
Index('idx_col34', mytable.c.col3, mytable.c.col4)

# place a unique index on col5, col6
Index('myindex', mytable.c.col5, mytable.c.col6, unique=True)

mytable.create(engine)
CREATE TABLE mytable (
    col1 INTEGER,
    col2 INTEGER,
    col3 INTEGER,
    col4 INTEGER,
    col5 INTEGER,
    col6 INTEGER
)
CREATE INDEX ix_mytable_col1 ON mytable (col1)
CREATE UNIQUE INDEX ix_mytable_col2 ON mytable (col2)
CREATE UNIQUE INDEX myindex ON mytable (col5, col6)
CREATE INDEX idx_col34 ON mytable (col3, col4)
```

Note in the example above, the *Index* construct is created externally to the table which it corresponds, using *Column* objects directly. *Index* also supports "inline" definition inside the *Table*, using string names to identify columns:

```
meta = MetaData()
mytable = Table('mytable', meta,
    Column('col1', Integer),

    Column('col2', Integer),

    Column('col3', Integer),
    Column('col4', Integer),

    # place an index on col1, col2
    Index('idx_col12', 'col1', 'col2'),

    # place a unique index on col3, col4
    Index('idx_col34', 'col3', 'col4', unique=True)
)
```

New in version 0.7: Support of "inline" definition inside the *Table* for *Index*.

The *Index* object also supports its own `create()` method:

```
i = Index('someindex', mytable.c.col5)
i.create(engine)
CREATE INDEX someindex ON mytable (col5)
```

Functional Indexes

Index supports SQL and function expressions, as supported by the target backend. To create an index against a column using a descending value, the *ColumnElement.desc()* modifier may be used:

```
from sqlalchemy import Index

Index('someindex', mytable.c.somecol.desc())
```

Or with a backend that supports functional indexes such as Postgresql, a "case insensitive" index can be created using the `lower()` function:

```
from sqlalchemy import func, Index

Index('someindex', func.lower(mytable.c.somecol))
```

New in version 0.8: *Index* supports SQL expressions and functions as well as plain columns.

Index API

class `sqlalchemy.schema.`**`Index`**(*name*, **expressions*, ***kw*)

Bases: *sqlalchemy.sql.base.DialectKWArgs*, *sqlalchemy.schema.ColumnCollectionMixin*, *sqlalchemy.schema.SchemaItem*

A table-level INDEX.

Defines a composite (one or more column) INDEX.

E.g.:

```
sometable = Table("sometable", metadata,
                Column("name", String(50)),
                Column("address", String(100))
            )

Index("some_index", sometable.c.name)
```

For a no-frills, single column index, adding *Column* also supports `index=True`:

```
sometable = Table("sometable", metadata,
                Column("name", String(50), index=True)
            )
```

For a composite index, multiple columns can be specified:

```
Index("some_index", sometable.c.name, sometable.c.address)
```

Functional indexes are supported as well, typically by using the *func* construct in conjunction with table-bound *Column* objects:

```
Index("some_index", func.lower(sometable.c.name))
```

New in version 0.8: support for functional and expression-based indexes.

An *Index* can also be manually associated with a *Table*, either through inline declaration or using *Table.append_constraint()*. When this approach is used, the names of the indexed columns can be specified as strings:

```
Table("sometable", metadata,
                Column("name", String(50)),
                Column("address", String(100)),
                Index("some_index", "name", "address")
        )
```

To support functional or expression-based indexes in this form, the `text()` construct may be used:

```
from sqlalchemy import text

Table("sometable", metadata,
                Column("name", String(50)),
                Column("address", String(100)),
                Index("some_index", text("lower(name)"))
        )
```

New in version 0.9.5: the `text()` construct may be used to specify `Index` expressions, provided the `Index` is explicitly associated with the `Table`.

See also:

Indexes - General information on `Index`.

Postgresql-Specific Index Options - PostgreSQL-specific options available for the `Index` construct.

MySQL Specific Index Options - MySQL-specific options available for the `Index` construct.

Clustered Index Support - MSSQL-specific options available for the `Index` construct.

__init__(*name*, **expressions*, ***kw*)

Construct an index object.

Parameters

- **name** – The name of the index
- ***expressions** – Column expressions to include in the index. The expressions are normally instances of `Column`, but may also be arbitrary SQL expressions which ultimately refer to a `Column`.
- **unique=False** – Keyword only argument; if True, create a unique index.
- **quote=None** – Keyword only argument; whether to apply quoting to the name of the index. Works in the same manner as that of `Column.quote`.
- **info=None** – Optional data dictionary which will be populated into the `SchemaItem.info` attribute of this object.

  New in version 1.0.0.
- ****kw** – Additional keyword arguments not mentioned above are dialect specific, and passed in the form `<dialectname>_<argname>`. See the documentation regarding an individual dialect at *Dialects* for detail on documented arguments.

argument_for(*dialect_name*, *argument_name*, *default*)

inherited from the `argument_for()` method of `DialectKWArgs`

Add a new kind of dialect-specific keyword argument for this class.

E.g.:

```
Index.argument_for("mydialect", "length", None)

some_index = Index('a', 'b', mydialect_length=5)
```

The `DialectKWArgs.argument_for()` method is a per-argument way adding extra arguments to the `DefaultDialect.construct_arguments` dictionary. This dictionary provides a list of argument names accepted by various schema-level constructs on behalf of a dialect.

New dialects should typically specify this dictionary all at once as a data member of the dialect class. The use case for ad-hoc addition of argument names is typically for end-user code that is also using a custom compilation scheme which consumes the additional arguments.

Parameters

- **dialect_name** – name of a dialect. The dialect must be locatable, else a `NoSuchModuleError` is raised. The dialect must also include an existing `DefaultDialect.construct_arguments` collection, indicating that it participates in the keyword-argument validation and default system, else `ArgumentError` is raised. If the dialect does not include this collection, then any keyword argument can be specified on behalf of this dialect already. All dialects packaged within SQLAlchemy include this collection, however for third party dialects, support may vary.
- **argument_name** – name of the parameter.
- **default** – default value of the parameter.

New in version 0.9.4.

bind

Return the connectable associated with this Index.

create(*bind=None*)

Issue a `CREATE` statement for this `Index`, using the given `Connectable` for connectivity.

See also:

`MetaData.create_all()`.

dialect_kwargs

inherited from the `dialect_kwargs` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

The arguments are present here in their original `<dialect>_<kwarg>` format. Only arguments that were actually passed are included; unlike the `DialectKWArgs.dialect_options` collection, which contains all options known by this dialect including defaults.

The collection is also writable; keys are accepted of the form `<dialect>_<kwarg>` where the value will be assembled into the list of options.

New in version 0.9.2.

Changed in version 0.9.4: The `DialectKWArgs.dialect_kwargs` collection is now writable.

See also:

`DialectKWArgs.dialect_options` - nested dictionary form

dialect_options

inherited from the `dialect_options` *attribute of* `DialectKWArgs`

A collection of keyword arguments specified as dialect-specific options to this construct.

This is a two-level nested registry, keyed to `<dialect_name>` and `<argument_name>`. For example, the `postgresql_where` argument would be locatable as:

```
arg = my_object.dialect_options['postgresql']['where']
```

New in version 0.9.2.

See also:

DialectKWArgs.dialect_kwargs - flat dictionary form

drop(*bind=None*)

Issue a DROP statement for this *Index*, using the given *Connectable* for connectivity.

See also:

MetaData.drop_all().

get_children(***kwargs*)

inherited from the get_children() method of SchemaItem

used to allow SchemaVisitor access

info

inherited from the info attribute of SchemaItem

Info dictionary associated with the object, allowing user-defined data to be associated with this *SchemaItem*.

The dictionary is automatically generated when first accessed. It can also be specified in the constructor of some objects, such as *Table* and *Column*.

kwargs

inherited from the kwargs attribute of DialectKWArgs

A synonym for *DialectKWArgs.dialect_kwargs*.

quote

inherited from the quote attribute of SchemaItem

Return the value of the quote flag passed to this schema object, for those schema items which have a name field.

Deprecated since version 0.9: Use <obj>.name.quote

3.3.5 Customizing DDL

In the preceding sections we've discussed a variety of schema constructs including *Table*, *ForeignKeyConstraint*, *CheckConstraint*, and *Sequence*. Throughout, we've relied upon the create() and *create_all()* methods of *Table* and *MetaData* in order to issue data definition language (DDL) for all constructs. When issued, a pre-determined order of operations is invoked, and DDL to create each table is created unconditionally including all constraints and other objects associated with it. For more complex scenarios where database-specific DDL is required, SQLAlchemy offers two techniques which can be used to add any DDL based on any condition, either accompanying the standard generation of tables or by itself.

Custom DDL

Custom DDL phrases are most easily achieved using the *DDL* construct. This construct works like all the other DDL elements except it accepts a string which is the text to be emitted:

```
event.listen(
    metadata,
    "after_create",
    DDL("ALTER TABLE users ADD CONSTRAINT "
        "cst_user_name_length "
        " CHECK (length(user_name) >= 8)")
)
```

A more comprehensive method of creating libraries of DDL constructs is to use custom compilation - see *Custom SQL Constructs and Compilation Extension* for details.

Controlling DDL Sequences

The `DDL` construct introduced previously also has the ability to be invoked conditionally based on inspection of the database. This feature is available using the `DDLElement.execute_if()` method. For example, if we wanted to create a trigger but only on the Postgresql backend, we could invoke this as:

```
mytable = Table(
    'mytable', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', String(50))
)

trigger = DDL(
    "CREATE TRIGGER dt_ins BEFORE INSERT ON mytable "
    "FOR EACH ROW BEGIN SET NEW.data='ins'; END"
)

event.listen(
    mytable,
    'after_create',
    trigger.execute_if(dialect='postgresql')
)
```

The `DDLElement.execute_if.dialect` keyword also accepts a tuple of string dialect names:

```
event.listen(
    mytable,
    "after_create",
    trigger.execute_if(dialect=('postgresql', 'mysql'))
)
event.listen(
    mytable,
    "before_drop",
    trigger.execute_if(dialect=('postgresql', 'mysql'))
)
```

The `DDLElement.execute_if()` method can also work against a callable function that will receive the database connection in use. In the example below, we use this to conditionally create a CHECK constraint, first looking within the Postgresql catalogs to see if it exists:

```
def should_create(ddl, target, connection, **kw):
    row = connection.execute(
        "select conname from pg_constraint where conname='%s'" %
        ddl.element.name).scalar()
```

```
    return not bool(row)

def should_drop(ddl, target, connection, **kw):
    return not should_create(ddl, target, connection, **kw)

event.listen(
    users,
    "after_create",
    DDL(
        "ALTER TABLE users ADD CONSTRAINT "
        "cst_user_name_length CHECK (length(user_name) >= 8)"
    ).execute_if(callable_=should_create)
)
event.listen(
    users,
    "before_drop",
    DDL(
        "ALTER TABLE users DROP CONSTRAINT cst_user_name_length"
    ).execute_if(callable_=should_drop)
)

users.create(engine)
CREATE TABLE users (
    user_id SERIAL NOT NULL,
    user_name VARCHAR(40) NOT NULL,
    PRIMARY KEY (user_id)
)

select conname from pg_constraint where conname='cst_user_name_length'
ALTER TABLE users ADD CONSTRAINT cst_user_name_length  CHECK (length(user_name) >= 8)

users.drop(engine)
select conname from pg_constraint where conname='cst_user_name_length'
ALTER TABLE users DROP CONSTRAINT cst_user_name_length
DROP TABLE users
```

Using the built-in DDLElement Classes

The `sqlalchemy.schema` package contains SQL expression constructs that provide DDL expressions. For example, to produce a `CREATE TABLE` statement:

```
from sqlalchemy.schema import CreateTable
engine.execute(CreateTable(mytable))
CREATE TABLE mytable (
    col1 INTEGER,
    col2 INTEGER,
    col3 INTEGER,
    col4 INTEGER,
    col5 INTEGER,
    col6 INTEGER
)
```

Above, the *`CreateTable`* construct works like any other expression construct (such as `select()`, `table.insert()`, etc.). All of SQLAlchemy's DDL oriented constructs are subclasses of the *`DDLElement`* base class; this is the base of all the objects corresponding to CREATE and DROP as well as ALTER, not only

in SQLAlchemy but in Alembic Migrations as well. A full reference of available constructs is in *DDL Expression Constructs API*.

User-defined DDL constructs may also be created as subclasses of `DDLElement` itself. The documentation in *Custom SQL Constructs and Compilation Extension* has several examples of this.

The event-driven DDL system described in the previous section *Controlling DDL Sequences* is available with other `DDLElement` objects as well. However, when dealing with the built-in constructs such as `CreateIndex`, `CreateSequence`, etc, the event system is of **limited** use, as methods like `Table.create()` and `MetaData.create_all()` will invoke these constructs unconditionally. In a future SQLAlchemy release, the DDL event system including conditional execution will taken into account for built-in constructs that currently invoke in all cases.

We can illustrate an event-driven example with the `AddConstraint` and `DropConstraint` constructs, as the event-driven system will work for CHECK and UNIQUE constraints, using these as we did in our previous example of `DDLElement.execute_if()`:

```
def should_create(ddl, target, connection, **kw):
    row = connection.execute(
        "select conname from pg_constraint where conname='%s'" %
        ddl.element.name).scalar()
    return not bool(row)

def should_drop(ddl, target, connection, **kw):
    return not should_create(ddl, target, connection, **kw)

event.listen(
    users,
    "after_create",
    AddConstraint(constraint).execute_if(callable_=should_create)
)
event.listen(
    users,
    "before_drop",
    DropConstraint(constraint).execute_if(callable_=should_drop)
)

users.create(engine)
CREATE TABLE users (
    user_id SERIAL NOT NULL,
    user_name VARCHAR(40) NOT NULL,
    PRIMARY KEY (user_id)
)

select conname from pg_constraint where conname='cst_user_name_length'
ALTER TABLE users ADD CONSTRAINT cst_user_name_length  CHECK (length(user_name) >= 8)

users.drop(engine)
select conname from pg_constraint where conname='cst_user_name_length'
ALTER TABLE users DROP CONSTRAINT cst_user_name_length
DROP TABLE users
```

While the above example is against the built-in `AddConstraint` and `DropConstraint` objects, the main usefulness of DDL events for now remains focused on the use of the `DDL` construct itself, as well as with user-defined subclasses of `DDLElement` that aren't already part of the `MetaData.create_all()`, `Table.create()`, and corresponding "drop" processes.

DDL Expression Constructs API

`sqlalchemy.schema.sort_tables`(*tables*, *skip_fn=None*, *extra_dependencies=None*)

sort a collection of `Table` objects based on dependency.

This is a dependency-ordered sort which will emit `Table` objects such that they will follow their dependent `Table` objects. Tables are dependent on another based on the presence of `ForeignKeyConstraint` objects as well as explicit dependencies added by `Table.add_is_dependent_on()`.

> **Warning:** The `sort_tables()` function cannot by itself accommodate automatic resolution of dependency cycles between tables, which are usually caused by mutually dependent foreign key constraints. To resolve these cycles, either the `ForeignKeyConstraint.use_alter` parameter may be appled to those constraints, or use the `sql.sort_tables_and_constraints()` function which will break out foreign key constraints involved in cycles separately.

Parameters

- **tables** – a sequence of `Table` objects.
- **skip_fn** – optional callable which will be passed a `ForeignKey` object; if it returns True, this constraint will not be considered as a dependency. Note this is **different** from the same parameter in `sort_tables_and_constraints()`, which is instead passed the owning `ForeignKeyConstraint` object.
- **extra_dependencies** – a sequence of 2-tuples of tables which will also be considered as dependent on each other.

See also:

`sort_tables_and_constraints()`

`MetaData.sorted_tables()` - uses this function to sort

`sqlalchemy.schema.sort_tables_and_constraints`(*tables*, *filter_fn=None*, *extra_dependencies=None*)

sort a collection of `Table` / `ForeignKeyConstraint` objects.

This is a dependency-ordered sort which will emit tuples of `(Table, [ForeignKeyConstraint, ...])` such that each `Table` follows its dependent `Table` objects. Remaining `ForeignKeyConstraint` objects that are separate due to dependency rules not satisifed by the sort are emitted afterwards as `(None, [ForeignKeyConstraint ...])`.

Tables are dependent on another based on the presence of `ForeignKeyConstraint` objects, explicit dependencies added by `Table.add_is_dependent_on()`, as well as dependencies stated here using the `skip_fn` and/or `extra_dependencies` parameters.

Parameters

- **tables** – a sequence of `Table` objects.
- **filter_fn** – optional callable which will be passed a `ForeignKeyConstraint` object, and returns a value based on whether this constraint should definitely be included or excluded as an inline constraint, or neither. If it returns False, the constraint will definitely be included as a dependency that cannot be subject to ALTER; if True, it will **only** be included as an ALTER result at the end. Returning None means the constraint is included in the table-based result unless it is detected as part of a dependency cycle.
- **extra_dependencies** – a sequence of 2-tuples of tables which will also be considered as dependent on each other.

New in version 1.0.0.

See also:

sort_tables()

class sqlalchemy.schema.**DDLElement**

Bases: *sqlalchemy.sql.expression.Executable*, sqlalchemy.schema._DDLCompiles

Base class for DDL expression constructs.

This class is the base for the general purpose *DDL* class, as well as the various create/drop clause constructs such as *CreateTable*, *DropTable*, *AddConstraint*, etc.

DDLElement integrates closely with SQLAlchemy events, introduced in *Events*. An instance of one is itself an event receiving callable:

```
event.listen(
    users,
    'after_create',
    AddConstraint(constraint).execute_if(dialect='postgresql')
)
```

See also:

DDL

DDLEvents

Events

Controlling DDL Sequences

__call__(*target*, *bind*, ***kw*)

Execute the DDL as a ddl_listener.

against(*target*)

Return a copy of this DDL against a specific schema item.

bind

callable_ = None

dialect = None

execute(*bind=None*, *target=None*)

Execute this DDL immediately.

Executes the DDL statement in isolation using the supplied *Connectable* or *Connectable* assigned to the `.bind` property, if not supplied. If the DDL has a conditional `on` criteria, it will be invoked with None as the event.

Parameters

- **bind** – Optional, an `Engine` or `Connection`. If not supplied, a valid *Connectable* must be present in the `.bind` property.
- **target** – Optional, defaults to None. The target SchemaItem for the execute call. Will be passed to the `on` callable if any, and may also provide string expansion data for the statement. See `execute_at` for more information.

execute_at(*event_name*, *target*)

Link execution of this DDL to the DDL lifecycle of a SchemaItem.

Deprecated since version 0.7: See *DDLEvents*, as well as *DDLElement.execute_if()*.

Links this `DDLElement` to a `Table` or `MetaData` instance, executing it when that schema item is created or dropped. The DDL statement will be executed using the same Connection and transactional context as the Table create/drop itself. The `.bind` property of this statement is ignored.

Parameters

- **event** – One of the events defined in the schema item's `.ddl_events`; e.g. 'before-create', 'after-create', 'before-drop' or 'after-drop'
- **target** – The Table or MetaData instance for which this DDLElement will be associated with.

A DDLElement instance can be linked to any number of schema items.

`execute_at` builds on the `append_ddl_listener` interface of *`MetaData`* and *`Table`* objects.

Caveat: Creating or dropping a Table in isolation will also trigger any DDL set to `execute_at` that Table's MetaData. This may change in a future release.

execute_if(*dialect=None*, *callable_=None*, *state=None*)

Return a callable that will execute this DDLElement conditionally.

Used to provide a wrapper for event listening:

```
event.listen(
            metadata,
            'before_create',
            DDL("my_ddl").execute_if(dialect='postgresql')
        )
```

Parameters

- **dialect** – May be a string, tuple or a callable predicate. If a string, it will be compared to the name of the executing database dialect:

```
DDL('something').execute_if(dialect='postgresql')
```

If a tuple, specifies multiple dialect names:

```
DDL('something').execute_if(dialect=('postgresql', 'mysql'))
```

- **callable_** – A callable, which will be invoked with four positional arguments as well as optional keyword arguments:

  ddl This DDL element.

  target The *`Table`* or *`MetaData`* object which is the target of this event. May be None if the DDL is executed explicitly.

  bind The *`Connection`* being used for DDL execution

  tables Optional keyword argument - a list of Table objects which are to be created/ dropped within a MetaData.create_all() or drop_all() method call.

  state Optional keyword argument - will be the `state` argument passed to this function.

  checkfirst Keyword argument, will be True if the 'checkfirst' flag was set during the call to `create()`, `create_all()`, `drop()`, `drop_all()`.

If the callable returns a true value, the DDL statement will be executed.

- **state** – any value which will be passed to the callable_ as the `state` keyword argument.

See also:

DDLEvents

Events

on = None

target = None

class `sqlalchemy.schema.`**DDL**(*statement*, *on=None*, *context=None*, *bind=None*)

Bases: *sqlalchemy.schema.DDLElement*

A literal DDL statement.

Specifies literal SQL DDL to be executed by the database. DDL objects function as DDL event listeners, and can be subscribed to those events listed in *DDLEvents*, using either *Table* or *MetaData* objects as targets. Basic templating support allows a single DDL instance to handle repetitive tasks for multiple tables.

Examples:

```
from sqlalchemy import event, DDL

tbl = Table('users', metadata, Column('uid', Integer))
event.listen(tbl, 'before_create', DDL('DROP TRIGGER users_trigger'))

spow = DDL('ALTER TABLE %(table)s SET secretpowers TRUE')
event.listen(tbl, 'after_create', spow.execute_if(dialect='somedb'))

drop_spow = DDL('ALTER TABLE users SET secretpowers FALSE')
connection.execute(drop_spow)
```

When operating on Table events, the following `statement` string substitions are available:

```
%(table)s  - the Table name, with any required quoting applied
%(schema)s - the schema name, with any required quoting applied
%(fullname)s - the Table name including schema, quoted if needed
```

The DDL's "context", if any, will be combined with the standard substitutions noted above. Keys present in the context will override the standard substitutions.

__init__(*statement*, *on=None*, *context=None*, *bind=None*)

Create a DDL statement.

Parameters

- **statement** – A string or unicode string to be executed. Statements will be processed with Python's string formatting operator. See the `context` argument and the `execute_at` method.

  A literal '%' in a statement must be escaped as '%%'.

  SQL bind parameters are not available in DDL statements.

- **on** – Deprecated since version 0.7: See *DDLElement.execute_if()*.

  Optional filtering criteria. May be a string, tuple or a callable predicate. If a string, it will be compared to the name of the executing database dialect:

```
DDL('something', on='postgresql')
```

If a tuple, specifies multiple dialect names:

```
DDL('something', on=('postgresql', 'mysql'))
```

If a callable, it will be invoked with four positional arguments as well as optional keyword arguments:

ddl This DDL element.

event The name of the event that has triggered this DDL, such as 'after-create' Will be None if the DDL is executed explicitly.

target The `Table` or `MetaData` object which is the target of this event. May be None if the DDL is executed explicitly.

connection The `Connection` being used for DDL execution

tables Optional keyword argument - a list of Table objects which are to be created/ dropped within a MetaData.create_all() or drop_all() method call.

If the callable returns a true value, the DDL statement will be executed.

- **context** – Optional dictionary, defaults to None. These values will be available for use in string substitutions on the DDL statement.
- **bind** – Optional. A *`Connectable`*, used by default when `execute()` is invoked without a bind argument.

See also:

`DDLEvents`

Events

class `sqlalchemy.schema.`**`_CreateDropBase`**(*element*, *on=None*, *bind=None*)

Bases: *`sqlalchemy.schema.DDLElement`*

Base class for DDL constructs that represent CREATE and DROP or equivalents.

The common theme of _CreateDropBase is a single `element` attribute which refers to the element to be created or dropped.

class `sqlalchemy.schema.`**`CreateTable`**(*element*, *on=None*, *bind=None*, *include_foreign_key_constraints=None*)

Bases: *`sqlalchemy.schema._CreateDropBase`*

Represent a CREATE TABLE statement.

`__init__`(*element*, *on=None*, *bind=None*, *include_foreign_key_constraints=None*)

Create a *`CreateTable`* construct.

Parameters

- **element** – a *`Table`* that's the subject of the CREATE
- **on** – See the description for 'on' in *`DDL`*.
- **bind** – See the description for 'bind' in *`DDL`*.

- **include_foreign_key_constraints** – optional sequence of *ForeignKeyConstraint* objects that will be included inline within the CREATE construct; if omitted, all foreign key constraints that do not specify use_alter=True are included.

  New in version 1.0.0.

class sqlalchemy.schema.**DropTable**(*element*, *on=None*, *bind=None*)

Bases: *sqlalchemy.schema._CreateDropBase*

Represent a DROP TABLE statement.

class sqlalchemy.schema.**CreateColumn**(*element*)

Bases: sqlalchemy.schema._DDLCompiles

Represent a *Column* as rendered in a CREATE TABLE statement, via the *CreateTable* construct.

This is provided to support custom column DDL within the generation of CREATE TABLE statements, by using the compiler extension documented in *Custom SQL Constructs and Compilation Extension* to extend *CreateColumn*.

Typical integration is to examine the incoming *Column* object, and to redirect compilation if a particular flag or condition is found:

```
from sqlalchemy import schema
from sqlalchemy.ext.compiler import compiles

@compiles(schema.CreateColumn)
def compile(element, compiler, **kw):
    column = element.element

    if "special" not in column.info:
        return compiler.visit_create_column(element, **kw)

    text = "%s SPECIAL DIRECTIVE %s" % (
            column.name,
            compiler.type_compiler.process(column.type)
        )
    default = compiler.get_column_default_string(column)
    if default is not None:
        text += " DEFAULT " + default

    if not column.nullable:
        text += " NOT NULL"

    if column.constraints:
        text += " ".join(
                    compiler.process(const)
                    for const in column.constraints)
    return text
```

The above construct can be applied to a *Table* as follows:

```
from sqlalchemy import Table, Metadata, Column, Integer, String
from sqlalchemy import schema

metadata = MetaData()

table = Table('mytable', MetaData(),
        Column('x', Integer, info={"special":True}, primary_key=True),
```

```
            Column('y', String(50)),
            Column('z', String(20), info={"special":True})
        )

    metadata.create_all(conn)
```

Above, the directives we've added to the *Column.info* collection will be detected by our custom compilation scheme:

```
CREATE TABLE mytable (
        x SPECIAL DIRECTIVE INTEGER NOT NULL,
        y VARCHAR(50),
        z SPECIAL DIRECTIVE VARCHAR(20),
    PRIMARY KEY (x)
)
```

The *CreateColumn* construct can also be used to skip certain columns when producing a `CREATE TABLE`. This is accomplished by creating a compilation rule that conditionally returns `None`. This is essentially how to produce the same effect as using the `system=True` argument on *Column*, which marks a column as an implicitly-present "system" column.

For example, suppose we wish to produce a *Table* which skips rendering of the Postgresql `xmin` column against the Postgresql backend, but on other backends does render it, in anticipation of a triggered rule. A conditional compilation rule could skip this name only on Postgresql:

```
from sqlalchemy.schema import CreateColumn

@compiles(CreateColumn, "postgresql")
def skip_xmin(element, compiler, **kw):
    if element.element.name == 'xmin':
        return None
    else:
        return compiler.visit_create_column(element, **kw)


my_table = Table('mytable', metadata,
            Column('id', Integer, primary_key=True),
            Column('xmin', Integer)
        )
```

Above, a *CreateTable* construct will generate a `CREATE TABLE` which only includes the `id` column in the string; the `xmin` column will be omitted, but only against the Postgresql backend.

New in version 0.8.3: The *CreateColumn* construct supports skipping of columns by returning `None` from a custom compilation rule.

New in version 0.8: The *CreateColumn* construct was added to support custom column creation styles.

class `sqlalchemy.schema.`**`CreateSequence`**(*element*, *on=None*, *bind=None*)

Bases: *sqlalchemy.schema._CreateDropBase*

Represent a CREATE SEQUENCE statement.

class `sqlalchemy.schema.`**`DropSequence`**(*element*, *on=None*, *bind=None*)

Bases: *sqlalchemy.schema._CreateDropBase*

Represent a DROP SEQUENCE statement.

class sqlalchemy.schema.**CreateIndex**(*element*, *on=None*, *bind=None*)
Bases: *sqlalchemy.schema._CreateDropBase*

Represent a CREATE INDEX statement.

class sqlalchemy.schema.**DropIndex**(*element*, *on=None*, *bind=None*)
Bases: *sqlalchemy.schema._CreateDropBase*

Represent a DROP INDEX statement.

class sqlalchemy.schema.**AddConstraint**(*element*, **args*, ***kw*)
Bases: *sqlalchemy.schema._CreateDropBase*

Represent an ALTER TABLE ADD CONSTRAINT statement.

class sqlalchemy.schema.**DropConstraint**(*element*, *cascade=False*, ***kw*)
Bases: *sqlalchemy.schema._CreateDropBase*

Represent an ALTER TABLE DROP CONSTRAINT statement.

class sqlalchemy.schema.**CreateSchema**(*name*, *quote=None*, ***kw*)
Bases: *sqlalchemy.schema._CreateDropBase*

Represent a CREATE SCHEMA statement.

New in version 0.7.4.

The argument here is the string name of the schema.

__init__(*name*, *quote=None*, ***kw*)
Create a new *CreateSchema* construct.

class sqlalchemy.schema.**DropSchema**(*name*, *quote=None*, *cascade=False*, ***kw*)
Bases: *sqlalchemy.schema._CreateDropBase*

Represent a DROP SCHEMA statement.

The argument here is the string name of the schema.

New in version 0.7.4.

__init__(*name*, *quote=None*, *cascade=False*, ***kw*)
Create a new *DropSchema* construct.

3.4 Column and Data Types

3.4.1 Column and Data Types

SQLAlchemy provides abstractions for most common database data types, and a mechanism for specifying your own custom data types.

The methods and attributes of type objects are rarely used directly. Type objects are supplied to *Table* definitions and can be supplied as type hints to *functions* for occasions where the database driver returns an incorrect type.

```
>>> users = Table('users', metadata,
...               Column('id', Integer, primary_key=True)
...               Column('login', String(32))
...              )
```

SQLAlchemy will use the `Integer` and `String(32)` type information when issuing a `CREATE TABLE` statement and will use it again when reading back rows `SELECTed` from the database. Functions that accept a type (such as *`Column()`*) will typically accept a type class or instance; `Integer` is equivalent to `Integer()` with no construction arguments in this case.

Generic Types

Generic types specify a column that can read, write and store a particular type of Python data. SQLAlchemy will choose the best database column type available on the target database when issuing a `CREATE TABLE` statement. For complete control over which column type is emitted in `CREATE TABLE`, such as `VARCHAR` see **'SQL Standard Types'_** and the other sections of this chapter.

class `sqlalchemy.types.`**`BigInteger`**

Bases: *`sqlalchemy.types.Integer`*

A type for bigger `int` integers.

Typically generates a `BIGINT` in DDL, and otherwise acts like a normal *`Integer`* on the Python side.

class `sqlalchemy.types.`**`Boolean`**(*create_constraint=True*, *name=None*, *_create_events=True*)

Bases: *`sqlalchemy.types.TypeEngine`*, *`sqlalchemy.types.SchemaType`*

A bool datatype.

Boolean typically uses BOOLEAN or SMALLINT on the DDL side, and on the Python side deals in `True` or `False`.

`__init__`(*create_constraint=True*, *name=None*, *_create_events=True*)

Construct a Boolean.

Parameters

- **`create_constraint`** – defaults to True. If the boolean is generated as an int/smallint, also create a CHECK constraint on the table that ensures 1 or 0 as a value.
- **`name`** – if a CHECK constraint is generated, specify the name of the constraint.

class `sqlalchemy.types.`**`Date`**

Bases: `sqlalchemy.types._DateAffinity`, *`sqlalchemy.types.TypeEngine`*

A type for `datetime.date()` objects.

class `sqlalchemy.types.`**`DateTime`**(*timezone=False*)

Bases: `sqlalchemy.types._DateAffinity`, *`sqlalchemy.types.TypeEngine`*

A type for `datetime.datetime()` objects.

Date and time types return objects from the Python `datetime` module. Most DBAPIs have built in support for the datetime module, with the noted exception of SQLite. In the case of SQLite, date and time types are stored as strings which are then converted back to datetime objects when rows are returned.

`__init__`(*timezone=False*)

Construct a new *`DateTime`*.

Parameters **`timezone`** – boolean. If True, and supported by the backend, will produce 'TIMESTAMP WITH TIMEZONE'. For backends that don't support timezone aware timestamps, has no effect.

class `sqlalchemy.types.`**`Enum`**(**enums*, ***kw*)

Bases: *`sqlalchemy.types.String`*, *`sqlalchemy.types.SchemaType`*

Generic Enum Type.

The Enum type provides a set of possible string values which the column is constrained towards.

By default, uses the backend's native ENUM type if available, else uses VARCHAR + a CHECK constraint.

See also:

ENUM - PostgreSQL-specific type, which has additional functionality.

__init__(**enums*, ***kw*)

Construct an enum.

Keyword arguments which don't apply to a specific backend are ignored by that backend.

Parameters

- ***enums** – string or unicode enumeration labels. If unicode labels are present, the *convert_unicode* flag is auto-enabled.
- **convert_unicode** – Enable unicode-aware bind parameter and result-set processing for this Enum's data. This is set automatically based on the presence of unicode label strings.
- **metadata** – Associate this type directly with a `MetaData` object. For types that exist on the target database as an independent schema construct (Postgresql), this type will be created and dropped within `create_all()` and `drop_all()` operations. If the type is not associated with any `MetaData` object, it will associate itself with each `Table` in which it is used, and will be created when any of those individual tables are created, after a check is performed for its existence. The type is only dropped when `drop_all()` is called for that `Table` object's metadata, however.
- **name** – The name of this type. This is required for Postgresql and any future supported database which requires an explicitly named type, or an explicitly named constraint in order to generate the type and/or a table that uses it.
- **native_enum** – Use the database's native ENUM type when available. Defaults to True. When False, uses VARCHAR + check constraint for all backends.
- **schema** – Schema name of this type. For types that exist on the target database as an independent schema construct (Postgresql), this parameter specifies the named schema in which the type is present.

  Note: The `schema` of the *Enum* type does not by default make use of the `schema` established on the owning *Table*. If this behavior is desired, set the `inherit_schema` flag to `True`.

- **quote** – Set explicit quoting preferences for the type's name.
- **inherit_schema** – When `True`, the "schema" from the owning *Table* will be copied to the "schema" attribute of this *Enum*, replacing whatever value was passed for the `schema` attribute. This also takes effect when using the *Table.tometadata()* operation.

  New in version 0.8.

create(*bind=None*, *checkfirst=False*)

Issue CREATE ddl for this type, if applicable.

drop(*bind=None*, *checkfirst=False*)

Issue DROP ddl for this type, if applicable.

class sqlalchemy.types.**Float**(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*, ***kwargs*)

Bases: *sqlalchemy.types.Numeric*

Type representing floating point types, such as FLOAT or REAL.

This type returns Python float objects by default, unless the *Float.asdecimal* flag is set to True, in which case they are coerced to decimal.Decimal objects.

Note: The *Float* type is designed to receive data from a database type that is explicitly known to be a floating point type (e.g. FLOAT, REAL, others) and not a decimal type (e.g. DECIMAL, NUMERIC, others). If the database column on the server is in fact a Numeric type, such as DECIMAL or NUMERIC, use the *Numeric* type or a subclass, otherwise numeric coercion between float/Decimal may or may not function as expected.

__init__(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*, ***kwargs*)

Construct a Float.

Parameters

- **precision** – the numeric precision for use in DDL CREATE TABLE.
- **asdecimal** – the same flag as that of *Numeric*, but defaults to False. Note that setting this flag to True results in floating point conversion.
- **decimal_return_scale** – Default scale to use when converting from floats to Python decimals. Floating point values will typically be much longer due to decimal inaccuracy, and most floating point database types don't have a notion of "scale", so by default the float type looks for the first ten decimal places when converting. Specfiying this value will override that length. Note that the MySQL float types, which do include "scale", will use "scale" as the default for decimal_return_scale, if not otherwise specified.

  New in version 0.9.0.
- ****kwargs** – deprecated. Additional arguments here are ignored by the default *Float* type. For database specific floats that support additional arguments, see that dialect's documentation for details, such as *sqlalchemy.dialects.mysql.FLOAT*.

class sqlalchemy.types.**Integer**

Bases: sqlalchemy.types._DateAffinity, *sqlalchemy.types.TypeEngine*

A type for int integers.

class sqlalchemy.types.**Interval**(*native=True*, *second_precision=None*, *day_precision=None*)

Bases: sqlalchemy.types._DateAffinity, *sqlalchemy.types.TypeDecorator*

A type for datetime.timedelta() objects.

The Interval type deals with datetime.timedelta objects. In PostgreSQL, the native INTERVAL type is used; for others, the value is stored as a date which is relative to the "epoch" (Jan. 1, 1970).

Note that the Interval type does not currently provide date arithmetic operations on platforms which do not support interval types natively. Such operations usually require transformation of both sides of the expression (such as, conversion of both sides into integer epoch values first) which currently is a manual procedure (such as via *func*).

__init__(*native=True*, *second_precision=None*, *day_precision=None*)

Construct an Interval object.

Parameters

- **native** – when True, use the actual INTERVAL type provided by the database, if supported (currently Postgresql, Oracle). Otherwise, represent the interval data as an epoch value regardless.
- **second_precision** – For native interval types which support a "fractional seconds precision" parameter, i.e. Oracle and Postgresql
- **day_precision** – for native interval types which support a "day precision" parameter, i.e. Oracle.

coerce_compared_value(*op*, *value*)
See *TypeEngine.coerce_compared_value()* for a description.

impl
alias of *DateTime*

class sqlalchemy.types.**LargeBinary**(*length=None*)
Bases: sqlalchemy.types._Binary

A type for large binary byte data.

The Binary type generates BLOB or BYTEA when tables are created, and also converts incoming values using the Binary callable provided by each DB-API.

__init__(*length=None*)
Construct a LargeBinary type.

Parameters **length** – optional, a length for the column for use in DDL statements, for those BLOB types that accept a length (i.e. MySQL). It does *not* produce a *lengthed* BINARY/VARBINARY type - use the BINARY/VARBINARY types specifically for those. May be safely omitted if no CREATE TABLE will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the CREATE TABLE DDL is issued.

class sqlalchemy.types.**MatchType**(*create_constraint=True*, *name=None*, *_create_events=True*)
Bases: *sqlalchemy.types.Boolean*

Refers to the return type of the MATCH operator.

As the *ColumnOperators.match()* is probably the most open-ended operator in generic SQLAlchemy Core, we can't assume the return type at SQL evaluation time, as MySQL returns a floating point, not a boolean, and other backends might do something different. So this type acts as a placeholder, currently subclassing *Boolean*. The type allows dialects to inject result-processing functionality if needed, and on MySQL will return floating-point values.

New in version 1.0.0.

class sqlalchemy.types.**Numeric**(*precision=None*, *scale=None*, *decimal_return_scale=None*, *asdecimal=True*)
Bases: sqlalchemy.types._DateAffinity, *sqlalchemy.types.TypeEngine*

A type for fixed precision numbers, such as NUMERIC or DECIMAL.

This type returns Python decimal.Decimal objects by default, unless the *Numeric.asdecimal* flag is set to False, in which case they are coerced to Python float objects.

Note: The *Numeric* type is designed to receive data from a database type that is explicitly known to be a decimal type (e.g. DECIMAL, NUMERIC, others) and not a floating point type (e.g. FLOAT, REAL, others). If the database column on the server is in fact a floating-point type type, such as FLOAT or REAL, use the *Float* type or a subclass, otherwise numeric coercion between float/Decimal may or may not function as expected.

Note: The Python `decimal.Decimal` class is generally slow performing; cPython 3.3 has now switched to use the cdecimal library natively. For older Python versions, the `cdecimal` library can be patched into any application where it will replace the `decimal` library fully, however this needs to be applied globally and before any other modules have been imported, as follows:

```
import sys
import cdecimal
sys.modules["decimal"] = cdecimal
```

Note that the `cdecimal` and `decimal` libraries are **not compatible with each other**, so patching `cdecimal` at the global level is the only way it can be used effectively with various DBAPIs that hardcode to import the `decimal` library.

__init__(*precision=None*, *scale=None*, *decimal_return_scale=None*, *asdecimal=True*)

Construct a Numeric.

Parameters

- **precision** – the numeric precision for use in DDL `CREATE TABLE`.
- **scale** – the numeric scale for use in DDL `CREATE TABLE`.
- **asdecimal** – default True. Return whether or not values should be sent as Python Decimal objects, or as floats. Different DBAPIs send one or the other based on datatypes - the Numeric type will ensure that return values are one or the other across DBAPIs consistently.
- **decimal_return_scale** – Default scale to use when converting from floats to Python decimals. Floating point values will typically be much longer due to decimal inaccuracy, and most floating point database types don't have a notion of "scale", so by default the float type looks for the first ten decimal places when converting. Specfiying this value will override that length. Types which do include an explicit ".scale" value, such as the base *Numeric* as well as the MySQL float types, will use the value of ".scale" as the default for decimal_return_scale, if not otherwise specified.

  New in version 0.9.0.

When using the `Numeric` type, care should be taken to ensure that the asdecimal setting is apppropriate for the DBAPI in use - when Numeric applies a conversion from Decimal->float or float-> Decimal, this conversion incurs an additional performance overhead for all result columns received.

DBAPIs that return Decimal natively (e.g. psycopg2) will have better accuracy and higher performance with a setting of `True`, as the native translation to Decimal reduces the amount of floating- point issues at play, and the Numeric type itself doesn't need to apply any further conversions. However, another DBAPI which returns floats natively *will* incur an additional conversion overhead, and is still subject to floating point data loss - in which case `asdecimal=False` will at least remove the extra conversion overhead.

class `sqlalchemy.types.`**PickleType**(*protocol=2*, *pickler=None*, *comparator=None*)

Bases: *sqlalchemy.types.TypeDecorator*

Holds Python objects, which are serialized using pickle.

PickleType builds upon the Binary type to apply Python's `pickle.dumps()` to incoming objects, and `pickle.loads()` on the way out, allowing any pickleable Python object to be stored as a serialized binary field.

To allow ORM change events to propagate for elements associated with *PickleType*, see *Mutation Tracking*.

__init__(*protocol=2*, *pickler=None*, *comparator=None*)
Construct a PickleType.

Parameters

- **protocol** – defaults to `pickle.HIGHEST_PROTOCOL`.
- **pickler** – defaults to cPickle.pickle or pickle.pickle if cPickle is not available. May be any object with pickle-compatible `dumps` and `loads` methods.
- **comparator** – a 2-arg callable predicate used to compare values of this type. If left as `None`, the Python "equals" operator is used to compare values.

impl
alias of *LargeBinary*

class sqlalchemy.types.**SchemaType**(*name=None*, *schema=None*, *metadata=None*, *inherit_schema=False*, *quote=None*, *_create_events=True*)
Bases: sqlalchemy.sql.expression.SchemaEventTarget

Mark a type as possibly requiring schema-level DDL for usage.

Supports types that must be explicitly created/dropped (i.e. PG ENUM type) as well as types that are complimented by table or schema level constraints, triggers, and other rules.

SchemaType classes can also be targets for the *DDLEvents.before_parent_attach()* and *DDLEvents.after_parent_attach()* events, where the events fire off surrounding the association of the type object with a parent *Column*.

See also:

Enum

Boolean

adapt(*impltype*, ***kw*)

bind

copy(***kw*)

create(*bind=None*, *checkfirst=False*)
Issue CREATE ddl for this type, if applicable.

drop(*bind=None*, *checkfirst=False*)
Issue DROP ddl for this type, if applicable.

class sqlalchemy.types.**SmallInteger**
Bases: *sqlalchemy.types.Integer*

A type for smaller `int` integers.

Typically generates a `SMALLINT` in DDL, and otherwise acts like a normal *Integer* on the Python side.

class sqlalchemy.types.**String**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)
Bases: *sqlalchemy.types.Concatenable*, *sqlalchemy.types.TypeEngine*

The base for all string and character types.

In SQL, corresponds to VARCHAR. Can also take Python unicode objects and encode to the database's encoding in bind params (and the reverse for result sets.)

The *length* field is usually required when the *String* type is used within a CREATE TABLE statement, as VARCHAR requires a length on most databases.

__init__(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

Create a string-holding type.

Parameters

- **length** – optional, a length for the column for use in DDL and CAST expressions. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a `length` for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued if a `VARCHAR` with no length is included. Whether the value is interpreted as bytes or characters is database specific.
- **collation** – Optional, a column-level collation for use in DDL and CAST expressions. Renders using the COLLATE keyword supported by SQLite, MySQL, and Postgresql. E.g.:

```
>>> from sqlalchemy import cast, select, String
>>> print select([cast('some string', String(collation='utf8'))])
SELECT CAST(:param_1 AS VARCHAR COLLATE utf8) AS anon_1
```

  New in version 0.8: Added support for COLLATE to all string types.

- **convert_unicode** – When set to `True`, the *String* type will assume that input is to be passed as Python `unicode` objects, and results returned as Python `unicode` objects. If the DBAPI in use does not support Python unicode (which is fewer and fewer these days), SQLAlchemy will encode/decode the value, using the value of the `encoding` parameter passed to *create_engine()* as the encoding.

  When using a DBAPI that natively supports Python unicode objects, this flag generally does not need to be set. For columns that are explicitly intended to store non-ASCII data, the *Unicode* or *UnicodeText* types should be used regardless, which feature the same behavior of `convert_unicode` but also indicate an underlying column type that directly supports unicode, such as `NVARCHAR`.

  For the extremely rare case that Python `unicode` is to be encoded/decoded by SQLAlchemy on a backend that does natively support Python `unicode`, the value `force` can be passed here which will cause SQLAlchemy's encode/decode services to be used unconditionally.

- **unicode_error** – Optional, a method to use to handle Unicode conversion errors. Behaves like the `errors` keyword argument to the standard library's `string.decode()` functions. This flag requires that `convert_unicode` is set to `force` - otherwise, SQLAlchemy is not guaranteed to handle the task of unicode conversion. Note that this flag adds significant performance overhead to row-fetching operations for backends that already return unicode objects natively (which most DBAPIs do). This flag should only be used as a last resort for reading strings from a column with varied or corrupted encodings.

class `sqlalchemy.types.`**Text**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

Bases: *sqlalchemy.types.String*

A variably sized string type.

In SQL, usually corresponds to CLOB or TEXT. Can also take Python unicode objects and encode to the database's encoding in bind params (and the reverse for result sets.) In general, TEXT objects do not have a length; while some databases will accept a length argument here, it will be rejected by others.

class sqlalchemy.types.**Time**(*timezone=False*)

Bases: sqlalchemy.types._DateAffinity, *sqlalchemy.types.TypeEngine*

A type for datetime.time() objects.

class sqlalchemy.types.**Unicode**(*length=None*, ***kwargs*)

Bases: *sqlalchemy.types.String*

A variable length Unicode string type.

The *Unicode* type is a *String* subclass that assumes input and output as Python unicode data, and in that regard is equivalent to the usage of the convert_unicode flag with the *String* type. However, unlike plain *String*, it also implies an underlying column type that is explicitly supporting of non-ASCII data, such as NVARCHAR on Oracle and SQL Server. This can impact the output of CREATE TABLE statements and CAST functions at the dialect level, and can also affect the handling of bound parameters in some specific DBAPI scenarios.

The encoding used by the *Unicode* type is usually determined by the DBAPI itself; most modern DBAPIs feature support for Python unicode objects as bound values and result set values, and the encoding should be configured as detailed in the notes for the target DBAPI in the *Dialects* section.

For those DBAPIs which do not support, or are not configured to accommodate Python unicode objects directly, SQLAlchemy does the encoding and decoding outside of the DBAPI. The encoding in this scenario is determined by the encoding flag passed to *create_engine()*.

When using the *Unicode* type, it is only appropriate to pass Python unicode objects, and not plain str. If a plain str is passed under Python 2, a warning is emitted. If you notice your application emitting these warnings but you're not sure of the source of them, the Python warnings filter, documented at http://docs.python.org/library/warnings.html, can be used to turn these warnings into exceptions which will illustrate a stack trace:

```
import warnings
warnings.simplefilter('error')
```

For an application that wishes to pass plain bytestrings and Python unicode objects to the Unicode type equally, the bytestrings must first be decoded into unicode. The recipe at *Coercing Encoded Strings to Unicode* illustrates how this is done.

See also:

UnicodeText - unlengthed textual counterpart to *Unicode*.

__init__(*length=None*, ***kwargs*)

Create a *Unicode* object.

Parameters are the same as that of *String*, with the exception that convert_unicode defaults to True.

class sqlalchemy.types.**UnicodeText**(*length=None*, ***kwargs*)

Bases: *sqlalchemy.types.Text*

An unbounded-length Unicode string type.

See *Unicode* for details on the unicode behavior of this object.

Like *Unicode*, usage the *UnicodeText* type implies a unicode-capable type being used on the backend, such as NCLOB, NTEXT.

__init__(*length=None*, ***kwargs*)

Create a Unicode-converting Text type.

Parameters are the same as that of *Text*, with the exception that convert_unicode defaults to True.

SQL Standard and Multiple Vendor Types

This category of types refers to types that are either part of the SQL standard, or are potentially found within a subset of database backends. Unlike the "generic" types, the SQL standard/multi-vendor types have **no** guarantee of working on all backends, and will only work on those backends that explicitly support them by name. That is, the type will always emit its exact name in DDL with `CREATE TABLE` is issued.

class `sqlalchemy.types.`**`ARRAY`**(*item_type*, *as_tuple=False*, *dimensions=None*, *zero_indexes=False*)

Bases: `sqlalchemy.types.Indexable`, `sqlalchemy.types.Concatenable`, `sqlalchemy.types.TypeEngine`

Represent a SQL Array type.

Note: This type serves as the basis for all ARRAY operations. However, currently **only the Postgresql backend has support for SQL arrays in SQLAlchemy**. It is recommended to use the `postgresql.ARRAY` type directly when using ARRAY types with PostgreSQL, as it provides additional operators specific to that backend.

`types.ARRAY` is part of the Core in support of various SQL standard functions such as `array_agg` which explicitly involve arrays; however, with the exception of the PostgreSQL backend and possibly some third-party dialects, no other SQLAlchemy built-in dialect has support for this type.

An `types.ARRAY` type is constructed given the "type" of element:

```
mytable = Table("mytable", metadata,
        Column("data", ARRAY(Integer))
    )
```

The above type represents an N-dimensional array, meaning a supporting backend such as Postgresql will interpret values with any number of dimensions automatically. To produce an INSERT construct that passes in a 1-dimensional array of integers:

```
connection.execute(
        mytable.insert(),
        data=[1,2,3]
)
```

The `types.ARRAY` type can be constructed given a fixed number of dimensions:

```
mytable = Table("mytable", metadata,
        Column("data", ARRAY(Integer, dimensions=2))
    )
```

Sending a number of dimensions is optional, but recommended if the datatype is to represent arrays of more than one dimension. This number is used:

- When emitting the type declaration itself to the database, e.g. `INTEGER[][]`
- When translating Python values to database values, and vice versa, e.g. an ARRAY of `Unicode` objects uses this number to efficiently access the string values inside of array structures without resorting to per-row type inspection
- When used with the Python `getitem` accessor, the number of dimensions serves to define the kind of type that the `[]` operator should return, e.g. for an ARRAY of INTEGER with two dimensions:

```
>>> expr = table.c.column[5]  # returns ARRAY(Integer, dimensions=1)
>>> expr = expr[6]  # returns Integer
```

For 1-dimensional arrays, an *types.ARRAY* instance with no dimension parameter will generally assume single-dimensional behaviors.

SQL expressions of type *types.ARRAY* have support for "index" and "slice" behavior. The Python `[]` operator works normally here, given integer indexes or slices. Arrays default to 1-based indexing. The operator produces binary expression constructs which will produce the appropriate SQL, both for SELECT statements:

```
select([mytable.c.data[5], mytable.c.data[2:7]])
```

as well as UPDATE statements when the *Update.values()* method is used:

```
mytable.update().values({
    mytable.c.data[5]: 7,
    mytable.c.data[2:7]: [1, 2, 3]
})
```

The *types.ARRAY* type also provides for the operators *types.ARRAY.Comparator.any()* and *types.ARRAY.Comparator.all()*. The PostgreSQL-specific version of *types.ARRAY* also provides additional operators.

New in version 1.1.0.

See also:

postgresql.ARRAY

class Comparator(*expr*)

Bases: `sqlalchemy.types.Comparator`, `sqlalchemy.types.Comparator`

Define comparison operations for *types.ARRAY*.

More operators are available on the dialect-specific form of this type. See *postgresql.ARRAY.Comparator*.

all(*other*, *operator=None*)

Return `other operator ALL (array)` clause.

Argument places are switched, because ALL requires array expression to be on the right hand-side.

E.g.:

```
from sqlalchemy.sql import operators

conn.execute(
    select([table.c.data]).where(
            table.c.data.all(7, operator=operators.lt)
        )
)
```

Parameters

- **other** – expression to be compared
- **operator** – an operator object from the `sqlalchemy.sql.operators` package, defaults to `operators.eq()`.

See also:

sql.expression.all_()

types.ARRAY.Comparator.any()

any(*other*, *operator=None*)

Return `other operator ANY (array)` clause.

Argument places are switched, because ANY requires array expression to be on the right hand-side.

E.g.:

```
from sqlalchemy.sql import operators

conn.execute(
    select([table.c.data]).where(
            table.c.data.any(7, operator=operators.lt)
        )
)
```

Parameters

- **other** – expression to be compared
- **operator** – an operator object from the `sqlalchemy.sql.operators` package, defaults to `operators.eq()`.

See also:

sql.expression.any_()

types.ARRAY.Comparator.all()

ARRAY.**__init__**(*item_type*, *as_tuple=False*, *dimensions=None*, *zero_indexes=False*)

Construct an *types.ARRAY*.

E.g.:

```
Column('myarray', ARRAY(Integer))
```

Arguments are:

Parameters

- **item_type** – The data type of items of this array. Note that dimensionality is irrelevant here, so multi-dimensional arrays like `INTEGER[][]`, are constructed as `ARRAY(Integer)`, not as `ARRAY(ARRAY(Integer))` or such.
- **as_tuple=False** – Specify whether return results should be converted to tuples from lists. This parameter is not generally needed as a Python list corresponds well to a SQL array.
- **dimensions** – if non-None, the ARRAY will assume a fixed number of dimensions. This impacts how the array is declared on the database, how it goes about interpreting Python and result values, as well as how expression behavior in conjunction with the "getitem" operator works. See the description at *types.ARRAY* for additional detail.
- **zero_indexes=False** – when True, index values will be converted between Python zero-based and SQL one-based indexes, e.g. a value of one will be added to all index values before passing to the database.

ARRAY.**comparator_factory**

alias of *Comparator*

ARRAY.**zero_indexes** = **False**

if True, Python zero-based indexes should be interpreted as one-based on the SQL expression side.

class sqlalchemy.types.**BIGINT**
Bases: *sqlalchemy.types.BigInteger*

The SQL BIGINT type.

class sqlalchemy.types.**BINARY**(*length=None*)
Bases: sqlalchemy.types._Binary

The SQL BINARY type.

class sqlalchemy.types.**BLOB**(*length=None*)
Bases: *sqlalchemy.types.LargeBinary*

The SQL BLOB type.

class sqlalchemy.types.**BOOLEAN**(*create_constraint=True*, *name=None*, *_create_events=True*)
Bases: *sqlalchemy.types.Boolean*

The SQL BOOLEAN type.

class sqlalchemy.types.**CHAR**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)
Bases: *sqlalchemy.types.String*

The SQL CHAR type.

class sqlalchemy.types.**CLOB**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)
Bases: *sqlalchemy.types.Text*

The CLOB type.

This type is found in Oracle and Informix.

class sqlalchemy.types.**DATE**
Bases: *sqlalchemy.types.Date*

The SQL DATE type.

class sqlalchemy.types.**DATETIME**(*timezone=False*)
Bases: *sqlalchemy.types.DateTime*

The SQL DATETIME type.

class sqlalchemy.types.**DECIMAL**(*precision=None*, *scale=None*, *decimal_return_scale=None*, *asdecimal=True*)
Bases: *sqlalchemy.types.Numeric*

The SQL DECIMAL type.

class sqlalchemy.types.**FLOAT**(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*, ***kwargs*)
Bases: *sqlalchemy.types.Float*

The SQL FLOAT type.

sqlalchemy.types.**INT**
alias of *INTEGER*

class sqlalchemy.types.**JSON**(*none_as_null=False*)
Bases: *sqlalchemy.types.Indexable*, *sqlalchemy.types.TypeEngine*

Represent a SQL JSON type.

Note: *types.JSON* is provided as a facade for vendor-specific JSON types. Since it supports JSON SQL

operations, it only works on backends that have an actual JSON type, currently Postgresql as well as certain versions of MySQL.

---

types.JSON is part of the Core in support of the growing popularity of native JSON datatypes.

The *types.JSON* type stores arbitrary JSON format data, e.g.:

```
data_table = Table('data_table', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', JSON)
)

with engine.connect() as conn:
    conn.execute(
        data_table.insert(),
        data = {"key1": "value1", "key2": "value2"}
    )
```

The base *types.JSON* provides these two operations:

•Keyed index operations:

```
data_table.c.data['some key']
```

•Integer index operations:

```
data_table.c.data[3]
```

•Path index operations:

```
data_table.c.data[('key_1', 'key_2', 5, ..., 'key_n')]
```

Additional operations are available from the dialect-specific versions of *types.JSON*, such as *postgresql.JSON* and *postgresql.JSONB*, each of which offer more operators than just the basic type.

Index operations return an expression object whose type defaults to *JSON* by default, so that further JSON-oriented instructions may be called upon the result type.

The *JSON* type, when used with the SQLAlchemy ORM, does not detect in-place mutations to the structure. In order to detect these, the *sqlalchemy.ext.mutable* extension must be used. This extension will allow "in-place" changes to the datastructure to produce events which will be detected by the unit of work. See the example at *HSTORE* for a simple example involving a dictionary.

When working with NULL values, the *JSON* type recommends the use of two specific constants in order to differentiate between a column that evaluates to SQL NULL, e.g. no value, vs. the JSON-encoded string of `"null"`. To insert or select against a value that is SQL NULL, use the constant *null()*:

```
from sqlalchemy import null
conn.execute(table.insert(), json_value=null())
```

To insert or select against a value that is JSON `"null"`, use the constant *JSON.NULL*:

```
conn.execute(table.insert(), json_value=JSON.NULL)
```

The *JSON* type supports a flag *JSON.none_as_null* which when set to True will result in the Python constant `None` evaluating to the value of SQL NULL, and when set to False results in the Python constant `None` evaluating to the value of JSON `"null"`. The Python value `None` may be used in conjunction with either *JSON.NULL* and *null()* in order to indicate NULL values, but care must be taken as to the value of the *JSON.none_as_null* in these cases.

See also:

postgresql.JSON

postgresql.JSONB

mysql.JSON

New in version 1.1.

class Comparator(*expr*)

Bases: `sqlalchemy.types.Comparator`, `sqlalchemy.types.Comparator`

Define comparison operations for *types.JSON*.

class JSON.**JSONIndexType**

Bases: *sqlalchemy.types.TypeEngine*

Placeholder for the datatype of a JSON index value.

This allows execution-time processing of JSON index values for special syntaxes.

class JSON.**JSONPathType**

Bases: *sqlalchemy.types.TypeEngine*

Placeholder type for JSON path operations.

This allows execution-time processing of a path-based index value into a specific SQL syntax.

JSON.**NULL = symbol('JSON_NULL')**

Describe the json value of NULL.

This value is used to force the JSON value of `"null"` to be used as the value. A value of Python `None` will be recognized either as SQL NULL or JSON `"null"`, based on the setting of the *JSON.none_as_null* flag; the *JSON.NULL* constant can be used to always resolve to JSON `"null"` regardless of this setting. This is in contrast to the `sql.null()` construct, which always resolves to SQL NULL. E.g.:

```
from sqlalchemy import null
from sqlalchemy.dialects.postgresql import JSON

obj1 = MyObject(json_value=null())  # will *always* insert SQL NULL
obj2 = MyObject(json_value=JSON.NULL)  # will *always* insert JSON string "null"

session.add_all([obj1, obj2])
session.commit()
```

JSON.**__init__**(*none_as_null=False*)

Construct a *types.JSON* type.

Parameters **none_as_null=False** – if True, persist the value `None` as a SQL NULL value, not the JSON encoding of `null`. Note that when this flag is False, the *null()* construct can still be used to persist a NULL value:

```
from sqlalchemy import null
conn.execute(table.insert(), data=null())
```

See also:

types.JSON.NULL

JSON.**comparator_factory**

alias of *Comparator*

class sqlalchemy.types.**INTEGER**

Bases: *sqlalchemy.types.Integer*

The SQL INT or INTEGER type.

class sqlalchemy.types.**NCHAR**(*length=None*, ***kwargs*)
Bases: *sqlalchemy.types.Unicode*

The SQL NCHAR type.

class sqlalchemy.types.**NVARCHAR**(*length=None*, ***kwargs*)
Bases: *sqlalchemy.types.Unicode*

The SQL NVARCHAR type.

class sqlalchemy.types.**NUMERIC**(*precision=None*, *scale=None*, *decimal_return_scale=None*, *asdecimal=True*)
Bases: *sqlalchemy.types.Numeric*

The SQL NUMERIC type.

class sqlalchemy.types.**REAL**(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*, ***kwargs*)
Bases: *sqlalchemy.types.Float*

The SQL REAL type.

class sqlalchemy.types.**SMALLINT**
Bases: *sqlalchemy.types.SmallInteger*

The SQL SMALLINT type.

class sqlalchemy.types.**TEXT**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)
Bases: *sqlalchemy.types.Text*

The SQL TEXT type.

class sqlalchemy.types.**TIME**(*timezone=False*)
Bases: *sqlalchemy.types.Time*

The SQL TIME type.

class sqlalchemy.types.**TIMESTAMP**(*timezone=False*)
Bases: *sqlalchemy.types.DateTime*

The SQL TIMESTAMP type.

class sqlalchemy.types.**VARBINARY**(*length=None*)
Bases: sqlalchemy.types._Binary

The SQL VARBINARY type.

class sqlalchemy.types.**VARCHAR**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)
Bases: *sqlalchemy.types.String*

The SQL VARCHAR type.

Vendor-Specific Types

Database-specific types are also available for import from each database's dialect module. See the *Dialects* reference for the database you're interested in.

For example, MySQL has a `BIGINT` type and PostgreSQL has an `INET` type. To use these, import them from the module explicitly:

```
from sqlalchemy.dialects import mysql

table = Table('foo', metadata,
    Column('id', mysql.BIGINT),
    Column('enumerates', mysql.ENUM('a', 'b', 'c'))
)
```

Or some PostgreSQL types:

```
from sqlalchemy.dialects import postgresql

table = Table('foo', metadata,
    Column('ipaddress', postgresql.INET),
    Column('elements', postgresql.ARRAY(String))
)
```

Each dialect provides the full set of typenames supported by that backend within its *__all__* collection, so that a simple *import* * or similar will import all supported types as implemented for that backend:

```
from sqlalchemy.dialects.postgresql import *

t = Table('mytable', metadata,
           Column('id', INTEGER, primary_key=True),
           Column('name', VARCHAR(300)),
           Column('inetaddr', INET)
)
```

Where above, the INTEGER and VARCHAR types are ultimately from sqlalchemy.types, and INET is specific to the Postgresql dialect.

Some dialect level types have the same name as the SQL standard type, but also provide additional arguments. For example, MySQL implements the full range of character and string types including additional arguments such as *collation* and *charset*:

```
from sqlalchemy.dialects.mysql import VARCHAR, TEXT

table = Table('foo', meta,
    Column('col1', VARCHAR(200, collation='binary')),
    Column('col2', TEXT(charset='latin1'))
)
```

3.4.2 Custom Types

A variety of methods exist to redefine the behavior of existing types as well as to provide new ones.

Overriding Type Compilation

A frequent need is to force the "string" version of a type, that is the one rendered in a CREATE TABLE statement or other SQL function like CAST, to be changed. For example, an application may want to force the rendering of `BINARY` for all platforms except for one, in which is wants `BLOB` to be rendered. Usage of an existing generic type, in this case *`LargeBinary`*, is preferred for most use cases. But to control types more accurately, a compilation directive that is per-dialect can be associated with any type:

```
from sqlalchemy.ext.compiler import compiles
from sqlalchemy.types import BINARY

@compiles(BINARY, "sqlite")
def compile_binary_sqlite(type_, compiler, **kw):
    return "BLOB"
```

The above code allows the usage of `types.BINARY`, which will produce the string `BINARY` against all backends except SQLite, in which case it will produce `BLOB`.

See the section *Changing Compilation of Types*, a subsection of *Custom SQL Constructs and Compilation Extension*, for additional examples.

Augmenting Existing Types

The `TypeDecorator` allows the creation of custom types which add bind-parameter and result-processing behavior to an existing type object. It is used when additional in-Python marshaling of data to and from the database is required.

Note: The bind- and result-processing of `TypeDecorator` is *in addition* to the processing already performed by the hosted type, which is customized by SQLAlchemy on a per-DBAPI basis to perform processing specific to that DBAPI. To change the DBAPI-level processing for an existing type, see the section *Replacing the Bind/Result Processing of Existing Types*.

class `sqlalchemy.types.`**`TypeDecorator`**(**args*, ***kwargs*)

Bases: `sqlalchemy.sql.expression.SchemaEventTarget`, `sqlalchemy.types.TypeEngine`

Allows the creation of types which add additional functionality to an existing type.

This method is preferred to direct subclassing of SQLAlchemy's built-in types as it ensures that all required functionality of the underlying type is kept in place.

Typical usage:

```
import sqlalchemy.types as types

class MyType(types.TypeDecorator):
    '''Prefixes Unicode values with "PREFIX:" on the way in and
    strips it off on the way out.
    '''

    impl = types.Unicode

    def process_bind_param(self, value, dialect):
        return "PREFIX:" + value

    def process_result_value(self, value, dialect):
        return value[7:]

    def copy(self, **kw):
        return MyType(self.impl.length)
```

The class-level "impl" attribute is required, and can reference any TypeEngine class. Alternatively, the load_dialect_impl() method can be used to provide different type classes based on the dialect given; in this case, the "impl" variable can reference `TypeEngine` as a placeholder.

Types that receive a Python type that isn't similar to the ultimate type used may want to define the *TypeDecorator.coerce_compared_value()* method. This is used to give the expression system a hint when coercing Python objects into bind parameters within expressions. Consider this expression:

```
mytable.c.somecol + datetime.date(2009, 5, 15)
```

Above, if "somecol" is an `Integer` variant, it makes sense that we're doing date arithmetic, where above is usually interpreted by databases as adding a number of days to the given date. The expression system does the right thing by not attempting to coerce the "date()" value into an integer-oriented bind parameter.

However, in the case of `TypeDecorator`, we are usually changing an incoming Python type to something new - `TypeDecorator` by default will "coerce" the non-typed side to be the same type as itself. Such as below, we define an "epoch" type that stores a date value as an integer:

```
class MyEpochType(types.TypeDecorator):
    impl = types.Integer

    epoch = datetime.date(1970, 1, 1)

    def process_bind_param(self, value, dialect):
        return (value - self.epoch).days

    def process_result_value(self, value, dialect):
        return self.epoch + timedelta(days=value)
```

Our expression of `somecol + date` with the above type will coerce the "date" on the right side to also be treated as `MyEpochType`.

This behavior can be overridden via the *coerce_compared_value()* method, which returns a type that should be used for the value of the expression. Below we set it such that an integer value will be treated as an `Integer`, and any other value is assumed to be a date and will be treated as a `MyEpochType`:

```
def coerce_compared_value(self, op, value):
    if isinstance(value, int):
        return Integer()
    else:
        return self
```

> **Warning:** Note that the **behavior of coerce_compared_value is not inherited by default from that of the base type**. If the *TypeDecorator* is augmenting a type that requires special logic for certain types of operators, this method **must** be overridden. A key example is when decorating the *postgresql.JSON* and *postgresql.JSONB* types; the default rules of *TypeEngine.coerce_compared_value()* should be used in order to deal with operators like index operations:

```
class MyJsonType(TypeDecorator):
    impl = postgresql.JSON

    def coerce_compared_value(self, op, value):
        return self.impl.coerce_compared_value(op, value)
```

> Without the above step, index operations such as `mycol['foo']` will cause the index value `'foo'` to be JSON encoded.

__init__(**args*, ***kwargs*)

Construct a *TypeDecorator*.

Arguments sent here are passed to the constructor of the class assigned to the `impl` class level attribute, assuming the `impl` is a callable, and the resulting object is assigned to the `self.impl` instance attribute (thus overriding the class attribute of the same name).

If the class level `impl` is not a callable (the unusual case), it will be assigned to the same instance attribute 'as-is', ignoring those arguments passed to the constructor.

Subclasses can override this to customize the generation of `self.impl` entirely.

adapt(*cls*, ***kw*)

inherited from the `adapt()` method of `TypeEngine`

Produce an "adapted" form of this type, given an "impl" class to work with.

This method is used internally to associate generic types with "implementation" types that are specific to a particular dialect.

bind_expression(*bindvalue*)

inherited from the `bind_expression()` method of `TypeEngine`

"Given a bind value (i.e. a `BindParameter` instance), return a SQL expression in its place.

This is typically a SQL function that wraps the existing bound parameter within the statement. It is used for special data types that require literals being wrapped in some special database function in order to coerce an application-level value into a database-specific format. It is the SQL analogue of the `TypeEngine.bind_processor()` method.

The method is evaluated at statement compile time, as opposed to statement construction time.

Note that this method, when implemented, should always return the exact same structure, without any conditional logic, as it may be used in an executemany() call against an arbitrary number of bound parameter sets.

See also:

Applying SQL-level Bind/Result Processing

bind_processor(*dialect*)

Provide a bound value processing function for the given `Dialect`.

This is the method that fulfills the `TypeEngine` contract for bound value conversion. `TypeDecorator` will wrap a user-defined implementation of `process_bind_param()` here.

User-defined code can override this method directly, though its likely best to use `process_bind_param()` so that the processing provided by `self.impl` is maintained.

Parameters **dialect** – Dialect instance in use.

This method is the reverse counterpart to the `result_processor()` method of this class.

coerce_compared_value(*op*, *value*)

Suggest a type for a 'coerced' Python value in an expression.

By default, returns self. This method is called by the expression system when an object using this type is on the left or right side of an expression against a plain Python object which does not yet have a SQLAlchemy type assigned:

```
expr = table.c.somecolumn + 35
```

Where above, if `somecolumn` uses this type, this method will be called with the value `operator.add` and `35`. The return value is whatever SQLAlchemy type should be used for `35` for this particular operation.

coerce_to_is_types = (<type 'NoneType'>,)

Specify those Python types which should be coerced at the expression level to "IS <constant>" when compared using == (and same for `IS NOT` in conjunction with `!=`.

For most SQLAlchemy types, this includes `NoneType`, as well as `bool`.

`TypeDecorator` modifies this list to only include `NoneType`, as typedecorator implementations that deal with boolean types are common.

Custom *`TypeDecorator`* classes can override this attribute to return an empty tuple, in which case no values will be coerced to constants.

..versionadded:: 0.8.2 Added *`TypeDecorator.coerce_to_is_types`* to allow for easier control of `__eq__()` `__ne__()` operations.

column_expression(*colexpr*)

inherited from the `column_expression()` method of `TypeEngine`

Given a SELECT column expression, return a wrapping SQL expression.

This is typically a SQL function that wraps a column expression as rendered in the columns clause of a SELECT statement. It is used for special data types that require columns to be wrapped in some special database function in order to coerce the value before being sent back to the application. It is the SQL analogue of the *`TypeEngine.result_processor()`* method.

The method is evaluated at statement compile time, as opposed to statement construction time.

See also:

Applying SQL-level Bind/Result Processing

compare_against_backend(*dialect*, *conn_type*)

inherited from the `compare_against_backend()` method of `TypeEngine`

Compare this type against the given backend type.

This function is currently not implemented for SQLAlchemy types, and for all built in types will return `None`. However, it can be implemented by a user-defined type where it can be consumed by schema comparison tools such as Alembic autogenerate.

A future release of SQLAlchemy will potentially impement this method for builtin types as well.

The function should return True if this type is equivalent to the given type; the type is typically reflected from the database so should be database specific. The dialect in use is also passed. It can also return False to assert that the type is not equivalent.

Parameters

- **dialect** – a *`Dialect`* that is involved in the comparison.
- **conn_type** – the type object reflected from the backend.

New in version 1.0.3.

compare_values(*x*, *y*)

Given two values, compare them for equality.

By default this calls upon *`TypeEngine.compare_values()`* of the underlying "impl", which in turn usually uses the Python equals operator ==.

This function is used by the ORM to compare an original-loaded value with an intercepted "changed" value, to determine if a net change has occurred.

compile(*dialect=None*)

inherited from the `compile()` method of `TypeEngine`

Produce a string-compiled form of this `TypeEngine`.

When called with no arguments, uses a "default" dialect to produce a string result.

Parameters **dialect** – a `Dialect` instance.

copy(***kw*)

Produce a copy of this `TypeDecorator` instance.

This is a shallow copy and is provided to fulfill part of the `TypeEngine` contract. It usually does not need to be overridden unless the user-defined `TypeDecorator` has local state that should be deep-copied.

dialect_impl(*dialect*)

inherited from the `dialect_impl()` *method of* `TypeEngine`

Return a dialect-specific implementation for this `TypeEngine`.

evaluates_none()

inherited from the `evaluates_none()` *method of* `TypeEngine`

Return a copy of this type which has the `should_evaluate_none` flag set to True.

E.g.:

```
Table(
    'some_table', metadata,
    Column(
        String(50).evaluates_none(),
        nullable=True,
        server_default='no value')
)
```

The ORM uses this flag to indicate that a positive value of `None` is passed to the column in an INSERT statement, rather than omitting the column from the INSERT statement which has the effect of firing off column-level defaults. It also allows for types which have special behavior associated with the Python None value to indicate that the value doesn't necessarily translate into SQL NULL; a prime example of this is a JSON type which may wish to persist the JSON value `'null'`.

In all cases, the actual NULL SQL value can be always be persisted in any column by using the `null` SQL construct in an INSERT statement or associated with an ORM-mapped attribute.

New in version 1.1.

See also:

Forcing NULL on a column with a default - in the ORM documentation

`postgresql.JSON.none_as_null` - Postgresql JSON interaction with this flag.

`TypeEngine.should_evaluate_none` - class-level flag

get_dbapi_type(*dbapi*)

Return the DBAPI type object represented by this `TypeDecorator`.

By default this calls upon `TypeEngine.get_dbapi_type()` of the underlying "impl".

literal_processor(*dialect*)

Provide a literal processing function for the given `Dialect`.

Subclasses here will typically override `TypeDecorator.process_literal_param()` instead of this method directly.

By default, this method makes use of `TypeDecorator.process_bind_param()` if that method is implemented, where `TypeDecorator.process_literal_param()` is not. The rationale here is that `TypeDecorator` typically deals with Python conversions of data that are above the layer of

database presentation. With the value converted by *TypeDecorator.process_bind_param()*, the underlying type will then handle whether it needs to be presented to the DBAPI as a bound parameter or to the database as an inline SQL value.

New in version 0.9.0.

load_dialect_impl(*dialect*)

Return a *TypeEngine* object corresponding to a dialect.

This is an end-user override hook that can be used to provide differing types depending on the given dialect. It is used by the *TypeDecorator* implementation of *type_engine()* to help determine what type should ultimately be returned for a given *TypeDecorator*.

By default returns `self.impl`.

process_bind_param(*value*, *dialect*)

Receive a bound parameter value to be converted.

Subclasses override this method to return the value that should be passed along to the underlying *TypeEngine* object, and from there to the DBAPI `execute()` method.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

This operation should be designed with the reverse operation in mind, which would be the process_result_value method of this class.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be `None`.
- **dialect** – the *Dialect* in use.

process_literal_param(*value*, *dialect*)

Receive a literal parameter value to be rendered inline within a statement.

This method is used when the compiler renders a literal value without using binds, typically within DDL such as in the "server default" of a column or an expression within a CHECK constraint.

The returned string will be rendered into the output string.

New in version 0.9.0.

process_result_value(*value*, *dialect*)

Receive a result-row column value to be converted.

Subclasses should implement this method to operate on data fetched from the database.

Subclasses override this method to return the value that should be passed back to the application, given a value that is already processed by the underlying *TypeEngine* object, originally from the DBAPI cursor method `fetchone()` or similar.

The operation could be anything desired to perform custom behavior, such as transforming or serializing data. This could also be used as a hook for validating logic.

Parameters

- **value** – Data to operate upon, of any type expected by this method in the subclass. Can be `None`.
- **dialect** – the *Dialect* in use.

This operation should be designed to be reversible by the "process_bind_param" method of this class.

python_type

inherited from the `python_type` *attribute of* `TypeEngine`

Return the Python type object expected to be returned by instances of this type, if known.

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like `int` for example), will return that type.

If a return type is not defined, raises `NotImplementedError`.

Note that any type also accommodates NULL in SQL which means you can also get back `None` from any type in practice.

result_processor(*dialect*, *coltype*)

Provide a result value processing function for the given `Dialect`.

This is the method that fulfills the `TypeEngine` contract for result value conversion. `TypeDecorator` will wrap a user-defined implementation of `process_result_value()` here.

User-defined code can override this method directly, though its likely best to use `process_result_value()` so that the processing provided by `self.impl` is maintained.

Parameters

- **dialect** – Dialect instance in use.
- **coltype** – An SQLAlchemy data type

This method is the reverse counterpart to the `bind_processor()` method of this class.

type_engine(*dialect*)

Return a dialect-specific `TypeEngine` instance for this `TypeDecorator`.

In most cases this returns a dialect-adapted form of the `TypeEngine` type represented by `self.impl`. Makes usage of `dialect_impl()` but also traverses into wrapped `TypeDecorator` instances. Behavior can be customized here by overriding `load_dialect_impl()`.

with_variant(*type_*, *dialect_name*)

inherited from the `with_variant()` *method of* `TypeEngine`

Produce a new type object that will utilize the given type when applied to the dialect of the given name.

e.g.:

```
from sqlalchemy.types import String
from sqlalchemy.dialects import mysql

s = String()

s = s.with_variant(mysql.VARCHAR(collation='foo'), 'mysql')
```

The construction of `TypeEngine.with_variant()` is always from the "fallback" type to that which is dialect specific. The returned type is an instance of `Variant`, which itself provides a `Variant.with_variant()` that can be called repeatedly.

Parameters

- **type_** – a `TypeEngine` that will be selected as a variant from the originating type, when a dialect of the given name is in use.
- **dialect_name** – base name of the dialect which uses this type. (i.e. `'postgresql'`, `'mysql'`, etc.)

New in version 0.7.2.

TypeDecorator Recipes

A few key *TypeDecorator* recipes follow.

Coercing Encoded Strings to Unicode

A common source of confusion regarding the *Unicode* type is that it is intended to deal *only* with Python `unicode` objects on the Python side, meaning values passed to it as bind parameters must be of the form `u'some string'` if using Python 2 and not 3. The encoding/decoding functions it performs are only to suit what the DBAPI in use requires, and are primarily a private implementation detail.

The use case of a type that can safely receive Python bytestrings, that is strings that contain non-ASCII characters and are not `u''` objects in Python 2, can be achieved using a *TypeDecorator* which coerces as needed:

```
from sqlalchemy.types import TypeDecorator, Unicode

class CoerceUTF8(TypeDecorator):
    """Safely coerce Python bytestrings to Unicode
    before passing off to the database."""

    impl = Unicode

    def process_bind_param(self, value, dialect):
        if isinstance(value, str):
            value = value.decode('utf-8')
        return value
```

Rounding Numerics

Some database connectors like those of SQL Server choke if a Decimal is passed with too many decimal places. Here's a recipe that rounds them down:

```
from sqlalchemy.types import TypeDecorator, Numeric
from decimal import Decimal

class SafeNumeric(TypeDecorator):
    """Adds quantization to Numeric."""

    impl = Numeric

    def __init__(self, *arg, **kw):
        TypeDecorator.__init__(self, *arg, **kw)
        self.quantize_int = - self.impl.scale
        self.quantize = Decimal(10) ** self.quantize_int

    def process_bind_param(self, value, dialect):
        if isinstance(value, Decimal) and \
            value.as_tuple()[2] < self.quantize_int:
            value = value.quantize(self.quantize)
        return value
```

Backend-agnostic GUID Type

Receives and returns Python uuid() objects. Uses the PG UUID type when using Postgresql, CHAR(32) on other backends, storing them in stringified hex format. Can be modified to store binary in CHAR(16) if desired:

```
from sqlalchemy.types import TypeDecorator, CHAR
from sqlalchemy.dialects.postgresql import UUID
import uuid

class GUID(TypeDecorator):
    """Platform-independent GUID type.

    Uses Postgresql's UUID type, otherwise uses
    CHAR(32), storing as stringified hex values.

    """
    impl = CHAR

    def load_dialect_impl(self, dialect):
        if dialect.name == 'postgresql':
            return dialect.type_descriptor(UUID())
        else:
            return dialect.type_descriptor(CHAR(32))

    def process_bind_param(self, value, dialect):
        if value is None:
            return value
        elif dialect.name == 'postgresql':
            return str(value)
        else:
            if not isinstance(value, uuid.UUID):
                return "%.32x" % uuid.UUID(value)
            else:
                # hexstring
                return "%.32x" % value

    def process_result_value(self, value, dialect):
        if value is None:
            return value
        else:
            return uuid.UUID(value)
```

Marshal JSON Strings

This type uses `simplejson` to marshal Python data structures to/from JSON. Can be modified to use Python's builtin json encoder:

```
from sqlalchemy.types import TypeDecorator, VARCHAR
import json

class JSONEncodedDict(TypeDecorator):
    """Represents an immutable structure as a json-encoded string.

    Usage::

        JSONEncodedDict(255)
```

```
    """

    impl = VARCHAR

    def process_bind_param(self, value, dialect):
        if value is not None:
            value = json.dumps(value)

        return value

    def process_result_value(self, value, dialect):
        if value is not None:
            value = json.loads(value)
        return value
```

Note that the ORM by default will not detect "mutability" on such a type - meaning, in-place changes to values will not be detected and will not be flushed. Without further steps, you instead would need to replace the existing value with a new one on each parent object to detect changes. Note that there's nothing wrong with this, as many applications may not require that the values are ever mutated once created. For those which do have this requirement, support for mutability is best applied using the `sqlalchemy.ext.mutable` extension - see the example in *Mutation Tracking*.

Replacing the Bind/Result Processing of Existing Types

Most augmentation of type behavior at the bind/result level is achieved using *`TypeDecorator`*. For the rare scenario where the specific processing applied by SQLAlchemy at the DBAPI level needs to be replaced, the SQLAlchemy type can be subclassed directly, and the `bind_processor()` or `result_processor()` methods can be overridden. Doing so requires that the `adapt()` method also be overridden. This method is the mechanism by which SQLAlchemy produces DBAPI-specific type behavior during statement execution. Overriding it allows a copy of the custom type to be used in lieu of a DBAPI-specific type. Below we subclass the *`types.TIME`* type to have custom result processing behavior. The `process()` function will receive `value` from the DBAPI cursor directly:

```
class MySpecialTime(TIME):
    def __init__(self, special_argument):
        super(MySpecialTime, self).__init__()
        self.special_argument = special_argument

    def result_processor(self, dialect, coltype):
        import datetime
        time = datetime.time
        def process(value):
            if value is not None:
                microseconds = value.microseconds
                seconds = value.seconds
                minutes = seconds / 60
                return time(
                          minutes / 60,
                          minutes % 60,
                          seconds - minutes * 60,
                          microseconds)
            else:
                return None
        return process

    def adapt(self, impltype):
        return MySpecialTime(self.special_argument)
```

Applying SQL-level Bind/Result Processing

As seen in the sections *Augmenting Existing Types* and *Replacing the Bind/Result Processing of Existing Types*, SQLAlchemy allows Python functions to be invoked both when parameters are sent to a statement, as well as when result rows are loaded from the database, to apply transformations to the values as they are sent to or from the database. It is also possible to define SQL-level transformations as well. The rationale here is when only the relational database contains a particular series of functions that are necessary to coerce incoming and outgoing data between an application and persistence format. Examples include using database-defined encryption/decryption functions, as well as stored procedures that handle geographic data. The Postgis extension to Postgresql includes an extensive array of SQL functions that are necessary for coercing data into particular formats.

Any `TypeEngine`, `UserDefinedType` or `TypeDecorator` subclass can include implementations of `TypeEngine.bind_expression()` and/or `TypeEngine.column_expression()`, which when defined to return a non-`None` value should return a `ColumnElement` expression to be injected into the SQL statement, either surrounding bound parameters or a column expression. For example, to build a `Geometry` type which will apply the Postgis function `ST_GeomFromText` to all outgoing values and the function `ST_AsText` to all incoming data, we can create our own subclass of `UserDefinedType` which provides these methods in conjunction with `func`:

```
from sqlalchemy import func
from sqlalchemy.types import UserDefinedType

class Geometry(UserDefinedType):
    def get_col_spec(self):
        return "GEOMETRY"

    def bind_expression(self, bindvalue):
        return func.ST_GeomFromText(bindvalue, type_=self)

    def column_expression(self, col):
        return func.ST_AsText(col, type_=self)
```

We can apply the `Geometry` type into `Table` metadata and use it in a `select()` construct:

```
geometry = Table('geometry', metadata,
              Column('geom_id', Integer, primary_key=True),
              Column('geom_data', Geometry)
            )

print select([geometry]).where(
  geometry.c.geom_data == 'LINESTRING(189412 252431,189631 259122)')
```

The resulting SQL embeds both functions as appropriate. `ST_AsText` is applied to the columns clause so that the return value is run through the function before passing into a result set, and `ST_GeomFromText` is run on the bound parameter so that the passed-in value is converted:

```
SELECT geometry.geom_id, ST_AsText(geometry.geom_data) AS geom_data_1
FROM geometry
WHERE geometry.geom_data = ST_GeomFromText(:geom_data_2)
```

The `TypeEngine.column_expression()` method interacts with the mechanics of the compiler such that the SQL expression does not interfere with the labeling of the wrapped expression. Such as, if we rendered a `select()` against a `label()` of our expression, the string label is moved to the outside of the wrapped expression:

```
print select([geometry.c.geom_data.label('my_data')])
```

Output:

```
SELECT ST_AsText(geometry.geom_data) AS my_data
FROM geometry
```

For an example of subclassing a built in type directly, we subclass *postgresql.BYTEA* to provide a `PGPString`, which will make use of the Postgresql `pgcrypto` extension to encrpyt/decrypt values transparently:

```
from sqlalchemy import create_engine, String, select, func, \
        MetaData, Table, Column, type_coerce

from sqlalchemy.dialects.postgresql import BYTEA

class PGPString(BYTEA):
    def __init__(self, passphrase, length=None):
        super(PGPString, self).__init__(length)
        self.passphrase = passphrase

    def bind_expression(self, bindvalue):
        # convert the bind's type from PGPString to
        # String, so that it's passed to psycopg2 as is without
        # a dbapi.Binary wrapper
        bindvalue = type_coerce(bindvalue, String)
        return func.pgp_sym_encrypt(bindvalue, self.passphrase)

    def column_expression(self, col):
        return func.pgp_sym_decrypt(col, self.passphrase)

metadata = MetaData()
message = Table('message', metadata,
                Column('username', String(50)),
                Column('message',
                    PGPString("this is my passphrase", length=1000)),
            )

engine = create_engine("postgresql://scott:tiger@localhost/test", echo=True)
with engine.begin() as conn:
    metadata.create_all(conn)

    conn.execute(message.insert(), username="some user",
                                message="this is my message")

    print conn.scalar(
            select([message.c.message]).\
                where(message.c.username == "some user")
        )
```

The `pgp_sym_encrypt` and `pgp_sym_decrypt` functions are applied to the INSERT and SELECT statements:

```
INSERT INTO message (username, message)
  VALUES (%(username)s, pgp_sym_encrypt(%(message)s, %(pgp_sym_encrypt_1)s))
  {'username': 'some user', 'message': 'this is my message',
    'pgp_sym_encrypt_1': 'this is my passphrase'}

SELECT pgp_sym_decrypt(message.message, %(pgp_sym_decrypt_1)s) AS message_1
  FROM message
  WHERE message.username = %(username_1)s
  {'pgp_sym_decrypt_1': 'this is my passphrase', 'username_1': 'some user'}
```

New in version 0.8: Added the *TypeEngine.bind_expression()* and *TypeEngine.column_expression()* methods.

See also:

PostGIS Integration

Redefining and Creating New Operators

SQLAlchemy Core defines a fixed set of expression operators available to all column expressions. Some of these operations have the effect of overloading Python's built in operators; examples of such operators include *ColumnOperators.__eq__()* (`table.c.somecolumn == 'foo'`), *ColumnOperators.__invert__()* (`~table.c.flag`), and *ColumnOperators.__add__()* (`table.c.x + table.c.y`). Other operators are exposed as explicit methods on column expressions, such as *ColumnOperators.in_()* (`table.c.value.in_(['x', 'y'])`) and *ColumnOperators.like()* (`table.c.value.like('%ed%')`).

The Core expression constructs in all cases consult the type of the expression in order to determine the behavior of existing operators, as well as to locate additional operators that aren't part of the built in set. The *TypeEngine* base class defines a root "comparison" implementation *TypeEngine.Comparator*, and many specific types provide their own sub-implementations of this class. User-defined *TypeEngine.Comparator* implementations can be built directly into a simple subclass of a particular type in order to override or define new operations. Below, we create a *Integer* subclass which overrides the *ColumnOperators.__add__()* operator:

```
from sqlalchemy import Integer

class MyInt(Integer):
    class comparator_factory(Integer.Comparator):
        def __add__(self, other):
            return self.op("goofy")(other)
```

The above configuration creates a new class `MyInt`, which establishes the *TypeEngine.comparator_factory* attribute as referring to a new class, subclassing the *TypeEngine.Comparator* class associated with the *Integer* type.

Usage:

```
>>> sometable = Table("sometable", metadata, Column("data", MyInt))
>>> print sometable.c.data + 5
sometable.data goofy :data_1
```

The implementation for *ColumnOperators.__add__()* is consulted by an owning SQL expression, by instantiating the *TypeEngine.Comparator* with itself as the `expr` attribute. The mechanics of the expression system are such that operations continue recursively until an expression object produces a new SQL expression construct. Above, we could just as well have said `self.expr.op("goofy")(other)` instead of `self.op("goofy")(other)`.

New methods added to a *TypeEngine.Comparator* are exposed on an owning SQL expression using a `__getattr__` scheme, which exposes methods added to *TypeEngine.Comparator* onto the owning *ColumnElement*. For example, to add a `log()` function to integers:

```
from sqlalchemy import Integer, func

class MyInt(Integer):
    class comparator_factory(Integer.Comparator):
        def log(self, other):
            return func.log(self.expr, other)
```

Using the above type:

```
>>> print sometable.c.data.log(5)
log(:log_1, :log_2)
```

Unary operations are also possible. For example, to add an implementation of the Postgresql factorial operator, we combine the *UnaryExpression* construct along with a *custom_op* to produce the factorial expression:

```
from sqlalchemy import Integer
from sqlalchemy.sql.expression import UnaryExpression
from sqlalchemy.sql import operators

class MyInteger(Integer):
    class comparator_factory(Integer.Comparator):
        def factorial(self):
            return UnaryExpression(self.expr,
                        modifier=operators.custom_op("!"),
                        type_=MyInteger)
```

Using the above type:

```
>>> from sqlalchemy.sql import column
>>> print column('x', MyInteger).factorial()
x !
```

See also:

TypeEngine.comparator_factory

New in version 0.8: The expression system was enhanced to support customization of operators on a per-type level.

Creating New Types

The *UserDefinedType* class is provided as a simple base class for defining entirely new database types. Use this to represent native database types not known by SQLAlchemy. If only Python translation behavior is needed, use *TypeDecorator* instead.

class sqlalchemy.types.**UserDefinedType**

Bases: *sqlalchemy.types.TypeEngine*

Base for user defined types.

This should be the base of new types. Note that for most cases, *TypeDecorator* is probably more appropriate:

```
import sqlalchemy.types as types

class MyType(types.UserDefinedType):
    def __init__(self, precision = 8):
        self.precision = precision

    def get_col_spec(self, **kw):
        return "MYTYPE(%s)" % self.precision

    def bind_processor(self, dialect):
        def process(value):
            return value
        return process
```

```
    def result_processor(self, dialect, coltype):
        def process(value):
            return value
        return process
```

Once the type is made, it's immediately usable:

```
table = Table('foo', meta,
    Column('id', Integer, primary_key=True),
    Column('data', MyType(16))
    )
```

The `get_col_spec()` method will in most cases receive a keyword argument `type_expression` which refers to the owning expression of the type as being compiled, such as a *Column* or *cast()* construct. This keyword is only sent if the method accepts keyword arguments (e.g. `**kw`) in its argument signature; introspection is used to check for this in order to support legacy forms of this function.

New in version 1.0.0: the owning expression is passed to the `get_col_spec()` method via the keyword argument `type_expression`, if it receives `**kw` in its signature.

coerce_compared_value(*op*, *value*)

Suggest a type for a 'coerced' Python value in an expression.

Default behavior for *UserDefinedType* is the same as that of *TypeDecorator*; by default it returns `self`, assuming the compared value should be coerced into the same type as this one. See *TypeDecorator.coerce_compared_value()* for more detail.

Changed in version 0.8: *UserDefinedType.coerce_compared_value()* now returns `self` by default, rather than falling onto the more fundamental behavior of *TypeEngine.coerce_compared_value()*.

3.4.3 Base Type API

class sqlalchemy.types.**TypeEngine**

Bases: `sqlalchemy.sql.visitors.Visitable`

The ultimate base class for all SQL datatypes.

Common subclasses of *TypeEngine* include *String*, *Integer*, and *Boolean*.

For an overview of the SQLAlchemy typing system, see *Column and Data Types*.

See also:

Column and Data Types

class Comparator(*expr*)

Bases: *sqlalchemy.sql.operators.ColumnOperators*

Base class for custom comparison operations defined at the type level. See *TypeEngine.comparator_factory*.

TypeEngine.**adapt**(*cls*, ***kw*)

Produce an "adapted" form of this type, given an "impl" class to work with.

This method is used internally to associate generic types with "implementation" types that are specific to a particular dialect.

`TypeEngine.bind_expression(bindvalue)`

"Given a bind value (i.e. a `BindParameter` instance), return a SQL expression in its place.

This is typically a SQL function that wraps the existing bound parameter within the statement. It is used for special data types that require literals being wrapped in some special database function in order to coerce an application-level value into a database-specific format. It is the SQL analogue of the `TypeEngine.bind_processor()` method.

The method is evaluated at statement compile time, as opposed to statement construction time.

Note that this method, when implemented, should always return the exact same structure, without any conditional logic, as it may be used in an executemany() call against an arbitrary number of bound parameter sets.

See also:

Applying SQL-level Bind/Result Processing

`TypeEngine.bind_processor(dialect)`

Return a conversion function for processing bind values.

Returns a callable which will receive a bind parameter value as the sole positional argument and will return a value to send to the DB-API.

If processing is not necessary, the method should return `None`.

Parameters **dialect** – Dialect instance in use.

`TypeEngine.coerce_compared_value(op, value)`

Suggest a type for a 'coerced' Python value in an expression.

Given an operator and value, gives the type a chance to return a type which the value should be coerced into.

The default behavior here is conservative; if the right-hand side is already coerced into a SQL type based on its Python type, it is usually left alone.

End-user functionality extension here should generally be via `TypeDecorator`, which provides more liberal behavior in that it defaults to coercing the other side of the expression into this type, thus applying special Python conversions above and beyond those needed by the DBAPI to both ides. It also provides the public method `TypeDecorator.coerce_compared_value()` which is intended for end-user customization of this behavior.

`TypeEngine.column_expression(colexpr)`

Given a SELECT column expression, return a wrapping SQL expression.

This is typically a SQL function that wraps a column expression as rendered in the columns clause of a SELECT statement. It is used for special data types that require columns to be wrapped in some special database function in order to coerce the value before being sent back to the application. It is the SQL analogue of the `TypeEngine.result_processor()` method.

The method is evaluated at statement compile time, as opposed to statement construction time.

See also:

Applying SQL-level Bind/Result Processing

`TypeEngine.comparator_factory`

Bases: `sqlalchemy.sql.operators.ColumnOperators`

A `TypeEngine.Comparator` class which will apply to operations performed by owning `ColumnElement` objects.

The *comparator_factory* attribute is a hook consulted by the core expression system when column and SQL expression operations are performed. When a *TypeEngine.Comparator* class is associated with this attribute, it allows custom re-definition of all existing operators, as well as definition of new operators. Existing operators include those provided by Python operator overloading such as *operators.ColumnOperators.__add__()* and *operators.ColumnOperators.__eq__()*, those provided as standard attributes of *operators.ColumnOperators* such as *operators.ColumnOperators.like()* and *operators.ColumnOperators.in_()*.

Rudimentary usage of this hook is allowed through simple subclassing of existing types, or alternatively by using *TypeDecorator*. See the documentation section *Redefining and Creating New Operators* for examples.

New in version 0.8: The expression system was enhanced to support customization of operators on a per-type level.

alias of *Comparator*

`TypeEngine.`**`compare_against_backend`**(*dialect*, *conn_type*)

Compare this type against the given backend type.

This function is currently not implemented for SQLAlchemy types, and for all built in types will return `None`. However, it can be implemented by a user-defined type where it can be consumed by schema comparison tools such as Alembic autogenerate.

A future release of SQLAlchemy will potentially impement this method for builtin types as well.

The function should return True if this type is equivalent to the given type; the type is typically reflected from the database so should be database specific. The dialect in use is also passed. It can also return False to assert that the type is not equivalent.

Parameters

- **dialect** – a *Dialect* that is involved in the comparison.
- **conn_type** – the type object reflected from the backend.

New in version 1.0.3.

`TypeEngine.`**`compare_values`**(*x*, *y*)

Compare two values for equality.

`TypeEngine.`**`compile`**(*dialect=None*)

Produce a string-compiled form of this *TypeEngine*.

When called with no arguments, uses a "default" dialect to produce a string result.

Parameters **dialect** – a *Dialect* instance.

`TypeEngine.`**`dialect_impl`**(*dialect*)

Return a dialect-specific implementation for this *TypeEngine*.

`TypeEngine.`**`evaluates_none`**()

Return a copy of this type which has the *should_evaluate_none* flag set to True.

E.g.:

```
Table(
    'some_table', metadata,
    Column(
        String(50).evaluates_none(),
        nullable=True,
        server_default='no value')
)
```

The ORM uses this flag to indicate that a positive value of `None` is passed to the column in an INSERT statement, rather than omitting the column from the INSERT statement which has the effect of firing off column-level defaults. It also allows for types which have special behavior associated with the Python None value to indicate that the value doesn't necessarily translate into SQL NULL; a prime example of this is a JSON type which may wish to persist the JSON value `'null'`.

In all cases, the actual NULL SQL value can be always be persisted in any column by using the *null* SQL construct in an INSERT statement or associated with an ORM-mapped attribute.

New in version 1.1.

See also:

Forcing NULL on a column with a default - in the ORM documentation

postgresql.JSON.none_as_null - Postgresql JSON interaction with this flag.

TypeEngine.should_evaluate_none - class-level flag

`TypeEngine.`**`get_dbapi_type`**(*dbapi*)

Return the corresponding type object from the underlying DB-API, if any.

This can be useful for calling `setinputsizes()`, for example.

`TypeEngine.`**`hashable`** = **True**

Flag, if False, means values from this type aren't hashable.

Used by the ORM when uniquing result lists.

`TypeEngine.`**`literal_processor`**(*dialect*)

Return a conversion function for processing literal values that are to be rendered directly without using binds.

This function is used when the compiler makes use of the "literal_binds" flag, typically used in DDL generation as well as in certain scenarios where backends don't accept bound parameters.

New in version 0.9.0.

`TypeEngine.`**`python_type`**

Return the Python type object expected to be returned by instances of this type, if known.

Basically, for those types which enforce a return type, or are known across the board to do such for all common DBAPIs (like `int` for example), will return that type.

If a return type is not defined, raises `NotImplementedError`.

Note that any type also accommodates NULL in SQL which means you can also get back `None` from any type in practice.

`TypeEngine.`**`result_processor`**(*dialect*, *coltype*)

Return a conversion function for processing result row values.

Returns a callable which will receive a result row column value as the sole positional argument and will return a value to return to the user.

If processing is not necessary, the method should return `None`.

Parameters

- **`dialect`** – Dialect instance in use.
- **`coltype`** – DBAPI coltype argument received in cursor.description.

`TypeEngine.`**`should_evaluate_none`** = **False**

If True, the Python constant `None` is considered to be handled explicitly by this type.

The ORM uses this flag to indicate that a positive value of `None` is passed to the column in an INSERT statement, rather than omitting the column from the INSERT statement which has the effect of firing off column-level defaults. It also allows types which have special behavior for Python None, such as a JSON type, to indicate that they'd like to handle the None value explicitly.

To set this flag on an existing type, use the *TypeEngine.evaluates_none()* method.

See also:

TypeEngine.evaluates_none()

New in version 1.1.

`TypeEngine.`**`with_variant`**(*type_*, *dialect_name*)

Produce a new type object that will utilize the given type when applied to the dialect of the given name.

e.g.:

```
from sqlalchemy.types import String
from sqlalchemy.dialects import mysql

s = String()

s = s.with_variant(mysql.VARCHAR(collation='foo'), 'mysql')
```

The construction of *TypeEngine.with_variant()* is always from the "fallback" type to that which is dialect specific. The returned type is an instance of *Variant*, which itself provides a `Variant.with_variant()` that can be called repeatedly.

Parameters

- **type_** – a *TypeEngine* that will be selected as a variant from the originating type, when a dialect of the given name is in use.
- **dialect_name** – base name of the dialect which uses this type. (i.e. `'postgresql'`, `'mysql'`, etc.)

New in version 0.7.2.

class `sqlalchemy.types.`**`Concatenable`**

A mixin that marks a type as supporting 'concatenation', typically strings.

class `sqlalchemy.types.`**`Indexable`**

A mixin that marks a type as supporting indexing operations, such as array or JSON structures.

New in version 1.1.0.

class `sqlalchemy.types.`**`NullType`**

Bases: *sqlalchemy.types.TypeEngine*

An unknown type.

NullType is used as a default type for those cases where a type cannot be determined, including:

- During table reflection, when the type of a column is not recognized by the *Dialect*
- When constructing SQL expressions using plain Python objects of unknown types (e.g. `somecolumn == my_special_object`)
- When a new *Column* is created, and the given type is passed as `None` or is not passed at all.

The *NullType* can be used within SQL expression invocation without issue, it just has no behavior either at the expression construction level or at the bind-parameter/result processing level. *NullType* will result in a *CompileError* if the compiler is asked to render the type itself, such as if it is used in a *cast()* operation or within a schema creation operation such as that invoked by *MetaData.create_all()* or the *CreateTable* construct.

class `sqlalchemy.types.`**`Variant`**(*base*, *mapping*)

Bases: `sqlalchemy.types.TypeDecorator`

A wrapping type that selects among a variety of implementations based on dialect in use.

The `Variant` type is typically constructed using the `TypeEngine.with_variant()` method.

New in version 0.7.2.

See also:

`TypeEngine.with_variant()` for an example of use.

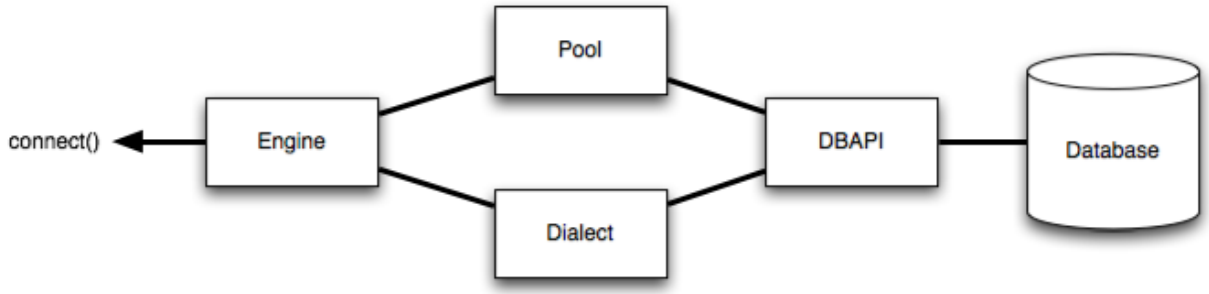
Members with_variant, __init__

3.5 Engine and Connection Use

3.5.1 Engine Configuration

The `Engine` is the starting point for any SQLAlchemy application. It's "home base" for the actual database and its DBAPI, delivered to the SQLAlchemy application through a connection pool and a `Dialect`, which describes how to talk to a specific kind of database/DBAPI combination.

The general structure can be illustrated as follows:



Where above, an `Engine` references both a `Dialect` and a `Pool`, which together interpret the DBAPI's module functions as well as the behavior of the database.

Creating an engine is just a matter of issuing a single call, `create_engine()`:

```
from sqlalchemy import create_engine
engine = create_engine('postgresql://scott:tiger@localhost:5432/mydatabase')
```

The above engine creates a `Dialect` object tailored towards PostgreSQL, as well as a `Pool` object which will establish a DBAPI connection at `localhost:5432` when a connection request is first received. Note that the `Engine` and its underlying `Pool` do **not** establish the first actual DBAPI connection until the `Engine.connect()` method is called, or an operation which is dependent on this method such as `Engine.execute()` is invoked. In this way, `Engine` and `Pool` can be said to have a *lazy initialization* behavior.

The `Engine`, once created, can either be used directly to interact with the database, or can be passed to a `Session` object to work with the ORM. This section covers the details of configuring an `Engine`. The next section, *Working with Engines and Connections*, will detail the usage API of the `Engine` and similar, typically for non-ORM applications.

Supported Databases

SQLAlchemy includes many `Dialect` implementations for various backends. Dialects for the most common databases are included with SQLAlchemy; a handful of others require an additional install of a separate dialect.

See the section *Dialects* for information on the various backends available.

Database Urls

The `create_engine()` function produces an `Engine` object based on a URL. These URLs follow RFC-1738, and usually can include username, password, hostname, database name as well as optional keyword arguments for additional configuration. In some cases a file path is accepted, and in others a "data source name" replaces the "host" and "database" portions. The typical form of a database URL is:

```
dialect+driver://username:password@host:port/database
```

Dialect names include the identifying name of the SQLAlchemy dialect, a name such as `sqlite`, `mysql`, `postgresql`, `oracle`, or `mssql`. The drivername is the name of the DBAPI to be used to connect to the database using all lowercase letters. If not specified, a "default" DBAPI will be imported if available - this default is typically the most widely known driver available for that backend.

Examples for common connection styles follow below. For a full index of detailed information on all included dialects as well as links to third-party dialects, see *Dialects*.

Postgresql

The Postgresql dialect uses psycopg2 as the default DBAPI. pg8000 is also available as a pure-Python substitute:

```
# default
engine = create_engine('postgresql://scott:tiger@localhost/mydatabase')

# psycopg2
engine = create_engine('postgresql+psycopg2://scott:tiger@localhost/mydatabase')

# pg8000
engine = create_engine('postgresql+pg8000://scott:tiger@localhost/mydatabase')
```

More notes on connecting to Postgresql at *PostgreSQL*.

MySQL

The MySQL dialect uses mysql-python as the default DBAPI. There are many MySQL DBAPIs available, including MySQL-connector-python and OurSQL:

```
# default
engine = create_engine('mysql://scott:tiger@localhost/foo')

# mysql-python
engine = create_engine('mysql+mysqldb://scott:tiger@localhost/foo')

# MySQL-connector-python
engine = create_engine('mysql+mysqlconnector://scott:tiger@localhost/foo')
```

```
# OurSQL
engine = create_engine('mysql+oursql://scott:tiger@localhost/foo')
```

More notes on connecting to MySQL at *MySQL*.

Oracle

The Oracle dialect uses cx_oracle as the default DBAPI:

```
engine = create_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')

engine = create_engine('oracle+cx_oracle://scott:tiger@tnsname')
```

More notes on connecting to Oracle at *Oracle*.

Microsoft SQL Server

The SQL Server dialect uses pyodbc as the default DBAPI. pymssql is also available:

```
# pyodbc
engine = create_engine('mssql+pyodbc://scott:tiger@mydsn')

# pymssql
engine = create_engine('mssql+pymssql://scott:tiger@hostname:port/dbname')
```

More notes on connecting to SQL Server at *Microsoft SQL Server*.

SQLite

SQLite connects to file-based databases, using the Python built-in module `sqlite3` by default.

As SQLite connects to local files, the URL format is slightly different. The "file" portion of the URL is the filename of the database. For a relative file path, this requires three slashes:

```
# sqlite://<nohostname>/<path>
# where <path> is relative:
engine = create_engine('sqlite:///foo.db')
```

And for an absolute file path, the three slashes are followed by the absolute path:

```
#Unix/Mac - 4 initial slashes in total
engine = create_engine('sqlite:////absolute/path/to/foo.db')
#Windows
engine = create_engine('sqlite:///C:\\path\\to\\foo.db')
#Windows alternative using raw string
engine = create_engine(r'sqlite:///C:\path\to\foo.db')
```

To use a SQLite `:memory:` database, specify an empty URL:

```
engine = create_engine('sqlite://')
```

More notes on connecting to SQLite at *SQLite*.

Others

See *Dialects*, the top-level page for all additional dialect documentation.

Engine Creation API

`sqlalchemy.create_engine(*args, **kwargs)`

Create a new *Engine* instance.

The standard calling form is to send the URL as the first positional argument, usually a string that indicates database dialect and connection arguments:

```
engine = create_engine("postgresql://scott:tiger@localhost/test")
```

Additional keyword arguments may then follow it which establish various options on the resulting *Engine* and its underlying *Dialect* and *Pool* constructs:

```
engine = create_engine("mysql://scott:tiger@hostname/dbname",
                       encoding='latin1', echo=True)
```

The string form of the URL is `dialect[+driver]://user:password@host/dbname[?key=value..]`, where `dialect` is a database name such as `mysql`, `oracle`, `postgresql`, etc., and `driver` the name of a DBAPI, such as `psycopg2`, `pyodbc`, `cx_oracle`, etc. Alternatively, the URL can be an instance of *URL*.

`**kwargs` takes a wide variety of options which are routed towards their appropriate components. Arguments may be specific to the *Engine*, the underlying *Dialect*, as well as the *Pool*. Specific dialects also accept keyword arguments that are unique to that dialect. Here, we describe the parameters that are common to most *create_engine()* usage.

Once established, the newly resulting *Engine* will request a connection from the underlying *Pool* once *Engine.connect()* is called, or a method which depends on it such as *Engine.execute()* is invoked. The *Pool* in turn will establish the first actual DBAPI connection when this request is received. The *create_engine()* call itself does **not** establish any actual DBAPI connections directly.

See also:

Engine Configuration

Dialects

Working with Engines and Connections

Parameters

- **case_sensitive=True** – if False, result column names will match in a case-insensitive fashion, that is, `row['SomeColumn']`.

  Changed in version 0.8: By default, result row names match case-sensitively. In version 0.7 and prior, all matches were case-insensitive.

- **connect_args** – a dictionary of options which will be passed directly to the DBAPI's `connect()` method as additional keyword arguments. See the example at *Custom DBAPI connect() arguments*.

- **convert_unicode=False** – if set to True, sets the default behavior of `convert_unicode` on the *String* type to `True`, regardless of a setting of `False` on an individual *String* type, thus causing all *String* -based columns to accommodate Python `unicode` objects. This flag is useful as an engine-wide setting when

using a DBAPI that does not natively support Python `unicode` objects and raises an error when one is received (such as pyodbc with FreeTDS).

See *`String`* for further details on what this flag indicates.

- **`creator`** – a callable which returns a DBAPI connection. This creation function will be passed to the underlying connection pool and will be used to create all new database connections. Usage of this function causes connection parameters specified in the URL argument to be bypassed.
- **`echo=False`** – if True, the Engine will log all statements as well as a repr() of their parameter lists to the engines logger, which defaults to sys.stdout. The `echo` attribute of `Engine` can be modified at any time to turn logging on and off. If set to the string `"debug"`, result rows will be printed to the standard output as well. This flag ultimately controls a Python logger; see *Configuring Logging* for information on how to configure logging directly.
- **`echo_pool=False`** – if True, the connection pool will log all checkouts/checkins to the logging stream, which defaults to sys.stdout. This flag ultimately controls a Python logger; see *Configuring Logging* for information on how to configure logging directly.
- **`encoding`** – Defaults to `utf-8`. This is the string encoding used by SQLAlchemy for string encode/decode operations which occur within SQLAlchemy, **outside of the DBAPI.** Most modern DBAPIs feature some degree of direct support for Python `unicode` objects, what you see in Python 2 as a string of the form `u'some string'`. For those scenarios where the DBAPI is detected as not supporting a Python `unicode` object, this encoding is used to determine the source/destination encoding. It is **not used** for those cases where the DBAPI handles unicode directly.

  To properly configure a system to accommodate Python `unicode` objects, the DBAPI should be configured to handle unicode to the greatest degree as is appropriate - see the notes on unicode pertaining to the specific target database in use at *Dialects*.

  Areas where string encoding may need to be accommodated outside of the DBAPI include zero or more of:

  - the values passed to bound parameters, corresponding to the *`Unicode`* type or the *`String`* type when `convert_unicode` is `True`;
  - the values returned in result set columns corresponding to the *`Unicode`* type or the *`String`* type when `convert_unicode` is `True`;
  - the string SQL statement passed to the DBAPI's `cursor.execute()` method;
  - the string names of the keys in the bound parameter dictionary passed to the DBAPI's `cursor.execute()` as well as `cursor.setinputsizes()` methods;
  - the string column names retrieved from the DBAPI's `cursor.description` attribute.

  When using Python 3, the DBAPI is required to support *all* of the above values as Python `unicode` objects, which in Python 3 are just known as `str`. In Python 2, the DBAPI does not specify unicode behavior at all, so SQLAlchemy must make decisions for each of the above values on a per-DBAPI basis - implementations are completely inconsistent in their behavior.
- **`execution_options`** – Dictionary execution options which will be applied to all connections. See *`execution_options()`*

- **implicit_returning=True** – When `True`, a RETURNING- compatible construct, if available, will be used to fetch newly generated primary key values when a single row INSERT statement is emitted with no existing returning() clause. This applies to those backends which support RETURNING or a compatible construct, including Postgresql, Firebird, Oracle, Microsoft SQL Server. Set this to `False` to disable the automatic usage of RETURNING.
- **isolation_level** – this string parameter is interpreted by various dialects in order to affect the transaction isolation level of the database connection. The parameter essentially accepts some subset of these string arguments: `"SERIALIZABLE"`, `"REPEATABLE_READ"`, `"READ_COMMITTED"`, `"READ_UNCOMMITTED"` and `"AUTOCOMMIT"`. Behavior here varies per backend, and individual dialects should be consulted directly.

  Note that the isolation level can also be set on a per-*Connection* basis as well, using the *Connection.execution_options.isolation_level* feature.

  See also:

  Connection.default_isolation_level - view default level

  Connection.execution_options.isolation_level - set per *Connection* isolation level

  SQLite Transaction Isolation

  Postgresql Transaction Isolation

  MySQL Transaction Isolation

  Setting Transaction Isolation Levels - for the ORM
- **label_length=None** – optional integer value which limits the size of dynamically generated column labels to that many characters. If less than 6, labels are generated as "_(counter)". If `None`, the value of `dialect.max_identifier_length` is used instead.
- **listeners** – A list of one or more *PoolListener* objects which will receive connection pool events.
- **logging_name** – String identifier which will be used within the "name" field of logging records generated within the "sqlalchemy.engine" logger. Defaults to a hexstring of the object's id.
- **max_overflow=10** – the number of connections to allow in connection pool "overflow", that is connections that can be opened above and beyond the pool_size setting, which defaults to five. this is only used with *QueuePool*.
- **module=None** – reference to a Python module object (the module itself, not its string name). Specifies an alternate DBAPI module to be used by the engine's dialect. Each sub-dialect references a specific DBAPI which will be imported before first connect. This parameter causes the import to be bypassed, and the given module to be used instead. Can be used for testing of DBAPIs as well as to inject "mock" DBAPI implementations into the *Engine*.
- **paramstyle=None** – The paramstyle to use when rendering bound parameters. This style defaults to the one recommended by the DBAPI itself, which is retrieved from the `.paramstyle` attribute of the DBAPI. However, most DBAPIs accept more than one paramstyle, and in particular it may be desirable to change a "named" paramstyle into a "positional" one, or vice versa. When this attribute is passed, it should be one of the

values `"qmark"`, `"numeric"`, `"named"`, `"format"` or `"pyformat"`, and should correspond to a parameter style known to be supported by the DBAPI in use.

- **pool=None** – an already-constructed instance of *Pool*, such as a *QueuePool* instance. If non-None, this pool will be used directly as the underlying connection pool for the engine, bypassing whatever connection parameters are present in the URL argument. For information on constructing connection pools manually, see *Connection Pooling*.
- **poolclass=None** – a *Pool* subclass, which will be used to create a connection pool instance using the connection parameters given in the URL. Note this differs from `pool` in that you don't actually instantiate the pool in this case, you just indicate what type of pool to be used.
- **pool_logging_name** – String identifier which will be used within the "name" field of logging records generated within the "sqlalchemy.pool" logger. Defaults to a hexstring of the object's id.
- **pool_size=5** – the number of connections to keep open inside the connection pool. This used with *QueuePool* as well as *SingletonThreadPool*. With *QueuePool*, a `pool_size` setting of 0 indicates no limit; to disable pooling, set `poolclass` to *NullPool* instead.
- **pool_recycle=-1** – this setting causes the pool to recycle connections after the given number of seconds has passed. It defaults to -1, or no timeout. For example, setting to 3600 means connections will be recycled after one hour. Note that MySQL in particular will disconnect automatically if no activity is detected on a connection for eight hours (although this is configurable with the MySQLDB connection itself and the server configuration as well).
- **pool_reset_on_return='rollback'** – set the "reset on return" behavior of the pool, which is whether `rollback()`, `commit()`, or nothing is called upon connections being returned to the pool. See the docstring for `reset_on_return` at *Pool*.

  New in version 0.7.6.
- **pool_timeout=30** – number of seconds to wait before giving up on getting a connection from the pool. This is only used with *QueuePool*.
- **strategy='plain'** – selects alternate engine implementations. Currently available are:
  - the `threadlocal` strategy, which is described in *Using the Threadlocal Execution Strategy*;
  - the `mock` strategy, which dispatches all statement execution to a function passed as the argument `executor`. See example in the FAQ.
- **executor=None** – a function taking arguments `(sql, *multiparams, **params)`, to which the `mock` strategy will dispatch all statement execution. Used only by `strategy='mock'`.

`sqlalchemy.`**`engine_from_config`**(*configuration*, *prefix='sqlalchemy.'*, ***kwargs*)

Create a new Engine instance using a configuration dictionary.

The dictionary is typically produced from a config file.

The keys of interest to `engine_from_config()` should be prefixed, e.g. `sqlalchemy.url`, `sqlalchemy.echo`, etc. The 'prefix' argument indicates the prefix to be searched for. Each match-

ing key (after the prefix is stripped) is treated as though it were the corresponding keyword argument to a `create_engine()` call.

The only required key is (assuming the default prefix) `sqlalchemy.url`, which provides the *database URL*.

A select set of keyword arguments will be "coerced" to their expected type based on string values. The set of arguments is extensible per-dialect using the `engine_config_types` accessor.

Parameters

- **configuration** – A dictionary (typically produced from a config file, but this is not a requirement). Items whose keys start with the value of 'prefix' will have that prefix stripped, and will then be passed to create_engine.
- **prefix** – Prefix to match and then strip from keys in 'configuration'.
- **kwargs** – Each keyword argument to `engine_from_config()` itself overrides the corresponding item taken from the 'configuration' dictionary. Keyword arguments should *not* be prefixed.

`sqlalchemy.engine.url.`**`make_url`**(*name_or_url*)

Given a string or unicode instance, produce a new URL instance.

The given string is parsed according to the RFC 1738 spec. If an existing URL object is passed, just returns the object.

class `sqlalchemy.engine.url.`**`URL`**(*drivername*, *username=None*, *password=None*, *host=None*, *port=None*, *database=None*, *query=None*)

Represent the components of a URL used to connect to a database.

This object is suitable to be passed directly to a `create_engine()` call. The fields of the URL are parsed from a string by the `make_url()` function. the string format of the URL is an RFC-1738-style string.

All initialization parameters are available as public attributes.

Parameters

- **drivername** – the name of the database backend. This name will correspond to a module in sqlalchemy/databases or a third party plug-in.
- **username** – The user name.
- **password** – database password.
- **host** – The name of the host.
- **port** – The port number.
- **database** – The database name.
- **query** – A dictionary of options to be passed to the dialect and/or the DBAPI upon connect.

`get_dialect`()

Return the SQLAlchemy database dialect class corresponding to this URL's driver name.

`translate_connect_args`(*names=[]*, ***kw*)

Translate url attributes into a dictionary of connection arguments.

Returns attributes of this url (*host*, *database*, *username*, *password*, *port*) as a plain dictionary. The attribute names are used as the keys by default. Unset or false attributes are omitted from the final dictionary.

Parameters

- ****kw** – Optional, alternate key names for url attributes.
- **names** – Deprecated. Same purpose as the keyword-based alternate names, but correlates the name to the original positionally.

Pooling

The *Engine* will ask the connection pool for a connection when the `connect()` or `execute()` methods are called. The default connection pool, *QueuePool*, will open connections to the database on an as-needed basis. As concurrent statements are executed, *QueuePool* will grow its pool of connections to a default size of five, and will allow a default "overflow" of ten. Since the *Engine* is essentially "home base" for the connection pool, it follows that you should keep a single *Engine* per database established within an application, rather than creating a new one for each connection.

Note: *QueuePool* is not used by default for SQLite engines. See *SQLite* for details on SQLite connection pool usage.

For more information on connection pooling, see *Connection Pooling*.

Custom DBAPI connect() arguments

Custom arguments used when issuing the `connect()` call to the underlying DBAPI may be issued in three distinct ways. String-based arguments can be passed directly from the URL string as query arguments:

```
db = create_engine('postgresql://scott:tiger@localhost/test?argument1=foo&argument2=bar')
```

If SQLAlchemy's database connector is aware of a particular query argument, it may convert its type from string to its proper type.

create_engine() also takes an argument `connect_args` which is an additional dictionary that will be passed to `connect()`. This can be used when arguments of a type other than string are required, and SQLAlchemy's database connector has no type conversion logic present for that parameter:

```
db = create_engine('postgresql://scott:tiger@localhost/test', connect_args = {'argument1':17, 'argume
```

The most customizable connection method of all is to pass a `creator` argument, which specifies a callable that returns a DBAPI connection:

```
def connect():
    return psycopg.connect(user='scott', host='localhost')

db = create_engine('postgresql://', creator=connect)
```

Configuring Logging

Python's standard logging module is used to implement informational and debug log output with SQLAlchemy. This allows SQLAlchemy's logging to integrate in a standard way with other applications and libraries. The `echo` and `echo_pool` flags that are present on *create_engine()*, as well as the `echo_uow` flag used on *Session*, all interact with regular loggers.

This section assumes familiarity with the above linked logging module. All logging performed by SQLAlchemy exists underneath the `sqlalchemy` namespace, as used by `logging.getLogger('sqlalchemy')`. When logging has been configured (i.e. such as via `logging.basicConfig()`), the general namespace of SA loggers that can be turned on is as follows:

- `sqlalchemy.engine` - controls SQL echoing. set to `logging.INFO` for SQL query output, `logging.DEBUG` for query + result set output.
- `sqlalchemy.dialects` - controls custom logging for SQL dialects. See the documentation of individual dialects for details.

- `sqlalchemy.pool` - controls connection pool logging. set to `logging.INFO` or lower to log connection pool checkouts/checkins.
- `sqlalchemy.orm` - controls logging of various ORM functions. set to `logging.INFO` for information on mapper configurations.

For example, to log SQL queries using Python logging instead of the `echo=True` flag:

```
import logging

logging.basicConfig()
logging.getLogger('sqlalchemy.engine').setLevel(logging.INFO)
```

By default, the log level is set to `logging.WARN` within the entire `sqlalchemy` namespace so that no log operations occur, even within an application that has logging enabled otherwise.

The `echo` flags present as keyword arguments to *`create_engine()`* and others as well as the `echo` property on *`Engine`*, when set to `True`, will first attempt to ensure that logging is enabled. Unfortunately, the `logging` module provides no way of determining if output has already been configured (note we are referring to if a logging configuration has been set up, not just that the logging level is set). For this reason, any `echo=True` flags will result in a call to `logging.basicConfig()` using sys.stdout as the destination. It also sets up a default format using the level name, timestamp, and logger name. Note that this configuration has the affect of being configured **in addition** to any existing logger configurations. Therefore, **when using Python logging, ensure all echo flags are set to False at all times**, to avoid getting duplicate log lines.

The logger name of instance such as an *`Engine`* or *`Pool`* defaults to using a truncated hex identifier string. To set this to a specific name, use the "logging_name" and "pool_logging_name" keyword arguments with *`sqlalchemy.create_engine()`*.

Note: The SQLAlchemy *`Engine`* conserves Python function call overhead by only emitting log statements when the current logging level is detected as `logging.INFO` or `logging.DEBUG`. It only checks this level when a new connection is procured from the connection pool. Therefore when changing the logging configuration for an already-running application, any *`Connection`* that's currently active, or more commonly a *`Session`* object that's active in a transaction, won't log any SQL according to the new configuration until a new *`Connection`* is procured (in the case of *`Session`*, this is after the current transaction ends and a new one begins).

3.5.2 Working with Engines and Connections

This section details direct usage of the *`Engine`*, *`Connection`*, and related objects. Its important to note that when using the SQLAlchemy ORM, these objects are not generally accessed; instead, the *`Session`* object is used as the interface to the database. However, for applications that are built around direct usage of textual SQL statements and/or SQL expression constructs without involvement by the ORM's higher level management services, the *`Engine`* and *`Connection`* are king (and queen?) - read on.

Basic Usage

Recall from Engine Configuration that an *`Engine`* is created via the *`create_engine()`* call:

```
engine = create_engine('mysql://scott:tiger@localhost/test')
```

The typical usage of *`create_engine()`* is once per particular database URL, held globally for the lifetime of a single application process. A single *`Engine`* manages many individual DBAPI connections on behalf of the process and is intended to be called upon in a concurrent fashion. The *`Engine`* is **not** synonymous to the DBAPI `connect`

function, which represents just one connection resource - the *Engine* is most efficient when created just once at the module level of an application, not per-object or per-function call.

For a multiple-process application that uses the `os.fork` system call, or for example the Python `multiprocessing` module, it's usually required that a separate *Engine* be used for each child process. This is because the *Engine* maintains a reference to a connection pool that ultimately references DBAPI connections - these tend to not be portable across process boundaries. An *Engine* that is configured not to use pooling (which is achieved via the usage of *NullPool*) does not have this requirement.

The engine can be used directly to issue SQL to the database. The most generic way is first procure a connection resource, which you get via the *Engine.connect()* method:

```
connection = engine.connect()
result = connection.execute("select username from users")
for row in result:
    print "username:", row['username']
connection.close()
```

The connection is an instance of *Connection*, which is a **proxy** object for an actual DBAPI connection. The DBAPI connection is retrieved from the connection pool at the point at which *Connection* is created.

The returned result is an instance of *ResultProxy*, which references a DBAPI cursor and provides a largely compatible interface with that of the DBAPI cursor. The DBAPI cursor will be closed by the *ResultProxy* when all of its result rows (if any) are exhausted. A *ResultProxy* that returns no rows, such as that of an UPDATE statement (without any returned rows), releases cursor resources immediately upon construction.

When the *close()* method is called, the referenced DBAPI connection is released to the connection pool. From the perspective of the database itself, nothing is actually "closed", assuming pooling is in use. The pooling mechanism issues a `rollback()` call on the DBAPI connection so that any transactional state or locks are removed, and the connection is ready for its next usage.

The above procedure can be performed in a shorthand way by using the *execute()* method of *Engine* itself:

```
result = engine.execute("select username from users")
for row in result:
    print "username:", row['username']
```

Where above, the *execute()* method acquires a new *Connection* on its own, executes the statement with that object, and returns the *ResultProxy*. In this case, the *ResultProxy* contains a special flag known as `close_with_result`, which indicates that when its underlying DBAPI cursor is closed, the *Connection* object itself is also closed, which again returns the DBAPI connection to the connection pool, releasing transactional resources.

If the *ResultProxy* potentially has rows remaining, it can be instructed to close out its resources explicitly:

```
result.close()
```

If the *ResultProxy* has pending rows remaining and is dereferenced by the application without being closed, Python garbage collection will ultimately close out the cursor as well as trigger a return of the pooled DBAPI connection resource to the pool (SQLAlchemy achieves this by the usage of weakref callbacks - *never* the `__del__` method) - however it's never a good idea to rely upon Python garbage collection to manage resources.

Our example above illustrated the execution of a textual SQL string. The *execute()* method can of course accommodate more than that, including the variety of SQL expression constructs described in *SQL Expression Language Tutorial*.

Using Transactions

Note: This section describes how to use transactions when working directly with *Engine* and *Connection* objects. When using the SQLAlchemy ORM, the public API for transaction control is via the *Session* object, which makes usage of the *Transaction* object internally. See *Managing Transactions* for further information.

The *Connection* object provides a *begin()* method which returns a *Transaction* object. This object is usually used within a try/except clause so that it is guaranteed to invoke *Transaction.rollback()* or *Transaction.commit()*:

```
connection = engine.connect()
trans = connection.begin()
try:
    r1 = connection.execute(table1.select())
    connection.execute(table1.insert(), col1=7, col2='this is some data')
    trans.commit()
except:
    trans.rollback()
    raise
```

The above block can be created more succinctly using context managers, either given an *Engine*:

```
# runs a transaction
with engine.begin() as connection:
    r1 = connection.execute(table1.select())
    connection.execute(table1.insert(), col1=7, col2='this is some data')
```

Or from the *Connection*, in which case the *Transaction* object is available as well:

```
with connection.begin() as trans:
    r1 = connection.execute(table1.select())
    connection.execute(table1.insert(), col1=7, col2='this is some data')
```

Nesting of Transaction Blocks

The *Transaction* object also handles "nested" behavior by keeping track of the outermost begin/commit pair. In this example, two functions both issue a transaction on a *Connection*, but only the outermost *Transaction* object actually takes effect when it is committed.

```
# method_a starts a transaction and calls method_b
def method_a(connection):
    trans = connection.begin() # open a transaction
    try:
        method_b(connection)
        trans.commit()  # transaction is committed here
    except:
        trans.rollback() # this rolls back the transaction unconditionally
        raise

# method_b also starts a transaction
def method_b(connection):
    trans = connection.begin() # open a transaction - this runs in the context of method_a's transac
    try:
        connection.execute("insert into mytable values ('bat', 'lala')")
```

```
        connection.execute(mytable.insert(), col1='bat', col2='lala')
        trans.commit()  # transaction is not committed yet
    except:
        trans.rollback() # this rolls back the transaction unconditionally
        raise

# open a Connection and call method_a
conn = engine.connect()
method_a(conn)
conn.close()
```

Above, `method_a` is called first, which calls `connection.begin()`. Then it calls `method_b`. When `method_b` calls `connection.begin()`, it just increments a counter that is decremented when it calls `commit()`. If either `method_a` or `method_b` calls `rollback()`, the whole transaction is rolled back. The transaction is not committed until `method_a` calls the `commit()` method. This "nesting" behavior allows the creation of functions which "guarantee" that a transaction will be used if one was not already available, but will automatically participate in an enclosing transaction if one exists.

Understanding Autocommit

The previous transaction example illustrates how to use *Transaction* so that several executions can take part in the same transaction. What happens when we issue an INSERT, UPDATE or DELETE call without using *Transaction*? While some DBAPI implementations provide various special "non-transactional" modes, the core behavior of DBAPI per PEP-0249 is that a *transaction is always in progress*, providing only `rollback()` and `commit()` methods but no `begin()`. SQLAlchemy assumes this is the case for any given DBAPI.

Given this requirement, SQLAlchemy implements its own "autocommit" feature which works completely consistently across all backends. This is achieved by detecting statements which represent data-changing operations, i.e. INSERT, UPDATE, DELETE, as well as data definition language (DDL) statements such as CREATE TABLE, ALTER TABLE, and then issuing a COMMIT automatically if no transaction is in progress. The detection is based on the presence of the `autocommit=True` execution option on the statement. If the statement is a text-only statement and the flag is not set, a regular expression is used to detect INSERT, UPDATE, DELETE, as well as a variety of other commands for a particular backend:

```
conn = engine.connect()
conn.execute("INSERT INTO users VALUES (1, 'john')")  # autocommits
```

The "autocommit" feature is only in effect when no *Transaction* has otherwise been declared. This means the feature is not generally used with the ORM, as the *Session* object by default always maintains an ongoing *Transaction*.

Full control of the "autocommit" behavior is available using the generative *Connection.execution_options()* method provided on *Connection*, *Engine*, *Executable*, using the "autocommit" flag which will turn on or off the autocommit for the selected scope. For example, a *text()* construct representing a stored procedure that commits might use it so that a SELECT statement will issue a COMMIT:

```
engine.execute(text("SELECT my_mutating_procedure()").execution_options(autocommit=True))
```

Connectionless Execution, Implicit Execution

Recall from the first section we mentioned executing with and without explicit usage of *Connection*. "Connectionless" execution refers to the usage of the `execute()` method on an object which is not a *Connection*. This was illustrated using the *execute()* method of *Engine*:

```
result = engine.execute("select username from users")
for row in result:
    print "username:", row['username']
```

In addition to "connectionless" execution, it is also possible to use the *execute()* method of any *Executable* construct, which is a marker for SQL expression objects that support execution. The SQL expression object itself references an *Engine* or *Connection* known as the **bind**, which it uses in order to provide so-called "implicit" execution services.

Given a table as below:

```
from sqlalchemy import MetaData, Table, Column, Integer

meta = MetaData()
users_table = Table('users', meta,
    Column('id', Integer, primary_key=True),
    Column('name', String(50))
)
```

Explicit execution delivers the SQL text or constructed SQL expression to the *execute()* method of *Connection*:

```
engine = create_engine('sqlite:///file.db')
connection = engine.connect()
result = connection.execute(users_table.select())
for row in result:
    # ....
connection.close()
```

Explicit, connectionless execution delivers the expression to the *execute()* method of *Engine*:

```
engine = create_engine('sqlite:///file.db')
result = engine.execute(users_table.select())
for row in result:
    # ....
result.close()
```

Implicit execution is also connectionless, and makes usage of the *execute()* method on the expression itself. This method is provided as part of the *Executable* class, which refers to a SQL statement that is sufficient for being invoked against the database. The method makes usage of the assumption that either an *Engine* or *Connection* has been **bound** to the expression object. By "bound" we mean that the special attribute *MetaData.bind* has been used to associate a series of *Table* objects and all SQL constructs derived from them with a specific engine:

```
engine = create_engine('sqlite:///file.db')
meta.bind = engine
result = users_table.select().execute()
for row in result:
    # ....
result.close()
```

Above, we associate an *Engine* with a *MetaData* object using the special attribute *MetaData.bind*. The *select()* construct produced from the *Table* object has a method *execute()*, which will search for an *Engine* that's "bound" to the *Table*.

Overall, the usage of "bound metadata" has three general effects:

- SQL statement objects gain an *Executable.execute()* method which automatically locates a "bind" with which to execute themselves.

- The ORM *Session* object supports using "bound metadata" in order to establish which *Engine* should be used to invoke SQL statements on behalf of a particular mapped class, though the *Session* also features its own explicit system of establishing complex *Engine*/ mapped class configurations.
- The *MetaData.create_all()*, *MetaData.drop_all()*, *Table.create()*, *Table.drop()*, and "autoload" features all make usage of the bound *Engine* automatically without the need to pass it explicitly.

Note: The concepts of "bound metadata" and "implicit execution" are not emphasized in modern SQLAlchemy. While they offer some convenience, they are no longer required by any API and are never necessary.

In applications where multiple *Engine* objects are present, each one logically associated with a certain set of tables (i.e. *vertical sharding*), the "bound metadata" technique can be used so that individual *Table* can refer to the appropriate *Engine* automatically; in particular this is supported within the ORM via the *Session* object as a means to associate *Table* objects with an appropriate *Engine*, as an alternative to using the bind arguments accepted directly by the *Session*.

However, the "implicit execution" technique is not at all appropriate for use with the ORM, as it bypasses the transactional context maintained by the *Session*.

Overall, in the *vast majority* of cases, "bound metadata" and "implicit execution" are **not useful**. While "bound metadata" has a marginal level of usefulness with regards to ORM configuration, "implicit execution" is a very old usage pattern that in most cases is more confusing than it is helpful, and its usage is discouraged. Both patterns seem to encourage the overuse of expedient "short cuts" in application design which lead to problems later on.

Modern SQLAlchemy usage, especially the ORM, places a heavy stress on working within the context of a transaction at all times; the "implicit execution" concept makes the job of associating statement execution with a particular transaction much more difficult. The *Executable.execute()* method on a particular SQL statement usually implies that the execution is not part of any particular transaction, which is usually not the desired effect.

In both "connectionless" examples, the *Connection* is created behind the scenes; the *ResultProxy* returned by the `execute()` call references the *Connection* used to issue the SQL statement. When the *ResultProxy* is closed, the underlying *Connection* is closed for us, resulting in the DBAPI connection being returned to the pool with transactional resources removed.

Translation of Schema Names

To support multi-tenancy applications that distribute common sets of tables into multiple schemas, the *Connection.execution_options.schema_translate_map* execution option may be used to repurpose a set of *Table* objects to render under different schema names without any changes.

Given a table:

```
user_table = Table(
    'user', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String(50))
)
```

The "schema" of this *Table* as defined by the *Table.schema* attribute is `None`. The *Connection.execution_options.schema_translate_map* can specify that all *Table* objects with a schema of `None` would instead render the schema as `user_schema_one`:

```
connection = engine.connect().execution_options(
    schema_translate_map={None: "user_schema_one"})

result = connection.execute(user_table.select())
```

The above code will invoke SQL on the database of the form:

```
SELECT user_schema_one.user.id, user_schema_one.user.name FROM
user_schema.user
```

That is, the schema name is substituted with our translated name. The map can specify any number of target->destination schemas:

```
connection = engine.connect().execution_options(
    schema_translate_map={
        None: "user_schema_one",      # no schema name -> "user_schema_one"
        "special": "special_schema", # schema="special" becomes "special_schema"
        "public": None               # Table objects with schema="public" will render with no schema
    })
```

The *Connection.execution_options.schema_translate_map* parameter affects all DDL and SQL constructs generated from the SQL expression language, as derived from the *Table* or *Sequence* objects. It does **not** impact literal string SQL used via the *expression.text()* construct nor via plain strings passed to *Connection.execute()*.

The feature takes effect **only** in those cases where the name of the schema is derived directly from that of a *Table* or *Sequence*; it does not impact methods where a string schema name is passed directly. By this pattern, it takes effect within the "can create" / "can drop" checks performed by methods such as *MetaData.create_all()* or *MetaData.drop_all()* are called, and it takes effect when using table reflection given a *Table* object. However it does **not** affect the operations present on the *Inspector* object, as the schema name is passed to these methods explicitly.

New in version 1.1.

Engine Disposal

The *Engine* refers to a connection pool, which means under normal circumstances, there are open database connections present while the *Engine* object is still resident in memory. When an *Engine* is garbage collected, its connection pool is no longer referred to by that *Engine*, and assuming none of its connections are still checked out, the pool and its connections will also be garbage collected, which has the effect of closing out the actual database connections as well. But otherwise, the *Engine* will hold onto open database connections assuming it uses the normally default pool implementation of *QueuePool*.

The *Engine* is intended to normally be a permanent fixture established up-front and maintained throughout the lifespan of an application. It is **not** intended to be created and disposed on a per-connection basis; it is instead a registry that maintains both a pool of connections as well as configurational information about the database and DBAPI in use, as well as some degree of internal caching of per-database resources.

However, there are many cases where it is desirable that all connection resources referred to by the *Engine* be completely closed out. It's generally not a good idea to rely on Python garbage collection for this to occur for these cases; instead, the *Engine* can be explicitly disposed using the *Engine.dispose()* method. This disposes of the engine's underlying connection pool and replaces it with a new one that's empty. Provided that the *Engine* is discarded at this point and no longer used, all **checked-in** connections which it refers to will also be fully closed.

Valid use cases for calling *Engine.dispose()* include:

- When a program wants to release any remaining checked-in connections held by the connection pool and expects to no longer be connected to that database at all for any future operations.
- When a program uses multiprocessing or `fork()`, and an *Engine* object is copied to the child process, *Engine.dispose()* should be called so that the engine creates brand new database connections local to that fork. Database connections generally do **not** travel across process boundaries.

- Within test suites or multitenancy scenarios where many ad-hoc, short-lived *Engine* objects may be created and disposed.

Connections that are **checked out** are **not** discarded when the engine is disposed or garbage collected, as these connections are still strongly referenced elsewhere by the application. However, after *Engine.dispose()* is called, those connections are no longer associated with that *Engine*; when they are closed, they will be returned to their now-orphaned connection pool which will ultimately be garbage collected, once all connections which refer to it are also no longer referenced anywhere. Since this process is not easy to control, it is strongly recommended that *Engine.dispose()* is called only after all checked out connections are checked in or otherwise de-associated from their pool.

An alternative for applications that are negatively impacted by the *Engine* object's use of connection pooling is to disable pooling entirely. This typically incurs only a modest performance impact upon the use of new connections, and means that when a connection is checked in, it is entirely closed out and is not held in memory. See *Switching Pool Implementations* for guidelines on how to disable pooling.

Using the Threadlocal Execution Strategy

The "threadlocal" engine strategy is an optional feature which can be used by non-ORM applications to associate transactions with the current thread, such that all parts of the application can participate in that transaction implicitly without the need to explicitly reference a *Connection*.

Note: The "threadlocal" feature is generally discouraged. It's designed for a particular pattern of usage which is generally considered as a legacy pattern. It has **no impact** on the "thread safety" of SQLAlchemy components or one's application. It also should not be used when using an ORM *Session* object, as the *Session* itself represents an ongoing transaction and itself handles the job of maintaining connection and transactional resources.

Enabling `threadlocal` is achieved as follows:

```
db = create_engine('mysql://localhost/test', strategy='threadlocal')
```

The above *Engine* will now acquire a *Connection* using connection resources derived from a thread-local variable whenever *Engine.execute()* or *Engine.contextual_connect()* is called. This connection resource is maintained as long as it is referenced, which allows multiple points of an application to share a transaction while using connectionless execution:

```
def call_operation1():
    engine.execute("insert into users values (?, ?)", 1, "john")

def call_operation2():
    users.update(users.c.user_id==5).execute(name='ed')

db.begin()
try:
    call_operation1()
    call_operation2()
    db.commit()
except:
    db.rollback()
```

Explicit execution can be mixed with connectionless execution by using the *Engine.connect()* method to acquire a *Connection* that is not part of the threadlocal scope:

```
db.begin()
conn = db.connect()
```

```
try:
    conn.execute(log_table.insert(), message="Operation started")
    call_operation1()
    call_operation2()
    db.commit()
    conn.execute(log_table.insert(), message="Operation succeeded")
except:
    db.rollback()
    conn.execute(log_table.insert(), message="Operation failed")
finally:
    conn.close()
```

To access the *Connection* that is bound to the threadlocal scope, call *Engine.contextual_connect()*:

```
conn = db.contextual_connect()
call_operation3(conn)
conn.close()
```

Calling *close()* on the "contextual" connection does not release its resources until all other usages of that resource are closed as well, including that any ongoing transactions are rolled back or committed.

Working with Raw DBAPI Connections

There are some cases where SQLAlchemy does not provide a genericized way at accessing some DBAPI functions, such as calling stored procedures as well as dealing with multiple result sets. In these cases, it's just as expedient to deal with the raw DBAPI connection directly.

The most common way to access the raw DBAPI connection is to get it from an already present *Connection* object directly. It is present using the *Connection.connection* attribute:

```
connection = engine.connect()
dbapi_conn = connection.connection
```

The DBAPI connection here is actually a "proxied" in terms of the originating connection pool, however this is an implementation detail that in most cases can be ignored. As this DBAPI connection is still contained within the scope of an owning *Connection* object, it is best to make use of the *Connection* object for most features such as transaction control as well as calling the *Connection.close()* method; if these operations are performed on the DBAPI connection directly, the owning *Connection* will not be aware of these changes in state.

To overcome the limitations imposed by the DBAPI connection that is maintained by an owning *Connection*, a DBAPI connection is also available without the need to procure a *Connection* first, using the *Engine.raw_connection()* method of *Engine*:

```
dbapi_conn = engine.raw_connection()
```

This DBAPI connection is again a "proxied" form as was the case before. The purpose of this proxying is now apparent, as when we call the `.close()` method of this connection, the DBAPI connection is typically not actually closed, but instead released back to the engine's connection pool:

```
dbapi_conn.close()
```

While SQLAlchemy may in the future add built-in patterns for more DBAPI use cases, there are diminishing returns as these cases tend to be rarely needed and they also vary highly dependent on the type of DBAPI in use, so in any case the direct DBAPI calling pattern is always there for those cases where it is needed.

Some recipes for DBAPI connection use follow.

Calling Stored Procedures

For stored procedures with special syntactical or parameter concerns, DBAPI-level callproc may be used:

```
connection = engine.raw_connection()
try:
    cursor = connection.cursor()
    cursor.callproc("my_procedure", ['x', 'y', 'z'])
    results = list(cursor.fetchall())
    cursor.close()
    connection.commit()
finally:
    connection.close()
```

Multiple Result Sets

Multiple result set support is available from a raw DBAPI cursor using the nextset method:

```
connection = engine.raw_connection()
try:
    cursor = connection.cursor()
    cursor.execute("select * from table1; select * from table2")
    results_one = cursor.fetchall()
    cursor.nextset()
    results_two = cursor.fetchall()
    cursor.close()
finally:
    connection.close()
```

Registering New Dialects

The *create_engine()* function call locates the given dialect using setuptools entrypoints. These entry points can be established for third party dialects within the setup.py script. For example, to create a new dialect "foodialect://", the steps are as follows:

1. Create a package called `foodialect`.
2. The package should have a module containing the dialect class, which is typically a subclass of *sqlalchemy.engine.default.DefaultDialect*. In this example let's say it's called `FooDialect` and its module is accessed via `foodialect.dialect`.
3. The entry point can be established in setup.py as follows:

```
    entry_points="""
    [sqlalchemy.dialects]
    foodialect = foodialect.dialect:FooDialect
    """
```

If the dialect is providing support for a particular DBAPI on top of an existing SQLAlchemy-supported database, the name can be given including a database-qualification. For example, if `FooDialect` were in fact a MySQL dialect, the entry point could be established like this:

```
entry_points="""
[sqlalchemy.dialects]
mysql.foodialect = foodialect.dialect:FooDialect
"""
```

The above entrypoint would then be accessed as `create_engine("mysql+foodialect://")`.

Registering Dialects In-Process

SQLAlchemy also allows a dialect to be registered within the current process, bypassing the need for separate installation. Use the `register()` function as follows:

```
from sqlalchemy.dialects import registry
registry.register("mysql.foodialect", "myapp.dialect", "MyMySQLDialect")
```

The above will respond to `create_engine("mysql+foodialect://")` and load the `MyMySQLDialect` class from the `myapp.dialect` module.

New in version 0.8.

Connection / Engine API

class `sqlalchemy.engine.`**`Connection`**(*engine*, *connection=None*, *close_with_result=False*, *_branch_from=None*, *_execution_options=None*, *_dispatch=None*, *_has_events=None*)

Bases: *`sqlalchemy.engine.Connectable`*

Provides high-level functionality for a wrapped DB-API connection.

Provides execution support for string-based SQL statements as well as *`ClauseElement`*, *`Compiled`* and *`DefaultGenerator`* objects. Provides a *`begin()`* method to return *`Transaction`* objects.

The Connection object is **not** thread-safe. While a Connection can be shared among threads using properly synchronized access, it is still possible that the underlying DBAPI connection may not support shared access between threads. Check the DBAPI documentation for details.

The Connection object represents a single dbapi connection checked out from the connection pool. In this state, the connection pool has no affect upon the connection, including its expiration or timeout state. For the connection pool to properly manage connections, connections should be returned to the connection pool (i.e. `connection.close()`) whenever the connection is not in use.

`__init__`(*engine*, *connection=None*, *close_with_result=False*, *_branch_from=None*, *_execution_options=None*, *_dispatch=None*, *_has_events=None*)

Construct a new Connection.

The constructor here is not public and is only called only by an *`Engine`*. See *`Engine.connect()`* and *`Engine.contextual_connect()`* methods.

`begin`()

Begin a transaction and return a transaction handle.

The returned object is an instance of *`Transaction`*. This object represents the "scope" of the transaction, which completes when either the *`Transaction.rollback()`* or *`Transaction.commit()`* method is called.

Nested calls to *`begin()`* on the same *`Connection`* will return new *`Transaction`* objects that represent an emulated transaction within the scope of the enclosing transaction, that is:

```
trans = conn.begin()    # outermost transaction
trans2 = conn.begin()   # "nested"
trans2.commit()         # does nothing
trans.commit()          # actually commits
```

Calls to `Transaction.commit()` only have an effect when invoked via the outermost `Transaction` object, though the `Transaction.rollback()` method of any of the `Transaction` objects will roll back the transaction.

See also:

`Connection.begin_nested()` - use a SAVEPOINT

`Connection.begin_twophase()` - use a two phase /XID transaction

`Engine.begin()` - context manager available from `Engine`.

begin_nested()

Begin a nested transaction and return a transaction handle.

The returned object is an instance of `NestedTransaction`.

Nested transactions require SAVEPOINT support in the underlying database. Any transaction in the hierarchy may `commit` and `rollback`, however the outermost transaction still controls the overall `commit` or `rollback` of the transaction of a whole.

See also `Connection.begin()`, `Connection.begin_twophase()`.

begin_twophase(*xid=None*)

Begin a two-phase or XA transaction and return a transaction handle.

The returned object is an instance of `TwoPhaseTransaction`, which in addition to the methods provided by `Transaction`, also provides a `prepare()` method.

Parameters **xid** – the two phase transaction id. If not supplied, a random id will be generated.

See also `Connection.begin()`, `Connection.begin_twophase()`.

close()

Close this `Connection`.

This results in a release of the underlying database resources, that is, the DBAPI connection referenced internally. The DBAPI connection is typically restored back to the connection-holding `Pool` referenced by the `Engine` that produced this `Connection`. Any transactional state present on the DBAPI connection is also unconditionally released via the DBAPI connection's `rollback()` method, regardless of any `Transaction` object that may be outstanding with regards to this `Connection`.

After `close()` is called, the `Connection` is permanently in a closed state, and will allow no further operations.

closed

Return True if this connection is closed.

connect()

Returns a branched version of this `Connection`.

The `Connection.close()` method on the returned `Connection` can be called and this `Connection` will remain open.

This method provides usage symmetry with `Engine.connect()`, including for usage with context managers.

connection

The underlying DB-API connection managed by this Connection.

See also:

Working with Raw DBAPI Connections

contextual_connect(***kwargs*)

Returns a branched version of this `Connection`.

The `Connection.close()` method on the returned `Connection` can be called and this `Connection` will remain open.

This method provides usage symmetry with `Engine.contextual_connect()`, including for usage with context managers.

default_isolation_level

The default isolation level assigned to this `Connection`.

This is the isolation level setting that the `Connection` has when first procured via the `Engine.connect()` method. This level stays in place until the `Connection.execution_options.isolation_level` is used to change the setting on a per-`Connection` basis.

Unlike `Connection.get_isolation_level()`, this attribute is set ahead of time from the first connection procured by the dialect, so SQL query is not invoked when this accessor is called.

New in version 0.9.9.

See also:

`Connection.get_isolation_level()` - view current level

`create_engine.isolation_level` - set per `Engine` isolation level

`Connection.execution_options.isolation_level` - set per `Connection` isolation level

detach()

Detach the underlying DB-API connection from its connection pool.

E.g.:

```
with engine.connect() as conn:
    conn.detach()
    conn.execute("SET search_path TO schema1, schema2")

    # work with connection

# connection is fully closed (since we used "with:", can
# also call .close())
```

This `Connection` instance will remain usable. When closed (or exited from a context manager context as above), the DB-API connection will be literally closed and not returned to its originating pool.

This method can be used to insulate the rest of an application from a modified state on a connection (such as a transaction isolation level or similar).

execute(*object*, **multiparams*, ***params*)

Executes the a SQL statement construct and returns a `ResultProxy`.

Parameters

- **object** – The statement to be executed. May be one of:
  - a plain string

- any `ClauseElement` construct that is also a subclass of `Executable`, such as a `select()` construct
- a `FunctionElement`, such as that generated by `func`, will be automatically wrapped in a SELECT statement, which is then executed.
- a `DDLElement` object
- a `DefaultGenerator` object
- a `Compiled` object

- **`*multiparams/**params`** – represent bound parameter values to be used in the execution. Typically, the format is either a collection of one or more dictionaries passed to *multiparams:

```
conn.execute(
    table.insert(),
    {"id":1, "value":"v1"},
    {"id":2, "value":"v2"}
)
```

...or individual key/values interpreted by **params:

```
conn.execute(
    table.insert(), id=1, value="v1"
)
```

In the case that a plain SQL string is passed, and the underlying DBAPI accepts positional bind parameters, a collection of tuples or individual values in *multiparams may be passed:

```
conn.execute(
    "INSERT INTO table (id, value) VALUES (?, ?)",
    (1, "v1"), (2, "v2")
)

conn.execute(
    "INSERT INTO table (id, value) VALUES (?, ?)",
    1, "v1"
)
```

Note above, the usage of a question mark "?" or other symbol is contingent upon the "paramstyle" accepted by the DBAPI in use, which may be any of "qmark", "named", "pyformat", "format", "numeric". See pep-249 for details on paramstyle.

To execute a textual SQL statement which uses bound parameters in a DBAPI-agnostic way, use the `text()` construct.

`execution_options`(***opt*)

Set non-SQL options for the connection which take effect during execution.

The method returns a copy of this `Connection` which references the same underlying DBAPI connection, but also defines the given execution options which will take effect for a call to `execute()`. As the new `Connection` references the same underlying resource, it's usually a good idea to ensure that the copies will be discarded immediately, which is implicit if used as in:

```
result = connection.execution_options(stream_results=True).\
                    execute(stmt)
```

Note that any key/value can be passed to `Connection.execution_options()`, and it will be stored in the `_execution_options` dictionary of the `Connection`. It is suitable for usage by end-user schemes to communicate with event listeners, for example.

The keywords that are currently recognized by SQLAlchemy itself include all those listed under `Executable.execution_options()`, as well as others that are specific to `Connection`.

Parameters

- **`autocommit`** – Available on: Connection, statement. When True, a COMMIT will be invoked after execution when executed in 'autocommit' mode, i.e. when an explicit transaction is not begun on the connection. Note that DBAPI connections by default are always in a transaction - SQLAlchemy uses rules applied to different kinds of statements to determine if COMMIT will be invoked in order to provide its "autocommit" feature. Typically, all INSERT/UPDATE/DELETE statements as well as CREATE/DROP statements have autocommit behavior enabled; SELECT constructs do not. Use this option when invoking a SELECT or other specific SQL construct where COMMIT is desired (typically when calling stored procedures and such), and an explicit transaction is not in progress.

- **`compiled_cache`** – Available on: Connection. A dictionary where `Compiled` objects will be cached when the `Connection` compiles a clause expression into a `Compiled` object. It is the user's responsibility to manage the size of this dictionary, which will have keys corresponding to the dialect, clause element, the column names within the VALUES or SET clause of an INSERT or UPDATE, as well as the "batch" mode for an INSERT or UPDATE statement. The format of this dictionary is not guaranteed to stay the same in future releases.

  Note that the ORM makes use of its own "compiled" caches for some operations, including flush operations. The caching used by the ORM internally supersedes a cache dictionary specified here.

- **`isolation_level`** – Available on: `Connection`. Set the transaction isolation level for the lifespan of this `Connection` object (*not* the underyling DBAPI connection, for which the level is reset to its original setting upon termination of this `Connection` object).

  Valid values include those string values accepted by the `create_engine.isolation_level` parameter passed to `create_engine()`. These levels are semi-database specific; see individual dialect documentation for valid levels.

  Note that this option necessarily affects the underlying DBAPI connection for the lifespan of the originating `Connection`, and is not per-execution. This setting is not removed until the underlying DBAPI connection is returned to the connection pool, i.e. the `Connection.close()` method is called.

  > **Warning:** The `isolation_level` execution option should **not** be used when a transaction is already established, that is, the `Connection.begin()` method or similar has been called. A database cannot change the isolation level on a transaction in progress, and different DBAPIs and/or SQLAlchemy dialects may implicitly roll back or commit the transaction, or not affect the connection at all.
  > Changed in version 0.9.9: A warning is emitted when the `isolation_level` execution option is used after a transaction has been started with `Connection.begin()` or similar.

Note: The `isolation_level` execution option is implicitly reset if the `Connection` is invalidated, e.g. via the `Connection.invalidate()` method, or if a disconnection error occurs. The new connection produced after the invalidation will not have the isolation level re-applied to it automatically.

See also:

`create_engine.isolation_level` - set per `Engine` isolation level

`Connection.get_isolation_level()` - view current level

SQLite Transaction Isolation

Postgresql Transaction Isolation

MySQL Transaction Isolation

Setting Transaction Isolation Levels - for the ORM

- **no_parameters** – When `True`, if the final parameter list or dictionary is totally empty, will invoke the statement on the cursor as `cursor.execute(statement)`, not passing the parameter collection at all. Some DBAPIs such as psycopg2 and mysql-python consider percent signs as significant only when parameters are present; this option allows code to generate SQL containing percent signs (and possibly other characters) that is neutral regarding whether it's executed by the DBAPI or piped into a script that's later invoked by command line tools.

  New in version 0.7.6.

- **stream_results** – Available on: Connection, statement. Indicate to the dialect that results should be "streamed" and not pre-buffered, if possible. This is a limitation of many DBAPIs. The flag is currently understood only by the psycopg2 dialect.

- **schema_translate_map** – Available on: Connection, Engine. A dictionary mapping schema names to schema names, that will be applied to the `Table.schema` element of each `Table` encountered when SQL or DDL expression elements are compiled into strings; the resulting schema name will be converted based on presence in the map of the original name.

  New in version 1.1.

  See also:

  Translation of Schema Names

get_isolation_level()

Return the current isolation level assigned to this `Connection`.

This will typically be the default isolation level as determined by the dialect, unless if the `Connection.execution_options.isolation_level` feature has been used to alter the isolation level on a per-`Connection` basis.

This attribute will typically perform a live SQL operation in order to procure the current isolation level, so the value returned is the actual level on the underlying DBAPI connection regardless of how this state was set. Compare to the `Connection.default_isolation_level` accessor which returns the dialect-level setting without performing a SQL query.

New in version 0.9.9.

See also:

`Connection.default_isolation_level` - view default level

`create_engine.isolation_level` - set per `Engine` isolation level

`Connection.execution_options.isolation_level` - set per `Connection` isolation level

in_transaction()

Return True if a transaction is in progress.

info

Info dictionary associated with the underlying DBAPI connection referred to by this `Connection`, allowing user-defined data to be associated with the connection.

The data here will follow along with the DBAPI connection including after it is returned to the connection pool and used again in subsequent instances of `Connection`.

invalidate(*exception=None*)

Invalidate the underlying DBAPI connection associated with this `Connection`.

The underlying DBAPI connection is literally closed (if possible), and is discarded. Its source connection pool will typically lazily create a new connection to replace it.

Upon the next use (where "use" typically means using the `Connection.execute()` method or similar), this `Connection` will attempt to procure a new DBAPI connection using the services of the `Pool` as a source of connectivty (e.g. a "reconnection").

If a transaction was in progress (e.g. the `Connection.begin()` method has been called) when `Connection.invalidate()` method is called, at the DBAPI level all state associated with this transaction is lost, as the DBAPI connection is closed. The `Connection` will not allow a reconnection to proceed until the `Transaction` object is ended, by calling the `Transaction.rollback()` method; until that point, any attempt at continuing to use the `Connection` will raise an `InvalidRequestError`. This is to prevent applications from accidentally continuing an ongoing transactional operations despite the fact that the transaction has been lost due to an invalidation.

The `Connection.invalidate()` method, just like auto-invalidation, will at the connection pool level invoke the `PoolEvents.invalidate()` event.

See also:

More on Invalidation

invalidated

Return True if this connection was invalidated.

run_callable(*callable_*, **args*, ***kwargs*)

Given a callable object or function, execute it, passing a `Connection` as the first argument.

The given *args and **kwargs are passed subsequent to the `Connection` argument.

This function, along with `Engine.run_callable()`, allows a function to be run with a `Connection` or `Engine` object without the need to know which one is being dealt with.

scalar(*object*, **multiparams*, ***params*)

Executes and returns the first column of the first row.

The underlying result/cursor is closed after execution.

schema_for_object = <sqlalchemy.sql.schema._SchemaTranslateMap object>

Return the ".schema" attribute for an object.

Used for `Table`, `Sequence` and similar objects, and takes into account the `Connection.execution_options.schema_translate_map` parameter.

New in version 1.1.

See also:

Translation of Schema Names

transaction(*callable_*, **args*, ***kwargs*)

Execute the given function within a transaction boundary.

The function is passed this `Connection` as the first argument, followed by the given *args and **kwargs, e.g.:

```
def do_something(conn, x, y):
    conn.execute("some statement", {'x':x, 'y':y})

conn.transaction(do_something, 5, 10)
```

The operations inside the function are all invoked within the context of a single `Transaction`. Upon success, the transaction is committed. If an exception is raised, the transaction is rolled back before propagating the exception.

Note: The `transaction()` method is superseded by the usage of the Python `with:` statement, which can be used with `Connection.begin()`:

```
with conn.begin():
    conn.execute("some statement", {'x':5, 'y':10})
```

As well as with `Engine.begin()`:

```
with engine.begin() as conn:
    conn.execute("some statement", {'x':5, 'y':10})
```

See also:

`Engine.begin()` - engine-level transactional context

`Engine.transaction()` - engine-level version of `Connection.transaction()`

class sqlalchemy.engine.**Connectable**

Interface for an object which supports execution of SQL constructs.

The two implementations of `Connectable` are `Connection` and `Engine`.

Connectable must also implement the 'dialect' member which references a `Dialect` instance.

connect(***kwargs*)

Return a `Connection` object.

Depending on context, this may be `self` if this object is already an instance of `Connection`, or a newly procured `Connection` if this object is an instance of `Engine`.

contextual_connect()

Return a `Connection` object which may be part of an ongoing context.

Depending on context, this may be `self` if this object is already an instance of `Connection`, or a newly procured `Connection` if this object is an instance of `Engine`.

create(*entity*, ***kwargs*)

Emit CREATE statements for the given schema entity.

Deprecated since version 0.7: Use the create() method on the given schema object directly, i.e. `Table.create()`, `Index.create()`, `MetaData.create_all()`

drop(*entity*, ***kwargs*)

Emit DROP statements for the given schema entity.

Deprecated since version 0.7: Use the drop() method on the given schema object directly, i.e. `Table.drop()`, `Index.drop()`, `MetaData.drop_all()`

execute(*object*, **multiparams*, ***params*)

Executes the given construct and returns a `ResultProxy`.

scalar(*object*, **multiparams*, ***params*)

Executes and returns the first column of the first row.

The underlying cursor is closed after execution.

class sqlalchemy.engine.**CreateEnginePlugin**(*url*, *kwargs*)

A set of hooks intended to augment the construction of an `Engine` object based on entrypoint names in a URL.

The purpose of `CreateEnginePlugin` is to allow third-party systems to apply engine, pool and dialect level event listeners without the need for the target application to be modified; instead, the plugin names can be added to the database URL. Target applications for `CreateEnginePlugin` include:

- connection and SQL performance tools, e.g. which use events to track number of checkouts and/or time spent with statements
- connectivity plugins such as proxies

Plugins are registered using entry points in a similar way as that of dialects:

```
entry_points={
    'sqlalchemy.plugins': [
        'myplugin = myapp.plugins:MyPlugin'
    ]
```

A plugin that uses the above names would be invoked from a database URL as in:

```
from sqlalchemy import create_engine

engine = create_engine(
  "mysql+pymysql://scott:tiger@localhost/test?plugin=myplugin")
```

The `plugin` argument supports multiple instances, so that a URL may specify multiple plugins; they are loaded in the order stated in the URL:

```
engine = create_engine(
  "mysql+pymysql://scott:tiger@localhost/"
  "test?plugin=plugin_one&plugin=plugin_twp&plugin=plugin_three")
```

A plugin can receive additional arguments from the URL string as well as from the keyword arguments passed to `create_engine()`. The `URL` object and the keyword dictionary are passed to the constructor so that these arguments can be extracted from the url's `URL.query` collection as well as from the dictionary:

```
class MyPlugin(CreateEnginePlugin):
    def __init__(self, url, kwargs):
        self.my_argument_one = url.query.pop('my_argument_one')
        self.my_argument_two = url.query.pop('my_argument_two')
        self.my_argument_three = kwargs.pop('my_argument_three', None)
```

Arguments like those illustrated above would be consumed from the following:

```
from sqlalchemy import create_engine

engine = create_engine(
  "mysql+pymysql://scott:tiger@localhost/"
  "test?plugin=myplugin&my_argument_one=foo&my_argument_two=bar",
  my_argument_three='bat')
```

The URL and dictionary are used for subsequent setup of the engine as they are, so the plugin can modify their arguments in-place. Arguments that are only understood by the plugin should be popped or otherwise removed so that they aren't interpreted as erroneous arguments afterwards.

When the engine creation process completes and produces the `Engine` object, it is again passed to the plugin via the `CreateEnginePlugin.engine_created()` hook. In this hook, additional changes can be made to the engine, most typically involving setup of events (e.g. those defined in *Core Events*).

New in version 1.1.

__init__(*url*, *kwargs*)

Contruct a new `CreateEnginePlugin`.

The plugin object is instantiated individually for each call to `create_engine()`. A single `Engine` will be passed to the `CreateEnginePlugin.engine_created()` method corresponding to this URL.

Parameters

- **url** – the `URL` object. The plugin should inspect what it needs here as well as remove its custom arguments from the `URL.query` collection. The URL can be modified in-place in any other way as well.
- **kwargs** – The keyword arguments passed to :func'.create_engine'. The plugin can read and modify this dictionary in-place, to affect the ultimate arguments used to create the engine. It should remove its custom arguments from the dictionary as well.

engine_created(*engine*)

Receive the `Engine` object when it is fully constructed.

The plugin may make additional changes to the engine, such as registering engine or connection pool events.

class sqlalchemy.engine.**Engine**(*pool*, *dialect*, *url*, *logging_name=None*, *echo=None*, *proxy=None*, *execution_options=None*)

Bases: `sqlalchemy.engine.Connectable`, `sqlalchemy.log.Identified`

Connects a `Pool` and `Dialect` together to provide a source of database connectivity and behavior.

An `Engine` object is instantiated publicly using the `create_engine()` function.

See also:

Engine Configuration

Working with Engines and Connections

begin(*close_with_result=False*)

Return a context manager delivering a `Connection` with a `Transaction` established.

E.g.:

```
with engine.begin() as conn:
    conn.execute("insert into table (x, y, z) values (1, 2, 3)")
    conn.execute("my_special_procedure(5)")
```

Upon successful operation, the `Transaction` is committed. If an error is raised, the `Transaction` is rolled back.

The `close_with_result` flag is normally `False`, and indicates that the `Connection` will be closed when the operation is complete. When set to `True`, it indicates the `Connection` is in "single use" mode, where the `ResultProxy` returned by the first call to `Connection.execute()` will close the `Connection` when that `ResultProxy` has exhausted all result rows.

New in version 0.7.6.

See also:

`Engine.connect()` - procure a `Connection` from an `Engine`.

`Connection.begin()` - start a `Transaction` for a particular `Connection`.

connect(***kwargs*)

Return a new `Connection` object.

The `Connection` object is a facade that uses a DBAPI connection internally in order to communicate with the database. This connection is procured from the connection-holding `Pool` referenced by this `Engine`. When the `close()` method of the `Connection` object is called, the underlying DBAPI connection is then returned to the connection pool, where it may be used again in a subsequent call to `connect()`.

contextual_connect(*close_with_result=False*, ***kwargs*)

Return a `Connection` object which may be part of some ongoing context.

By default, this method does the same thing as `Engine.connect()`. Subclasses of `Engine` may override this method to provide contextual behavior.

Parameters **close_with_result** – When True, the first `ResultProxy` created by the `Connection` will call the `Connection.close()` method of that connection as soon as any pending result rows are exhausted. This is used to supply the "connectionless execution" behavior provided by the `Engine.execute()` method.

dispose()

Dispose of the connection pool used by this `Engine`.

This has the effect of fully closing all **currently checked in** database connections. Connections that are still checked out will **not** be closed, however they will no longer be associated with this `Engine`, so when they are closed individually, eventually the `Pool` which they are associated with will be garbage collected and they will be closed out fully, if not already closed on checkin.

A new connection pool is created immediately after the old one has been disposed. This new pool, like all SQLAlchemy connection pools, does not make any actual connections to the database until one is first requested, so as long as the `Engine` isn't used again, no new connections will be made.

See also:

Engine Disposal

driver

Driver name of the `Dialect` in use by this `Engine`.

execute(*statement*, **multiparams*, ***params*)

Executes the given construct and returns a `ResultProxy`.

The arguments are the same as those used by `Connection.execute()`.

Here, a *Connection* is acquired using the *contextual_connect()* method, and the statement executed with that connection. The returned *ResultProxy* is flagged such that when the *ResultProxy* is exhausted and its underlying cursor is closed, the *Connection* created here will also be closed, which allows its associated DBAPI connection resource to be returned to the connection pool.

execution_options(***opt*)

Return a new *Engine* that will provide *Connection* objects with the given execution options.

The returned *Engine* remains related to the original *Engine* in that it shares the same connection pool and other state:

- The *Pool* used by the new *Engine* is the same instance. The *Engine.dispose()* method will replace the connection pool instance for the parent engine as well as this one.
- Event listeners are "cascaded" - meaning, the new *Engine* inherits the events of the parent, and new events can be associated with the new *Engine* individually.
- The logging configuration and logging_name is copied from the parent *Engine*.

The intent of the *Engine.execution_options()* method is to implement "sharding" schemes where multiple *Engine* objects refer to the same connection pool, but are differentiated by options that would be consumed by a custom event:

```
primary_engine = create_engine("mysql://")
shard1 = primary_engine.execution_options(shard_id="shard1")
shard2 = primary_engine.execution_options(shard_id="shard2")
```

Above, the `shard1` engine serves as a factory for *Connection* objects that will contain the execution option `shard_id=shard1`, and `shard2` will produce *Connection* objects that contain the execution option `shard_id=shard2`.

An event handler can consume the above execution option to perform a schema switch or other operation, given a connection. Below we emit a MySQL `use` statement to switch databases, at the same time keeping track of which database we've established using the *Connection.info* dictionary, which gives us a persistent storage space that follows the DBAPI connection:

```
from sqlalchemy import event
from sqlalchemy.engine import Engine

shards = {"default": "base", shard_1: "db1", "shard_2": "db2"}

@event.listens_for(Engine, "before_cursor_execute")
def _switch_shard(conn, cursor, stmt,
        params, context, executemany):
    shard_id = conn._execution_options.get('shard_id', "default")
    current_shard = conn.info.get("current_shard", None)

    if current_shard != shard_id:
        cursor.execute("use %s" % shards[shard_id])
        conn.info["current_shard"] = shard_id
```

New in version 0.8.

See also:

Connection.execution_options() - update execution options on a *Connection* object.

Engine.update_execution_options() - update the execution options for a given *Engine* in place.

has_table(*table_name*, *schema=None*)

Return True if the given backend has a table of the given name.

See also:

Fine Grained Reflection with Inspector - detailed schema inspection using the `Inspector` interface.

`quoted_name` - used to pass quoting information along with a schema identifier.

name

String name of the `Dialect` in use by this `Engine`.

raw_connection(*_connection=None*)

Return a "raw" DBAPI connection from the connection pool.

The returned object is a proxied version of the DBAPI connection object used by the underlying driver in use. The object will have all the same behavior as the real DBAPI connection, except that its `close()` method will result in the connection being returned to the pool, rather than being closed for real.

This method provides direct DBAPI connection access for special situations when the API provided by `Connection` is not needed. When a `Connection` object is already present, the DBAPI connection is available using the `Connection.connection` accessor.

See also:

Working with Raw DBAPI Connections

run_callable(*callable_*, **args*, ***kwargs*)

Given a callable object or function, execute it, passing a `Connection` as the first argument.

The given *args and **kwargs are passed subsequent to the `Connection` argument.

This function, along with `Connection.run_callable()`, allows a function to be run with a `Connection` or `Engine` object without the need to know which one is being dealt with.

schema_for_object = <sqlalchemy.sql.schema._SchemaTranslateMap object>

Return the ".schema" attribute for an object.

Used for `Table`, `Sequence` and similar objects, and takes into account the `Connection.execution_options.schema_translate_map` parameter.

New in version 1.1.

See also:

Translation of Schema Names

table_names(*schema=None*, *connection=None*)

Return a list of all table names available in the database.

Parameters

- **schema** – Optional, retrieve names from a non-default schema.
- **connection** – Optional, use a specified connection. Default is the `contextual_connect` for this `Engine`.

transaction(*callable_*, **args*, ***kwargs*)

Execute the given function within a transaction boundary.

The function is passed a `Connection` newly procured from `Engine.contextual_connect()` as the first argument, followed by the given *args and **kwargs.

e.g.:

```
def do_something(conn, x, y):
    conn.execute("some statement", {'x':x, 'y':y})

engine.transaction(do_something, 5, 10)
```

The operations inside the function are all invoked within the context of a single *Transaction*. Upon success, the transaction is committed. If an exception is raised, the transaction is rolled back before propagating the exception.

Note: The *transaction()* method is superseded by the usage of the Python `with:` statement, which can be used with *Engine.begin()*:

```
with engine.begin() as conn:
    conn.execute("some statement", {'x':5, 'y':10})
```

See also:

Engine.begin() - engine-level transactional context

Connection.transaction() - connection-level version of *Engine.transaction()*

update_execution_options(***opt*)

Update the default execution_options dictionary of this *Engine*.

The given keys/values in **opt are added to the default execution options that will be used for all connections. The initial contents of this dictionary can be sent via the `execution_options` parameter to *create_engine()*.

See also:

Connection.execution_options()

Engine.execution_options()

class sqlalchemy.engine.**ExceptionContext**

Encapsulate information about an error condition in progress.

This object exists solely to be passed to the *ConnectionEvents.handle_error()* event, supporting an interface that can be extended without backwards-incompatibility.

New in version 0.9.7.

chained_exception = None

The exception that was returned by the previous handler in the exception chain, if any.

If present, this exception will be the one ultimately raised by SQLAlchemy unless a subsequent handler replaces it.

May be None.

connection = None

The *Connection* in use during the exception.

This member is present, except in the case of a failure when first connecting.

See also:

ExceptionContext.engine

cursor = None

The DBAPI cursor object.

May be None.

engine = None

The *Engine* in use during the exception.

This member should always be present, even in the case of a failure when first connecting.

New in version 1.0.0.

execution_context = None

The *ExecutionContext* corresponding to the execution operation in progress.

This is present for statement execution operations, but not for operations such as transaction begin/end. It also is not present when the exception was raised before the *ExecutionContext* could be constructed.

Note that the *ExceptionContext.statement* and *ExceptionContext.parameters* members may represent a different value than that of the *ExecutionContext*, potentially in the case where a *ConnectionEvents.before_cursor_execute()* event or similar modified the statement/parameters to be sent.

May be None.

invalidate_pool_on_disconnect = True

Represent whether all connections in the pool should be invalidated when a "disconnect" condition is in effect.

Setting this flag to False within the scope of the *ConnectionEvents.handle_error()* event will have the effect such that the full collection of connections in the pool will not be invalidated during a disconnect; only the current connection that is the subject of the error will actually be invalidated.

The purpose of this flag is for custom disconnect-handling schemes where the invalidation of other connections in the pool is to be performed based on other conditions, or even on a per-connection basis.

New in version 1.0.3.

is_disconnect = None

Represent whether the exception as occurred represents a "disconnect" condition.

This flag will always be True or False within the scope of the *ConnectionEvents.handle_error()* handler.

SQLAlchemy will defer to this flag in order to determine whether or not the connection should be invalidated subsequently. That is, by assigning to this flag, a "disconnect" event which then results in a connection and pool invalidation can be invoked or prevented by changing this flag.

original_exception = None

The exception object which was caught.

This member is always present.

parameters = None

Parameter collection that was emitted directly to the DBAPI.

May be None.

sqlalchemy_exception = None

The *sqlalchemy.exc.StatementError* which wraps the original, and will be raised if exception handling is not circumvented by the event.

May be None, as not all exception types are wrapped by SQLAlchemy. For DBAPI-level exceptions that subclass the dbapi's Error class, this field will always be present.

statement = None

String SQL statement that was emitted directly to the DBAPI.

May be None.

class sqlalchemy.engine.**NestedTransaction**(*connection*, *parent*)

Bases: *sqlalchemy.engine.Transaction*

Represent a 'nested', or SAVEPOINT transaction.

A new *NestedTransaction* object may be procured using the *Connection.begin_nested()* method.

The interface is the same as that of *Transaction*.

class sqlalchemy.engine.**ResultProxy**(*context*)

Wraps a DB-API cursor object to provide easier access to row columns.

Individual columns may be accessed by their integer position, case-insensitive column name, or by schema.Column object. e.g.:

```
row = fetchone()

col1 = row[0]    # access via integer position

col2 = row['col2']   # access via name

col3 = row[mytable.c.mycol] # access via Column object.
```

ResultProxy also handles post-processing of result column data using TypeEngine objects, which are referenced from the originating SQL statement that produced this result set.

_cursor_description()

May be overridden by subclasses.

_process_row

alias of *RowProxy*

_soft_close(*_autoclose_connection=True*)

Soft close this *ResultProxy*.

This releases all DBAPI cursor resources, but leaves the ResultProxy "open" from a semantic perspective, meaning the fetchXXX() methods will continue to return empty results.

This method is called automatically when:

- all result rows are exhausted using the fetchXXX() methods.
- cursor.description is None.

This method is **not public**, but is documented in order to clarify the "autoclose" process used.

New in version 1.0.0.

See also:

ResultProxy.close()

close()

Close this ResultProxy.

This closes out the underlying DBAPI cursor corresonding to the statement execution, if one is stil present. Note that the DBAPI cursor is automatically released when the *ResultProxy* exhausts all available rows. *ResultProxy.close()* is generally an optional method except in the case when discarding a *ResultProxy* that still has additional rows pending for fetch.

In the case of a result that is the product of *connectionless execution*, the underlying *Connection* object is also closed, which releases DBAPI connection resources.

After this method is called, it is no longer valid to call upon the fetch methods, which will raise a *ResourceClosedError* on subsequent use.

Changed in version 1.0.0: - the *ResultProxy.close()* method has been separated out from the process that releases the underlying DBAPI cursor resource. The "auto close" feature of the

`Connection` now performs a so-called "soft close", which releases the underlying DBAPI cursor, but allows the `ResultProxy` to still behave as an open-but-exhausted result set; the actual `ResultProxy.close()` method is never called. It is still safe to discard a `ResultProxy` that has been fully exhausted without calling this method.

See also:

Working with Engines and Connections

`ResultProxy._soft_close()`

fetchall()

Fetch all rows, just like DB-API `cursor.fetchall()`.

After all rows have been exhausted, the underlying DBAPI cursor resource is released, and the object may be safely discarded.

Subsequent calls to `ResultProxy.fetchall()` will return an empty list. After the `ResultProxy.close()` method is called, the method will raise `ResourceClosedError`.

Changed in version 1.0.0: - Added "soft close" behavior which allows the result to be used in an "exhausted" state prior to calling the `ResultProxy.close()` method.

fetchmany(*size=None*)

Fetch many rows, just like DB-API `cursor.fetchmany(size=cursor.arraysize)`.

After all rows have been exhausted, the underlying DBAPI cursor resource is released, and the object may be safely discarded.

Calls to `ResultProxy.fetchmany()` after all rows have been exhuasted will return an empty list. After the `ResultProxy.close()` method is called, the method will raise `ResourceClosedError`.

Changed in version 1.0.0: - Added "soft close" behavior which allows the result to be used in an "exhausted" state prior to calling the `ResultProxy.close()` method.

fetchone()

Fetch one row, just like DB-API `cursor.fetchone()`.

After all rows have been exhausted, the underlying DBAPI cursor resource is released, and the object may be safely discarded.

Calls to `ResultProxy.fetchone()` after all rows have been exhausted will return `None`. After the `ResultProxy.close()` method is called, the method will raise `ResourceClosedError`.

Changed in version 1.0.0: - Added "soft close" behavior which allows the result to be used in an "exhausted" state prior to calling the `ResultProxy.close()` method.

first()

Fetch the first row and then close the result set unconditionally.

Returns None if no row is present.

After calling this method, the object is fully closed, e.g. the `ResultProxy.close()` method will have been called.

inserted_primary_key

Return the primary key for the row just inserted.

The return value is a list of scalar values corresponding to the list of primary key columns in the target table.

This only applies to single row `insert()` constructs which did not explicitly specify `Insert.returning()`.

Note that primary key columns which specify a server_default clause, or otherwise do not qualify as "autoincrement" columns (see the notes at *Column*), and were generated using the database-side default, will appear in this list as `None` unless the backend supports "returning" and the insert statement executed with the "implicit returning" enabled.

Raises *InvalidRequestError* if the executed statement is not a compiled expression construct or is not an insert() construct.

is_insert

True if this *ResultProxy* is the result of a executing an expression language compiled *expression.insert()* construct.

When True, this implies that the *inserted_primary_key* attribute is accessible, assuming the statement did not include a user defined "returning" construct.

keys()

Return the current set of string keys for rows.

last_inserted_params()

Return the collection of inserted parameters from this execution.

Raises *InvalidRequestError* if the executed statement is not a compiled expression construct or is not an insert() construct.

last_updated_params()

Return the collection of updated parameters from this execution.

Raises *InvalidRequestError* if the executed statement is not a compiled expression construct or is not an update() construct.

lastrow_has_defaults()

Return `lastrow_has_defaults()` from the underlying *ExecutionContext*.

See *ExecutionContext* for details.

lastrowid

return the 'lastrowid' accessor on the DBAPI cursor.

This is a DBAPI specific method and is only functional for those backends which support it, for statements where it is appropriate. It's behavior is not consistent across backends.

Usage of this method is normally unnecessary when using insert() expression constructs; the *inserted_primary_key* attribute provides a tuple of primary key values for a newly inserted row, regardless of database backend.

postfetch_cols()

Return `postfetch_cols()` from the underlying *ExecutionContext*.

See *ExecutionContext* for details.

Raises *InvalidRequestError* if the executed statement is not a compiled expression construct or is not an insert() or update() construct.

prefetch_cols()

Return `prefetch_cols()` from the underlying *ExecutionContext*.

See *ExecutionContext* for details.

Raises *InvalidRequestError* if the executed statement is not a compiled expression construct or is not an insert() or update() construct.

returned_defaults

Return the values of default columns that were fetched using the *ValuesBase.return_defaults()* feature.

The value is an instance of *RowProxy*, or None if *ValuesBase.return_defaults()* was not used or if the backend does not support RETURNING.

New in version 0.9.0.

See also:

ValuesBase.return_defaults()

returns_rows

True if this *ResultProxy* returns rows.

I.e. if it is legal to call the methods *fetchone()*, *fetchmany()* *fetchall()*.

rowcount

Return the 'rowcount' for this result.

The 'rowcount' reports the number of rows *matched* by the WHERE criterion of an UPDATE or DELETE statement.

Note: Notes regarding *ResultProxy.rowcount*:

- This attribute returns the number of rows *matched*, which is not necessarily the same as the number of rows that were actually *modified* - an UPDATE statement, for example, may have no net change on a given row if the SET values given are the same as those present in the row already. Such a row would be matched but not modified. On backends that feature both styles, such as MySQL, rowcount is configured by default to return the match count in all cases.
- *ResultProxy.rowcount* is *only* useful in conjunction with an UPDATE or DELETE statement. Contrary to what the Python DBAPI says, it does *not* return the number of rows available from the results of a SELECT statement as DBAPIs cannot support this functionality when rows are unbuffered.
- *ResultProxy.rowcount* may not be fully implemented by all dialects. In particular, most DBAPIs do not support an aggregate rowcount result from an executemany call. The *ResultProxy.supports_sane_rowcount()* and *ResultProxy.supports_sane_multi_rowcount()* methods will report from the dialect if each usage is known to be supported.
- Statements that use RETURNING may not return a correct rowcount.

---

scalar()

Fetch the first column of the first row, and close the result set.

Returns None if no row is present.

After calling this method, the object is fully closed, e.g. the *ResultProxy.close()* method will have been called.

supports_sane_multi_rowcount()

Return supports_sane_multi_rowcount from the dialect.

See *ResultProxy.rowcount* for background.

supports_sane_rowcount()

Return supports_sane_rowcount from the dialect.

See *ResultProxy.rowcount* for background.

class sqlalchemy.engine.**RowProxy**(*parent*, *row*, *processors*, *keymap*)

Bases: sqlalchemy.engine.BaseRowProxy

Proxy values from a single cursor row.

Mostly follows "ordered dictionary" behavior, mapping result values to the string-based column name, the integer position of the result in the row, as well as Column instances which can be mapped to the original Columns that produced this result set (for results that correspond to constructed SQL expressions).

has_key(*key*)
Return True if this RowProxy contains the given key.

items()
Return a list of tuples, each tuple containing a key/value pair.

keys()
Return the list of keys as strings represented by this RowProxy.

class sqlalchemy.engine.**Transaction**(*connection*, *parent*)
Represent a database transaction in progress.

The *Transaction* object is procured by calling the *begin()* method of *Connection*:

```
from sqlalchemy import create_engine
engine = create_engine("postgresql://scott:tiger@localhost/test")
connection = engine.connect()
trans = connection.begin()
connection.execute("insert into x (a, b) values (1, 2)")
trans.commit()
```

The object provides *rollback()* and *commit()* methods in order to control transaction boundaries. It also implements a context manager interface so that the Python `with` statement can be used with the *Connection.begin()* method:

```
with connection.begin():
    connection.execute("insert into x (a, b) values (1, 2)")
```

The Transaction object is **not** threadsafe.

See also: *Connection.begin()*, *Connection.begin_twophase()*, *Connection.begin_nested()*.

close()
Close this *Transaction*.

If this transaction is the base transaction in a begin/commit nesting, the transaction will rollback(). Otherwise, the method returns.

This is used to cancel a Transaction without affecting the scope of an enclosing transaction.

commit()
Commit this *Transaction*.

rollback()
Roll back this *Transaction*.

class sqlalchemy.engine.**TwoPhaseTransaction**(*connection*, *xid*)
Bases: *sqlalchemy.engine.Transaction*

Represent a two-phase transaction.

A new *TwoPhaseTransaction* object may be procured using the *Connection.begin_twophase()* method.

The interface is the same as that of *Transaction* with the addition of the *prepare()* method.

prepare()

Prepare this *TwoPhaseTransaction*.

After a PREPARE, the transaction can be committed.

3.5.3 Connection Pooling

A connection pool is a standard technique used to maintain long running connections in memory for efficient re-use, as well as to provide management for the total number of connections an application might use simultaneously.

Particularly for server-side web applications, a connection pool is the standard way to maintain a "pool" of active database connections in memory which are reused across requests.

SQLAlchemy includes several connection pool implementations which integrate with the *Engine*. They can also be used directly for applications that want to add pooling to an otherwise plain DBAPI approach.

Connection Pool Configuration

The *Engine* returned by the *create_engine()* function in most cases has a *QueuePool* integrated, pre-configured with reasonable pooling defaults. If you're reading this section only to learn how to enable pooling - congratulations! You're already done.

The most common *QueuePool* tuning parameters can be passed directly to *create_engine()* as keyword arguments: `pool_size`, `max_overflow`, `pool_recycle` and `pool_timeout`. For example:

```
engine = create_engine('postgresql://me@localhost/mydb',
                       pool_size=20, max_overflow=0)
```

In the case of SQLite, the *SingletonThreadPool* or *NullPool* are selected by the dialect to provide greater compatibility with SQLite's threading and locking model, as well as to provide a reasonable default behavior to SQLite "memory" databases, which maintain their entire dataset within the scope of a single connection.

All SQLAlchemy pool implementations have in common that none of them "pre create" connections - all implementations wait until first use before creating a connection. At that point, if no additional concurrent checkout requests for more connections are made, no additional connections are created. This is why it's perfectly fine for *create_engine()* to default to using a *QueuePool* of size five without regard to whether or not the application really needs five connections queued up - the pool would only grow to that size if the application actually used five connections concurrently, in which case the usage of a small pool is an entirely appropriate default behavior.

Switching Pool Implementations

The usual way to use a different kind of pool with *create_engine()* is to use the `poolclass` argument. This argument accepts a class imported from the `sqlalchemy.pool` module, and handles the details of building the pool for you. Common options include specifying *QueuePool* with SQLite:

```
from sqlalchemy.pool import QueuePool
engine = create_engine('sqlite:///file.db', poolclass=QueuePool)
```

Disabling pooling using *NullPool*:

```
from sqlalchemy.pool import NullPool
engine = create_engine(
          'postgresql+psycopg2://scott:tiger@localhost/test',
          poolclass=NullPool)
```

Using a Custom Connection Function

All *Pool* classes accept an argument `creator` which is a callable that creates a new connection. *create_engine()* accepts this function to pass onto the pool via an argument of the same name:

```
import sqlalchemy.pool as pool
import psycopg2

def getconn():
    c = psycopg2.connect(username='ed', host='127.0.0.1', dbname='test')
    # do things with 'c' to set up
    return c

engine = create_engine('postgresql+psycopg2://', creator=getconn)
```

For most "initialize on connection" routines, it's more convenient to use the *PoolEvents* event hooks, so that the usual URL argument to *create_engine()* is still usable. `creator` is there as a last resort for when a DBAPI has some form of `connect` that is not at all supported by SQLAlchemy.

Constructing a Pool

To use a *Pool* by itself, the `creator` function is the only argument that's required and is passed first, followed by any additional options:

```
import sqlalchemy.pool as pool
import psycopg2

def getconn():
    c = psycopg2.connect(username='ed', host='127.0.0.1', dbname='test')
    return c

mypool = pool.QueuePool(getconn, max_overflow=10, pool_size=5)
```

DBAPI connections can then be procured from the pool using the *Pool.connect()* function. The return value of this method is a DBAPI connection that's contained within a transparent proxy:

```
# get a connection
conn = mypool.connect()

# use it
cursor = conn.cursor()
cursor.execute("select foo")
```

The purpose of the transparent proxy is to intercept the `close()` call, such that instead of the DBAPI connection being closed, it is returned to the pool:

```
# "close" the connection.  Returns
# it to the pool.
conn.close()
```

The proxy also returns its contained DBAPI connection to the pool when it is garbage collected, though it's not deterministic in Python that this occurs immediately (though it is typical with cPython).

The `close()` step also performs the important step of calling the `rollback()` method of the DBAPI connection. This is so that any existing transaction on the connection is removed, not only ensuring that no existing state remains

on next usage, but also so that table and row locks are released as well as that any isolated data snapshots are removed. This behavior can be disabled using the `reset_on_return` option of *Pool*.

A particular pre-created *Pool* can be shared with one or more engines by passing it to the `pool` argument of *create_engine()*:

```
e = create_engine('postgresql://', pool=mypool)
```

Pool Events

Connection pools support an event interface that allows hooks to execute upon first connect, upon each new connection, and upon checkout and checkin of connections. See *PoolEvents* for details.

Dealing with Disconnects

The connection pool has the ability to refresh individual connections as well as its entire set of connections, setting the previously pooled connections as "invalid". A common use case is allow the connection pool to gracefully recover when the database server has been restarted, and all previously established connections are no longer functional. There are two approaches to this.

Disconnect Handling - Optimistic

The most common approach is to let SQLAlchemy handle disconnects as they occur, at which point the pool is refreshed. This assumes the *Pool* is used in conjunction with a *Engine*. The *Engine* has logic which can detect disconnection events and refresh the pool automatically.

When the *Connection* attempts to use a DBAPI connection, and an exception is raised that corresponds to a "disconnect" event, the connection is invalidated. The *Connection* then calls the *Pool.recreate()* method, effectively invalidating all connections not currently checked out so that they are replaced with new ones upon next checkout:

```
from sqlalchemy import create_engine, exc
e = create_engine(...)
c = e.connect()

try:
    # suppose the database has been restarted.
    c.execute("SELECT * FROM table")
    c.close()
except exc.DBAPIError, e:
    # an exception is raised, Connection is invalidated.
    if e.connection_invalidated:
        print "Connection was invalidated!"

# after the invalidate event, a new connection
# starts with a new Pool
c = e.connect()
c.execute("SELECT * FROM table")
```

The above example illustrates that no special intervention is needed, the pool continues normally after a disconnection event is detected. However, an exception is raised. In a typical web application using an ORM Session, the above condition would correspond to a single request failing with a 500 error, then the web application continuing normally beyond that. Hence the approach is "optimistic" in that frequent database restarts are not anticipated.

Setting Pool Recycle An additional setting that can augment the "optimistic" approach is to set the pool recycle parameter. This parameter prevents the pool from using a particular connection that has passed a certain age, and is appropriate for database backends such as MySQL that automatically close connections that have been stale after a particular period of time:

```
from sqlalchemy import create_engine
e = create_engine("mysql://scott:tiger@localhost/test", pool_recycle=3600)
```

Above, any DBAPI connection that has been open for more than one hour will be invalidated and replaced, upon next checkout. Note that the invalidation **only** occurs during checkout - not on any connections that are held in a checked out state. `pool_recycle` is a function of the *Pool* itself, independent of whether or not an *Engine* is in use.

Disconnect Handling - Pessimistic

At the expense of some extra SQL emitted for each connection checked out from the pool, a "ping" operation established by a checkout event handler can detect an invalid connection before it is used. In modern SQLAlchemy, the best way to do this is to make use of the *ConnectionEvents.engine_connect()* event, assuming the use of a *Engine* and not just a raw *Pool* object:

```
from sqlalchemy import exc
from sqlalchemy import event
from sqlalchemy import select

some_engine = create_engine(...)

@event.listens_for(some_engine, "engine_connect")
def ping_connection(connection, branch):
    if branch:
        # "branch" refers to a sub-connection of a connection,
        # we don't want to bother pinging on these.
        return

    try:
        # run a SELECT 1.   use a core select() so that
        # the SELECT of a scalar value without a table is
        # appropriately formatted for the backend
        connection.scalar(select([1]))
    except exc.DBAPIError as err:
        # catch SQLAlchemy's DBAPIError, which is a wrapper
        # for the DBAPI's exception.  It includes a .connection_invalidated
        # attribute which specifies if this connection is a "disconnect"
        # condition, which is based on inspection of the original exception
        # by the dialect in use.
        if err.connection_invalidated:
            # run the same SELECT again - the connection will re-validate
            # itself and establish a new connection.  The disconnect detection
            # here also causes the whole connection pool to be invalidated
            # so that all stale connections are discarded.
            connection.scalar(select([1]))
        else:
            raise
```

The above recipe has the advantage that we are making use of SQLAlchemy's facilities for detecting those DBAPI exceptions that are known to indicate a "disconnect" situation, as well as the *Engine* object's ability to correctly invalidate the current connection pool when this condition occurs and allowing the current *Connection* to re-validate onto a new DBAPI connection.

For the much less common case of where a *Pool* is being used without an *Engine*, an older approach may be used as below:

```
from sqlalchemy import exc
from sqlalchemy import event
from sqlalchemy.pool import Pool

@event.listens_for(Pool, "checkout")
def ping_connection(dbapi_connection, connection_record, connection_proxy):
    cursor = dbapi_connection.cursor()
    try:
        cursor.execute("SELECT 1")
    except:
        # raise DisconnectionError - pool will try
        # connecting again up to three times before raising.
        raise exc.DisconnectionError()
    cursor.close()
```

Above, the *Pool* object specifically catches *DisconnectionError* and attempts to create a new DBAPI connection, up to three times, before giving up and then raising *InvalidRequestError*, failing the connection. The disadvantage of the above approach is that we don't have any easy way of determining if the exception raised is in fact a "disconnect" situation, since there is no *Engine* or *Dialect* in play, and also the above error would occur individually for all stale connections still in the pool.

More on Invalidation

The *Pool* provides "connection invalidation" services which allow both explicit invalidation of a connection as well as automatic invalidation in response to conditions that are determined to render a connection unusable.

"Invalidation" means that a particular DBAPI connection is removed from the pool and discarded. The `.close()` method is called on this connection if it is not clear that the connection itself might not be closed, however if this method fails, the exception is logged but the operation still proceeds.

When using a *Engine*, the *Connection.invalidate()* method is the usual entrypoint to explicit invalidation. Other conditions by which a DBAPI connection might be invalidated include:

- a DBAPI exception such as *OperationalError*, raised when a method like `connection.execute()` is called, is detected as indicating a so-called "disconnect" condition. As the Python DBAPI provides no standard system for determining the nature of an exception, all SQLAlchemy dialects include a system called `is_disconnect()` which will examine the contents of an exception object, including the string message and any potential error codes included with it, in order to determine if this exception indicates that the connection is no longer usable. If this is the case, the *_ConnectionFairy.invalidate()* method is called and the DBAPI connection is then discarded.
- When the connection is returned to the pool, and calling the `connection.rollback()` or `connection.commit()` methods, as dictated by the pool's "reset on return" behavior, throws an exception. A final attempt at calling `.close()` on the connection will be made, and it is then discarded.
- When a listener implementing *PoolEvents.checkout()* raises the *DisconnectionError* exception, indicating that the connection won't be usable and a new connection attempt needs to be made.

All invalidations which occur will invoke the *PoolEvents.invalidate()* event.

Using Connection Pools with Multiprocessing

It's critical that when using a connection pool, and by extension when using an *Engine* created via *create_engine()*, that the pooled connections **are not shared to a forked process**. TCP connections are repre-

sented as file descriptors, which usually work across process boundaries, meaning this will cause concurrent access to the file descriptor on behalf of two or more entirely independent Python interpreter states.

There are two approaches to dealing with this.

The first is, either create a new *Engine* within the child process, or upon an existing *Engine*, call *Engine.dispose()* before the child process uses any connections. This will remove all existing connections from the pool so that it makes all new ones. Below is a simple version using `multiprocessing.Process`, but this idea should be adapted to the style of forking in use:

```
eng = create_engine("...")

def run_in_process():
  eng.dispose()

  with eng.connect() as conn:
      conn.execute("...")

p = Process(target=run_in_process)
```

The next approach is to instrument the *Pool* itself with events so that connections are automatically invalidated in the subprocess. This is a little more magical but probably more foolproof:

```
from sqlalchemy import event
from sqlalchemy import exc
import os

eng = create_engine("...")

@event.listens_for(engine, "connect")
def connect(dbapi_connection, connection_record):
    connection_record.info['pid'] = os.getpid()

@event.listens_for(engine, "checkout")
def checkout(dbapi_connection, connection_record, connection_proxy):
    pid = os.getpid()
    if connection_record.info['pid'] != pid:
        connection_record.connection = connection_proxy.connection = None
        raise exc.DisconnectionError(
                "Connection record belongs to pid %s, "
                "attempting to check out in pid %s" %
                (connection_record.info['pid'], pid)
        )
```

Above, we use an approach similar to that described in *Disconnect Handling - Pessimistic* to treat a DBAPI connection that originated in a different parent process as an "invalid" connection, coercing the pool to recycle the connection record to make a new connection.

API Documentation - Available Pool Implementations

class `sqlalchemy.pool.`**`Pool`**(*creator*, *recycle=-1*, *echo=None*, *use_threadlocal=False*, *logging_name=None*, *reset_on_return=True*, *listeners=None*, *events=None*, *_dispatch=None*, *_dialect=None*)

Bases: *sqlalchemy.log.Identified*

Abstract base class for connection pools.

__init__(*creator*, *recycle=-1*, *echo=None*, *use_threadlocal=False*, *logging_name=None*, *reset_on_return=True*, *listeners=None*, *events=None*, *_dispatch=None*, *_dialect=None*)

Construct a Pool.

Parameters

- **creator** – a callable function that returns a DB-API connection object. The function will be called with parameters.
- **recycle** – If set to non -1, number of seconds between connection recycling, which means upon checkout, if this timeout is surpassed the connection will be closed and replaced with a newly opened connection. Defaults to -1.
- **logging_name** – String identifier which will be used within the "name" field of logging records generated within the "sqlalchemy.pool" logger. Defaults to a hexstring of the object's id.
- **echo** – If True, connections being pulled and retrieved from the pool will be logged to the standard output, as well as pool sizing information. Echoing can also be achieved by enabling logging for the "sqlalchemy.pool" namespace. Defaults to False.
- **use_threadlocal** – If set to True, repeated calls to `connect()` within the same application thread will be guaranteed to return the same connection object, if one has already been retrieved from the pool and has not been returned yet. Offers a slight performance advantage at the cost of individual transactions by default. The `Pool.unique_connection()` method is provided to return a consistenty unique connection to bypass this behavior when the flag is set.

  > **Warning:** The `Pool.use_threadlocal` flag **does not affect the behavior** of `Engine.connect()`. `Engine.connect()` makes use of the `Pool.unique_connection()` method which **does not use thread local context**. To produce a `Connection` which refers to the `Pool.connect()` method, use `Engine.contextual_connect()`.
  > Note that other SQLAlchemy connectivity systems such as `Engine.execute()` as well as the orm `Session` make use of `Engine.contextual_connect()` internally, so these functions are compatible with the `Pool.use_threadlocal` setting.

  See also:

  Using the Threadlocal Execution Strategy - contains detail on the "threadlocal" engine strategy, which provides a more comprehensive approach to "threadlocal" connectivity for the specific use case of using `Engine` and `Connection` objects directly.

- **reset_on_return** – Determine steps to take on connections as they are returned to the pool. reset_on_return can have any of these values:
  - `"rollback"` - call rollback() on the connection, to release locks and transaction resources. This is the default value. The vast majority of use cases should leave this value set.
  - `True` - same as 'rollback', this is here for backwards compatibility.
  - `"commit"` - call commit() on the connection, to release locks and transaction resources. A commit here may be desirable for databases that cache query plans if a commit is emitted, such as Microsoft SQL Server. However, this

value is more dangerous than 'rollback' because any data changes present on the transaction are committed unconditionally.

- `None` - don't do anything on the connection. This setting should only be made on a database that has no transaction support at all, namely MySQL MyISAM. By not doing anything, performance can be improved. This setting should **never be selected** for a database that supports transactions, as it will lead to deadlocks and stale state.
- `"none"` - same as `None`

  New in version 0.9.10.
- `False` - same as None, this is here for backwards compatibility.

Changed in version 0.7.6: *Pool.reset_on_return* accepts `"rollback"` and `"commit"` arguments.

- **events** – a list of 2-tuples, each of the form `(callable, target)` which will be passed to *event.listen()* upon construction. Provided here so that event listeners can be assigned via *create_engine()* before dialect-level listeners are applied.
- **listeners** – Deprecated. A list of *PoolListener*-like objects or dictionaries of callables that receive events when DB-API connections are created, checked out and checked in to the pool. This has been superseded by *listen()*.

connect()

Return a DBAPI connection from the pool.

The connection is instrumented such that when its `close()` method is called, the connection will be returned to the pool.

dispose()

Dispose of this pool.

This method leaves the possibility of checked-out connections remaining open, as it only affects connections that are idle in the pool.

See also the *Pool.recreate()* method.

recreate()

Return a new *Pool*, of the same class as this one and configured with identical creation arguments.

This method is used in conjunction with *dispose()* to close out an entire *Pool* and create a new one in its place.

unique_connection()

Produce a DBAPI connection that is not referenced by any thread-local context.

This method is equivalent to *Pool.connect()* when the *Pool.use_threadlocal* flag is not set to True. When *Pool.use_threadlocal* is True, the *Pool.unique_connection()* method provides a means of bypassing the threadlocal context.

class sqlalchemy.pool.**QueuePool**(*creator*, *pool_size=5*, *max_overflow=10*, *timeout=30*, ***kw*)

Bases: *sqlalchemy.pool.Pool*

A *Pool* that imposes a limit on the number of open connections.

QueuePool is the default pooling implementation used for all *Engine* objects, unless the SQLite dialect is in use.

__init__(*creator*, *pool_size=5*, *max_overflow=10*, *timeout=30*, ***kw*)

Construct a QueuePool.

Parameters

- **creator** – a callable function that returns a DB-API connection object, same as that of `Pool.creator`.
- **pool_size** – The size of the pool to be maintained, defaults to 5. This is the largest number of connections that will be kept persistently in the pool. Note that the pool begins with no connections; once this number of connections is requested, that number of connections will remain. `pool_size` can be set to 0 to indicate no size limit; to disable pooling, use a `NullPool` instead.
- **max_overflow** – The maximum overflow size of the pool. When the number of checked-out connections reaches the size set in pool_size, additional connections will be returned up to this limit. When those additional connections are returned to the pool, they are disconnected and discarded. It follows then that the total number of simultaneous connections the pool will allow is pool_size + *max_overflow*, and the total number of "sleeping" connections the pool will allow is pool_size. *max_overflow* can be set to -1 to indicate no overflow limit; no limit will be placed on the total number of concurrent connections. Defaults to 10.
- **timeout** – The number of seconds to wait before giving up on returning a connection. Defaults to 30.
- ****kw** – Other keyword arguments including `Pool.recycle`, `Pool.echo`, `Pool.reset_on_return` and others are passed to the `Pool` constructor.

connect()

Return a DBAPI connection from the pool.

The connection is instrumented such that when its `close()` method is called, the connection will be returned to the pool.

unique_connection()

Produce a DBAPI connection that is not referenced by any thread-local context.

This method is equivalent to `Pool.connect()` when the `Pool.use_threadlocal` flag is not set to True. When `Pool.use_threadlocal` is True, the `Pool.unique_connection()` method provides a means of bypassing the threadlocal context.

class sqlalchemy.pool.**SingletonThreadPool**(*creator*, *pool_size=5*, ***kw*)

Bases: `sqlalchemy.pool.Pool`

A Pool that maintains one connection per thread.

Maintains one connection per each thread, never moving a connection to a thread other than the one which it was created in.

> **Warning:** the `SingletonThreadPool` will call `.close()` on arbitrary connections that exist beyond the size setting of `pool_size`, e.g. if more unique **thread identities** than what `pool_size` states are used. This cleanup is non-deterministic and not sensitive to whether or not the connections linked to those thread identities are currently in use.
> `SingletonThreadPool` may be improved in a future release, however in its current status it is generally used only for test scenarios using a SQLite `:memory:` database and is not recommended for production use.

Options are the same as those of `Pool`, as well as:

Parameters **pool_size** – The number of threads in which to maintain connections at once. Defaults to five.

`SingletonThreadPool` is used by the SQLite dialect automatically when a memory-based database is used. See *SQLite*.

__init__(*creator*, *pool_size=5*, ***kw*)

class sqlalchemy.pool.**AssertionPool**(**args*, ***kw*)

Bases: sqlalchemy.pool.Pool

A Pool that allows at most one checked out connection at any given time.

This will raise an exception if more than one connection is checked out at a time. Useful for debugging code that is using more connections than desired.

Changed in version 0.7: AssertionPool also logs a traceback of where the original connection was checked out, and reports this in the assertion error raised.

class sqlalchemy.pool.**NullPool**(*creator*, *recycle=-1*, *echo=None*, *use_threadlocal=False*, *logging_name=None*, *reset_on_return=True*, *listeners=None*, *events=None*, *_dispatch=None*, *_dialect=None*)

Bases: sqlalchemy.pool.Pool

A Pool which does not pool connections.

Instead it literally opens and closes the underlying DB-API connection per each connection open/close.

Reconnect-related functions such as recycle and connection invalidation are not supported by this Pool implementation, since no connections are held persistently.

Changed in version 0.7: NullPool is used by the SQlite dialect automatically when a file-based database is used. See *SQLite*.

class sqlalchemy.pool.**StaticPool**(*creator*, *recycle=-1*, *echo=None*, *use_threadlocal=False*, *logging_name=None*, *reset_on_return=True*, *listeners=None*, *events=None*, *_dispatch=None*, *_dialect=None*)

Bases: sqlalchemy.pool.Pool

A Pool of exactly one connection, used for all requests.

Reconnect-related functions such as recycle and connection invalidation (which is also used to support auto-reconnect) are not currently supported by this Pool implementation but may be implemented in a future release.

class sqlalchemy.pool.**_ConnectionFairy**(*dbapi_connection*, *connection_record*, *echo*)

Proxies a DBAPI connection and provides return-on-dereference support.

This is an internal object used by the Pool implementation to provide context management to a DBAPI connection delivered by that Pool.

The name "fairy" is inspired by the fact that the _ConnectionFairy object's lifespan is transitory, as it lasts only for the length of a specific DBAPI connection being checked out from the pool, and additionally that as a transparent proxy, it is mostly invisible.

See also:

_ConnectionRecord

_connection_record = None

A reference to the _ConnectionRecord object associated with the DBAPI connection.

This is currently an internal accessor which is subject to change.

connection = None

A reference to the actual DBAPI connection being tracked.

cursor(**args*, ***kwargs*)

Return a new DBAPI cursor for the underlying connection.

This method is a proxy for the connection.cursor() DBAPI method.

detach()

Separate this connection from its Pool.

This means that the connection will no longer be returned to the pool when closed, and will instead be literally closed. The containing ConnectionRecord is separated from the DB-API connection, and will create a new connection when next used.

Note that any overall connection limiting constraints imposed by a Pool implementation may be violated after a detach, as the detached connection is removed from the pool's knowledge and control.

info

Info dictionary associated with the underlying DBAPI connection referred to by this `ConnectionFairy`, allowing user-defined data to be associated with the connection.

The data here will follow along with the DBAPI connection including after it is returned to the connection pool and used again in subsequent instances of *_ConnectionFairy*. It is shared with the *_ConnectionRecord.info* and *Connection.info* accessors.

invalidate(*e=None*, *soft=False*)

Mark this connection as invalidated.

This method can be called directly, and is also called as a result of the *Connection.invalidate()* method. When invoked, the DBAPI connection is immediately closed and discarded from further use by the pool. The invalidation mechanism proceeds via the *_ConnectionRecord.invalidate()* internal method.

Parameters

- **e** – an exception object indicating a reason for the invalidation.
- **soft** – if True, the connection isn't closed; instead, this connection will be recycled on next checkout.

  New in version 1.0.3.

See also:

More on Invalidation

is_valid

Return True if this *_ConnectionFairy* still refers to an active DBAPI connection.

class sqlalchemy.pool.**_ConnectionRecord**(*pool*)

Internal object which maintains an individual DBAPI connection referenced by a *Pool*.

The *_ConnectionRecord* object always exists for any particular DBAPI connection whether or not that DBAPI connection has been "checked out". This is in contrast to the *_ConnectionFairy* which is only a public facade to the DBAPI connection while it is checked out.

A *_ConnectionRecord* may exist for a span longer than that of a single DBAPI connection. For example, if the *_ConnectionRecord.invalidate()* method is called, the DBAPI connection associated with this *_ConnectionRecord* will be discarded, but the *_ConnectionRecord* may be used again, in which case a new DBAPI connection is produced when the *Pool* next uses this record.

The *_ConnectionRecord* is delivered along with connection pool events, including *PoolEvents.connect()* and *PoolEvents.checkout()*, however *_ConnectionRecord* still remains an internal object whose API and internals may change.

See also:

_ConnectionFairy

connection = None

A reference to the actual DBAPI connection being tracked.

May be `None` if this *_ConnectionRecord* has been marked as invalidated; a new DBAPI connection may replace it if the owning pool calls upon this *_ConnectionRecord* to reconnect.

info

The `.info` dictionary associated with the DBAPI connection.

This dictionary is shared among the *_ConnectionFairy.info* and *Connection.info* accessors.

invalidate(*e=None*, *soft=False*)

Invalidate the DBAPI connection held by this *_ConnectionRecord*.

This method is called for all connection invalidations, including when the *_ConnectionFairy.invalidate()* or *Connection.invalidate()* methods are called, as well as when any so-called "automatic invalidation" condition occurs.

Parameters

- **e** – an exception object indicating a reason for the invalidation.
- **soft** – if True, the connection isn't closed; instead, this connection will be recycled on next checkout.

  New in version 1.0.3.

See also:

More on Invalidation

Pooling Plain DB-API Connections

Any **PEP 249** DB-API module can be "proxied" through the connection pool transparently. Usage of the DB-API is exactly as before, except the `connect()` method will consult the pool. Below we illustrate this with `psycopg2`:

```
import sqlalchemy.pool as pool
import psycopg2 as psycopg

psycopg = pool.manage(psycopg)

# then connect normally
connection = psycopg.connect(database='test', username='scott',
                             password='tiger')
```

This produces a `_DBProxy` object which supports the same `connect()` function as the original DB-API module. Upon connection, a connection proxy object is returned, which delegates its calls to a real DB-API connection object. This connection object is stored persistently within a connection pool (an instance of *Pool*) that corresponds to the exact connection arguments sent to the `connect()` function.

The connection proxy supports all of the methods on the original connection object, most of which are proxied via `__getattr__()`. The `close()` method will return the connection to the pool, and the `cursor()` method will return a proxied cursor object. Both the connection proxy and the cursor proxy will also return the underlying connection to the pool after they have both been garbage collected, which is detected via weakref callbacks (`__del__` is not used).

Additionally, when connections are returned to the pool, a `rollback()` is issued on the connection unconditionally. This is to release any locks still held by the connection that may have resulted from normal activity.

By default, the `connect()` method will return the same connection that is already checked out in the current thread. This allows a particular connection to be used in a given thread without needing to pass it around between functions. To disable this behavior, specify `use_threadlocal=False` to the `manage()` function.

`sqlalchemy.pool.manage`(*module*, ***params*)

Return a proxy for a DB-API module that automatically pools connections.

Given a DB-API 2.0 module and pool management parameters, returns a proxy for the module that will automatically pool connections, creating new connection pools for each distinct set of connection arguments sent to the decorated module's connect() function.

Parameters

- **module** – a DB-API 2.0 database module
- **poolclass** – the class used by the pool module to provide pooling. Defaults to *QueuePool*.
- ****params** – will be passed through to *poolclass*

`sqlalchemy.pool.clear_managers`()

Remove all current DB-API 2.0 managers.

All pools and connections are disposed.

3.5.4 Core Events

This section describes the event interfaces provided in SQLAlchemy Core. For an introduction to the event listening API, see *Events*. ORM events are described in *ORM Events*.

class `sqlalchemy.event.base.Events`

Define event listening functions for a particular target type.

Connection Pool Events

class `sqlalchemy.events.PoolEvents`

Bases: *sqlalchemy.event.base.Events*

Available events for *Pool*.

The methods here define the name of an event as well as the names of members that are passed to listener functions.

e.g.:

```
from sqlalchemy import event

def my_on_checkout(dbapi_conn, connection_rec, connection_proxy):
    "handle an on checkout event"

event.listen(Pool, 'checkout', my_on_checkout)
```

In addition to accepting the *Pool* class and *Pool* instances, *PoolEvents* also accepts *Engine* objects and the *Engine* class as targets, which will be resolved to the `.pool` attribute of the given engine or the *Pool* class:

```
engine = create_engine("postgresql://scott:tiger@localhost/test")

# will associate with engine.pool
event.listen(engine, 'checkout', my_on_checkout)
```

checkin(*dbapi_connection*, *connection_record*)

Called when a connection returns to the pool.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'checkin')
def receive_checkin(dbapi_connection, connection_record):
    "listen for the 'checkin' event"

    # ... (event handling logic) ...
```

Note that the connection may be closed, and may be None if the connection has been invalidated. `checkin` will not be called for detached connections. (They do not return to the pool.)

Parameters

- **dbapi_connection** – a DBAPI connection.
- **connection_record** – the *_ConnectionRecord* managing the DBAPI connection.

checkout(*dbapi_connection*, *connection_record*, *connection_proxy*)

Called when a connection is retrieved from the Pool.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'checkout')
def receive_checkout(dbapi_connection, connection_record, connection_proxy):
    "listen for the 'checkout' event"

    # ... (event handling logic) ...
```

Parameters

- **dbapi_connection** – a DBAPI connection.
- **connection_record** – the *_ConnectionRecord* managing the DBAPI connection.
- **connection_proxy** – the *_ConnectionFairy* object which will proxy the public interface of the DBAPI connection for the lifespan of the checkout.

If you raise a *DisconnectionError*, the current connection will be disposed and a fresh connection retrieved. Processing of all checkout listeners will abort and restart using the new connection.

See also:

ConnectionEvents.engine_connect() - a similar event which occurs upon creation of a new *Connection*.

connect(*dbapi_connection*, *connection_record*)

Called at the moment a particular DBAPI connection is first created for a given *Pool*.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'connect')
def receive_connect(dbapi_connection, connection_record):
    "listen for the 'connect' event"

    # ... (event handling logic) ...
```

This event allows one to capture the point directly after which the DBAPI module-level `.connect()` method has been used in order to produce a new DBAPI connection.

Parameters

- **dbapi_connection** – a DBAPI connection.
- **connection_record** – the *_ConnectionRecord* managing the DBAPI connection.

first_connect(*dbapi_connection*, *connection_record*)

Called exactly once for the first time a DBAPI connection is checked out from a particular *Pool*.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'first_connect')
def receive_first_connect(dbapi_connection, connection_record):
    "listen for the 'first_connect' event"

    # ... (event handling logic) ...
```

The rationale for *PoolEvents.first_connect()* is to determine information about a particular series of database connections based on the settings used for all connections. Since a particular *Pool* refers to a single "creator" function (which in terms of a *Engine* refers to the URL and connection options used), it is typically valid to make observations about a single connection that can be safely assumed to be valid about all subsequent connections, such as the database version, the server and client encoding settings, collation settings, and many others.

Parameters

- **dbapi_connection** – a DBAPI connection.
- **connection_record** – the *_ConnectionRecord* managing the DBAPI connection.

invalidate(*dbapi_connection*, *connection_record*, *exception*)

Called when a DBAPI connection is to be "invalidated".

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'invalidate')
def receive_invalidate(dbapi_connection, connection_record, exception):
    "listen for the 'invalidate' event"

    # ... (event handling logic) ...
```

This event is called any time the `_ConnectionRecord.invalidate()` method is invoked, either from API usage or via "auto-invalidation", without the `soft` flag.

The event occurs before a final attempt to call `.close()` on the connection occurs.

Parameters

- **dbapi_connection** – a DBAPI connection.
- **connection_record** – the `_ConnectionRecord` managing the DBAPI connection.
- **exception** – the exception object corresponding to the reason for this invalidation, if any. May be `None`.

New in version 0.9.2: Added support for connection invalidation listening.

See also:

More on Invalidation

reset(*dbapi_connection*, *connection_record*)

Called before the "reset" action occurs for a pooled connection.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngineOrPool, 'reset')
def receive_reset(dbapi_connection, connection_record):
    "listen for the 'reset' event"

    # ... (event handling logic) ...
```

This event represents when the `rollback()` method is called on the DBAPI connection before it is returned to the pool. The behavior of "reset" can be controlled, including disabled, using the `reset_on_return` pool argument.

The `PoolEvents.reset()` event is usually followed by the `PoolEvents.checkin()` event is called, except in those cases where the connection is discarded immediately after reset.

Parameters

- **dbapi_connection** – a DBAPI connection.
- **connection_record** – the `_ConnectionRecord` managing the DBAPI connection.

New in version 0.8.

See also:

`ConnectionEvents.rollback()`

`ConnectionEvents.commit()`

soft_invalidate(*dbapi_connection*, *connection_record*, *exception*)

Called when a DBAPI connection is to be "soft invalidated".

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
```

```
@event.listens_for(SomeEngineOrPool, 'soft_invalidate')
def receive_soft_invalidate(dbapi_connection, connection_record, exception):
    "listen for the 'soft_invalidate' event"

    # ... (event handling logic) ...
```

This event is called any time the `_ConnectionRecord.invalidate()` method is invoked with the `soft` flag.

Soft invalidation refers to when the connection record that tracks this connection will force a reconnect after the current connection is checked in. It does not actively close the dbapi_connection at the point at which it is called.

New in version 1.0.3.

SQL Execution and Connection Events

class `sqlalchemy.events.`**`ConnectionEvents`**

Bases: `sqlalchemy.event.base.Events`

Available events for `Connectable`, which includes `Connection` and `Engine`.

The methods here define the name of an event as well as the names of members that are passed to listener functions.

An event listener can be associated with any `Connectable` class or instance, such as an `Engine`, e.g.:

```
from sqlalchemy import event, create_engine

def before_cursor_execute(conn, cursor, statement, parameters, context,
                                                    executemany):
    log.info("Received statement: %s" % statement)

engine = create_engine('postgresql://scott:tiger@localhost/test')
event.listen(engine, "before_cursor_execute", before_cursor_execute)
```

or with a specific `Connection`:

```
with engine.begin() as conn:
    @event.listens_for(conn, 'before_cursor_execute')
    def before_cursor_execute(conn, cursor, statement, parameters,
                                        context, executemany):
        log.info("Received statement: %s" % statement)
```

When the methods are called with a *statement* parameter, such as in `after_cursor_execute()`, `before_cursor_execute()` and `dbapi_error()`, the statement is the exact SQL string that was prepared for transmission to the DBAPI `cursor` in the connection's `Dialect`.

The `before_execute()` and `before_cursor_execute()` events can also be established with the `retval=True` flag, which allows modification of the statement and parameters to be sent to the database. The `before_cursor_execute()` event is particularly useful here to add ad-hoc string transformations, such as comments, to all executions:

```
from sqlalchemy.engine import Engine
from sqlalchemy import event

@event.listens_for(Engine, "before_cursor_execute", retval=True)
```

```
def comment_sql_calls(conn, cursor, statement, parameters,
                                  context, executemany):
    statement = statement + " -- some comment"
    return statement, parameters
```

Note: *ConnectionEvents* can be established on any combination of *Engine*, *Connection*, as well as instances of each of those classes. Events across all four scopes will fire off for a given instance of *Connection*. However, for performance reasons, the *Connection* object determines at instantiation time whether or not its parent *Engine* has event listeners established. Event listeners added to the *Engine* class or to an instance of *Engine* *after* the instantiation of a dependent *Connection* instance will usually *not* be available on that *Connection* instance. The newly added listeners will instead take effect for *Connection* instances created subsequent to those event listeners being established on the parent *Engine* class or instance.

Parameters **retval=False** – Applies to the *before_execute()* and *before_cursor_execute()* events only. When True, the user-defined event function must have a return value, which is a tuple of parameters that replace the given statement and parameters. See those methods for a description of specific return arguments.

Changed in version 0.8: *ConnectionEvents* can now be associated with any *Connectable* including *Connection*, in addition to the existing support for *Engine*.

after_cursor_execute(*conn*, *cursor*, *statement*, *parameters*, *context*, *executemany*)

Intercept low-level cursor execute() events after execution.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'after_cursor_execute')
def receive_after_cursor_execute(conn, cursor, statement, parameters, context, executemany)
    "listen for the 'after_cursor_execute' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeEngine, 'after_cursor_execute', named=True)
def receive_after_cursor_execute(**kw):
    "listen for the 'after_cursor_execute' event"
    conn = kw['conn']
    cursor = kw['cursor']

    # ... (event handling logic) ...
```

Parameters

- **conn** – *Connection* object
- **cursor** – DBAPI cursor object. Will have results pending if the statement was a SELECT, but these should not be consumed as they will be needed by the *ResultProxy*.
- **statement** – string SQL statement, as passed to the DBAPI
- **parameters** – Dictionary, tuple, or list of parameters being passed to the `execute()` or `executemany()` method of the DBAPI `cursor`. In some cases may be `None`.

- **context** – *ExecutionContext* object in use. May be None.
- **executemany** – boolean, if True, this is an executemany() call, if False, this is an execute() call.

after_execute(*conn*, *clauseelement*, *multiparams*, *params*, *result*)

Intercept high level execute() events after execute.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'after_execute')
def receive_after_execute(conn, clauseelement, multiparams, params, result):
    "listen for the 'after_execute' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeEngine, 'after_execute', named=True)
def receive_after_execute(**kw):
    "listen for the 'after_execute' event"
    conn = kw['conn']
    clauseelement = kw['clauseelement']

    # ... (event handling logic) ...
```

Parameters

- **conn** – *Connection* object
- **clauseelement** – SQL expression construct, *Compiled* instance, or string statement passed to *Connection.execute()*.
- **multiparams** – Multiple parameter sets, a list of dictionaries.
- **params** – Single parameter set, a single dictionary.
- **result** – *ResultProxy* generated by the execution.

before_cursor_execute(*conn*, *cursor*, *statement*, *parameters*, *context*, *executemany*)

Intercept low-level cursor execute() events before execution, receiving the string SQL statement and DBAPI-specific parameter list to be invoked against a cursor.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'before_cursor_execute')
def receive_before_cursor_execute(conn, cursor, statement, parameters, context, executemany
    "listen for the 'before_cursor_execute' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeEngine, 'before_cursor_execute', named=True)
def receive_before_cursor_execute(**kw):
    "listen for the 'before_cursor_execute' event"
    conn = kw['conn']
```

```
        cursor = kw['cursor']

        # ... (event handling logic) ...
```

This event is a good choice for logging as well as late modifications to the SQL string. It's less ideal for parameter modifications except for those which are specific to a target backend.

This event can be optionally established with the `retval=True` flag. The `statement` and `parameters` arguments should be returned as a two-tuple in this case:

```
    @event.listens_for(Engine, "before_cursor_execute", retval=True)
    def before_cursor_execute(conn, cursor, statement,
                              parameters, context, executemany):
        # do something with statement, parameters
        return statement, parameters
```

See the example at *ConnectionEvents*.

Parameters

- **conn** – *Connection* object
- **cursor** – DBAPI cursor object
- **statement** – string SQL statement, as to be passed to the DBAPI
- **parameters** – Dictionary, tuple, or list of parameters being passed to the `execute()` or `executemany()` method of the DBAPI `cursor`. In some cases may be `None`.
- **context** – *ExecutionContext* object in use. May be `None`.
- **executemany** – boolean, if `True`, this is an `executemany()` call, if `False`, this is an `execute()` call.

See also:

before_execute()

after_cursor_execute()

before_execute(*conn*, *clauseelement*, *multiparams*, *params*)

Intercept high level execute() events, receiving uncompiled SQL constructs and other objects prior to rendering into SQL.

Example argument forms:

```
    from sqlalchemy import event

    # standard decorator style
    @event.listens_for(SomeEngine, 'before_execute')
    def receive_before_execute(conn, clauseelement, multiparams, params):
        "listen for the 'before_execute' event"

        # ... (event handling logic) ...

    # named argument style (new in 0.9)
    @event.listens_for(SomeEngine, 'before_execute', named=True)
    def receive_before_execute(**kw):
        "listen for the 'before_execute' event"
        conn = kw['conn']
        clauseelement = kw['clauseelement']
```

```
        # ... (event handling logic) ...
```

This event is good for debugging SQL compilation issues as well as early manipulation of the parameters being sent to the database, as the parameter lists will be in a consistent format here.

This event can be optionally established with the `retval=True` flag. The `clauseelement`, `multiparams`, and `params` arguments should be returned as a three-tuple in this case:

```
@event.listens_for(Engine, "before_execute", retval=True)
def before_execute(conn, conn, clauseelement, multiparams, params):
    # do something with clauseelement, multiparams, params
    return clauseelement, multiparams, params
```

Parameters

- **conn** – *Connection* object
- **clauseelement** – SQL expression construct, *Compiled* instance, or string statement passed to *Connection.execute()*.
- **multiparams** – Multiple parameter sets, a list of dictionaries.
- **params** – Single parameter set, a single dictionary.

See also:

before_cursor_execute()

begin(*conn*)

Intercept begin() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'begin')
def receive_begin(conn):
    "listen for the 'begin' event"

    # ... (event handling logic) ...
```

Parameters **conn** – *Connection* object

begin_twophase(*conn*, *xid*)

Intercept begin_twophase() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'begin_twophase')
def receive_begin_twophase(conn, xid):
    "listen for the 'begin_twophase' event"

    # ... (event handling logic) ...
```

Parameters

- **conn** – *Connection* object
- **xid** – two-phase XID identifier

commit(*conn*)

Intercept commit() events, as initiated by a *Transaction*.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'commit')
def receive_commit(conn):
    "listen for the 'commit' event"

    # ... (event handling logic) ...
```

Note that the *Pool* may also "auto-commit" a DBAPI connection upon checkin, if the `reset_on_return` flag is set to the value `'commit'`. To intercept this commit, use the *PoolEvents.reset()* hook.

Parameters **conn** – *Connection* object

commit_twophase(*conn*, *xid*, *is_prepared*)

Intercept commit_twophase() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'commit_twophase')
def receive_commit_twophase(conn, xid, is_prepared):
    "listen for the 'commit_twophase' event"

    # ... (event handling logic) ...
```

Parameters

- **conn** – *Connection* object
- **xid** – two-phase XID identifier
- **is_prepared** – boolean, indicates if *TwoPhaseTransaction.prepare()* was called.

dbapi_error(*conn*, *cursor*, *statement*, *parameters*, *context*, *exception*)

Intercept a raw DBAPI error.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'dbapi_error')
def receive_dbapi_error(conn, cursor, statement, parameters, context, exception):
    "listen for the 'dbapi_error' event"

    # ... (event handling logic) ...
```

```
# named argument style (new in 0.9)
@event.listens_for(SomeEngine, 'dbapi_error', named=True)
def receive_dbapi_error(**kw):
    "listen for the 'dbapi_error' event"
    conn = kw['conn']
    cursor = kw['cursor']

    # ... (event handling logic) ...
```

This event is called with the DBAPI exception instance received from the DBAPI itself, *before* SQLAlchemy wraps the exception with it's own exception wrappers, and before any other operations are performed on the DBAPI cursor; the existing transaction remains in effect as well as any state on the cursor.

The use case here is to inject low-level exception handling into an *Engine*, typically for logging and debugging purposes.

> **Warning:** Code should **not** modify any state or throw any exceptions here as this will interfere with SQLAlchemy's cleanup and error handling routines. For exception modification, please refer to the new *ConnectionEvents.handle_error()* event.

Subsequent to this hook, SQLAlchemy may attempt any number of operations on the connection/cursor, including closing the cursor, rolling back of the transaction in the case of connectionless execution, and disposing of the entire connection pool if a "disconnect" was detected. The exception is then wrapped in a SQLAlchemy DBAPI exception wrapper and re-thrown.

Parameters

- **conn** – *Connection* object
- **cursor** – DBAPI cursor object
- **statement** – string SQL statement, as passed to the DBAPI
- **parameters** – Dictionary, tuple, or list of parameters being passed to the `execute()` or `executemany()` method of the DBAPI `cursor`. In some cases may be `None`.
- **context** – *ExecutionContext* object in use. May be `None`.
- **exception** – The **unwrapped** exception emitted directly from the DBAPI. The class here is specific to the DBAPI module in use.

Deprecated since version 0.9.7: - replaced by *ConnectionEvents.handle_error()*

engine_connect(*conn*, *branch*)

Intercept the creation of a new *Connection*.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'engine_connect')
def receive_engine_connect(conn, branch):
    "listen for the 'engine_connect' event"

    # ... (event handling logic) ...
```

This event is called typically as the direct result of calling the *Engine.connect()* method.

It differs from the `PoolEvents.connect()` method, which refers to the actual connection to a database at the DBAPI level; a DBAPI connection may be pooled and reused for many operations. In contrast, this event refers only to the production of a higher level `Connection` wrapper around such a DBAPI connection.

It also differs from the `PoolEvents.checkout()` event in that it is specific to the `Connection` object, not the DBAPI connection that `PoolEvents.checkout()` deals with, although this DBAPI connection is available here via the `Connection.connection` attribute. But note there can in fact be multiple `PoolEvents.checkout()` events within the lifespan of a single `Connection` object, if that `Connection` is invalidated and re-established. There can also be multiple `Connection` objects generated for the same already-checked-out DBAPI connection, in the case that a "branch" of a `Connection` is produced.

Parameters

- **conn** – `Connection` object.
- **branch** – if True, this is a "branch" of an existing `Connection`. A branch is generated within the course of a statement execution to invoke supplemental statements, most typically to pre-execute a SELECT of a default value for the purposes of an INSERT statement.

New in version 0.9.0.

See also:

Disconnect Handling - Pessimistic - illustrates how to use `ConnectionEvents.engine_connect()` to transparently ensure pooled connections are connected to the database.

`PoolEvents.checkout()` the lower-level pool checkout event for an individual DBAPI connection

`ConnectionEvents.set_connection_execution_options()` - a copy of a `Connection` is also made when the `Connection.execution_options()` method is called.

engine_disposed(*engine*)

Intercept when the `Engine.dispose()` method is called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'engine_disposed')
def receive_engine_disposed(engine):
    "listen for the 'engine_disposed' event"

    # ... (event handling logic) ...
```

The `Engine.dispose()` method instructs the engine to "dispose" of it's connection pool (e.g. `Pool`), and replaces it with a new one. Disposing of the old pool has the effect that existing checked-in connections are closed. The new pool does not establish any new connections until it is first used.

This event can be used to indicate that resources related to the `Engine` should also be cleaned up, keeping in mind that the `Engine` can still be used for new requests in which case it re-acquires connection resources.

New in version 1.0.5.

handle_error(*exception_context*)

Intercept all exceptions processed by the `Connection`.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'handle_error')
def receive_handle_error(exception_context):
    "listen for the 'handle_error' event"

    # ... (event handling logic) ...
```

This includes all exceptions emitted by the DBAPI as well as within SQLAlchemy's statement invocation process, including encoding errors and other statement validation errors. Other areas in which the event is invoked include transaction begin and end, result row fetching, cursor creation.

Note that `handle_error()` may support new kinds of exceptions and new calling scenarios at *any time*. Code which uses this event must expect new calling patterns to be present in minor releases.

To support the wide variety of members that correspond to an exception, as well as to allow extensibility of the event without backwards incompatibility, the sole argument received is an instance of `ExceptionContext`. This object contains data members representing detail about the exception.

Use cases supported by this hook include:

- read-only, low-level exception handling for logging and debugging purposes
- exception re-writing

The hook is called while the cursor from the failed operation (if any) is still open and accessible. Special cleanup operations can be called on this cursor; SQLAlchemy will attempt to close this cursor subsequent to this hook being invoked. If the connection is in "autocommit" mode, the transaction also remains open within the scope of this hook; the rollback of the per-statement transaction also occurs after the hook is called.

The user-defined event handler has two options for replacing the SQLAlchemy-constructed exception into one that is user defined. It can either raise this new exception directly, in which case all further event listeners are bypassed and the exception will be raised, after appropriate cleanup as taken place:

```
@event.listens_for(Engine, "handle_error")
def handle_exception(context):
    if isinstance(context.original_exception,
        psycopg2.OperationalError) and \
        "failed" in str(context.original_exception):
        raise MySpecialException("failed operation")
```

> **Warning:** Because the `ConnectionEvents.handle_error()` event specifically provides for exceptions to be re-thrown as the ultimate exception raised by the failed statement, **stack traces will be misleading** if the user-defined event handler itself fails and throws an unexpected exception; the stack trace may not illustrate the actual code line that failed! It is advised to code carefully here and use logging and/or inline debugging if unexpected exceptions are occurring.

Alternatively, a "chained" style of event handling can be used, by configuring the handler with the `retval=True` modifier and returning the new exception instance from the function. In this case, event handling will continue onto the next handler. The "chained" exception is available using `ExceptionContext.chained_exception`:

```
@event.listens_for(Engine, "handle_error", retval=True)
def handle_exception(context):
    if context.chained_exception is not None and \
        "special" in context.chained_exception.message:
```

```
                return MySpecialException("failed",
                    cause=context.chained_exception)
```

Handlers that return `None` may remain within this chain; the last non-`None` return value is the one that continues to be passed to the next handler.

When a custom exception is raised or returned, SQLAlchemy raises this new exception as-is, it is not wrapped by any SQLAlchemy object. If the exception is not a subclass of *sqlalchemy.exc.StatementError*, certain features may not be available; currently this includes the ORM's feature of adding a detail hint about "autoflush" to exceptions raised within the autoflush process.

Parameters **context** – an *ExceptionContext* object. See this class for details on all available members.

New in version 0.9.7: Added the *ConnectionEvents.handle_error()* hook.

Changed in version 1.0.0: The *handle_error()* event is now invoked when an *Engine* fails during the initial call to *Engine.connect()*, as well as when a *Connection* object encounters an error during a reconnect operation.

Changed in version 1.0.0: The *handle_error()* event is not fired off when a dialect makes use of the `skip_user_error_events` execution option. This is used by dialects which intend to catch SQLAlchemy-specific exceptions within specific operations, such as when the MySQL dialect detects a table not present within the `has_table()` dialect method. Prior to 1.0.0, code which implements *handle_error()* needs to ensure that exceptions thrown in these scenarios are re-raised without modification.

prepare_twophase(*conn*, *xid*)

Intercept prepare_twophase() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'prepare_twophase')
def receive_prepare_twophase(conn, xid):
    "listen for the 'prepare_twophase' event"

    # ... (event handling logic) ...
```

Parameters

- **conn** – *Connection* object
- **xid** – two-phase XID identifier

release_savepoint(*conn*, *name*, *context*)

Intercept release_savepoint() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'release_savepoint')
def receive_release_savepoint(conn, name, context):
    "listen for the 'release_savepoint' event"
```

```
        # ... (event handling logic) ...
```

Parameters

- **conn** – *Connection* object
- **name** – specified name used for the savepoint.
- **context** – *ExecutionContext* in use. May be `None`.

rollback(*conn*)

Intercept rollback() events, as initiated by a *Transaction*.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'rollback')
def receive_rollback(conn):
    "listen for the 'rollback' event"

    # ... (event handling logic) ...
```

Note that the *Pool* also "auto-rolls back" a DBAPI connection upon checkin, if the `reset_on_return` flag is set to its default value of `'rollback'`. To intercept this rollback, use the *PoolEvents.reset()* hook.

Parameters **conn** – *Connection* object

See also:

PoolEvents.reset()

rollback_savepoint(*conn*, *name*, *context*)

Intercept rollback_savepoint() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'rollback_savepoint')
def receive_rollback_savepoint(conn, name, context):
    "listen for the 'rollback_savepoint' event"

    # ... (event handling logic) ...
```

Parameters

- **conn** – *Connection* object
- **name** – specified name used for the savepoint.
- **context** – *ExecutionContext* in use. May be `None`.

rollback_twophase(*conn*, *xid*, *is_prepared*)

Intercept rollback_twophase() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'rollback_twophase')
def receive_rollback_twophase(conn, xid, is_prepared):
    "listen for the 'rollback_twophase' event"

    # ... (event handling logic) ...
```

Parameters

- **conn** – *Connection* object
- **xid** – two-phase XID identifier
- **is_prepared** – boolean, indicates if *TwoPhaseTransaction.prepare()* was called.

savepoint(*conn*, *name*)

Intercept savepoint() events.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'savepoint')
def receive_savepoint(conn, name):
    "listen for the 'savepoint' event"

    # ... (event handling logic) ...
```

Parameters

- **conn** – *Connection* object
- **name** – specified name used for the savepoint.

set_connection_execution_options(*conn*, *opts*)

Intercept when the *Connection.execution_options()* method is called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'set_connection_execution_options')
def receive_set_connection_execution_options(conn, opts):
    "listen for the 'set_connection_execution_options' event"

    # ... (event handling logic) ...
```

This method is called after the new *Connection* has been produced, with the newly updated execution options collection, but before the *Dialect* has acted upon any of those new options.

Note that this method is not called when a new *Connection* is produced which is inheriting execution options from its parent *Engine*; to intercept this condition, use the *ConnectionEvents.engine_connect()* event.

Parameters

- **conn** – The newly copied *Connection* object
- **opts** – dictionary of options that were passed to the *Connection.execution_options()* method.

New in version 0.9.0.

See also:

ConnectionEvents.set_engine_execution_options() - event which is called when *Engine.execution_options()* is called.

set_engine_execution_options(*engine*, *opts*)

Intercept when the *Engine.execution_options()* method is called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'set_engine_execution_options')
def receive_set_engine_execution_options(engine, opts):
    "listen for the 'set_engine_execution_options' event"

    # ... (event handling logic) ...
```

The *Engine.execution_options()* method produces a shallow copy of the *Engine* which stores the new options. That new *Engine* is passed here. A particular application of this method is to add a *ConnectionEvents.engine_connect()* event handler to the given *Engine* which will perform some per- *Connection* task specific to these execution options.

Parameters

- **conn** – The newly copied *Engine* object
- **opts** – dictionary of options that were passed to the *Connection.execution_options()* method.

New in version 0.9.0.

See also:

ConnectionEvents.set_connection_execution_options() - event which is called when *Connection.execution_options()* is called.

class sqlalchemy.events.**DialectEvents**

Bases: *sqlalchemy.event.base.Events*

event interface for execution-replacement functions.

These events allow direct instrumentation and replacement of key dialect functions which interact with the DBAPI.

Note: *DialectEvents* hooks should be considered **semi-public** and experimental. These hooks are not for general use and are only for those situations where intricate re-statement of DBAPI mechanics must be injected onto an existing dialect. For general-use statement-interception events, please use the *ConnectionEvents* interface.

See also:

ConnectionEvents.before_cursor_execute()

ConnectionEvents.before_execute()

ConnectionEvents.after_cursor_execute()

ConnectionEvents.after_execute()

New in version 0.9.4.

do_connect(*dialect*, *conn_rec*, *cargs*, *cparams*)

Receive connection arguments before a connection is made.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'do_connect')
def receive_do_connect(dialect, conn_rec, cargs, cparams):
    "listen for the 'do_connect' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeEngine, 'do_connect', named=True)
def receive_do_connect(**kw):
    "listen for the 'do_connect' event"
    dialect = kw['dialect']
    conn_rec = kw['conn_rec']

    # ... (event handling logic) ...
```

Return a DBAPI connection to halt further events from invoking; the returned connection will be used.

Alternatively, the event can manipulate the cargs and/or cparams collections; cargs will always be a Python list that can be mutated in-place and cparams a Python dictionary. Return None to allow control to pass to the next event handler and ultimately to allow the dialect to connect normally, given the updated arguments.

New in version 1.0.3.

do_execute(*cursor*, *statement*, *parameters*, *context*)

Receive a cursor to have execute() called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'do_execute')
def receive_do_execute(cursor, statement, parameters, context):
    "listen for the 'do_execute' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeEngine, 'do_execute', named=True)
def receive_do_execute(**kw):
    "listen for the 'do_execute' event"
    cursor = kw['cursor']
    statement = kw['statement']

    # ... (event handling logic) ...
```

Return the value True to halt further events from invoking, and to indicate that the cursor execution has already taken place within the event handler.

do_execute_no_params(*cursor*, *statement*, *context*)
Receive a cursor to have execute() with no parameters called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'do_execute_no_params')
def receive_do_execute_no_params(cursor, statement, context):
    "listen for the 'do_execute_no_params' event"

    # ... (event handling logic) ...
```

Return the value True to halt further events from invoking, and to indicate that the cursor execution has already taken place within the event handler.

do_executemany(*cursor*, *statement*, *parameters*, *context*)
Receive a cursor to have executemany() called.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeEngine, 'do_executemany')
def receive_do_executemany(cursor, statement, parameters, context):
    "listen for the 'do_executemany' event"

    # ... (event handling logic) ...

# named argument style (new in 0.9)
@event.listens_for(SomeEngine, 'do_executemany', named=True)
def receive_do_executemany(**kw):
    "listen for the 'do_executemany' event"
    cursor = kw['cursor']
    statement = kw['statement']

    # ... (event handling logic) ...
```

Return the value True to halt further events from invoking, and to indicate that the cursor execution has already taken place within the event handler.

Schema Events

class sqlalchemy.events.**DDLEvents**
Bases: *sqlalchemy.event.base.Events*

Define event listeners for schema objects, that is, *SchemaItem* and other *SchemaEventTarget* subclasses, including *MetaData*, *Table*, *Column*.

MetaData and *Table* support events specifically regarding when CREATE and DROP DDL is emitted to the database.

Attachment events are also provided to customize behavior whenever a child schema element is associated with a parent, such as, when a *Column* is associated with its *Table*, when a *ForeignKeyConstraint* is associated with a *Table*, etc.

Example using the `after_create` event:

```
from sqlalchemy import event
from sqlalchemy import Table, Column, Metadata, Integer

m = MetaData()
some_table = Table('some_table', m, Column('data', Integer))

def after_create(target, connection, **kw):
    connection.execute("ALTER TABLE %s SET name=foo_%s" %
                            (target.name, target.name))

event.listen(some_table, "after_create", after_create)
```

DDL events integrate closely with the *DDL* class and the *DDLElement* hierarchy of DDL clause constructs, which are themselves appropriate as listener callables:

```
from sqlalchemy import DDL
event.listen(
    some_table,
    "after_create",
    DDL("ALTER TABLE %(table)s SET name=foo_%(table)s")
)
```

The methods here define the name of an event as well as the names of members that are passed to listener functions.

See also:

Events

DDLElement

DDL

Controlling DDL Sequences

after_create(*target*, *connection*, ***kw*)

Called after CREATE statements are emitted.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSchemaClassOrObject, 'after_create')
def receive_after_create(target, connection, **kw):
    "listen for the 'after_create' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the *MetaData* or *Table* object which is the target of the event.
- **connection** – the *Connection* where the CREATE statement or statements have been emitted.
- ****kw** – additional keyword arguments relevant to the event. The contents of this dictionary may vary across releases, and include the list of tables being generated for a metadata-level event, the checkfirst flag, and other elements used by internal events.

after_drop(*target*, *connection*, ***kw*)

Called after DROP statements are emitted.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSchemaClassOrObject, 'after_drop')
def receive_after_drop(target, connection, **kw):
    "listen for the 'after_drop' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the *MetaData* or *Table* object which is the target of the event.
- **connection** – the *Connection* where the DROP statement or statements have been emitted.
- ****kw** – additional keyword arguments relevant to the event. The contents of this dictionary may vary across releases, and include the list of tables being generated for a metadata-level event, the checkfirst flag, and other elements used by internal events.

after_parent_attach(*target*, *parent*)

Called after a *SchemaItem* is associated with a parent *SchemaItem*.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSchemaClassOrObject, 'after_parent_attach')
def receive_after_parent_attach(target, parent):
    "listen for the 'after_parent_attach' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the target object
- **parent** – the parent to which the target is being attached.

event.listen() also accepts a modifier for this event:

Parameters **propagate=False** – When True, the listener function will be established for any copies made of the target object, i.e. those copies that are generated when *Table.tometadata()* is used.

before_create(*target*, *connection*, ***kw*)

Called before CREATE statements are emitted.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSchemaClassOrObject, 'before_create')
```

```
def receive_before_create(target, connection, **kw):
    "listen for the 'before_create' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the *MetaData* or *Table* object which is the target of the event.
- **connection** – the *Connection* where the CREATE statement or statements will be emitted.
- ****kw** – additional keyword arguments relevant to the event. The contents of this dictionary may vary across releases, and include the list of tables being generated for a metadata-level event, the checkfirst flag, and other elements used by internal events.

before_drop(*target*, *connection*, ***kw*)

Called before DROP statements are emitted.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSchemaClassOrObject, 'before_drop')
def receive_before_drop(target, connection, **kw):
    "listen for the 'before_drop' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the *MetaData* or *Table* object which is the target of the event.
- **connection** – the *Connection* where the DROP statement or statements will be emitted.
- ****kw** – additional keyword arguments relevant to the event. The contents of this dictionary may vary across releases, and include the list of tables being generated for a metadata-level event, the checkfirst flag, and other elements used by internal events.

before_parent_attach(*target*, *parent*)

Called before a *SchemaItem* is associated with a parent *SchemaItem*.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSchemaClassOrObject, 'before_parent_attach')
def receive_before_parent_attach(target, parent):
    "listen for the 'before_parent_attach' event"

    # ... (event handling logic) ...
```

Parameters

- **target** – the target object
- **parent** – the parent to which the target is being attached.

event.listen() also accepts a modifier for this event:

Parameters **propagate=False** – When True, the listener function will be established for any copies made of the target object, i.e. those copies that are generated when *Table.tometadata()* is used.

column_reflect(*inspector*, *table*, *column_info*)

Called for each unit of 'column info' retrieved when a *Table* is being reflected.

Example argument forms:

```
from sqlalchemy import event

# standard decorator style
@event.listens_for(SomeSchemaClassOrObject, 'column_reflect')
def receive_column_reflect(inspector, table, column_info):
    "listen for the 'column_reflect' event"

    # ... (event handling logic) ...
```

The dictionary of column information as returned by the dialect is passed, and can be modified. The dictionary is that returned in each element of the list returned by *reflection.Inspector.get_columns()*.

The event is called before any action is taken against this dictionary, and the contents can be modified. The *Column* specific arguments `info`, `key`, and `quote` can also be added to the dictionary and will be passed to the constructor of *Column*.

Note that this event is only meaningful if either associated with the *Table* class across the board, e.g.:

```
from sqlalchemy.schema import Table
from sqlalchemy import event

def listen_for_reflect(inspector, table, column_info):
    "receive a column_reflect event"
    # ...

event.listen(
        Table,
        'column_reflect',
        listen_for_reflect)
```

...or with a specific *Table* instance using the `listeners` argument:

```
def listen_for_reflect(inspector, table, column_info):
    "receive a column_reflect event"
    # ...

t = Table(
    'sometable',
    autoload=True,
    listeners=[
        ('column_reflect', listen_for_reflect)
    ])
```

This because the reflection process initiated by `autoload=True` completes within the scope of the constructor for *Table*.

class `sqlalchemy.events.`**`SchemaEventTarget`**

Base class for elements that are the targets of *DDLEvents* events.

This includes *SchemaItem* as well as *SchemaType*.

3.6 Core API Basics

3.6.1 Events

SQLAlchemy includes an event API which publishes a wide variety of hooks into the internals of both SQLAlchemy Core and ORM.

New in version 0.7: The system supersedes the previous system of "extension", "proxy", and "listener" classes.

Event Registration

Subscribing to an event occurs through a single API point, the *listen()* function, or alternatively the *listens_for()* decorator. These functions accept a target, a string identifier which identifies the event to be intercepted, and a user-defined listening function. Additional positional and keyword arguments to these two functions may be supported by specific types of events, which may specify alternate interfaces for the given event function, or provide instructions regarding secondary event targets based on the given target.

The name of an event and the argument signature of a corresponding listener function is derived from a class bound specification method, which exists bound to a marker class that's described in the documentation. For example, the documentation for *PoolEvents.connect()* indicates that the event name is `"connect"` and that a user-defined listener function should receive two positional arguments:

```
from sqlalchemy.event import listen
from sqlalchemy.pool import Pool

def my_on_connect(dbapi_con, connection_record):
    print "New DBAPI connection:", dbapi_con

listen(Pool, 'connect', my_on_connect)
```

To listen with the *listens_for()* decorator looks like:

```
from sqlalchemy.event import listens_for
from sqlalchemy.pool import Pool

@listens_for(Pool, "connect")
def my_on_connect(dbapi_con, connection_record):
    print "New DBAPI connection:", dbapi_con
```

Named Argument Styles

There are some varieties of argument styles which can be accepted by listener functions. Taking the example of *PoolEvents.connect()*, this function is documented as receiving `dbapi_connection` and `connection_record` arguments. We can opt to receive these arguments by name, by establishing a listener function that accepts `**keyword` arguments, by passing `named=True` to either *listen()* or *listens_for()*:

```
from sqlalchemy.event import listens_for
from sqlalchemy.pool import Pool

@listens_for(Pool, "connect", named=True)
def my_on_connect(**kw):
    print("New DBAPI connection:", kw['dbapi_connection'])
```

When using named argument passing, the names listed in the function argument specification will be used as keys in the dictionary.

Named style passes all arguments by name regardless of the function signature, so specific arguments may be listed as well, in any order, as long as the names match up:

```
from sqlalchemy.event import listens_for
from sqlalchemy.pool import Pool

@listens_for(Pool, "connect", named=True)
def my_on_connect(dbapi_connection, **kw):
    print("New DBAPI connection:", dbapi_connection)
    print("Connection record:", kw['connection_record'])
```

Above, the presence of `**kw` tells *listens_for()* that arguments should be passed to the function by name, rather than positionally.

New in version 0.9.0: Added optional `named` argument dispatch to event calling.

Targets

The *listen()* function is very flexible regarding targets. It generally accepts classes, instances of those classes, and related classes or objects from which the appropriate target can be derived. For example, the above mentioned `"connect"` event accepts *Engine* classes and objects as well as *Pool* classes and objects:

```
from sqlalchemy.event import listen
from sqlalchemy.pool import Pool, QueuePool
from sqlalchemy import create_engine
from sqlalchemy.engine import Engine
import psycopg2

def connect():
    return psycopg2.connect(username='ed', host='127.0.0.1', dbname='test')

my_pool = QueuePool(connect)
my_engine = create_engine('postgresql://ed@localhost/test')

# associate listener with all instances of Pool
listen(Pool, 'connect', my_on_connect)

# associate listener with all instances of Pool
# via the Engine class
listen(Engine, 'connect', my_on_connect)

# associate listener with my_pool
listen(my_pool, 'connect', my_on_connect)

# associate listener with my_engine.pool
listen(my_engine, 'connect', my_on_connect)
```

Modifiers

Some listeners allow modifiers to be passed to `listen()`. These modifiers sometimes provide alternate calling signatures for listeners. Such as with ORM events, some event listeners can have a return value which modifies the subsequent handling. By default, no listener ever requires a return value, but by passing `retval=True` this value can be supported:

```
def validate_phone(target, value, oldvalue, initiator):
    """Strip non-numeric characters from a phone number"""

    return re.sub(r'(?![0-9])', '', value)

# setup listener on UserContact.phone attribute, instructing
# it to use the return value
listen(UserContact.phone, 'set', validate_phone, retval=True)
```

Event Reference

Both SQLAlchemy Core and SQLAlchemy ORM feature a wide variety of event hooks:

- **Core Events** - these are described in *Core Events* and include event hooks specific to connection pool lifecycle, SQL statement execution, transaction lifecycle, and schema creation and teardown.
- **ORM Events** - these are described in *ORM Events*, and include event hooks specific to class and attribute instrumentation, object initialization hooks, attribute on-change hooks, session state, flush, and commit hooks, mapper initialization, object/result population, and per-instance persistence hooks.

API Reference

`sqlalchemy.event.`**`listen`**(*target*, *identifier*, *fn*, **args*, ***kw*)

Register a listener function for the given target.

e.g.:

```
from sqlalchemy import event
from sqlalchemy.schema import UniqueConstraint

def unique_constraint_name(const, table):
    const.name = "uq_%s_%s" % (
        table.name,
        list(const.columns)[0].name
    )
event.listen(
        UniqueConstraint,
        "after_parent_attach",
        unique_constraint_name)
```

A given function can also be invoked for only the first invocation of the event using the `once` argument:

```
def on_config():
    do_config()

event.listen(Mapper, "before_configure", on_config, once=True)
```

New in version 0.9.4: Added `once=True` to *event.listen()* and *event.listens_for()*.

Note: The *listen()* function cannot be called at the same time that the target event is being run. This has implications for thread safety, and also means an event cannot be added from inside the listener function for itself. The list of events to be run are present inside of a mutable collection that can't be changed during iteration.

Event registration and removal is not intended to be a "high velocity" operation; it is a configurational operation. For systems that need to quickly associate and deassociate with events at high scale, use a mutable structure that is handled from inside of a single listener.

Changed in version 1.0.0: - a `collections.deque()` object is now used as the container for the list of events, which explicitly disallows collection mutation while the collection is being iterated.

See also:

listens_for()

remove()

`sqlalchemy.event.`**`listens_for`**(*target*, *identifier*, **args*, ***kw*)

Decorate a function as a listener for the given target + identifier.

e.g.:

```
from sqlalchemy import event
from sqlalchemy.schema import UniqueConstraint

@event.listens_for(UniqueConstraint, "after_parent_attach")
def unique_constraint_name(const, table):
    const.name = "uq_%s_%s" % (
        table.name,
        list(const.columns)[0].name
    )
```

A given function can also be invoked for only the first invocation of the event using the `once` argument:

```
@event.listens_for(Mapper, "before_configure", once=True)
def on_config():
    do_config()
```

New in version 0.9.4: Added `once=True` to *event.listen()* and *event.listens_for()*.

See also:

listen() - general description of event listening

`sqlalchemy.event.`**`remove`**(*target*, *identifier*, *fn*)

Remove an event listener.

The arguments here should match exactly those which were sent to *listen()*; all the event registration which proceeded as a result of this call will be reverted by calling *remove()* with the same arguments.

e.g.:

```
# if a function was registered like this...
@event.listens_for(SomeMappedClass, "before_insert", propagate=True)
def my_listener_function(*arg):
    pass
```

```
# ... it's removed like this
event.remove(SomeMappedClass, "before_insert", my_listener_function)
```

Above, the listener function associated with `SomeMappedClass` was also propagated to subclasses of `SomeMappedClass`; the *remove()* function will revert all of these operations.

New in version 0.9.0.

Note: The *remove()* function cannot be called at the same time that the target event is being run. This has implications for thread safety, and also means an event cannot be removed from inside the listener function for itself. The list of events to be run are present inside of a mutable collection that can't be changed during iteration.

Event registration and removal is not intended to be a "high velocity" operation; it is a configurational operation. For systems that need to quickly associate and deassociate with events at high scale, use a mutable structure that is handled from inside of a single listener.

Changed in version 1.0.0: - a `collections.deque()` object is now used as the container for the list of events, which explicitly disallows collection mutation while the collection is being iterated.

See also:

listen()

`sqlalchemy.event.`**`contains`**(*target*, *identifier*, *fn*)

Return True if the given target/ident/fn is set up to listen.

New in version 0.9.0.

3.6.2 Runtime Inspection API

The inspection module provides the *inspect()* function, which delivers runtime information about a wide variety of SQLAlchemy objects, both within the Core as well as the ORM.

The *inspect()* function is the entry point to SQLAlchemy's public API for viewing the configuration and construction of in-memory objects. Depending on the type of object passed to *inspect()*, the return value will either be a related object which provides a known interface, or in many cases it will return the object itself.

The rationale for *inspect()* is twofold. One is that it replaces the need to be aware of a large variety of "information getting" functions in SQLAlchemy, such as *Inspector.from_engine()*, *orm.attributes.instance_state()*, *orm.class_mapper()*, and others. The other is that the return value of *inspect()* is guaranteed to obey a documented API, thus allowing third party tools which build on top of SQLAlchemy configurations to be constructed in a forwards-compatible way.

New in version 0.8: The *inspect()* system is introduced as of version 0.8.

`sqlalchemy.inspection.`**`inspect`**(*subject*, *raiseerr=True*)

Produce an inspection object for the given target.

The returned value in some cases may be the same object as the one given, such as if a *Mapper* object is passed. In other cases, it will be an instance of the registered inspection type for the given object, such as if an *engine.Engine* is passed, an *Inspector* object is returned.

Parameters

- **`subject`** – the subject to be inspected.
- **`raiseerr`** – When `True`, if the given subject does not correspond to a known SQLAlchemy inspected type, *sqlalchemy.exc.NoInspectionAvailable* is raised. If `False`, `None` is returned.

Available Inspection Targets

Below is a listing of many of the most common inspection targets.

- *Connectable* (i.e. *Engine*, *Connection*) - returns an *Inspector* object.
- *ClauseElement* - all SQL expression components, including *Table*, *Column*, serve as their own inspection objects, meaning any of these objects passed to *inspect()* return themselves.
- object - an object given will be checked by the ORM for a mapping - if so, an *InstanceState* is returned representing the mapped state of the object. The *InstanceState* also provides access to per attribute state via the *AttributeState* interface as well as the per-flush "history" of any attribute via the *History* object.
- type (i.e. a class) - a class given will be checked by the ORM for a mapping - if so, a *Mapper* for that class is returned.
- mapped attribute - passing a mapped attribute to *inspect()*, such as inspect(MyClass.some_attribute), returns a *QueryableAttribute* object, which is the descriptor associated with a mapped class. This descriptor refers to a *MapperProperty*, which is usually an instance of *ColumnProperty* or *RelationshipProperty*, via its *QueryableAttribute.property* attribute.
- *AliasedClass* - returns an *AliasedInsp* object.

3.6.3 Deprecated Event Interfaces

This section describes the class-based core event interface introduced in SQLAlchemy 0.5. The ORM analogue is described at *Deprecated ORM Event Interfaces*.

Deprecated since version 0.7: The new event system described in *Events* replaces the extension/proxy/listener system, providing a consistent interface to all events without the need for subclassing.

Execution, Connection and Cursor Events

class sqlalchemy.interfaces.**ConnectionProxy**

Allows interception of statement execution by Connections.

Note: *ConnectionProxy* is deprecated. Please refer to *ConnectionEvents*.

Either or both of the execute() and cursor_execute() may be implemented to intercept compiled statement and cursor level executions, e.g.:

```
class MyProxy(ConnectionProxy):
    def execute(self, conn, execute, clauseelement,
                *multiparams, **params):
        print "compiled statement:", clauseelement
        return execute(clauseelement, *multiparams, **params)

    def cursor_execute(self, execute, cursor, statement,
                       parameters, context, executemany):
        print "raw statement:", statement
        return execute(cursor, statement, parameters, context)
```

The execute argument is a function that will fulfill the default execution behavior for the operation. The signature illustrated in the example should be used.

The proxy is installed into an *Engine* via the proxy argument:

```
e = create_engine('someurl://', proxy=MyProxy())
```

begin(*conn*, *begin*)
Intercept begin() events.

begin_twophase(*conn*, *begin_twophase*, *xid*)
Intercept begin_twophase() events.

commit(*conn*, *commit*)
Intercept commit() events.

commit_twophase(*conn*, *commit_twophase*, *xid*, *is_prepared*)
Intercept commit_twophase() events.

cursor_execute(*execute*, *cursor*, *statement*, *parameters*, *context*, *executemany*)
Intercept low-level cursor execute() events.

execute(*conn*, *execute*, *clauseelement*, **multiparams*, ***params*)
Intercept high level execute() events.

prepare_twophase(*conn*, *prepare_twophase*, *xid*)
Intercept prepare_twophase() events.

release_savepoint(*conn*, *release_savepoint*, *name*, *context*)
Intercept release_savepoint() events.

rollback(*conn*, *rollback*)
Intercept rollback() events.

rollback_savepoint(*conn*, *rollback_savepoint*, *name*, *context*)
Intercept rollback_savepoint() events.

rollback_twophase(*conn*, *rollback_twophase*, *xid*, *is_prepared*)
Intercept rollback_twophase() events.

savepoint(*conn*, *savepoint*, *name=None*)
Intercept savepoint() events.

Connection Pool Events

class sqlalchemy.interfaces.**PoolListener**
Hooks into the lifecycle of connections in a *Pool*.

Note: *PoolListener* is deprecated. Please refer to *PoolEvents*.

Usage:

```
class MyListener(PoolListener):
    def connect(self, dbapi_con, con_record):
        '''perform connect operations'''
    # etc.

# create a new pool with a listener
p = QueuePool(..., listeners=[MyListener()])

# add a listener after the fact
p.add_listener(MyListener())
```

```
# usage with create_engine()
e = create_engine("url://", listeners=[MyListener()])
```

All of the standard connection `Pool` types can accept event listeners for key connection lifecycle events: creation, pool check-out and check-in. There are no events fired when a connection closes.

For any given DB-API connection, there will be one `connect` event, *n* number of `checkout` events, and either *n* or *n - 1* `checkin` events. (If a `Connection` is detached from its pool via the `detach()` method, it won't be checked back in.)

These are low-level events for low-level objects: raw Python DB-API connections, without the conveniences of the SQLAlchemy `Connection` wrapper, `Dialect` services or `ClauseElement` execution. If you execute SQL through the connection, explicitly closing all cursors and other resources is recommended.

Events also receive a `_ConnectionRecord`, a long-lived internal `Pool` object that basically represents a "slot" in the connection pool. `_ConnectionRecord` objects have one public attribute of note: `info`, a dictionary whose contents are scoped to the lifetime of the DB-API connection managed by the record. You can use this shared storage area however you like.

There is no need to subclass `PoolListener` to handle events. Any class that implements one or more of these methods can be used as a pool listener. The `Pool` will inspect the methods provided by a listener object and add the listener to one or more internal event queues based on its capabilities. In terms of efficiency and function call overhead, you're much better off only providing implementations for the hooks you'll be using.

checkin(*dbapi_con*, *con_record*)

Called when a connection returns to the pool.

Note that the connection may be closed, and may be None if the connection has been invalidated. `checkin` will not be called for detached connections. (They do not return to the pool.)

dbapi_con A raw DB-API connection

con_record The `_ConnectionRecord` that persistently manages the connection

checkout(*dbapi_con*, *con_record*, *con_proxy*)

Called when a connection is retrieved from the Pool.

dbapi_con A raw DB-API connection

con_record The `_ConnectionRecord` that persistently manages the connection

con_proxy The `_ConnectionFairy` which manages the connection for the span of the current checkout.

If you raise an `exc.DisconnectionError`, the current connection will be disposed and a fresh connection retrieved. Processing of all checkout listeners will abort and restart using the new connection.

connect(*dbapi_con*, *con_record*)

Called once for each new DB-API connection or Pool's `creator()`.

dbapi_con A newly connected raw DB-API connection (not a SQLAlchemy `Connection` wrapper).

con_record The `_ConnectionRecord` that persistently manages the connection

first_connect(*dbapi_con*, *con_record*)

Called exactly once for the first DB-API connection.

dbapi_con A newly connected raw DB-API connection (not a SQLAlchemy `Connection` wrapper).

con_record The `_ConnectionRecord` that persistently manages the connection

3.6.4 Core Exceptions

Exceptions used with SQLAlchemy.

The base exception class is *SQLAlchemyError*. Exceptions which are raised as a result of DBAPI exceptions are all subclasses of *DBAPIError*.

exception sqlalchemy.exc.**AmbiguousForeignKeysError**
Raised when more than one foreign key matching can be located between two selectables during a join.

exception sqlalchemy.exc.**ArgumentError**
Raised when an invalid or conflicting function argument is supplied.

This error generally corresponds to construction time state errors.

exception sqlalchemy.exc.**CircularDependencyError**(*message*, *cycles*, *edges*, *msg=None*)
Raised by topological sorts when a circular dependency is detected.

There are two scenarios where this error occurs:

- In a Session flush operation, if two objects are mutually dependent on each other, they can not be inserted or deleted via INSERT or DELETE statements alone; an UPDATE will be needed to post-associate or pre-deassociate one of the foreign key constrained values. The `post_update` flag described at *Rows that point to themselves / Mutually Dependent Rows* can resolve this cycle.
- In a *MetaData.sorted_tables* operation, two *ForeignKey* or *ForeignKeyConstraint* objects mutually refer to each other. Apply the `use_alter=True` flag to one or both, see *Creating/Dropping Foreign Key Constraints via ALTER*.

exception sqlalchemy.exc.**CompileError**
Raised when an error occurs during SQL compilation

exception sqlalchemy.exc.**DBAPIError**(*statement*, *params*, *orig*, *connection_invalidated=False*)
Raised when the execution of a database operation fails.

Wraps exceptions raised by the DB-API underlying the database operation. Driver-specific implementations of the standard DB-API exception types are wrapped by matching sub-types of SQLAlchemy's *DBAPIError* when possible. DB-API's `Error` type maps to *DBAPIError* in SQLAlchemy, otherwise the names are identical. Note that there is no guarantee that different DB-API implementations will raise the same exception type for any given error condition.

DBAPIError features *statement* and *params* attributes which supply context regarding the specifics of the statement which had an issue, for the typical case when the error was raised within the context of emitting a SQL statement.

The wrapped exception object is available in the *orig* attribute. Its type and properties are DB-API implementation specific.

exception sqlalchemy.exc.**DataError**(*statement*, *params*, *orig*, *connection_invalidated=False*)
Wraps a DB-API DataError.

exception sqlalchemy.exc.**DatabaseError**(*statement*, *params*, *orig*, *connection_invalidated=False*)
Wraps a DB-API DatabaseError.

exception sqlalchemy.exc.**DisconnectionError**
A disconnect is detected on a raw DB-API connection.

This error is raised and consumed internally by a connection pool. It can be raised by the *PoolEvents.checkout()* event so that the host pool forces a retry; the exception will be caught three times in a row before the pool gives up and raises *InvalidRequestError* regarding the connection attempt.

class `sqlalchemy.exc.`**`DontWrapMixin`**

A mixin class which, when applied to a user-defined Exception class, will not be wrapped inside of `StatementError` if the error is emitted within the process of executing a statement.

E.g.:

```
from sqlalchemy.exc import DontWrapMixin

class MyCustomException(Exception, DontWrapMixin):
    pass

class MySpecialType(TypeDecorator):
    impl = String

    def process_bind_param(self, value, dialect):
        if value == 'invalid':
            raise MyCustomException("invalid!")
```

exception `sqlalchemy.exc.`**`IdentifierError`**

Raised when a schema name is beyond the max character limit

exception `sqlalchemy.exc.`**`IntegrityError`**(*statement*, *params*, *orig*, *connection_invalidated=False*)

Wraps a DB-API IntegrityError.

exception `sqlalchemy.exc.`**`InterfaceError`**(*statement*, *params*, *orig*, *connection_invalidated=False*)

Wraps a DB-API InterfaceError.

exception `sqlalchemy.exc.`**`InternalError`**(*statement*, *params*, *orig*, *connection_invalidated=False*)

Wraps a DB-API InternalError.

exception `sqlalchemy.exc.`**`InvalidRequestError`**

SQLAlchemy was asked to do something it can't do.

This error generally corresponds to runtime state errors.

exception `sqlalchemy.exc.`**`NoForeignKeysError`**

Raised when no foreign keys can be located between two selectables during a join.

exception `sqlalchemy.exc.`**`NoInspectionAvailable`**

A subject passed to `sqlalchemy.inspection.inspect()` produced no context for inspection.

exception `sqlalchemy.exc.`**`NoReferenceError`**

Raised by `ForeignKey` to indicate a reference cannot be resolved.

exception `sqlalchemy.exc.`**`NoReferencedColumnError`**(*message*, *tname*, *cname*)

Raised by `ForeignKey` when the referred `Column` cannot be located.

exception `sqlalchemy.exc.`**`NoReferencedTableError`**(*message*, *tname*)

Raised by `ForeignKey` when the referred `Table` cannot be located.

exception `sqlalchemy.exc.`**`NoSuchColumnError`**

A nonexistent column is requested from a `RowProxy`.

exception `sqlalchemy.exc.`**`NoSuchModuleError`**

Raised when a dynamically-loaded module (usually a database dialect) of a particular name cannot be located.

exception `sqlalchemy.exc.`**`NoSuchTableError`**

Table does not exist or is not visible to a connection.

exception sqlalchemy.exc.**NotSupportedError**(*statement*, *params*, *orig*, *connection_invalidated=False*)

Wraps a DB-API NotSupportedError.

exception sqlalchemy.exc.**OperationalError**(*statement*, *params*, *orig*, *connection_invalidated=False*)

Wraps a DB-API OperationalError.

exception sqlalchemy.exc.**ProgrammingError**(*statement*, *params*, *orig*, *connection_invalidated=False*)

Wraps a DB-API ProgrammingError.

exception sqlalchemy.exc.**ResourceClosedError**

An operation was requested from a connection, cursor, or other object that's in a closed state.

exception sqlalchemy.exc.**SADeprecationWarning**

Issued once per usage of a deprecated API.

exception sqlalchemy.exc.**SAPendingDeprecationWarning**

Issued once per usage of a deprecated API.

exception sqlalchemy.exc.**SAWarning**

Issued at runtime.

exception sqlalchemy.exc.**SQLAlchemyError**

Generic error class.

exception sqlalchemy.exc.**StatementError**(*message*, *statement*, *params*, *orig*)

An error occurred during execution of a SQL statement.

`StatementError` wraps the exception raised during execution, and features `statement` and `params` attributes which supply context regarding the specifics of the statement which had an issue.

The wrapped exception object is available in the `orig` attribute.

orig = None

The DBAPI exception object.

params = None

The parameter list being used when this exception occurred.

statement = None

The string SQL statement being invoked when this exception occurred.

exception sqlalchemy.exc.**TimeoutError**

Raised when a connection pool times out on getting a connection.

exception sqlalchemy.exc.**UnboundExecutionError**

SQL was attempted without a database connection to execute it on.

exception sqlalchemy.exc.**UnsupportedCompilationError**(*compiler*, *element_type*)

Raised when an operation is not supported by the given compiler.

New in version 0.8.3.

3.6.5 Core Internals

Some key internal constructs are listed here.

class sqlalchemy.engine.interfaces.**Compiled**(*dialect*, *statement*, *bind=None*, *schema_translate_map=None*, *compile_kwargs=immutabledict({})*)

Represent a compiled SQL or DDL expression.

The `__str__` method of the `Compiled` object should produce the actual text of the statement. `Compiled` objects are specific to their underlying database dialect, and also may or may not be specific to the columns referenced within a particular set of bind parameters. In no case should the `Compiled` object be dependent on the actual values of those bind parameters, even though it may reference those values as defaults.

__init__(*dialect*, *statement*, *bind=None*, *schema_translate_map=None*, *compile_kwargs=immutabledict({})*)

Construct a new `Compiled` object.

Parameters

- **dialect** – `Dialect` to compile against.
- **statement** – `ClauseElement` to be compiled.
- **bind** – Optional Engine or Connection to compile this statement against.
- **schema_translate_map** – dictionary of schema names to be translated when forming the resultant SQL

  New in version 1.1.

  See also:

  Translation of Schema Names

- **compile_kwargs** – additional kwargs that will be passed to the initial call to `Compiled.process()`.

compile()

Produce the internal string representation of this element.

Deprecated since version 0.7: `Compiled` objects now compile within the constructor.

construct_params(*params=None*)

Return the bind params for this compiled object.

Parameters **params** – a dict of string/object pairs whose values will override bind values compiled in to the statement.

execute(**multiparams*, ***params*)

Execute this compiled object.

params

Return the bind params for this compiled object.

scalar(**multiparams*, ***params*)

Execute this compiled object and return the result's scalar value.

sql_compiler

Return a Compiled that is capable of processing SQL expressions.

If this compiler is one, it would likely just return 'self'.

class `sqlalchemy.sql.compiler.`**DDLCompiler**(*dialect*, *statement*, *bind=None*, *schema_translate_map=None*, *compile_kwargs=immutabledict({})*)

Bases: `sqlalchemy.sql.compiler.Compiled`

__init__(*dialect*, *statement*, *bind=None*, *schema_translate_map=None*, *compile_kwargs=immutabledict({})*)

inherited from the `__init__()` *method of* `Compiled`

Construct a new `Compiled` object.

Parameters

- **dialect** – *Dialect* to compile against.
- **statement** – *ClauseElement* to be compiled.
- **bind** – Optional Engine or Connection to compile this statement against.
- **schema_translate_map** – dictionary of schema names to be translated when forming the resultant SQL

  New in version 1.1.

  See also:

  Translation of Schema Names

- **compile_kwargs** – additional kwargs that will be passed to the initial call to `Compiled.process()`.

compile()

inherited from the `compile()` *method of* `Compiled`

Produce the internal string representation of this element.

Deprecated since version 0.7: *Compiled* objects now compile within the constructor.

define_constraint_remote_table(*constraint*, *table*, *preparer*)

Format the remote table clause of a CREATE CONSTRAINT clause.

execute(**multiparams*, ***params*)

inherited from the `execute()` *method of* `Compiled`

Execute this compiled object.

params

inherited from the `params` *attribute of* `Compiled`

Return the bind params for this compiled object.

scalar(**multiparams*, ***params*)

inherited from the `scalar()` *method of* `Compiled`

Execute this compiled object and return the result's scalar value.

class `sqlalchemy.engine.default.`**DefaultDialect**(*convert_unicode=False*, *encoding='utf-8'*, *paramstyle=None*, *dbapi=None*, *implicit_returning=None*, *supports_right_nested_joins=None*, *case_sensitive=True*, *supports_native_boolean=None*, *label_length=None*, ***kwargs*)

Bases: *sqlalchemy.engine.interfaces.Dialect*

Default implementation of Dialect

construct_arguments = None

Optional set of argument specifiers for various SQLAlchemy constructs, typically schema items.

To implement, establish as a series of tuples, as in:

```
construct_arguments = [
    (schema.Index, {
        "using": False,
        "where": None,
        "ops": None
```

```
        })
    ]
```

If the above construct is established on the Postgresql dialect, the `Index` construct will now accept the keyword arguments `postgresql_using`, `postgresql_where`, nad `postgresql_ops`. Any other argument specified to the constructor of `Index` which is prefixed with `postgresql_` will raise `ArgumentError`.

A dialect which does not include a `construct_arguments` member will not participate in the argument validation system. For such a dialect, any argument name is accepted by all participating constructs, within the namespace of arguments prefixed with that dialect name. The rationale here is so that third-party dialects that haven't yet implemented this feature continue to function in the old way.

New in version 0.9.2.

See also:

`DialectKWArgs` - implementing base class which consumes `DefaultDialect.construct_arguments`

create_xid()

Create a random two-phase transaction ID.

This id will be passed to do_begin_twophase(), do_rollback_twophase(), do_commit_twophase(). Its format is unspecified.

dbapi_exception_translation_map = immutabledict({})

mapping used in the extremely unusual case that a DBAPI's published exceptions don't actually have the __name__ that they are linked towards.

New in version 1.0.5.

denormalize_name(*name*)

inherited from the `denormalize_name()` *method of* `Dialect`

convert the given name to a case insensitive identifier for the backend if it is an all-lowercase name.

this method is only used if the dialect defines requires_name_normalize=True.

do_begin_twophase(*connection*, *xid*)

inherited from the `do_begin_twophase()` *method of* `Dialect`

Begin a two phase transaction on the given connection.

Parameters

- **connection** – a `Connection`.
- **xid** – xid

do_commit_twophase(*connection*, *xid*, *is_prepared=True*, *recover=False*)

inherited from the `do_commit_twophase()` *method of* `Dialect`

Commit a two phase transaction on the given connection.

Parameters

- **connection** – a `Connection`.
- **xid** – xid
- **is_prepared** – whether or not `TwoPhaseTransaction.prepare()` was called.
- **recover** – if the recover flag was passed.

do_prepare_twophase(*connection*, *xid*)
inherited from the `do_prepare_twophase()` *method of* `Dialect`

Prepare a two phase transaction on the given connection.

Parameters

- **connection** – a `Connection`.
- **xid** – xid

do_recover_twophase(*connection*)
inherited from the `do_recover_twophase()` *method of* `Dialect`

Recover list of uncommited prepared two phase transaction identifiers on the given connection.

Parameters **connection** – a `Connection`.

do_rollback_twophase(*connection*, *xid*, *is_prepared=True*, *recover=False*)
inherited from the `do_rollback_twophase()` *method of* `Dialect`

Rollback a two phase transaction on the given connection.

Parameters

- **connection** – a `Connection`.
- **xid** – xid
- **is_prepared** – whether or not `TwoPhaseTransaction.prepare()` was called.
- **recover** – if the recover flag was passed.

engine_created(*engine*)
inherited from the `engine_created()` *method of* `Dialect`

A convenience hook called before returning the final `Engine`.

If the dialect returned a different class from the `get_dialect_cls()` method, then the hook is called on both classes, first on the dialect class returned by the `get_dialect_cls()` method and then on the class on which the method was called.

The hook should be used by dialects and/or wrappers to apply special events to the engine or its components. In particular, it allows a dialect-wrapping class to apply dialect-level events.

New in version 1.0.3.

execute_sequence_format
alias of `tuple`

get_columns(*connection*, *table_name*, *schema=None*, ***kw*)
inherited from the `get_columns()` *method of* `Dialect`

Return information about columns in *table_name*.

Given a `Connection`, a string *table_name*, and an optional string *schema*, return column information as a list of dictionaries with these keys:

name the column's name

type [sqlalchemy.types#TypeEngine]

nullable boolean

default the column's default value

autoincrement boolean

sequence

a dictionary of the form

{'name' [str, 'start' :int, 'increment': int, 'minvalue': int,] 'maxvalue': int, 'nominvalue': bool, 'nomaxvalue': bool, 'cycle': bool}

Additional column attributes may be present.

get_dialect_cls(*url*)
inherited from the `get_dialect_cls()` *method of* `Dialect`

Given a URL, return the `Dialect` that will be used.

This is a hook that allows an external plugin to provide functionality around an existing dialect, by allowing the plugin to be loaded from the url based on an entrypoint, and then the plugin returns the actual dialect to be used.

By default this just returns the cls.

New in version 1.0.3.

get_foreign_keys(*connection*, *table_name*, *schema=None*, ***kw*)
inherited from the `get_foreign_keys()` *method of* `Dialect`

Return information about foreign_keys in *table_name*.

Given a `Connection`, a string *table_name*, and an optional string *schema*, return foreign key information as a list of dicts with these keys:

name the constraint's name

constrained_columns a list of column names that make up the foreign key

referred_schema the name of the referred schema

referred_table the name of the referred table

referred_columns a list of column names in the referred table that correspond to constrained_columns

get_indexes(*connection*, *table_name*, *schema=None*, ***kw*)
inherited from the `get_indexes()` *method of* `Dialect`

Return information about indexes in *table_name*.

Given a `Connection`, a string *table_name* and an optional string *schema*, return index information as a list of dictionaries with these keys:

name the index's name

column_names list of column names in order

unique boolean

get_isolation_level(*dbapi_conn*)
inherited from the `get_isolation_level()` *method of* `Dialect`

Given a DBAPI connection, return its isolation level.

When working with a `Connection` object, the corresponding DBAPI connection may be procured using the `Connection.connection` accessor.

Note that this is a dialect-level method which is used as part of the implementation of the `Connection` and `Engine` isolation level facilities; these APIs should be preferred for most typical use cases.

See also:

`Connection.get_isolation_level()` - view current level

`Connection.default_isolation_level` - view default level

`Connection.execution_options.isolation_level` - set per `Connection` isolation level

`create_engine.isolation_level` - set per `Engine` isolation level

get_pk_constraint(*conn*, *table_name*, *schema=None*, ***kw*)
Compatibility method, adapts the result of get_primary_keys() for those dialects which don't implement get_pk_constraint().

get_primary_keys(*connection*, *table_name*, *schema=None*, ***kw*)
inherited from the `get_primary_keys()` *method of* `Dialect`

Return information about primary keys in *table_name*.

Deprecated. This method is only called by the default implementation of `Dialect.get_pk_constraint()`. Dialects should instead implement the `Dialect.get_pk_constraint()` method directly.

get_table_names(*connection*, *schema=None*, ***kw*)
inherited from the `get_table_names()` *method of* `Dialect`

Return a list of table names for *schema*.

get_temp_table_names(*connection*, *schema=None*, ***kw*)
inherited from the `get_temp_table_names()` *method of* `Dialect`

Return a list of temporary table names on the given connection, if supported by the underlying backend.

get_temp_view_names(*connection*, *schema=None*, ***kw*)
inherited from the `get_temp_view_names()` *method of* `Dialect`

Return a list of temporary view names on the given connection, if supported by the underlying backend.

get_unique_constraints(*connection*, *table_name*, *schema=None*, ***kw*)
inherited from the `get_unique_constraints()` *method of* `Dialect`

Return information about unique constraints in *table_name*.

Given a string *table_name* and an optional string *schema*, return unique constraint information as a list of dicts with these keys:

name the unique constraint's name

column_names list of column names in order

****kw** other options passed to the dialect's get_unique_constraints() method.

New in version 0.9.0.

get_view_definition(*connection*, *view_name*, *schema=None*, ***kw*)
inherited from the `get_view_definition()` *method of* `Dialect`

Return view definition.

Given a `Connection`, a string *view_name*, and an optional string *schema*, return the view definition.

get_view_names(*connection*, *schema=None*, ***kw*)
inherited from the `get_view_names()` *method of* `Dialect`

Return a list of all view names available in the database.

schema: Optional, retrieve names from a non-default schema.

has_sequence(*connection*, *sequence_name*, *schema=None*)
inherited from the `has_sequence()` *method of* `Dialect`

Check the existence of a particular sequence in the database.

Given a `Connection` object and a string *sequence_name*, return True if the given sequence exists in the database, False otherwise.

has_table(*connection*, *table_name*, *schema=None*)
inherited from the `has_table()` *method of* `Dialect`

Check the existence of a particular table in the database.

Given a `Connection` object and a string *table_name*, return True if the given table (possibly within the specified *schema*) exists in the database, False otherwise.

normalize_name(*name*)
inherited from the `normalize_name()` *method of* `Dialect`

convert the given name to lowercase if it is detected as case insensitive.

this method is only used if the dialect defines requires_name_normalize=True.

on_connect()
return a callable which sets up a newly created DBAPI connection.

This is used to set dialect-wide per-connection options such as isolation modes, unicode modes, etc.

If a callable is returned, it will be assembled into a pool listener that receives the direct DBAPI connection, with all wrappers removed.

If None is returned, no listener will be generated.

preparer
alias of `IdentifierPreparer`

set_isolation_level(*dbapi_conn*, *level*)
inherited from the `set_isolation_level()` *method of* `Dialect`

Given a DBAPI connection, set its isolation level.

Note that this is a dialect-level method which is used as part of the implementation of the `Connection` and `Engine` isolation level facilities; these APIs should be preferred for most typical use cases.

See also:

`Connection.get_isolation_level()` - view current level

`Connection.default_isolation_level` - view default level

`Connection.execution_options.isolation_level` - set per `Connection` isolation level

`create_engine.isolation_level` - set per `Engine` isolation level

statement_compiler
alias of `SQLCompiler`

type_descriptor(*typeobj*)
Provide a database-specific `TypeEngine` object, given the generic object which comes from the types module.

This method looks for a dictionary called `colspecs` as a class or instance-level variable, and passes on to `types.adapt_type()`.

class `sqlalchemy.engine.interfaces.`**Dialect**
Define the behavior of a specific database and DB-API combination.

Any aspect of metadata definition, SQL query generation, execution, result-set handling, or anything else which varies between databases is defined under the general category of the Dialect. The Dialect acts as a factory for

other database-specific object implementations including ExecutionContext, Compiled, DefaultGenerator, and TypeEngine.

All Dialects implement the following attributes:

name identifying name for the dialect from a DBAPI-neutral point of view (i.e. 'sqlite')

driver identifying name for the dialect's DBAPI

positional True if the paramstyle for this Dialect is positional.

paramstyle the paramstyle to be used (some DB-APIs support multiple paramstyles).

convert_unicode True if Unicode conversion should be applied to all `str` types.

encoding type of encoding to use for unicode, usually defaults to 'utf-8'.

statement_compiler a *`Compiled`* class used to compile SQL statements

ddl_compiler a *`Compiled`* class used to compile DDL statements

server_version_info a tuple containing a version number for the DB backend in use. This value is only available for supporting dialects, and is typically populated during the initial connection to the database.

default_schema_name the name of the default schema. This value is only available for supporting dialects, and is typically populated during the initial connection to the database.

execution_ctx_cls a *`ExecutionContext`* class used to handle statement execution

execute_sequence_format either the 'tuple' or 'list' type, depending on what cursor.execute() accepts for the second argument (they vary).

preparer a *`IdentifierPreparer`* class used to quote identifiers.

supports_alter `True` if the database supports `ALTER TABLE`.

max_identifier_length The maximum length of identifier names.

supports_unicode_statements Indicate whether the DB-API can receive SQL statements as Python unicode strings

supports_unicode_binds Indicate whether the DB-API can receive string bind parameters as Python unicode strings

supports_sane_rowcount Indicate whether the dialect properly implements rowcount for `UPDATE` and `DELETE` statements.

supports_sane_multi_rowcount Indicate whether the dialect properly implements rowcount for `UPDATE` and `DELETE` statements when executed via executemany.

preexecute_autoincrement_sequences True if 'implicit' primary key functions must be executed separately in order to get their value. This is currently oriented towards Postgresql.

implicit_returning use RETURNING or equivalent during INSERT execution in order to load newly generated primary keys and other column defaults in one execution, which are then available via inserted_primary_key. If an insert statement has returning() specified explicitly, the "implicit" functionality is not used and inserted_primary_key will not be available.

dbapi_type_map A mapping of DB-API type objects present in this Dialect's DB-API implementation mapped to TypeEngine implementations used by the dialect.

This is used to apply types to result sets based on the DB-API types present in cursor.description; it only takes effect for result sets against textual statements where no explicit typemap was present.

colspecs A dictionary of TypeEngine classes from sqlalchemy.types mapped to subclasses that are specific to the dialect class. This dictionary is class-level only and is not accessed from the dialect instance itself.

supports_default_values Indicates if the construct `INSERT INTO tablename DEFAULT VALUES` is supported

supports_sequences Indicates if the dialect supports CREATE SEQUENCE or similar.

sequences_optional If True, indicates if the "optional" flag on the Sequence() construct should signal to not generate a CREATE SEQUENCE. Applies only to dialects that support sequences. Currently used only to allow Postgresql SERIAL to be used on a column that specifies Sequence() for usage on other backends.

supports_native_enum Indicates if the dialect supports a native ENUM construct. This will prevent types.Enum from generating a CHECK constraint when that type is used.

supports_native_boolean Indicates if the dialect supports a native boolean construct. This will prevent types.Boolean from generating a CHECK constraint when that type is used.

dbapi_exception_translation_map A dictionary of names that will contain as values the names of pep-249 exceptions ("IntegrityError", "OperationalError", etc) keyed to alternate class names, to support the case

where a DBAPI has exception classes that aren't named as they are referred to (e.g. IntegrityError = MyException). In the vast majority of cases this dictionary is empty.

New in version 1.0.5.

connect()

return a callable which sets up a newly created DBAPI connection.

The callable accepts a single argument "conn" which is the DBAPI connection itself. It has no return value.

This is used to set dialect-wide per-connection options such as isolation modes, unicode modes, etc.

If a callable is returned, it will be assembled into a pool listener that receives the direct DBAPI connection, with all wrappers removed.

If None is returned, no listener will be generated.

create_connect_args(*url*)

Build DB-API compatible connection arguments.

Given a `URL` object, returns a tuple consisting of a **args/**kwargs* suitable to send directly to the dbapi's connect function.

create_xid()

Create a two-phase transaction ID.

This id will be passed to do_begin_twophase(), do_rollback_twophase(), do_commit_twophase(). Its format is unspecified.

denormalize_name(*name*)

convert the given name to a case insensitive identifier for the backend if it is an all-lowercase name.

this method is only used if the dialect defines requires_name_normalize=True.

do_begin(*dbapi_connection*)

Provide an implementation of `connection.begin()`, given a DB-API connection.

The DBAPI has no dedicated "begin" method and it is expected that transactions are implicit. This hook is provided for those DBAPIs that might need additional help in this area.

Note that `Dialect.do_begin()` is not called unless a `Transaction` object is in use. The `Dialect.do_autocommit()` hook is provided for DBAPIs that need some extra commands emitted after a commit in order to enter the next transaction, when the SQLAlchemy `Connection` is used in its default "autocommit" mode.

Parameters **dbapi_connection** – a DBAPI connection, typically proxied within a `ConnectionFairy`.

do_begin_twophase(*connection*, *xid*)

Begin a two phase transaction on the given connection.

Parameters

- **connection** – a `Connection`.
- **xid** – xid

do_close(*dbapi_connection*)

Provide an implementation of `connection.close()`, given a DBAPI connection.

This hook is called by the `Pool` when a connection has been detached from the pool, or is being returned beyond the normal capacity of the pool.

New in version 0.8.

do_commit(*dbapi_connection*)
Provide an implementation of `connection.commit()`, given a DB-API connection.

Parameters **dbapi_connection** – a DBAPI connection, typically proxied within a `ConnectionFairy`.

do_commit_twophase(*connection*, *xid*, *is_prepared=True*, *recover=False*)
Commit a two phase transaction on the given connection.

Parameters

- **connection** – a *Connection*.
- **xid** – xid
- **is_prepared** – whether or not *TwoPhaseTransaction.prepare()* was called.
- **recover** – if the recover flag was passed.

do_execute(*cursor*, *statement*, *parameters*, *context=None*)
Provide an implementation of `cursor.execute(statement, parameters)`.

do_execute_no_params(*cursor*, *statement*, *parameters*, *context=None*)
Provide an implementation of `cursor.execute(statement)`.

The parameter collection should not be sent.

do_executemany(*cursor*, *statement*, *parameters*, *context=None*)
Provide an implementation of `cursor.executemany(statement, parameters)`.

do_prepare_twophase(*connection*, *xid*)
Prepare a two phase transaction on the given connection.

Parameters

- **connection** – a *Connection*.
- **xid** – xid

do_recover_twophase(*connection*)
Recover list of uncommited prepared two phase transaction identifiers on the given connection.

Parameters **connection** – a *Connection*.

do_release_savepoint(*connection*, *name*)
Release the named savepoint on a connection.

Parameters

- **connection** – a *Connection*.
- **name** – savepoint name.

do_rollback(*dbapi_connection*)
Provide an implementation of `connection.rollback()`, given a DB-API connection.

Parameters **dbapi_connection** – a DBAPI connection, typically proxied within a `ConnectionFairy`.

do_rollback_to_savepoint(*connection*, *name*)
Rollback a connection to the named savepoint.

Parameters

- **connection** – a *Connection*.

- **name** – savepoint name.

do_rollback_twophase(*connection*, *xid*, *is_prepared=True*, *recover=False*)
Rollback a two phase transaction on the given connection.

Parameters

- **connection** – a `Connection`.
- **xid** – xid
- **is_prepared** – whether or not `TwoPhaseTransaction.prepare()` was called.
- **recover** – if the recover flag was passed.

do_savepoint(*connection*, *name*)
Create a savepoint with the given name.

Parameters

- **connection** – a `Connection`.
- **name** – savepoint name.

classmethod engine_created(*engine*)
A convenience hook called before returning the final `Engine`.

If the dialect returned a different class from the `get_dialect_cls()` method, then the hook is called on both classes, first on the dialect class returned by the `get_dialect_cls()` method and then on the class on which the method was called.

The hook should be used by dialects and/or wrappers to apply special events to the engine or its components. In particular, it allows a dialect-wrapping class to apply dialect-level events.

New in version 1.0.3.

get_columns(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about columns in *table_name*.

Given a `Connection`, a string *table_name*, and an optional string *schema*, return column information as a list of dictionaries with these keys:

name the column's name

type [sqlalchemy.types#TypeEngine]

nullable boolean

default the column's default value

autoincrement boolean

sequence

a dictionary of the form

{'name' [str, 'start' :int, 'increment': int, 'minvalue': int,] 'maxvalue': int, 'nominvalue': bool, 'nomaxvalue': bool, 'cycle': bool}

Additional column attributes may be present.

classmethod get_dialect_cls(*url*)
Given a URL, return the `Dialect` that will be used.

This is a hook that allows an external plugin to provide functionality around an existing dialect, by allowing the plugin to be loaded from the url based on an entrypoint, and then the plugin returns the actual dialect to be used.

By default this just returns the cls.

New in version 1.0.3.

get_foreign_keys(*connection*, *table_name*, *schema=None*, ***kw*)

Return information about foreign_keys in *table_name*.

Given a `Connection`, a string *table_name*, and an optional string *schema*, return foreign key information as a list of dicts with these keys:

name the constraint's name

constrained_columns a list of column names that make up the foreign key

referred_schema the name of the referred schema

referred_table the name of the referred table

referred_columns a list of column names in the referred table that correspond to constrained_columns

get_indexes(*connection*, *table_name*, *schema=None*, ***kw*)

Return information about indexes in *table_name*.

Given a `Connection`, a string *table_name* and an optional string *schema*, return index information as a list of dictionaries with these keys:

name the index's name

column_names list of column names in order

unique boolean

get_isolation_level(*dbapi_conn*)

Given a DBAPI connection, return its isolation level.

When working with a `Connection` object, the corresponding DBAPI connection may be procured using the `Connection.connection` accessor.

Note that this is a dialect-level method which is used as part of the implementation of the `Connection` and `Engine` isolation level facilities; these APIs should be preferred for most typical use cases.

See also:

`Connection.get_isolation_level()` - view current level

`Connection.default_isolation_level` - view default level

`Connection.execution_options.isolation_level` - set per `Connection` isolation level

`create_engine.isolation_level` - set per `Engine` isolation level

get_pk_constraint(*connection*, *table_name*, *schema=None*, ***kw*)

Return information about the primary key constraint on table_name‘.

Given a `Connection`, a string *table_name*, and an optional string *schema*, return primary key information as a dictionary with these keys:

constrained_columns a list of column names that make up the primary key

name optional name of the primary key constraint.

get_primary_keys(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about primary keys in *table_name*.

Deprecated. This method is only called by the default implementation of `Dialect.get_pk_constraint()`. Dialects should instead implement the `Dialect.get_pk_constraint()` method directly.

get_table_names(*connection*, *schema=None*, ***kw*)
Return a list of table names for *schema*.

get_temp_table_names(*connection*, *schema=None*, ***kw*)
Return a list of temporary table names on the given connection, if supported by the underlying backend.

get_temp_view_names(*connection*, *schema=None*, ***kw*)
Return a list of temporary view names on the given connection, if supported by the underlying backend.

get_unique_constraints(*connection*, *table_name*, *schema=None*, ***kw*)
Return information about unique constraints in *table_name*.

Given a string *table_name* and an optional string *schema*, return unique constraint information as a list of dicts with these keys:

name the unique constraint's name

column_names list of column names in order

****kw** other options passed to the dialect's get_unique_constraints() method.

New in version 0.9.0.

get_view_definition(*connection*, *view_name*, *schema=None*, ***kw*)
Return view definition.

Given a `Connection`, a string *view_name*, and an optional string *schema*, return the view definition.

get_view_names(*connection*, *schema=None*, ***kw*)
Return a list of all view names available in the database.

schema: Optional, retrieve names from a non-default schema.

has_sequence(*connection*, *sequence_name*, *schema=None*)
Check the existence of a particular sequence in the database.

Given a `Connection` object and a string *sequence_name*, return True if the given sequence exists in the database, False otherwise.

has_table(*connection*, *table_name*, *schema=None*)
Check the existence of a particular table in the database.

Given a `Connection` object and a string *table_name*, return True if the given table (possibly within the specified *schema*) exists in the database, False otherwise.

initialize(*connection*)
Called during strategized creation of the dialect with a connection.

Allows dialects to configure options based on server version info or other properties.

The connection passed here is a SQLAlchemy Connection object, with full capabilities.

The initialize() method of the base dialect should be called via super().

is_disconnect(*e*, *connection*, *cursor*)
Return True if the given DB-API error indicates an invalid connection

normalize_name(*name*)
convert the given name to lowercase if it is detected as case insensitive.

this method is only used if the dialect defines requires_name_normalize=True.

reflecttable(*connection*, *table*, *include_columns*, *exclude_columns*)
Load table description from the database.

Given a `Connection` and a `Table` object, reflect its columns and properties from the database.

The implementation of this method is provided by `DefaultDialect.reflecttable()`, which makes use of `Inspector` to retrieve column information.

Dialects should **not** seek to implement this method, and should instead implement individual schema inspection operations such as `Dialect.get_columns()`, `Dialect.get_pk_constraint()`, etc.

reset_isolation_level(*dbapi_conn*)
Given a DBAPI connection, revert its isolation to the default.

Note that this is a dialect-level method which is used as part of the implementation of the `Connection` and `Engine` isolation level facilities; these APIs should be preferred for most typical use cases.

See also:

`Connection.get_isolation_level()` - view current level

`Connection.default_isolation_level` - view default level

`Connection.execution_options.isolation_level` - set per `Connection` isolation level

`create_engine.isolation_level` - set per `Engine` isolation level

set_isolation_level(*dbapi_conn*, *level*)
Given a DBAPI connection, set its isolation level.

Note that this is a dialect-level method which is used as part of the implementation of the `Connection` and `Engine` isolation level facilities; these APIs should be preferred for most typical use cases.

See also:

`Connection.get_isolation_level()` - view current level

`Connection.default_isolation_level` - view default level

`Connection.execution_options.isolation_level` - set per `Connection` isolation level

`create_engine.isolation_level` - set per `Engine` isolation level

classmethod type_descriptor(*typeobj*)
Transform a generic type to a dialect-specific type.

Dialect classes will usually use the `types.adapt_type()` function in the types module to accomplish this.

The returned result is cached *per dialect class* so can contain no dialect-instance state.

class sqlalchemy.engine.default.**DefaultExecutionContext**
Bases: `sqlalchemy.engine.interfaces.ExecutionContext`

get_lastrowid()
return self.cursor.lastrowid, or equivalent, after an INSERT.

This may involve calling special cursor functions, issuing a new SELECT on the cursor (or a new one), or returning a stored value that was calculated within post_exec().

This function will only be called for dialects which support "implicit" primary key generation, keep pre-execute_autoincrement_sequences set to False, and when no explicit id value was bound to the statement.

The function is called once, directly after post_exec() and before the transaction is committed or ResultProxy is generated. If the post_exec() method assigns a value to *self._lastrowid*, the value is used in place of calling get_lastrowid().

Note that this method is *not* equivalent to the `lastrowid` method on `ResultProxy`, which is a direct proxy to the DBAPI `lastrowid` accessor in all cases.

get_result_processor(*type_*, *colname*, *coltype*)

Return a 'result processor' for a given type as present in cursor.description.

This has a default implementation that dialects can override for context-sensitive result type handling.

set_input_sizes(*translate=None*, *exclude_types=None*)

Given a cursor and ClauseParameters, call the appropriate style of `setinputsizes()` on the cursor, using DB-API types from the bind parameter's `TypeEngine` objects.

This method only called by those dialects which require it, currently cx_oracle.

class sqlalchemy.engine.interfaces.**ExecutionContext**

A messenger object for a Dialect that corresponds to a single execution.

ExecutionContext should have these data members:

connection Connection object which can be freely used by default value generators to execute SQL. This Connection should reference the same underlying connection/transactional resources of root_connection.

root_connection Connection object which is the source of this ExecutionContext. This Connection may have close_with_result=True set, in which case it can only be used once.

dialect dialect which created this ExecutionContext.

cursor DB-API cursor procured from the connection,

compiled if passed to constructor, sqlalchemy.engine.base.Compiled object being executed,

statement string version of the statement to be executed. Is either passed to the constructor, or must be created from the sql.Compiled object by the time pre_exec() has completed.

parameters bind parameters passed to the execute() method. For compiled statements, this is a dictionary or list of dictionaries. For textual statements, it should be in a format suitable for the dialect's paramstyle (i.e. dict or list of dicts for non positional, list or list of lists/tuples for positional).

isinsert True if the statement is an INSERT.

isupdate True if the statement is an UPDATE.

should_autocommit True if the statement is a "committable" statement.

prefetch_cols a list of Column objects for which a client-side default was fired off. Applies to inserts and updates.

postfetch_cols a list of Column objects for which a server-side default or inline SQL expression value was fired off. Applies to inserts and updates.

create_cursor()

Return a new cursor generated from this ExecutionContext's connection.

Some dialects may wish to change the behavior of connection.cursor(), such as postgresql which may return a PG "server side" cursor.

exception = None

A DBAPI-level exception that was caught when this ExecutionContext attempted to execute a statement.

This attribute is meaningful only within the *`ConnectionEvents.dbapi_error()`* event.

New in version 0.9.7.

See also:

`ExecutionContext.is_disconnect`

ConnectionEvents.dbapi_error()

get_rowcount()
Return the DBAPI `cursor.rowcount` value, or in some cases an interpreted value.

See *ResultProxy.rowcount* for details on this.

handle_dbapi_exception(*e*)
Receive a DBAPI exception which occurred upon execute, result fetch, etc.

is_disconnect = None
Boolean flag set to True or False when a DBAPI-level exception is caught when this ExecutionContext attempted to execute a statement.

This attribute is meaningful only within the *ConnectionEvents.dbapi_error()* event.

New in version 0.9.7.

See also:

ExecutionContext.exception

ConnectionEvents.dbapi_error()

lastrow_has_defaults()
Return True if the last INSERT or UPDATE row contained inlined or database-side defaults.

post_exec()
Called after the execution of a compiled statement.

If a compiled statement was passed to this ExecutionContext, the *last_insert_ids*, *last_inserted_params*, etc. datamembers should be available after this method completes.

pre_exec()
Called before an execution of a compiled statement.

If a compiled statement was passed to this ExecutionContext, the *statement* and *parameters* datamembers must be initialized after this statement is complete.

result()
Return a result object corresponding to this ExecutionContext.

Returns a ResultProxy.

should_autocommit_text(*statement*)
Parse the given textual statement and return True if it refers to a "committable" statement

class sqlalchemy.log.**Identified**

class sqlalchemy.sql.compiler.**IdentifierPreparer**(*dialect*, *initial_quote=""*, *final_quote=None*, *escape_quote=""*, *omit_schema=False*)
Handle quoting and case-folding of identifiers based on options.

__init__(*dialect*, *initial_quote=""*, *final_quote=None*, *escape_quote=""*, *omit_schema=False*)
Construct a new `IdentifierPreparer` object.

initial_quote Character that begins a delimited identifier.

final_quote Character that ends a delimited identifier. Defaults to *initial_quote*.

omit_schema Prevent prepending schema name. Useful for databases that do not support schemae.

format_column(*column*, *use_table=False*, *name=None*, *table_name=None*)
Prepare a quoted column name.

format_schema(*name*, *quote=None*)
Prepare a quoted schema name.

format_table(*table*, *use_schema=True*, *name=None*)
Prepare a quoted table and schema name.

format_table_seq(*table*, *use_schema=True*)
Format table name and schema as a tuple.

quote(*ident*, *force=None*)
Conditionally quote an identifier.

the 'force' flag should be considered deprecated.

quote_identifier(*value*)
Quote an identifier.

Subclasses should override this to provide database-dependent quoting behavior.

quote_schema(*schema*, *force=None*)
Conditionally quote a schema.

Subclasses can override this to provide database-dependent quoting behavior for schema names.

the 'force' flag should be considered deprecated.

unformat_identifiers(*identifiers*)
Unpack 'schema.table.column'-like strings into components.

class sqlalchemy.sql.compiler.**SQLCompiler**(*dialect*, *statement*, *column_keys=None*, *inline=False*, ***kwargs*)

Bases: sqlalchemy.sql.compiler.Compiled

Default implementation of *Compiled*.

Compiles *ClauseElement* objects into SQL strings.

__init__(*dialect*, *statement*, *column_keys=None*, *inline=False*, ***kwargs*)
Construct a new *SQLCompiler* object.

Parameters

- **dialect** – *Dialect* to be used
- **statement** – *ClauseElement* to be compiled
- **column_keys** – a list of column names to be compiled into an INSERT or UPDATE statement.
- **inline** – whether to generate INSERT statements as "inline", e.g. not formatted to return any generated defaults
- **kwargs** – additional keyword arguments to be consumed by the superclass.

ansi_bind_rules = False
SQL 92 doesn't allow bind parameters to be used in the columns clause of a SELECT, nor does it allow ambiguous expressions like "? = ?". A compiler subclass can set this flag to False if the target driver/DB enforces this

construct_params(*params=None*, *_group_number=None*, *_check=True*)
return a dictionary of bind parameter keys and values

default_from()
Called when a SELECT statement has no froms, and no FROM clause is to be appended.

Gives Oracle a chance to tack on a FROM DUAL to the string output.

escape_literal_column(*text*)
provide escaping for the literal_column() construct.

get_select_precolumns(*select*, ***kw*)
Called when building a `SELECT` statement, position is just before column list.

isdelete = False
class-level defaults which can be set at the instance level to define if this Compiled instance represents INSERT/UPDATE/DELETE

isinsert = False
class-level defaults which can be set at the instance level to define if this Compiled instance represents INSERT/UPDATE/DELETE

isupdate = False
class-level defaults which can be set at the instance level to define if this Compiled instance represents INSERT/UPDATE/DELETE

params
Return the bind param dictionary embedded into this compiled object, for those values that are present.

render_literal_value(*value*, *type_*)
Render the value of a bind parameter as a quoted literal.

This is used for statement sections that do not accept bind parameters on the target driver/database.

This should be implemented by subclasses using the quoting services of the DBAPI.

render_table_with_column_in_update_from = False
set to True classwide to indicate the SET clause in a multi-table UPDATE statement should qualify columns with the table name (i.e. MySQL only)

returning = None
holds the "returning" collection of columns if the statement is CRUD and defines returning columns either implicitly or explicitly

returning_precedes_values = False
set to True classwide to generate RETURNING clauses before the VALUES or WHERE clause (i.e. MSSQL)

update_from_clause(*update_stmt*, *from_table*, *extra_froms*, *from_hints*, ***kw*)
Provide a hook to override the generation of an UPDATE..FROM clause.

MySQL and MSSQL override this.

update_limit_clause(*update_stmt*)
Provide a hook for MySQL to add LIMIT to the UPDATE

update_tables_clause(*update_stmt*, *from_table*, *extra_froms*, ***kw*)
Provide a hook to override the initial table clause in an UPDATE statement.

MySQL overrides this.

CHAPTER 4

Dialects

The **dialect** is the system SQLAlchemy uses to communicate with various types of DBAPI implementations and databases. The sections that follow contain reference documentation and notes specific to the usage of each backend, as well as notes for the various DBAPIs.

All dialects require that an appropriate DBAPI driver is installed.

4.1 Included Dialects

4.1.1 Firebird

Support for the Firebird database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- *fdb*
- *kinterbasdb*

Firebird Dialects

Firebird offers two distinct dialects (not to be confused with a SQLAlchemy `Dialect`):

dialect 1 This is the old syntax and behaviour, inherited from Interbase pre-6.0.

dialect 3 This is the newer and supported syntax, introduced in Interbase 6.0.

The SQLAlchemy Firebird dialect detects these versions and adjusts its representation of SQL accordingly. However, support for dialect 1 is not well tested and probably has incompatibilities.

Locking Behavior

Firebird locks tables aggressively. For this reason, a DROP TABLE may hang until other transactions are released. SQLAlchemy does its best to release transactions as quickly as possible. The most common cause of hanging transactions is a non-fully consumed result set, i.e.:

```
result = engine.execute("select * from table")
row = result.fetchone()
return
```

Where above, the `ResultProxy` has not been fully consumed. The connection will be returned to the pool and the transactional state rolled back once the Python garbage collector reclaims the objects which hold onto the connection, which often occurs asynchronously. The above use case can be alleviated by calling `first()` on the `ResultProxy` which will fetch the first row and immediately close all remaining cursor/connection resources.

RETURNING support

Firebird 2.0 supports returning a result set from inserts, and 2.1 extends that to deletes and updates. This is generically exposed by the SQLAlchemy `returning()` method, such as:

```
# INSERT..RETURNING
result = table.insert().returning(table.c.col1, table.c.col2).\
               values(name='foo')
print result.fetchall()

# UPDATE..RETURNING
raises = empl.update().returning(empl.c.id, empl.c.salary).\
              where(empl.c.sales>100).\
              values(dict(salary=empl.c.salary * 1.1))
print raises.fetchall()
```

fdb

Support for the Firebird database via the fdb driver. fdb is a kinterbasdb compatible DBAPI for Firebird.

New in version 0.8: - Support for the fdb Firebird driver.

Changed in version 0.9: - The fdb dialect is now the default dialect under the `firebird://` URL space, as `fdb` is now the official Python driver for Firebird.

DBAPI

Documentation and download information (if applicable) for fdb is available at: http://pypi.python.org/pypi/fdb/

Connecting

Connect String:

```
firebird+fdb://user:password@host:port/path/to/db[?key=value&key=value...]
```

Arguments

The `fdb` dialect is based on the *`sqlalchemy.dialects.firebird.kinterbasdb`* dialect, however does not accept every argument that Kinterbasdb does.

- `enable_rowcount` - True by default, setting this to False disables the usage of "cursor.rowcount" with the Kinterbasdb dialect, which SQLAlchemy ordinarily calls upon automatically after any UPDATE or DELETE statement. When disabled, SQLAlchemy's ResultProxy will return -1 for result.rowcount. The rationale here is that Kinterbasdb requires a second round trip to the database when .rowcount is called - since SQLA's resultproxy automatically closes the cursor after a non-result-returning statement, rowcount must be called, if at all, before the result object is returned. Additionally, cursor.rowcount may not return correct results with older versions of Firebird, and setting this flag to False will also cause the SQLAlchemy ORM to ignore its usage. The behavior can also be controlled on a per-execution basis using the `enable_rowcount` option with *`Connection.execution_options()`*:

```
conn = engine.connect().execution_options(enable_rowcount=True)
r = conn.execute(stmt)
print r.rowcount
```

- `retaining` - False by default. Setting this to True will pass the `retaining=True` keyword argument to the `.commit()` and `.rollback()` methods of the DBAPI connection, which can improve performance in some situations, but apparently with significant caveats. Please read the fdb and/or kinterbasdb DBAPI documentation in order to understand the implications of this flag.

  New in version 0.8.2: - `retaining` keyword argument specifying transaction retaining behavior - in 0.8 it defaults to `True` for backwards compatibility.

  Changed in version 0.9.0: - the `retaining` flag defaults to `False`. In 0.8 it defaulted to `True`.

  See also:

  http://pythonhosted.org/fdb/usage-guide.html#retaining-transactions - information on the "retaining" flag.

kinterbasdb

Support for the Firebird database via the kinterbasdb driver.

DBAPI

Documentation and download information (if applicable) for kinterbasdb is available at: http://firebirdsql.org/index.php?op=devel&sub=python

Connecting

Connect String:

```
firebird+kinterbasdb://user:password@host:port/path/to/db[?key=value&key=value...]
```

Arguments

The Kinterbasdb backend accepts the `enable_rowcount` and `retaining` arguments accepted by the *`sqlalchemy.dialects.firebird.fdb`* dialect. In addition, it also accepts the following:

- `type_conv` - select the kind of mapping done on the types: by default SQLAlchemy uses 200 with Unicode, datetime and decimal support. See the linked documents below for further information.
- `concurrency_level` - set the backend policy with regards to threading issues: by default SQLAlchemy uses policy 1. See the linked documents below for further information.

See also:

http://sourceforge.net/projects/kinterbasdb

http://kinterbasdb.sourceforge.net/dist_docs/usage.html#adv_param_conv_dynamic_type_translation

http://kinterbasdb.sourceforge.net/dist_docs/usage.html#special_issue_concurrency

4.1.2 Microsoft SQL Server

Support for the Microsoft SQL Server database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- *PyODBC*
- *mxODBC*
- *pymssql*
- *zxJDBC for Jython*
- *adodbapi*

Auto Increment Behavior

SQL Server provides so-called "auto incrementing" behavior using the `IDENTITY` construct, which can be placed on an integer primary key. SQLAlchemy considers `IDENTITY` within its default "autoincrement" behavior, described at `Column.autoincrement`; this means that by default, the first integer primary key column in a `Table` will be considered to be the identity column and will generate DDL as such:

```
from sqlalchemy import Table, MetaData, Column, Integer

m = MetaData()
t = Table('t', m,
        Column('id', Integer, primary_key=True),
        Column('x', Integer))
m.create_all(engine)
```

The above example will generate DDL as:

```
CREATE TABLE t (
    id INTEGER NOT NULL IDENTITY(1,1),
    x INTEGER NULL,
    PRIMARY KEY (id)
)
```

For the case where this default generation of `IDENTITY` is not desired, specify `autoincrement=False` on all integer primary key columns:

```
m = MetaData()
t = Table('t', m,
        Column('id', Integer, primary_key=True, autoincrement=False),
        Column('x', Integer))
m.create_all(engine)
```

Note: An INSERT statement which refers to an explicit value for such a column is prohibited by SQL Server, however SQLAlchemy will detect this and modify the `IDENTITY_INSERT` flag accordingly at statement execution time. As this is not a high performing process, care should be taken to set the `autoincrement` flag appropriately for columns that will not actually require IDENTITY behavior.

Controlling "Start" and "Increment"

Specific control over the parameters of the `IDENTITY` value is supported using the *`schema.Sequence`* object. While this object normally represents an explicit "sequence" for supporting backends, on SQL Server it is re-purposed to specify behavior regarding the identity column, including support of the "start" and "increment" values:

```
from sqlalchemy import Table, Integer, Sequence, Column

Table('test', metadata,
       Column('id', Integer,
              Sequence('blah', start=100, increment=10),
              primary_key=True),
       Column('name', String(20))
     ).create(some_engine)
```

would yield:

```
CREATE TABLE test (
  id INTEGER NOT NULL IDENTITY(100,10) PRIMARY KEY,
  name VARCHAR(20) NULL,
  )
```

Note that the `start` and `increment` values for sequences are optional and will default to 1,1.

INSERT behavior

Handling of the `IDENTITY` column at INSERT time involves two key techniques. The most common is being able to fetch the "last inserted value" for a given `IDENTITY` column, a process which SQLAlchemy performs implicitly in many cases, most importantly within the ORM.

The process for fetching this value has several variants:

- In the vast majority of cases, RETURNING is used in conjunction with INSERT statements on SQL Server in order to get newly generated primary key values:

```
INSERT INTO t (x) OUTPUT inserted.id VALUES (?)
```

- When RETURNING is not available or has been disabled via `implicit_returning=False`, either the `scope_identity()` function or the `@@identity` variable is used; behavior varies by backend:
  - when using PyODBC, the phrase `; select scope_identity()` will be appended to the end of the INSERT statement; a second result set will be fetched in order to receive the value. Given a table as:

```
t = Table('t', m, Column('id', Integer, primary_key=True),
        Column('x', Integer),
        implicit_returning=False)
```

an INSERT will look like:

```
INSERT INTO t (x) VALUES (?); select scope_identity()
```

- Other dialects such as pymssql will call upon `SELECT scope_identity() AS lastrowid` subsequent to an INSERT statement. If the flag `use_scope_identity=False` is passed to *create_engine()*, the statement `SELECT @@identity AS lastrowid` is used instead.

A table that contains an `IDENTITY` column will prohibit an INSERT statement that refers to the identity column explicitly. The SQLAlchemy dialect will detect when an INSERT construct, created using a core *insert()* construct (not a plain string SQL), refers to the identity column, and in this case will emit `SET IDENTITY_INSERT ON` prior to the insert statement proceeding, and `SET IDENTITY_INSERT OFF` subsequent to the execution. Given this example:

```
m = MetaData()
t = Table('t', m, Column('id', Integer, primary_key=True),
                Column('x', Integer))
m.create_all(engine)

engine.execute(t.insert(), {'id': 1, 'x':1}, {'id':2, 'x':2})
```

The above column will be created with IDENTITY, however the INSERT statement we emit is specifying explicit values. In the echo output we can see how SQLAlchemy handles this:

```
CREATE TABLE t (
    id INTEGER NOT NULL IDENTITY(1,1),
    x INTEGER NULL,
    PRIMARY KEY (id)
)

COMMIT
SET IDENTITY_INSERT t ON
INSERT INTO t (id, x) VALUES (?, ?)
((1, 1), (2, 2))
SET IDENTITY_INSERT t OFF
COMMIT
```

This is an auxilliary use case suitable for testing and bulk insert scenarios.

Collation Support

Character collations are supported by the base string types, specified by the string argument "collation":

```
from sqlalchemy import VARCHAR
Column('login', VARCHAR(32, collation='Latin1_General_CI_AS'))
```

When such a column is associated with a *Table*, the CREATE TABLE statement for this column will yield:

```
login VARCHAR(32) COLLATE Latin1_General_CI_AS NULL
```

New in version 0.8: Character collations are now part of the base string types.

LIMIT/OFFSET Support

MSSQL has no support for the LIMIT or OFFSET keywords. LIMIT is supported directly through the `TOP` Transact SQL keyword:

```
select.limit
```

will yield:

```
SELECT TOP n
```

If using SQL Server 2005 or above, LIMIT with OFFSET support is available through the `ROW_NUMBER OVER` construct. For versions below 2005, LIMIT with OFFSET usage will fail.

Nullability

MSSQL has support for three levels of column nullability. The default nullability allows nulls and is explicit in the CREATE TABLE construct:

```
name VARCHAR(20) NULL
```

If `nullable=None` is specified then no specification is made. In other words the database's configured default is used. This will render:

```
name VARCHAR(20)
```

If `nullable` is `True` or `False` then the column will be `NULL` or `NOT NULL` respectively.

Date / Time Handling

DATE and TIME are supported. Bind parameters are converted to datetime.datetime() objects as required by most MSSQL drivers, and results are processed from strings if needed. The DATE and TIME types are not available for MSSQL 2005 and previous - if a server version below 2008 is detected, DDL for these types will be issued as DATETIME.

Large Text/Binary Type Deprecation

Per SQL Server 2012/2014 Documentation, the `NTEXT`, `TEXT` and `IMAGE` datatypes are to be removed from SQL Server in a future release. SQLAlchemy normally relates these types to the *UnicodeText*, *Text* and *LargeBinary* datatypes.

In order to accommodate this change, a new flag `deprecate_large_types` is added to the dialect, which will be automatically set based on detection of the server version in use, if not otherwise set by the user. The behavior of this flag is as follows:

- When this flag is `True`, the *UnicodeText*, *Text* and *LargeBinary* datatypes, when used to render DDL, will render the types `NVARCHAR(max)`, `VARCHAR(max)`, and `VARBINARY(max)`, respectively. This is a new behavior as of the addition of this flag.
- When this flag is `False`, the *UnicodeText*, *Text* and *LargeBinary* datatypes, when used to render DDL, will render the types `NTEXT`, `TEXT`, and `IMAGE`, respectively. This is the long-standing behavior of these types.
- The flag begins with the value `None`, before a database connection is established. If the dialect is used to render DDL without the flag being set, it is interpreted the same as `False`.
- On first connection, the dialect detects if SQL Server version 2012 or greater is in use; if the flag is still at `None`, it sets it to `True` or `False` based on whether 2012 or greater is detected.

- The flag can be set to either `True` or `False` when the dialect is created, typically via *create_engine()*:

```
eng = create_engine("mssql+pymssql://user:pass@host/db",
                    deprecate_large_types=True)
```

- Complete control over whether the "old" or "new" types are rendered is available in all SQLAlchemy versions by using the UPPERCASE type objects instead: *NVARCHAR*, *VARCHAR*, *types.VARBINARY*, *TEXT*, *mssql.NTEXT*, *mssql.IMAGE* will always remain fixed and always output exactly that type.

New in version 1.0.0.

Legacy Schema Mode

Very old versions of the MSSQL dialect introduced the behavior such that a schema-qualified table would be auto-aliased when used in a SELECT statement; given a table:

```
account_table = Table(
    'account', metadata,
    Column('id', Integer, primary_key=True),
    Column('info', String(100)),
    schema="customer_schema"
)
```

this legacy mode of rendering would assume that "customer_schema.account" would not be accepted by all parts of the SQL statement, as illustrated below:

```
>>> eng = create_engine("mssql+pymssql://mydsn", legacy_schema_aliasing=True)
>>> print(account_table.select().compile(eng))
SELECT account_1.id, account_1.info
FROM customer_schema.account AS account_1
```

This mode of behavior is now off by default, as it appears to have served no purpose; however in the case that legacy applications rely upon it, it is available using the `legacy_schema_aliasing` argument to *create_engine()* as illustrated above.

Changed in version 1.1: the `legacy_schema_aliasing` flag introduced in version 1.0.5 to allow disabling of legacy mode for schemas now defaults to False.

Clustered Index Support

The MSSQL dialect supports clustered indexes (and primary keys) via the `mssql_clustered` option. This option is available to *Index*, *UniqueConstraint*. and *PrimaryKeyConstraint*.

To generate a clustered index:

```
Index("my_index", table.c.x, mssql_clustered=True)
```

which renders the index as `CREATE CLUSTERED INDEX my_index ON table (x)`.

New in version 0.8.

To generate a clustered primary key use:

```
Table('my_table', metadata,
      Column('x', ...),
      Column('y', ...),
      PrimaryKeyConstraint("x", "y", mssql_clustered=True))
```

which will render the table, for example, as:

```
CREATE TABLE my_table (x INTEGER NOT NULL, y INTEGER NOT NULL,
                       PRIMARY KEY CLUSTERED (x, y))
```

Similarly, we can generate a clustered unique constraint using:

```
  Table('my_table', metadata,
        Column('x', ...),
        Column('y', ...),
        PrimaryKeyConstraint("x"),
        UniqueConstraint("y", mssql_clustered=True),
        )

.. versionadded:: 0.9.2
```

MSSQL-Specific Index Options

In addition to clustering, the MSSQL dialect supports other special options for *Index*.

INCLUDE

The `mssql_include` option renders INCLUDE(colname) for the given string names:

```
Index("my_index", table.c.x, mssql_include=['y'])
```

would render the index as `CREATE INDEX my_index ON table (x) INCLUDE (y)`

New in version 0.8.

Index ordering

Index ordering is available via functional expressions, such as:

```
Index("my_index", table.c.x.desc())
```

would render the index as `CREATE INDEX my_index ON table (x DESC)`

New in version 0.8.

See also:

Functional Indexes

Compatibility Levels

MSSQL supports the notion of setting compatibility levels at the database level. This allows, for instance, to run a database that is compatible with SQL2000 while running on a SQL2005 database server. `server_version_info` will always return the database server version information (in this case SQL2005) and not the compatibility level information. Because of this, if running under a backwards compatibility mode SQAlchemy may attempt to use T-SQL statements that are unable to be parsed by the database server.

Triggers

SQLAlchemy by default uses OUTPUT INSERTED to get at newly generated primary key values via IDENTITY columns or other server side defaults. MS-SQL does not allow the usage of OUTPUT INSERTED on tables that have triggers. To disable the usage of OUTPUT INSERTED on a per-table basis, specify `implicit_returning=False` for each *Table* which has triggers:

```
Table('mytable', metadata,
    Column('id', Integer, primary_key=True),
    # ...,
    implicit_returning=False
)
```

Declarative form:

```
class MyClass(Base):
    # ...
    __table_args__ = {'implicit_returning':False}
```

This option can also be specified engine-wide using the `implicit_returning=False` argument on *create_engine()*.

Enabling Snapshot Isolation

Not necessarily specific to SQLAlchemy, SQL Server has a default transaction isolation mode that locks entire tables, and causes even mildly concurrent applications to have long held locks and frequent deadlocks. Enabling snapshot isolation for the database as a whole is recommended for modern levels of concurrency support. This is accomplished via the following ALTER DATABASE commands executed at the SQL prompt:

```
ALTER DATABASE MyDatabase SET ALLOW_SNAPSHOT_ISOLATION ON

ALTER DATABASE MyDatabase SET READ_COMMITTED_SNAPSHOT ON
```

Background on SQL Server snapshot isolation is available at http://msdn.microsoft.com/en-us/library/ms175095.aspx.

Known Issues

- No support for more than one `IDENTITY` column per table
- reflection of indexes does not work with versions older than SQL Server 2005

SQL Server Data Types

As with all SQLAlchemy dialects, all UPPERCASE types that are known to be valid with SQL server are importable from the top level dialect, whether they originate from *sqlalchemy.types* or from the local dialect:

```
from sqlalchemy.dialects.mssql import \
    BIGINT, BINARY, BIT, CHAR, DATE, DATETIME, DATETIME2, \
    DATETIMEOFFSET, DECIMAL, FLOAT, IMAGE, INTEGER, MONEY, \
    NCHAR, NTEXT, NUMERIC, NVARCHAR, REAL, SMALLDATETIME, \
    SMALLINT, SMALLMONEY, SQL_VARIANT, TEXT, TIME, \
    TIMESTAMP, TINYINT, UNIQUEIDENTIFIER, VARBINARY, VARCHAR
```

Types which are specific to SQL Server, or have SQL Server-specific construction arguments, are as follows:

class sqlalchemy.dialects.mssql.**BIT**

Bases: *sqlalchemy.types.TypeEngine*

__init__

x.__init__(...) initializes x; see help(type(x)) for signature

class sqlalchemy.dialects.mssql.**CHAR**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

Bases: *sqlalchemy.types.String*

The SQL CHAR type.

__init__(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

Create a string-holding type.

Parameters

- **length** – optional, a length for the column for use in DDL and CAST expressions. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a `length` for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued if a `VARCHAR` with no length is included. Whether the value is interpreted as bytes or characters is database specific.
- **collation** – Optional, a column-level collation for use in DDL and CAST expressions. Renders using the COLLATE keyword supported by SQLite, MySQL, and Postgresql. E.g.:

```
>>> from sqlalchemy import cast, select, String
>>> print select([cast('some string', String(collation='utf8'))])
SELECT CAST(:param_1 AS VARCHAR COLLATE utf8) AS anon_1
```

  New in version 0.8: Added support for COLLATE to all string types.

- **convert_unicode** – When set to `True`, the *String* type will assume that input is to be passed as Python `unicode` objects, and results returned as Python `unicode` objects. If the DBAPI in use does not support Python unicode (which is fewer and fewer these days), SQLAlchemy will encode/decode the value, using the value of the `encoding` parameter passed to *create_engine()* as the encoding.

  When using a DBAPI that natively supports Python unicode objects, this flag generally does not need to be set. For columns that are explicitly intended to store non-ASCII data, the *Unicode* or *UnicodeText* types should be used regardless, which feature the same behavior of `convert_unicode` but also indicate an underlying column type that directly supports unicode, such as `NVARCHAR`.

For the extremely rare case that Python `unicode` is to be encoded/decoded by SQLAlchemy on a backend that does natively support Python `unicode`, the value `force` can be passed here which will cause SQLAlchemy's encode/decode services to be used unconditionally.

- **`unicode_error`** – Optional, a method to use to handle Unicode conversion errors. Behaves like the `errors` keyword argument to the standard library's `string.decode()` functions. This flag requires that `convert_unicode` is set to `force` - otherwise, SQLAlchemy is not guaranteed to handle the task of unicode conversion. Note that this flag adds significant performance overhead to row-fetching operations for backends that already return unicode objects natively (which most DBAPIs do). This flag should only be used as a last resort for reading strings from a column with varied or corrupted encodings.

class `sqlalchemy.dialects.mssql.`**`DATETIME2`**(*precision=None*, ***kw*)
Bases: `sqlalchemy.dialects.mssql.base._DateTimeBase`, *`sqlalchemy.types.DateTime`*

class `sqlalchemy.dialects.mssql.`**`DATETIMEOFFSET`**(*precision=None*, ***kwargs*)
Bases: *`sqlalchemy.types.TypeEngine`*

class `sqlalchemy.dialects.mssql.`**`IMAGE`**(*length=None*)
Bases: *`sqlalchemy.types.LargeBinary`*

`__init__`(*length=None*)
Construct a LargeBinary type.

Parameters **`length`** – optional, a length for the column for use in DDL statements, for those BLOB types that accept a length (i.e. MySQL). It does *not* produce a *lengthed* BINARY/VARBINARY type - use the BINARY/VARBINARY types specifically for those. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued.

class `sqlalchemy.dialects.mssql.`**`MONEY`**
Bases: *`sqlalchemy.types.TypeEngine`*

`__init__`
x.__init__(...) initializes x; see help(type(x)) for signature

class `sqlalchemy.dialects.mssql.`**`NCHAR`**(*length=None*, ***kwargs*)
Bases: *`sqlalchemy.types.Unicode`*

The SQL NCHAR type.

`__init__`(*length=None*, ***kwargs*)
Create a *`Unicode`* object.

Parameters are the same as that of *`String`*, with the exception that `convert_unicode` defaults to `True`.

class `sqlalchemy.dialects.mssql.`**`NTEXT`**(*length=None*, ***kwargs*)
Bases: *`sqlalchemy.types.UnicodeText`*

MSSQL NTEXT type, for variable-length unicode text up to 2^30 characters.

`__init__`(*length=None*, ***kwargs*)
Create a Unicode-converting Text type.

Parameters are the same as that of *`Text`*, with the exception that `convert_unicode` defaults to `True`.

class `sqlalchemy.dialects.mssql.`**`NVARCHAR`**(*length=None*, ***kwargs*)
Bases: *`sqlalchemy.types.Unicode`*

The SQL NVARCHAR type.

__init__(*length=None*, ***kwargs*)

Create a `Unicode` object.

Parameters are the same as that of `String`, with the exception that `convert_unicode` defaults to `True`.

class sqlalchemy.dialects.mssql.**REAL**(***kw*)

Bases: `sqlalchemy.types.REAL`

class sqlalchemy.dialects.mssql.**SMALLDATETIME**(*timezone=False*)

Bases: sqlalchemy.dialects.mssql.base._DateTimeBase, `sqlalchemy.types.DateTime`

__init__(*timezone=False*)

Construct a new `DateTime`.

Parameters timezone – boolean. If True, and supported by the backend, will produce 'TIMESTAMP WITH TIMEZONE'. For backends that don't support timezone aware timestamps, has no effect.

class sqlalchemy.dialects.mssql.**SMALLMONEY**

Bases: `sqlalchemy.types.TypeEngine`

__init__

x.__init__(...) initializes x; see help(type(x)) for signature

class sqlalchemy.dialects.mssql.**SQL_VARIANT**

Bases: `sqlalchemy.types.TypeEngine`

__init__

x.__init__(...) initializes x; see help(type(x)) for signature

class sqlalchemy.dialects.mssql.**TEXT**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

Bases: `sqlalchemy.types.Text`

The SQL TEXT type.

__init__(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

Create a string-holding type.

Parameters

- **length** – optional, a length for the column for use in DDL and CAST expressions. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a `length` for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued if a `VARCHAR` with no length is included. Whether the value is interpreted as bytes or characters is database specific.
- **collation** – Optional, a column-level collation for use in DDL and CAST expressions. Renders using the COLLATE keyword supported by SQLite, MySQL, and Postgresql. E.g.:

```
>>> from sqlalchemy import cast, select, String
>>> print select([cast('some string', String(collation='utf8'))])
SELECT CAST(:param_1 AS VARCHAR COLLATE utf8) AS anon_1
```

New in version 0.8: Added support for COLLATE to all string types.

- **convert_unicode** – When set to `True`, the *String* type will assume that input is to be passed as Python `unicode` objects, and results returned as Python `unicode` objects. If the DBAPI in use does not support Python unicode (which is fewer and fewer these days), SQLAlchemy will encode/decode the value, using the value of the `encoding` parameter passed to *create_engine()* as the encoding.

  When using a DBAPI that natively supports Python unicode objects, this flag generally does not need to be set. For columns that are explicitly intended to store non-ASCII data, the *Unicode* or *UnicodeText* types should be used regardless, which feature the same behavior of `convert_unicode` but also indicate an underlying column type that directly supports unicode, such as `NVARCHAR`.

  For the extremely rare case that Python `unicode` is to be encoded/decoded by SQLAlchemy on a backend that does natively support Python `unicode`, the value `force` can be passed here which will cause SQLAlchemy's encode/decode services to be used unconditionally.

- **unicode_error** – Optional, a method to use to handle Unicode conversion errors. Behaves like the `errors` keyword argument to the standard library's `string.decode()` functions. This flag requires that `convert_unicode` is set to `force` - otherwise, SQLAlchemy is not guaranteed to handle the task of unicode conversion. Note that this flag adds significant performance overhead to row-fetching operations for backends that already return unicode objects natively (which most DBAPIs do). This flag should only be used as a last resort for reading strings from a column with varied or corrupted encodings.

class `sqlalchemy.dialects.mssql.`**TIME**(*precision=None*, ***kwargs*)

Bases: *sqlalchemy.types.TIME*

class `sqlalchemy.dialects.mssql.`**TINYINT**

Bases: *sqlalchemy.types.Integer*

__init__

x.__init__(...) initializes x; see help(type(x)) for signature

class `sqlalchemy.dialects.mssql.`**UNIQUEIDENTIFIER**

Bases: *sqlalchemy.types.TypeEngine*

__init__

x.__init__(...) initializes x; see help(type(x)) for signature

class `sqlalchemy.dialects.mssql.`**VARCHAR**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

Bases: *sqlalchemy.types.String*

The SQL VARCHAR type.

__init__(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

Create a string-holding type.

Parameters

- **length** – optional, a length for the column for use in DDL and CAST expressions. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a `length` for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued if a `VARCHAR` with no length is included. Whether the value is interpreted as bytes or characters is database specific.

- **collation** – Optional, a column-level collation for use in DDL and CAST expressions. Renders using the COLLATE keyword supported by SQLite, MySQL, and Postgresql. E.g.:

```
>>> from sqlalchemy import cast, select, String
>>> print select([cast('some string', String(collation='utf8'))])
SELECT CAST(:param_1 AS VARCHAR COLLATE utf8) AS anon_1
```

  New in version 0.8: Added support for COLLATE to all string types.

- **convert_unicode** – When set to `True`, the *String* type will assume that input is to be passed as Python `unicode` objects, and results returned as Python `unicode` objects. If the DBAPI in use does not support Python unicode (which is fewer and fewer these days), SQLAlchemy will encode/decode the value, using the value of the `encoding` parameter passed to *create_engine()* as the encoding.

  When using a DBAPI that natively supports Python unicode objects, this flag generally does not need to be set. For columns that are explicitly intended to store non-ASCII data, the *Unicode* or *UnicodeText* types should be used regardless, which feature the same behavior of `convert_unicode` but also indicate an underlying column type that directly supports unicode, such as `NVARCHAR`.

  For the extremely rare case that Python `unicode` is to be encoded/decoded by SQLAlchemy on a backend that does natively support Python `unicode`, the value `force` can be passed here which will cause SQLAlchemy's encode/decode services to be used unconditionally.

- **unicode_error** – Optional, a method to use to handle Unicode conversion errors. Behaves like the `errors` keyword argument to the standard library's `string.decode()` functions. This flag requires that `convert_unicode` is set to `force` - otherwise, SQLAlchemy is not guaranteed to handle the task of unicode conversion. Note that this flag adds significant performance overhead to row-fetching operations for backends that already return unicode objects natively (which most DBAPIs do). This flag should only be used as a last resort for reading strings from a column with varied or corrupted encodings.

PyODBC

Support for the Microsoft SQL Server database via the PyODBC driver.

DBAPI

Documentation and download information (if applicable) for PyODBC is available at: http://pypi.python.org/pypi/pyodbc/

Connecting

Connect String:

```
mssql+pyodbc://<username>:<password>@<dsnname>
```

Connecting to PyODBC

The URL here is to be translated to PyODBC connection strings, as detailed in ConnectionStrings.

DSN Connections A DSN-based connection is **preferred** overall when using ODBC. A basic DSN-based connection looks like:

```
engine = create_engine("mssql+pyodbc://scott:tiger@some_dsn")
```

Which above, will pass the following connection string to PyODBC:

```
dsn=mydsn;UID=user;PWD=pass
```

If the username and password are omitted, the DSN form will also add the `Trusted_Connection=yes` directive to the ODBC string.

Hostname Connections Hostname-based connections are **not preferred**, however are supported. The ODBC driver name must be explicitly specified:

```
engine = create_engine("mssql+pyodbc://scott:tiger@myhost:port/databasename?driver=SQL+Server+Native-
```

Changed in version 1.0.0: Hostname-based PyODBC connections now require the SQL Server driver name specified explicitly. SQLAlchemy cannot choose an optimal default here as it varies based on platform and installed drivers.

Other keywords interpreted by the Pyodbc dialect to be passed to `pyodbc.connect()` in both the DSN and hostname cases include: `odbc_autotranslate`, `ansi`, `unicode_results`, `autocommit`.

Pass through exact Pyodbc string A PyODBC connection string can also be sent exactly as specified in ConnectionStrings into the driver using the parameter `odbc_connect`. The delimeters must be URL escaped, however, as illustrated below using `urllib.quote_plus`:

```
import urllib
params = urllib.quote_plus("DRIVER={SQL Server Native Client 10.0};SERVER=dagger;DATABASE=test;UID=us

engine = create_engine("mssql+pyodbc:///?odbc_connect=%s" % params)
```

Unicode Binds

The current state of PyODBC on a unix backend with FreeTDS and/or EasySoft is poor regarding unicode; different OS platforms and versions of UnixODBC versus IODBC versus FreeTDS/EasySoft versus PyODBC itself dramatically alter how strings are received. The PyODBC dialect attempts to use all the information it knows to determine whether or not a Python unicode literal can be passed directly to the PyODBC driver or not; while SQLAlchemy can encode these to bytestrings first, some users have reported that PyODBC mis-handles bytestrings for certain encodings and requires a Python unicode object, while the author has observed widespread cases where a Python unicode is completely misinterpreted by PyODBC, particularly when dealing with the information schema tables used in table reflection, and the value must first be encoded to a bytestring.

It is for this reason that whether or not unicode literals for bound parameters be sent to PyODBC can be controlled using the `supports_unicode_binds` parameter to `create_engine()`. When left at its default of `None`, the PyODBC dialect will use its best guess as to whether or not the driver deals with unicode literals well. When `False`, unicode literals will be encoded first, and when `True` unicode literals will be passed straight through. This is an interim flag that hopefully should not be needed when the unicode situation stabilizes for unix + PyODBC.

New in version 0.7.7: `supports_unicode_binds` parameter to `create_engine()`.

mxODBC

Support for the Microsoft SQL Server database via the mxODBC driver.

DBAPI

Documentation and download information (if applicable) for mxODBC is available at: http://www.egenix.com/

Connecting

Connect String:

```
mssql+mxodbc://<username>:<password>@<dsnname>
```

Execution Modes

mxODBC features two styles of statement execution, using the `cursor.execute()` and `cursor.executedirect()` methods (the second being an extension to the DBAPI specification). The former makes use of a particular API call specific to the SQL Server Native Client ODBC driver known SQLDescribeParam, while the latter does not.

mxODBC apparently only makes repeated use of a single prepared statement when SQLDescribeParam is used. The advantage to prepared statement reuse is one of performance. The disadvantage is that SQLDescribeParam has a limited set of scenarios in which bind parameters are understood, including that they cannot be placed within the argument lists of function calls, anywhere outside the FROM, or even within subqueries within the FROM clause - making the usage of bind parameters within SELECT statements impossible for all but the most simplistic statements.

For this reason, the mxODBC dialect uses the "native" mode by default only for INSERT, UPDATE, and DELETE statements, and uses the escaped string mode for all other statements.

This behavior can be controlled via *`execution_options()`* using the `native_odbc_execute` flag with a value of `True` or `False`, where a value of `True` will unconditionally use native bind parameters and a value of `False` will unconditionally use string-escaped parameters.

pymssql

Support for the Microsoft SQL Server database via the pymssql driver.

DBAPI

Documentation and download information (if applicable) for pymssql is available at: http://pymssql.org/

Connecting

Connect String:

```
mssql+pymssql://<username>:<password>@<freetds_name>?charset=utf8
```

pymssql is a Python module that provides a Python DBAPI interface around FreeTDS. Compatible builds are available for Linux, MacOSX and Windows platforms.

zxjdbc

Support for the Microsoft SQL Server database via the zxJDBC for Jython driver.

Note: Jython is not supported by current versions of SQLAlchemy. The zxjdbc dialect should be considered as experimental.

DBAPI

Drivers for this database are available at: http://jtds.sourceforge.net/

Connecting

Connect String:

```
mssql+zxjdbc://user:pass@host:port/dbname[?key=value&key=value...]
```

AdoDBAPI

Support for the Microsoft SQL Server database via the adodbapi driver.

DBAPI

Documentation and download information (if applicable) for adodbapi is available at: http://adodbapi.sourceforge.net/

Connecting

Connect String:

```
mssql+adodbapi://<username>:<password>@<dsnname>
```

Note: The adodbapi dialect is not implemented SQLAlchemy versions 0.6 and above at this time.

4.1.3 MySQL

Support for the MySQL database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- *MySQL-Python*
- *PyMySQL*
- *MySQL Connector/Python*
- *CyMySQL*
- *OurSQL*
- *Google Cloud SQL*
- *PyODBC*
- *zxjdbc for Jython*

Supported Versions and Features

SQLAlchemy supports MySQL starting with version 4.1 through modern releases. However, no heroic measures are taken to work around major missing SQL features - if your server version does not support sub-selects, for example, they won't work in SQLAlchemy either.

See the official MySQL documentation for detailed information about features supported in any given server release.

Connection Timeouts

MySQL features an automatic connection close behavior, for connections that have been idle for eight hours or more. To circumvent having this issue, use the `pool_recycle` option which controls the maximum age of any connection:

```
engine = create_engine('mysql+mysqldb://...', pool_recycle=3600)
```

See also:

Setting Pool Recycle - full description of the pool recycle feature.

CREATE TABLE arguments including Storage Engines

MySQL's CREATE TABLE syntax includes a wide array of special options, including `ENGINE`, `CHARSET`, `MAX_ROWS`, `ROW_FORMAT`, `INSERT_METHOD`, and many more. To accommodate the rendering of these arguments, specify the form `mysql_argument_name="value"`. For example, to specify a table with `ENGINE` of `InnoDB`, `CHARSET` of `utf8`, and `KEY_BLOCK_SIZE` of `1024`:

```
Table('mytable', metadata,
      Column('data', String(32)),
      mysql_engine='InnoDB',
      mysql_charset='utf8',
      mysql_key_block_size="1024"
     )
```

The MySQL dialect will normally transfer any keyword specified as `mysql_keyword_name` to be rendered as `KEYWORD_NAME` in the `CREATE TABLE` statement. A handful of these names will render with a space instead of an underscore; to support this, the MySQL dialect has awareness of these particular names, which include

DATA DIRECTORY (e.g. mysql_data_directory), CHARACTER SET (e.g. mysql_character_set) and INDEX DIRECTORY (e.g. mysql_index_directory).

The most common argument is mysql_engine, which refers to the storage engine for the table. Historically, MySQL server installations would default to MyISAM for this value, although newer versions may be defaulting to InnoDB. The InnoDB engine is typically preferred for its support of transactions and foreign keys.

A *Table* that is created in a MySQL database with a storage engine of MyISAM will be essentially non-transactional, meaning any INSERT/UPDATE/DELETE statement referring to this table will be invoked as autocommit. It also will have no support for foreign key constraints; while the CREATE TABLE statement accepts foreign key options, when using the MyISAM storage engine these arguments are discarded. Reflecting such a table will also produce no foreign key constraint information.

For fully atomic transactions as well as support for foreign key constraints, all participating CREATE TABLE statements must specify a transactional engine, which in the vast majority of cases is InnoDB.

See also:

The InnoDB Storage Engine - on the MySQL website.

Case Sensitivity and Table Reflection

MySQL has inconsistent support for case-sensitive identifier names, basing support on specific details of the underlying operating system. However, it has been observed that no matter what case sensitivity behavior is present, the names of tables in foreign key declarations are *always* received from the database as all-lower case, making it impossible to accurately reflect a schema where inter-related tables use mixed-case identifier names.

Therefore it is strongly advised that table names be declared as all lower case both within SQLAlchemy as well as on the MySQL database itself, especially if database reflection features are to be used.

Transaction Isolation Level

create_engine() accepts an *create_engine.isolation_level* parameter which results in the command SET SESSION TRANSACTION ISOLATION LEVEL <level> being invoked for every new connection. Valid values for this parameter are READ COMMITTED, READ UNCOMMITTED, REPEATABLE READ, and SERIALIZABLE:

```
engine = create_engine(
                "mysql://scott:tiger@localhost/test",
                isolation_level="READ UNCOMMITTED"
            )
```

New in version 0.7.6.

AUTO_INCREMENT Behavior

When creating tables, SQLAlchemy will automatically set AUTO_INCREMENT on the first *Integer* primary key column which is not marked as a foreign key:

```
>>> t = Table('mytable', metadata,
...   Column('mytable_id', Integer, primary_key=True)
... )
>>> t.create()
CREATE TABLE mytable (
        id INTEGER NOT NULL AUTO_INCREMENT,
```

```
        PRIMARY KEY (id)
)
```

You can disable this behavior by passing `False` to the *`autoincrement`* argument of *`Column`*. This flag can also be used to enable auto-increment on a secondary column in a multi-column key for some storage engines:

```
Table('mytable', metadata,
      Column('gid', Integer, primary_key=True, autoincrement=False),
      Column('id', Integer, primary_key=True)
     )
```

Unicode

Charset Selection

Most MySQL DBAPIs offer the option to set the client character set for a connection. This is typically delivered using the `charset` parameter in the URL, such as:

```
e = create_engine("mysql+pymysql://scott:tiger@localhost/test?charset=utf8")
```

This charset is the **client character set** for the connection. Some MySQL DBAPIs will default this to a value such as `latin1`, and some will make use of the `default-character-set` setting in the `my.cnf` file as well. Documentation for the DBAPI in use should be consulted for specific behavior.

The encoding used for Unicode has traditionally been `'utf8'`. However, for MySQL versions 5.5.3 on forward, a new MySQL-specific encoding `'utf8mb4'` has been introduced. The rationale for this new encoding is due to the fact that MySQL's utf-8 encoding only supports codepoints up to three bytes instead of four. Therefore, when communicating with a MySQL database that includes codepoints more than three bytes in size, this new charset is preferred, if supported by both the database as well as the client DBAPI, as in:

```
e = create_engine("mysql+pymysql://scott:tiger@localhost/test?charset=utf8mb4")
```

At the moment, up-to-date versions of MySQLdb and PyMySQL support the `utf8mb4` charset. Other DBAPIs such as MySQL-Connector and OurSQL may **not** support it as of yet.

In order to use `utf8mb4` encoding, changes to the MySQL schema and/or server configuration may be required.

See also:

The utf8mb4 Character Set - in the MySQL documentation

Unicode Encoding / Decoding

All modern MySQL DBAPIs all offer the service of handling the encoding and decoding of unicode data between the Python application space and the database. As this was not always the case, SQLAlchemy also includes a comprehensive system of performing the encode/decode task as well. As only one of these systems should be in use at at time, SQLAlchemy has long included functionality to automatically detect upon first connection whether or not the DBAPI is automatically handling unicode.

Whether or not the MySQL DBAPI will handle encoding can usually be configured using a DBAPI flag `use_unicode`, which is known to be supported at least by MySQLdb, PyMySQL, and MySQL-Connector. Setting this value to `0` in the "connect args" or query string will have the effect of disabling the DBAPI's handling of unicode, such that it instead will return data of the `str` type or `bytes` type, with data in the configured charset:

```
# connect while disabling the DBAPI's unicode encoding/decoding
e = create_engine("mysql+mysqldb://scott:tiger@localhost/test?charset=utf8&use_unicode=0")
```

Current recommendations for modern DBAPIs are as follows:

- It is generally always safe to leave the `use_unicode` flag set at its default; that is, don't use it at all.
- Under Python 3, the `use_unicode=0` flag should **never be used**. SQLAlchemy under Python 3 generally assumes the DBAPI receives and returns string values as Python 3 strings, which are inherently unicode objects.
- Under Python 2 with MySQLdb, the `use_unicode=0` flag will **offer superior performance**, as MySQLdb's unicode converters under Python 2 only have been observed to have unusually slow performance compared to SQLAlchemy's fast C-based encoders/decoders.

In short: don't specify `use_unicode` *at all*, with the possible exception of `use_unicode=0` on MySQLdb with Python 2 **only** for a potential performance gain.

Ansi Quoting Style

MySQL features two varieties of identifier "quoting style", one using backticks and the other using quotes, e.g. `` `some_identifier` `` vs. `"some_identifier"`. All MySQL dialects detect which version is in use by checking the value of `sql_mode` when a connection is first established with a particular *Engine*. This quoting style comes into play when rendering table and column names as well as when reflecting existing database structures. The detection is entirely automatic and no special configuration is needed to use either quoting style.

Changed in version 0.6: detection of ANSI quoting style is entirely automatic, there's no longer any end-user `create_engine()` options in this regard.

MySQL SQL Extensions

Many of the MySQL SQL extensions are handled through SQLAlchemy's generic function and operator support:

```
table.select(table.c.password==func.md5('plaintext'))
table.select(table.c.username.op('regexp')('^[a-d]'))
```

And of course any valid MySQL statement can be executed as a string as well.

Some limited direct support for MySQL extensions to SQL is currently available.

- SELECT pragma:

```
select(..., prefixes=['HIGH_PRIORITY', 'SQL_SMALL_RESULT'])
```

- UPDATE with LIMIT:

```
update(..., mysql_limit=10)
```

rowcount Support

SQLAlchemy standardizes the DBAPI `cursor.rowcount` attribute to be the usual definition of "number of rows matched by an UPDATE or DELETE" statement. This is in contradiction to the default setting on most MySQL DBAPI drivers, which is "number of rows actually modified/deleted". For this reason, the SQLAlchemy MySQL dialects always add the `constants.CLIENT.FOUND_ROWS` flag, or whatever is equivalent for the target dialect, upon connection. This setting is currently hardcoded.

See also:

ResultProxy.rowcount

CAST Support

MySQL documents the CAST operator as available in version 4.0.2. When using the SQLAlchemy *cast()* function, SQLAlchemy will not render the CAST token on MySQL before this version, based on server version detection, instead rendering the internal expression directly.

CAST may still not be desirable on an early MySQL version post-4.0.2, as it didn't add all datatype support until 4.1.1. If your application falls into this narrow area, the behavior of CAST can be controlled using the *Custom SQL Constructs and Compilation Extension* system, as per the recipe below:

```
from sqlalchemy.sql.expression import Cast
from sqlalchemy.ext.compiler import compiles

@compiles(Cast, 'mysql')
def _check_mysql_version(element, compiler, **kw):
    if compiler.dialect.server_version_info < (4, 1, 0):
        return compiler.process(element.clause, **kw)
    else:
        return compiler.visit_cast(element, **kw)
```

The above function, which only needs to be declared once within an application, overrides the compilation of the *cast()* construct to check for version 4.1.0 before fully rendering CAST; else the internal element of the construct is rendered directly.

MySQL Specific Index Options

MySQL-specific extensions to the *Index* construct are available.

Index Length

MySQL provides an option to create index entries with a certain length, where "length" refers to the number of characters or bytes in each value which will become part of the index. SQLAlchemy provides this feature via the `mysql_length` parameter:

```
Index('my_index', my_table.c.data, mysql_length=10)

Index('a_b_idx', my_table.c.a, my_table.c.b, mysql_length={'a': 4,
                                                           'b': 9})
```

Prefix lengths are given in characters for nonbinary string types and in bytes for binary string types. The value passed to the keyword argument *must* be either an integer (and, thus, specify the same prefix length value for all columns of the index) or a dict in which keys are column names and values are prefix length values for corresponding columns. MySQL only allows a length for a column of an index if it is for a CHAR, VARCHAR, TEXT, BINARY, VARBINARY and BLOB.

New in version 0.8.2: `mysql_length` may now be specified as a dictionary for use with composite indexes.

Index Types

Some MySQL storage engines permit you to specify an index type when creating an index or primary key constraint. SQLAlchemy provides this feature via the `mysql_using` parameter on *Index*:

```
Index('my_index', my_table.c.data, mysql_using='hash')
```

As well as the `mysql_using` parameter on *PrimaryKeyConstraint*:

```
PrimaryKeyConstraint("data", mysql_using='hash')
```

The value passed to the keyword argument will be simply passed through to the underlying CREATE INDEX or PRIMARY KEY clause, so it *must* be a valid index type for your MySQL storage engine.

More information can be found at:

http://dev.mysql.com/doc/refman/5.0/en/create-index.html

http://dev.mysql.com/doc/refman/5.0/en/create-table.html

MySQL Foreign Keys

MySQL's behavior regarding foreign keys has some important caveats.

Foreign Key Arguments to Avoid

MySQL does not support the foreign key arguments "DEFERRABLE", "INITIALLY", or "MATCH". Using the `deferrable` or `initially` keyword argument with *ForeignKeyConstraint* or *ForeignKey* will have the effect of these keywords being rendered in a DDL expression, which will then raise an error on MySQL. In order to use these keywords on a foreign key while having them ignored on a MySQL backend, use a custom compile rule:

```
from sqlalchemy.ext.compiler import compiles
from sqlalchemy.schema import ForeignKeyConstraint

@compiles(ForeignKeyConstraint, "mysql")
def process(element, compiler, **kw):
    element.deferrable = element.initially = None
    return compiler.visit_foreign_key_constraint(element, **kw)
```

Changed in version 0.9.0: - the MySQL backend no longer silently ignores the `deferrable` or `initially` keyword arguments of *ForeignKeyConstraint* and *ForeignKey*.

The "MATCH" keyword is in fact more insidious, and is explicitly disallowed by SQLAlchemy in conjunction with the MySQL backend. This argument is silently ignored by MySQL, but in addition has the effect of ON UPDATE and ON DELETE options also being ignored by the backend. Therefore MATCH should never be used with the MySQL backend; as is the case with DEFERRABLE and INITIALLY, custom compilation rules can be used to correct a MySQL ForeignKeyConstraint at DDL definition time.

New in version 0.9.0: - the MySQL backend will raise a *CompileError* when the `match` keyword is used with *ForeignKeyConstraint* or *ForeignKey*.

Reflection of Foreign Key Constraints

Not all MySQL storage engines support foreign keys. When using the very common `MyISAM` MySQL storage engine, the information loaded by table reflection will not include foreign keys. For these tables, you may supply a `ForeignKeyConstraint` at reflection time:

```
Table('mytable', metadata,
      ForeignKeyConstraint(['other_id'], ['othertable.other_id']),
      autoload=True
     )
```

See also:

CREATE TABLE arguments including Storage Engines

MySQL Unique Constraints and Reflection

SQLAlchemy supports both the *Index* construct with the flag `unique=True`, indicating a UNIQUE index, as well as the *UniqueConstraint* construct, representing a UNIQUE constraint. Both objects/syntaxes are supported by MySQL when emitting DDL to create these constraints. However, MySQL does not have a unique constraint construct that is separate from a unique index; that is, the "UNIQUE" constraint on MySQL is equivalent to creating a "UNIQUE INDEX".

When reflecting these constructs, the *Inspector.get_indexes()* and the *Inspector.get_unique_constraints()* methods will **both** return an entry for a UNIQUE index in MySQL. However, when performing full table reflection using `Table(..., autoload=True)`, the *UniqueConstraint* construct is **not** part of the fully reflected *Table* construct under any circumstances; this construct is always represented by a *Index* with the `unique=True` setting present in the `Table.indexes` collection.

TIMESTAMP Columns and NULL

MySQL historically enforces that a column which specifies the TIMESTAMP datatype implicitly includes a default value of CURRENT_TIMESTAMP, even though this is not stated, and additionally sets the column as NOT NULL, the opposite behavior vs. that of all other datatypes:

```
mysql> CREATE TABLE ts_test (
    -> a INTEGER,
    -> b INTEGER NOT NULL,
    -> c TIMESTAMP,
    -> d TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    -> e TIMESTAMP NULL);
Query OK, 0 rows affected (0.03 sec)

mysql> SHOW CREATE TABLE ts_test;
+---------+-----------------------------------------------------
| Table   | Create Table
+---------+-----------------------------------------------------
| ts_test | CREATE TABLE `ts_test` (
  `a` int(11) DEFAULT NULL,
  `b` int(11) NOT NULL,
  `c` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  `d` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `e` timestamp NULL DEFAULT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1
```

Above, we see that an INTEGER column defaults to NULL, unless it is specified with NOT NULL. But when the column is of type TIMESTAMP, an implicit default of CURRENT_TIMESTAMP is generated which also coerces the column to be a NOT NULL, even though we did not specify it as such.

This behavior of MySQL can be changed on the MySQL side using the explicit_defaults_for_timestamp configuration flag introduced in MySQL 5.6. With this server setting enabled, TIMESTAMP columns behave like any other datatype on the MySQL side with regards to defaults and nullability.

However, to accommodate the vast majority of MySQL databases that do not specify this new flag, SQLAlchemy emits the "NULL" specifier explicitly with any TIMESTAMP column that does not specify `nullable=False`. In order to accommodate newer databases that specify `explicit_defaults_for_timestamp`, SQLAlchemy also emits NOT NULL for TIMESTAMP columns that do specify `nullable=False`. The following example illustrates:

```
from sqlalchemy import MetaData, Integer, Table, Column, text
from sqlalchemy.dialects.mysql import TIMESTAMP

m = MetaData()
t = Table('ts_test', m,
        Column('a', Integer),
        Column('b', Integer, nullable=False),
        Column('c', TIMESTAMP),
        Column('d', TIMESTAMP, nullable=False)
    )


from sqlalchemy import create_engine
e = create_engine("mysql://scott:tiger@localhost/test", echo=True)
m.create_all(e)
```

output:

```
CREATE TABLE ts_test (
    a INTEGER,
    b INTEGER NOT NULL,
    c TIMESTAMP NULL,
    d TIMESTAMP NOT NULL
)
```

Changed in version 1.0.0: - SQLAlchemy now renders NULL or NOT NULL in all cases for TIMESTAMP columns, to accommodate `explicit_defaults_for_timestamp`. Prior to this version, it will not render "NOT NULL" for a TIMESTAMP column that is `nullable=False`.

MySQL Data Types

As with all SQLAlchemy dialects, all UPPERCASE types that are known to be valid with MySQL are importable from the top level dialect:

```
from sqlalchemy.dialects.mysql import \
        BIGINT, BINARY, BIT, BLOB, BOOLEAN, CHAR, DATE, \
        DATETIME, DECIMAL, DECIMAL, DOUBLE, ENUM, FLOAT, INTEGER, \
        LONGBLOB, LONGTEXT, MEDIUMBLOB, MEDIUMINT, MEDIUMTEXT, NCHAR, \
        NUMERIC, NVARCHAR, REAL, SET, SMALLINT, TEXT, TIME, TIMESTAMP, \
        TINYBLOB, TINYINT, TINYTEXT, VARBINARY, VARCHAR, YEAR
```

Types which are specific to MySQL, or have MySQL-specific construction arguments, are as follows:

class `sqlalchemy.dialects.mysql.`**`BIGINT`**(*display_width=None*, ***kw*)

Bases: `sqlalchemy.dialects.mysql.types._IntegerType`, *sqlalchemy.types.BIGINT*

MySQL BIGINTEGER type.

`__init__`(*display_width=None*, ***kw*)

Construct a BIGINTEGER.

Parameters

- **`display_width`** – Optional, maximum display width for this number.
- **`unsigned`** – a boolean, optional.
- **`zerofill`** – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class `sqlalchemy.dialects.mysql.`**`BINARY`**(*length=None*)

Bases: `sqlalchemy.types._Binary`

The SQL BINARY type.

class `sqlalchemy.dialects.mysql.`**`BIT`**(*length=None*)

Bases: *sqlalchemy.types.TypeEngine*

MySQL BIT type.

This type is for MySQL 5.0.3 or greater for MyISAM, and 5.0.5 or greater for MyISAM, MEMORY, InnoDB and BDB. For older versions, use a MSTinyInteger() type.

`__init__`(*length=None*)

Construct a BIT.

Parameters **`length`** – Optional, number of bits.

class `sqlalchemy.dialects.mysql.`**`BLOB`**(*length=None*)

Bases: *sqlalchemy.types.LargeBinary*

The SQL BLOB type.

`__init__`(*length=None*)

Construct a LargeBinary type.

Parameters **`length`** – optional, a length for the column for use in DDL statements, for those BLOB types that accept a length (i.e. MySQL). It does *not* produce a *lengthed* BINARY/VARBINARY type - use the BINARY/VARBINARY types specifically for those. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued.

class `sqlalchemy.dialects.mysql.`**`BOOLEAN`**(*create_constraint=True*, *name=None*, *_create_events=True*)

Bases: *sqlalchemy.types.Boolean*

The SQL BOOLEAN type.

`__init__`(*create_constraint=True*, *name=None*, *_create_events=True*)

Construct a Boolean.

Parameters

- **`create_constraint`** – defaults to True. If the boolean is generated as an int/smallint, also create a CHECK constraint on the table that ensures 1 or 0 as a value.

- **name** – if a CHECK constraint is generated, specify the name of the constraint.

class sqlalchemy.dialects.mysql.**CHAR**(*length=None*, ***kwargs*)

Bases: sqlalchemy.dialects.mysql.types._StringType, *sqlalchemy.types.CHAR*

MySQL CHAR type, for fixed-length character data.

__init__(*length=None*, ***kwargs*)

Construct a CHAR.

Parameters

- **length** – Maximum data length, in characters.
- **binary** – Optional, use the default binary collation for the national character set. This does not affect the type of data stored, use a BINARY type for binary data.
- **collation** – Optional, request a particular collation. Must be compatible with the national character set.

class sqlalchemy.dialects.mysql.**DATE**

Bases: *sqlalchemy.types.Date*

The SQL DATE type.

__init__

x.__init__(...) initializes x; see help(type(x)) for signature

class sqlalchemy.dialects.mysql.**DATETIME**(*timezone=False*, *fsp=None*)

Bases: *sqlalchemy.types.DATETIME*

MySQL DATETIME type.

__init__(*timezone=False*, *fsp=None*)

Construct a MySQL DATETIME type.

Parameters

- **timezone** – not used by the MySQL dialect.
- **fsp** – fractional seconds precision value. MySQL 5.6.4 supports storage of fractional seconds; this parameter will be used when emitting DDL for the DATETIME type.

Note: DBAPI driver support for fractional seconds may be limited; current support includes MySQL Connector/Python.

New in version 0.8.5: Added MySQL-specific *mysql.DATETIME* with fractional seconds support.

class sqlalchemy.dialects.mysql.**DECIMAL**(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)

Bases: sqlalchemy.dialects.mysql.types._NumericType, *sqlalchemy.types.DECIMAL*

MySQL DECIMAL type.

__init__(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)

Construct a DECIMAL.

Parameters

- **precision** – Total digits in this number. If scale and precision are both None, values are stored to limits allowed by the server.
- **scale** – The number of digits after the decimal point.
- **unsigned** – a boolean, optional.

- **zerofill** – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class `sqlalchemy.dialects.mysql.`**`DOUBLE`**(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)

Bases: `sqlalchemy.dialects.mysql.types._FloatType`

MySQL DOUBLE type.

`__init__`(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)

Construct a DOUBLE.

Note: The *DOUBLE* type by default converts from float to Decimal, using a truncation that defaults to 10 digits. Specify either `scale=n` or `decimal_return_scale=n` in order to change this scale, or `asdecimal=False` to return values directly as Python floating points.

Parameters

- **precision** – Total digits in this number. If scale and precision are both None, values are stored to limits allowed by the server.
- **scale** – The number of digits after the decimal point.
- **unsigned** – a boolean, optional.
- **zerofill** – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class `sqlalchemy.dialects.mysql.`**`ENUM`**(**enums*, ***kw*)

Bases: *`sqlalchemy.types.Enum`*, `sqlalchemy.dialects.mysql.enumerated._EnumeratedValues`

MySQL ENUM type.

`__init__`(**enums*, ***kw*)

Construct an ENUM.

E.g.:

```
Column('myenum', ENUM("foo", "bar", "baz"))
```

Parameters

- **enums** – The range of valid values for this ENUM. Values will be quoted when generating the schema according to the quoting flag (see below).
- **strict** – Defaults to False: ensure that a given value is in this ENUM's range of permissible values when inserting or updating rows. Note that MySQL will not raise a fatal error if you attempt to store an out of range value- an alternate value will be stored instead. (See MySQL ENUM documentation.)
- **charset** – Optional, a column-level character set for this string value. Takes precedence to 'ascii' or 'unicode' short-hand.
- **collation** – Optional, a column-level collation for this string value. Takes precedence to 'binary' short-hand.
- **ascii** – Defaults to False: short-hand for the `latin1` character set, generates ASCII in schema.

- **unicode** – Defaults to False: short-hand for the `ucs2` character set, generates UNICODE in schema.
- **binary** – Defaults to False: short-hand, pick the binary collation type that matches the column's character set. Generates BINARY in schema. This does not affect the type of data stored, only the collation of character data.
- **quoting** – Defaults to 'auto': automatically determine enum value quoting. If all enum values are surrounded by the same quoting character, then use 'quoted' mode. Otherwise, use 'unquoted' mode.

  'quoted': values in enums are already quoted, they will be used directly when generating the schema - this usage is deprecated.

  'unquoted': values in enums are not quoted, they will be escaped and surrounded by single quotes when generating the schema.

  Previous versions of this type always required manually quoted values to be supplied; future versions will always quote the string literals for you. This is a transitional option.

class `sqlalchemy.dialects.mysql.`**`FLOAT`**(*precision=None*, *scale=None*, *asdecimal=False*, ***kw*)

Bases: `sqlalchemy.dialects.mysql.types._FloatType`, *`sqlalchemy.types.FLOAT`*

MySQL FLOAT type.

`__init__`(*precision=None*, *scale=None*, *asdecimal=False*, ***kw*)

Construct a FLOAT.

Parameters

- **precision** – Total digits in this number. If scale and precision are both None, values are stored to limits allowed by the server.
- **scale** – The number of digits after the decimal point.
- **unsigned** – a boolean, optional.
- **zerofill** – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class `sqlalchemy.dialects.mysql.`**`INTEGER`**(*display_width=None*, ***kw*)

Bases: `sqlalchemy.dialects.mysql.types._IntegerType`, *`sqlalchemy.types.INTEGER`*

MySQL INTEGER type.

`__init__`(*display_width=None*, ***kw*)

Construct an INTEGER.

Parameters

- **display_width** – Optional, maximum display width for this number.
- **unsigned** – a boolean, optional.
- **zerofill** – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class `sqlalchemy.dialects.mysql.`**`JSON`**(*none_as_null=False*)

Bases: *`sqlalchemy.types.JSON`*

MySQL JSON type.

MySQL supports JSON as of version 5.7. Note that MariaDB does **not** support JSON at the time of this writing.

The *mysql.JSON* type supports persistence of JSON values as well as the core index operations provided by *types.JSON* datatype, by adapting the operations to render the `JSON_EXTRACT` function at the database level.

New in version 1.1.

class `sqlalchemy.dialects.mysql.`**`LONGBLOB`**(*length=None*)
Bases: `sqlalchemy.types._Binary`

MySQL LONGBLOB type, for binary data up to 2^32 bytes.

class `sqlalchemy.dialects.mysql.`**`LONGTEXT`**(***kwargs*)
Bases: `sqlalchemy.dialects.mysql.types._StringType`

MySQL LONGTEXT type, for text up to 2^32 characters.

`__init__`(***kwargs*)
Construct a LONGTEXT.

Parameters

- **`charset`** – Optional, a column-level character set for this string value. Takes precedence to 'ascii' or 'unicode' short-hand.
- **`collation`** – Optional, a column-level collation for this string value. Takes precedence to 'binary' short-hand.
- **`ascii`** – Defaults to False: short-hand for the `latin1` character set, generates ASCII in schema.
- **`unicode`** – Defaults to False: short-hand for the `ucs2` character set, generates UNICODE in schema.
- **`national`** – Optional. If true, use the server's configured national character set.
- **`binary`** – Defaults to False: short-hand, pick the binary collation type that matches the column's character set. Generates BINARY in schema. This does not affect the type of data stored, only the collation of character data.

class `sqlalchemy.dialects.mysql.`**`MEDIUMBLOB`**(*length=None*)
Bases: `sqlalchemy.types._Binary`

MySQL MEDIUMBLOB type, for binary data up to 2^24 bytes.

class `sqlalchemy.dialects.mysql.`**`MEDIUMINT`**(*display_width=None*, ***kw*)
Bases: `sqlalchemy.dialects.mysql.types._IntegerType`

MySQL MEDIUMINTEGER type.

`__init__`(*display_width=None*, ***kw*)
Construct a MEDIUMINTEGER

Parameters

- **`display_width`** – Optional, maximum display width for this number.
- **`unsigned`** – a boolean, optional.
- **`zerofill`** – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class `sqlalchemy.dialects.mysql.`**`MEDIUMTEXT`**(***kwargs*)

Bases: `sqlalchemy.dialects.mysql.types._StringType`

MySQL MEDIUMTEXT type, for text up to 2^24 characters.

`__init__`(***kwargs*)

Construct a MEDIUMTEXT.

Parameters

- **`charset`** – Optional, a column-level character set for this string value. Takes precedence to 'ascii' or 'unicode' short-hand.
- **`collation`** – Optional, a column-level collation for this string value. Takes precedence to 'binary' short-hand.
- **`ascii`** – Defaults to False: short-hand for the `latin1` character set, generates ASCII in schema.
- **`unicode`** – Defaults to False: short-hand for the `ucs2` character set, generates UNICODE in schema.
- **`national`** – Optional. If true, use the server's configured national character set.
- **`binary`** – Defaults to False: short-hand, pick the binary collation type that matches the column's character set. Generates BINARY in schema. This does not affect the type of data stored, only the collation of character data.

class `sqlalchemy.dialects.mysql.`**`NCHAR`**(*length=None*, ***kwargs*)

Bases: `sqlalchemy.dialects.mysql.types._StringType`, *`sqlalchemy.types.NCHAR`*

MySQL NCHAR type.

For fixed-length character data in the server's configured national character set.

`__init__`(*length=None*, ***kwargs*)

Construct an NCHAR.

Parameters

- **`length`** – Maximum data length, in characters.
- **`binary`** – Optional, use the default binary collation for the national character set. This does not affect the type of data stored, use a BINARY type for binary data.
- **`collation`** – Optional, request a particular collation. Must be compatible with the national character set.

class `sqlalchemy.dialects.mysql.`**`NUMERIC`**(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)

Bases: `sqlalchemy.dialects.mysql.types._NumericType`, *`sqlalchemy.types.NUMERIC`*

MySQL NUMERIC type.

`__init__`(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)

Construct a NUMERIC.

Parameters

- **`precision`** – Total digits in this number. If scale and precision are both None, values are stored to limits allowed by the server.
- **`scale`** – The number of digits after the decimal point.
- **`unsigned`** – a boolean, optional.

- **zerofill** – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class sqlalchemy.dialects.mysql.**NVARCHAR**(*length=None*, ***kwargs*)

Bases: sqlalchemy.dialects.mysql.types._StringType, *sqlalchemy.types.NVARCHAR*

MySQL NVARCHAR type.

For variable-length character data in the server's configured national character set.

__init__(*length=None*, ***kwargs*)

Construct an NVARCHAR.

Parameters

- **length** – Maximum data length, in characters.
- **binary** – Optional, use the default binary collation for the national character set. This does not affect the type of data stored, use a BINARY type for binary data.
- **collation** – Optional, request a particular collation. Must be compatible with the national character set.

class sqlalchemy.dialects.mysql.**REAL**(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)

Bases: sqlalchemy.dialects.mysql.types._FloatType, *sqlalchemy.types.REAL*

MySQL REAL type.

__init__(*precision=None*, *scale=None*, *asdecimal=True*, ***kw*)

Construct a REAL.

Note: The *REAL* type by default converts from float to Decimal, using a truncation that defaults to 10 digits. Specify either `scale=n` or `decimal_return_scale=n` in order to change this scale, or `asdecimal=False` to return values directly as Python floating points.

Parameters

- **precision** – Total digits in this number. If scale and precision are both None, values are stored to limits allowed by the server.
- **scale** – The number of digits after the decimal point.
- **unsigned** – a boolean, optional.
- **zerofill** – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class sqlalchemy.dialects.mysql.**SET**(**values*, ***kw*)

Bases: sqlalchemy.dialects.mysql.enumerated._EnumeratedValues

MySQL SET type.

__init__(**values*, ***kw*)

Construct a SET.

E.g.:

```
Column('myset', SET("foo", "bar", "baz"))
```

The list of potential values is required in the case that this set will be used to generate DDL for a table, or if the *SET.retrieve_as_bitwise* flag is set to True.

Parameters

- **values** – The range of valid values for this SET.
- **convert_unicode** – Same flag as that of *String.convert_unicode*.
- **collation** – same as that of *String.collation*
- **charset** – same as that of *VARCHAR.charset*.
- **ascii** – same as that of *VARCHAR.ascii*.
- **unicode** – same as that of *VARCHAR.unicode*.
- **binary** – same as that of *VARCHAR.binary*.
- **quoting** – Defaults to 'auto': automatically determine set value quoting. If all values are surrounded by the same quoting character, then use 'quoted' mode. Otherwise, use 'unquoted' mode.

  'quoted': values in enums are already quoted, they will be used directly when generating the schema - this usage is deprecated.

  'unquoted': values in enums are not quoted, they will be escaped and surrounded by single quotes when generating the schema.

  Previous versions of this type always required manually quoted values to be supplied; future versions will always quote the string literals for you. This is a transitional option.

  New in version 0.9.0.
- **retrieve_as_bitwise** – if True, the data for the set type will be persisted and selected using an integer value, where a set is coerced into a bitwise mask for persistence. MySQL allows this mode which has the advantage of being able to store values unambiguously, such as the blank string `''`. The datatype will appear as the expression `col + 0` in a SELECT statement, so that the value is coerced into an integer value in result sets. This flag is required if one wishes to persist a set that can store the blank string `''` as a value.

  > **Warning:** When using *mysql.SET.retrieve_as_bitwise*, it is essential that the list of set values is expressed in the **exact same order** as exists on the MySQL database.

  New in version 1.0.0.

class `sqlalchemy.dialects.mysql.`**`SMALLINT`**(*display_width=None*, ***kw*)

Bases: `sqlalchemy.dialects.mysql.types._IntegerType`, *sqlalchemy.types.SMALLINT*

MySQL SMALLINTEGER type.

`__init__`(*display_width=None*, ***kw*)

Construct a SMALLINTEGER.

Parameters

- **display_width** – Optional, maximum display width for this number.
- **unsigned** – a boolean, optional.
- **zerofill** – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class `sqlalchemy.dialects.mysql.`**`TEXT`**(*length=None*, ***kw*)

Bases: `sqlalchemy.dialects.mysql.types._StringType`, *`sqlalchemy.types.TEXT`*

MySQL TEXT type, for text up to 2^16 characters.

`__init__`(*length=None*, ***kw*)

Construct a TEXT.

Parameters

- **`length`** – Optional, if provided the server may optimize storage by substituting the smallest TEXT type sufficient to store `length` characters.
- **`charset`** – Optional, a column-level character set for this string value. Takes precedence to 'ascii' or 'unicode' short-hand.
- **`collation`** – Optional, a column-level collation for this string value. Takes precedence to 'binary' short-hand.
- **`ascii`** – Defaults to False: short-hand for the `latin1` character set, generates ASCII in schema.
- **`unicode`** – Defaults to False: short-hand for the `ucs2` character set, generates UNICODE in schema.
- **`national`** – Optional. If true, use the server's configured national character set.
- **`binary`** – Defaults to False: short-hand, pick the binary collation type that matches the column's character set. Generates BINARY in schema. This does not affect the type of data stored, only the collation of character data.

class `sqlalchemy.dialects.mysql.`**`TIME`**(*timezone=False*, *fsp=None*)

Bases: *`sqlalchemy.types.TIME`*

MySQL TIME type.

`__init__`(*timezone=False*, *fsp=None*)

Construct a MySQL TIME type.

Parameters

- **`timezone`** – not used by the MySQL dialect.
- **`fsp`** – fractional seconds precision value. MySQL 5.6 supports storage of fractional seconds; this parameter will be used when emitting DDL for the TIME type.

Note: DBAPI driver support for fractional seconds may be limited; current support includes MySQL Connector/Python.

New in version 0.8: The MySQL-specific TIME type as well as fractional seconds support.

class `sqlalchemy.dialects.mysql.`**`TIMESTAMP`**(*timezone=False*, *fsp=None*)

Bases: *`sqlalchemy.types.TIMESTAMP`*

MySQL TIMESTAMP type.

`__init__`(*timezone=False*, *fsp=None*)

Construct a MySQL TIMESTAMP type.

Parameters

- **`timezone`** – not used by the MySQL dialect.

- **fsp** – fractional seconds precision value. MySQL 5.6.4 supports storage of fractional seconds; this parameter will be used when emitting DDL for the TIMESTAMP type.

  Note: DBAPI driver support for fractional seconds may be limited; current support includes MySQL Connector/Python.

New in version 0.8.5: Added MySQL-specific `mysql.TIMESTAMP` with fractional seconds support.

class `sqlalchemy.dialects.mysql.`**`TINYBLOB`**(*length=None*)
Bases: `sqlalchemy.types._Binary`

MySQL TINYBLOB type, for binary data up to 2^8 bytes.

class `sqlalchemy.dialects.mysql.`**`TINYINT`**(*display_width=None*, ***kw*)
Bases: `sqlalchemy.dialects.mysql.types._IntegerType`

MySQL TINYINT type.

`__init__`(*display_width=None*, ***kw*)
Construct a TINYINT.

Parameters

- **display_width** – Optional, maximum display width for this number.
- **unsigned** – a boolean, optional.
- **zerofill** – Optional. If true, values will be stored as strings left-padded with zeros. Note that this does not effect the values returned by the underlying database API, which continue to be numeric.

class `sqlalchemy.dialects.mysql.`**`TINYTEXT`**(***kwargs*)
Bases: `sqlalchemy.dialects.mysql.types._StringType`

MySQL TINYTEXT type, for text up to 2^8 characters.

`__init__`(***kwargs*)
Construct a TINYTEXT.

Parameters

- **charset** – Optional, a column-level character set for this string value. Takes precedence to 'ascii' or 'unicode' short-hand.
- **collation** – Optional, a column-level collation for this string value. Takes precedence to 'binary' short-hand.
- **ascii** – Defaults to False: short-hand for the `latin1` character set, generates ASCII in schema.
- **unicode** – Defaults to False: short-hand for the `ucs2` character set, generates UNICODE in schema.
- **national** – Optional. If true, use the server's configured national character set.
- **binary** – Defaults to False: short-hand, pick the binary collation type that matches the column's character set. Generates BINARY in schema. This does not affect the type of data stored, only the collation of character data.

class `sqlalchemy.dialects.mysql.`**`VARBINARY`**(*length=None*)
Bases: `sqlalchemy.types._Binary`

The SQL VARBINARY type.

class `sqlalchemy.dialects.mysql.`**`VARCHAR`**(*length=None*, ***kwargs*)

Bases: `sqlalchemy.dialects.mysql.types._StringType`, *`sqlalchemy.types.VARCHAR`*

MySQL VARCHAR type, for variable-length character data.

`__init__`(*length=None*, ***kwargs*)

Construct a VARCHAR.

Parameters

- **`charset`** – Optional, a column-level character set for this string value. Takes precedence to 'ascii' or 'unicode' short-hand.
- **`collation`** – Optional, a column-level collation for this string value. Takes precedence to 'binary' short-hand.
- **`ascii`** – Defaults to False: short-hand for the `latin1` character set, generates ASCII in schema.
- **`unicode`** – Defaults to False: short-hand for the `ucs2` character set, generates UNICODE in schema.
- **`national`** – Optional. If true, use the server's configured national character set.
- **`binary`** – Defaults to False: short-hand, pick the binary collation type that matches the column's character set. Generates BINARY in schema. This does not affect the type of data stored, only the collation of character data.

class `sqlalchemy.dialects.mysql.`**`YEAR`**(*display_width=None*)

Bases: *`sqlalchemy.types.TypeEngine`*

MySQL YEAR type, for single byte storage of years 1901-2155.

MySQL-Python

Support for the MySQL database via the MySQL-Python driver.

DBAPI

Documentation and download information (if applicable) for MySQL-Python is available at: http://sourceforge.net/projects/mysql-python

Connecting

Connect String:

```
mysql+mysqldb://<user>:<password>@<host>[:<port>]/<dbname>
```

Unicode

Please see *Unicode* for current recommendations on unicode handling.

Py3K Support

Currently, MySQLdb only runs on Python 2 and development has been stopped. mysqlclient is fork of MySQLdb and provides Python 3 support as well as some bugfixes.

Using MySQLdb with Google Cloud SQL

Google Cloud SQL now recommends use of the MySQLdb dialect. Connect using a URL like the following:

```
mysql+mysqldb://root@/<dbname>?unix_socket=/cloudsql/<projectid>:<instancename>
```

pymysql

Support for the MySQL database via the PyMySQL driver.

DBAPI

Documentation and download information (if applicable) for PyMySQL is available at: http://www.pymysql.org/

Connecting

Connect String:

```
mysql+pymysql://<username>:<password>@<host>/<dbname>[?<options>]
```

Unicode

Please see *Unicode* for current recommendations on unicode handling.

MySQL-Python Compatibility

The pymysql DBAPI is a pure Python port of the MySQL-python (MySQLdb) driver, and targets 100% compatibility. Most behavioral notes for MySQL-python apply to the pymysql driver as well.

MySQL-Connector

Support for the MySQL database via the MySQL Connector/Python driver.

DBAPI

Documentation and download information (if applicable) for MySQL Connector/Python is available at: http://dev.mysql.com/downloads/connector/python/

Connecting

Connect String:

```
mysql+mysqlconnector://<user>:<password>@<host>[:<port>]/<dbname>
```

Unicode

Please see *Unicode* for current recommendations on unicode handling.

cymysql

Support for the MySQL database via the CyMySQL driver.

DBAPI

Documentation and download information (if applicable) for CyMySQL is available at: https://github.com/nakagami/CyMySQL

Connecting

Connect String:

```
mysql+cymysql://<username>:<password>@<host>/<dbname>[?<options>]
```

OurSQL

Support for the MySQL database via the OurSQL driver.

DBAPI

Documentation and download information (if applicable) for OurSQL is available at: http://packages.python.org/oursql/

Connecting

Connect String:

```
mysql+oursql://<user>:<password>@<host>[:<port>]/<dbname>
```

Unicode

Please see *Unicode* for current recommendations on unicode handling.

Google App Engine

Support for the MySQL database via the Google Cloud SQL driver. This dialect is based primarily on the *mysql.mysqldb* dialect with minimal changes.

New in version 0.7.8.

Deprecated since version 1.0: This dialect is **no longer necessary** for Google Cloud SQL; the MySQLdb dialect can be used directly. Cloud SQL now recommends creating connections via the mysql dialect using the URL format

```
mysql+mysqldb://root@/<dbname>?unix_socket=/cloudsql/<projectid>:<instancename>
```

DBAPI

Documentation and download information (if applicable) for Google Cloud SQL is available at: https://developers.google.com/appengine/docs/python/cloud-sql/developers-guide

Connecting

Connect String:

```
mysql+gaerdbms:///<dbname>?instance=<instancename>
```

Pooling

Google App Engine connections appear to be randomly recycled, so the dialect does not pool connections. The *NullPool* implementation is installed within the *Engine* by default.

pyodbc

Support for the MySQL database via the PyODBC driver.

Note: The PyODBC for MySQL dialect is not well supported, and is subject to unresolved character encoding issues which exist within the current ODBC drivers available. (see http://code.google.com/p/pyodbc/issues/detail?id=25). Other dialects for MySQL are recommended.

DBAPI

Documentation and download information (if applicable) for PyODBC is available at: http://pypi.python.org/pypi/pyodbc/

Connecting

Connect String:

```
mysql+pyodbc://<username>:<password>@<dsnname>
```

zxjdbc

Support for the MySQL database via the zxjdbc for Jython driver.

Note: Jython is not supported by current versions of SQLAlchemy. The zxjdbc dialect should be considered as experimental.

DBAPI

Drivers for this database are available at: http://dev.mysql.com/downloads/connector/j/

Connecting

Connect String:

```
mysql+zxjdbc://<user>:<password>@<hostname>[:<port>]/<database>
```

Character Sets

SQLAlchemy zxjdbc dialects pass unicode straight through to the zxjdbc/JDBC layer. To allow multiple character sets to be sent from the MySQL Connector/J JDBC driver, by default SQLAlchemy sets its `characterEncoding` connection property to `UTF-8`. It may be overridden via a `create_engine` URL parameter.

4.1.4 Oracle

Support for the Oracle database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- *cx-Oracle*
- *zxJDBC for Jython*

Connect Arguments

The dialect supports several *`create_engine()`* arguments which affect the behavior of the dialect regardless of driver in use.

- `use_ansi` - Use ANSI JOIN constructs (see the section on Oracle 8). Defaults to `True`. If `False`, Oracle-8 compatible constructs are used for joins.
- `optimize_limits` - defaults to `False`. see the section on LIMIT/OFFSET.
- `use_binds_for_limits` - defaults to `True`. see the section on LIMIT/OFFSET.

Auto Increment Behavior

SQLAlchemy Table objects which include integer primary keys are usually assumed to have "autoincrementing" behavior, meaning they can generate their own primary key values upon INSERT. Since Oracle has no "autoincrement" feature, SQLAlchemy relies upon sequences to produce these values. With the Oracle dialect, *a sequence must always be explicitly specified to enable autoincrement*. This is divergent with the majority of documentation examples which assume the usage of an autoincrement-capable database. To specify sequences, use the sqlalchemy.schema.Sequence object which is passed to a Column construct:

```
t = Table('mytable', metadata,
      Column('id', Integer, Sequence('id_seq'), primary_key=True),
      Column(...), ...
)
```

This step is also required when using table reflection, i.e. autoload=True:

```
t = Table('mytable', metadata,
      Column('id', Integer, Sequence('id_seq'), primary_key=True),
      autoload=True
)
```

Identifier Casing

In Oracle, the data dictionary represents all case insensitive identifier names using UPPERCASE text. SQLAlchemy on the other hand considers an all-lower case identifier name to be case insensitive. The Oracle dialect converts all case insensitive identifiers to and from those two formats during schema level communication, such as reflection of tables and indexes. Using an UPPERCASE name on the SQLAlchemy side indicates a case sensitive identifier, and SQLAlchemy will quote the name - this will cause mismatches against data dictionary data received from Oracle, so unless identifier names have been truly created as case sensitive (i.e. using quoted names), all lowercase names should be used on the SQLAlchemy side.

LIMIT/OFFSET Support

Oracle has no support for the LIMIT or OFFSET keywords. SQLAlchemy uses a wrapped subquery approach in conjunction with ROWNUM. The exact methodology is taken from http://www.oracle.com/technology/oramag/oracle/06-sep/o56asktom.html .

There are two options which affect its behavior:

- the "FIRST ROWS()" optimization keyword is not used by default. To enable the usage of this optimization directive, specify `optimize_limits=True` to *`create_engine()`*.
- the values passed for the limit/offset are sent as bound parameters. Some users have observed that Oracle produces a poor query plan when the values are sent as binds and not rendered literally. To render the limit/offset values literally within the SQL statement, specify `use_binds_for_limits=False` to *`create_engine()`*.

Some users have reported better performance when the entirely different approach of a window query is used, i.e. ROW_NUMBER() OVER (ORDER BY), to provide LIMIT/OFFSET (note that the majority of users don't observe this). To suit this case the method used for LIMIT/OFFSET can be replaced entirely. See the recipe at http://www.sqlalchemy.org/trac/wiki/UsageRecipes/WindowFunctionsByDefault which installs a select compiler that overrides the generation of limit/offset with a window function.

RETURNING Support

The Oracle database supports a limited form of RETURNING, in order to retrieve result sets of matched rows from INSERT, UPDATE and DELETE statements. Oracle's RETURNING..INTO syntax only supports one row being returned, as it relies upon OUT parameters in order to function. In addition, supported DBAPIs have further limitations (see *RETURNING Support*).

SQLAlchemy's "implicit returning" feature, which employs RETURNING within an INSERT and sometimes an UPDATE statement in order to fetch newly generated primary key values and other SQL defaults and expressions, is normally enabled on the Oracle backend. By default, "implicit returning" typically only fetches the value of a single `nextval(some_seq)` expression embedded into an INSERT in order to increment a sequence within an INSERT statement and get the value back at the same time. To disable this feature across the board, specify `implicit_returning=False` to *`create_engine()`*:

```
engine = create_engine("oracle://scott:tiger@dsn",
                       implicit_returning=False)
```

Implicit returning can also be disabled on a table-by-table basis as a table option:

```
# Core Table
my_table = Table("my_table", metadata, ..., implicit_returning=False)


# declarative
class MyClass(Base):
    __tablename__ = 'my_table'
    __table_args__ = {"implicit_returning": False}
```

See also:

RETURNING Support - additional cx_oracle-specific restrictions on implicit returning.

ON UPDATE CASCADE

Oracle doesn't have native ON UPDATE CASCADE functionality. A trigger based solution is available at http://asktom.oracle.com/tkyte/update_cascade/index.html .

When using the SQLAlchemy ORM, the ORM has limited ability to manually issue cascading updates - specify ForeignKey objects using the "deferrable=True, initially='deferred'" keyword arguments, and specify "passive_updates=False" on each relationship().

Oracle 8 Compatibility

When Oracle 8 is detected, the dialect internally configures itself to the following behaviors:

- the use_ansi flag is set to False. This has the effect of converting all JOIN phrases into the WHERE clause, and in the case of LEFT OUTER JOIN makes use of Oracle's (+) operator.
- the NVARCHAR2 and NCLOB datatypes are no longer generated as DDL when the *`Unicode`* is used - VARCHAR2 and CLOB are issued instead. This because these types don't seem to work correctly on Oracle 8 even though they are available. The *`NVARCHAR`* and *`NCLOB`* types will always generate NVARCHAR2 and NCLOB.
- the "native unicode" mode is disabled when using cx_oracle, i.e. SQLAlchemy encodes all Python unicode objects to "string" before passing in as bind parameters.

Synonym/DBLINK Reflection

When using reflection with Table objects, the dialect can optionally search for tables indicated by synonyms, either in local or remote schemas or accessed over DBLINK, by passing the flag `oracle_resolve_synonyms=True` as a keyword argument to the *Table* construct:

```
some_table = Table('some_table', autoload=True,
                           autoload_with=some_engine,
                           oracle_resolve_synonyms=True)
```

When this flag is set, the given name (such as `some_table` above) will be searched not just in the `ALL_TABLES` view, but also within the `ALL_SYNONYMS` view to see if this name is actually a synonym to another name. If the synonym is located and refers to a DBLINK, the oracle dialect knows how to locate the table's information using DBLINK syntax(e.g. `@dblink`).

`oracle_resolve_synonyms` is accepted wherever reflection arguments are accepted, including methods such as *MetaData.reflect()* and *Inspector.get_columns()*.

If synonyms are not in use, this flag should be left disabled.

DateTime Compatibility

Oracle has no datatype known as `DATETIME`, it instead has only `DATE`, which can actually store a date and time value. For this reason, the Oracle dialect provides a type *oracle.DATE* which is a subclass of *DateTime*. This type has no special behavior, and is only present as a "marker" for this type; additionally, when a database column is reflected and the type is reported as `DATE`, the time-supporting *oracle.DATE* type is used.

Changed in version 0.9.4: Added *oracle.DATE* to subclass *DateTime*. This is a change as previous versions would reflect a `DATE` column as *types.DATE*, which subclasses *Date*. The only significance here is for schemes that are examining the type of column for use in special Python translations or for migrating schemas to other database backends.

Oracle Table Options

The CREATE TABLE phrase supports the following options with Oracle in conjunction with the *Table* construct:

- `ON COMMIT`:

```
Table(
    "some_table", metadata, ...,
    prefixes=['GLOBAL TEMPORARY'], oracle_on_commit='PRESERVE ROWS')
```

New in version 1.0.0.

- `COMPRESS`:

```
 Table('mytable', metadata, Column('data', String(32)),
     oracle_compress=True)

 Table('mytable', metadata, Column('data', String(32)),
     oracle_compress=6)

The ``oracle_compress`` parameter accepts either an integer compression
level, or ``True`` to use the default compression level.
```

New in version 1.0.0.

Oracle Specific Index Options

Bitmap Indexes

You can specify the `oracle_bitmap` parameter to create a bitmap index instead of a B-tree index:

```
Index('my_index', my_table.c.data, oracle_bitmap=True)
```

Bitmap indexes cannot be unique and cannot be compressed. SQLAlchemy will not check for such limitations, only the database will.

New in version 1.0.0.

Index compression

Oracle has a more efficient storage mode for indexes containing lots of repeated values. Use the `oracle_compress` parameter to turn on key c ompression:

```
Index('my_index', my_table.c.data, oracle_compress=True)

Index('my_index', my_table.c.data1, my_table.c.data2, unique=True,
       oracle_compress=1)
```

The `oracle_compress` parameter accepts either an integer specifying the number of prefix columns to compress, or `True` to use the default (all columns for non-unique indexes, all but the last column for unique indexes).

New in version 1.0.0.

Oracle Data Types

As with all SQLAlchemy dialects, all UPPERCASE types that are known to be valid with Oracle are importable from the top level dialect, whether they originate from *sqlalchemy.types* or from the local dialect:

```
from sqlalchemy.dialects.oracle import \
            BFILE, BLOB, CHAR, CLOB, DATE, \
            DOUBLE_PRECISION, FLOAT, INTERVAL, LONG, NCLOB, \
            NUMBER, NVARCHAR, NVARCHAR2, RAW, TIMESTAMP, VARCHAR, \
            VARCHAR2
```

Types which are specific to Oracle, or have Oracle-specific construction arguments, are as follows:

class `sqlalchemy.dialects.oracle.`**`BFILE`**(*length=None*)

Bases: *sqlalchemy.types.LargeBinary*

`__init__`(*length=None*)

Construct a LargeBinary type.

Parameters **`length`** – optional, a length for the column for use in DDL statements, for those BLOB types that accept a length (i.e. MySQL). It does *not* produce a *lengthed* BINARY/VARBINARY type - use the BINARY/VARBINARY types specifically for those. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued.

class sqlalchemy.dialects.oracle.**DATE**(*timezone=False*)

Bases: sqlalchemy.types.DateTime

Provide the oracle DATE type.

This type has no special Python behavior, except that it subclasses types.DateTime; this is to suit the fact that the Oracle DATE type supports a time value.

New in version 0.9.4.

__init__(*timezone=False*)

Construct a new DateTime.

Parameters timezone – boolean. If True, and supported by the backend, will produce 'TIMESTAMP WITH TIMEZONE'. For backends that don't support timezone aware timestamps, has no effect.

class sqlalchemy.dialects.oracle.**DOUBLE_PRECISION**(*precision=None*, *scale=None*, *asdecimal=None*)

Bases: sqlalchemy.types.Numeric

class sqlalchemy.dialects.oracle.**INTERVAL**(*day_precision=None*, *second_precision=None*)

Bases: sqlalchemy.types.TypeEngine

__init__(*day_precision=None*, *second_precision=None*)

Construct an INTERVAL.

Note that only DAY TO SECOND intervals are currently supported. This is due to a lack of support for YEAR TO MONTH intervals within available DBAPIs (cx_oracle and zxjdbc).

Parameters

- **day_precision** – the day precision value. this is the number of digits to store for the day field. Defaults to "2"
- **second_precision** – the second precision value. this is the number of digits to store for the fractional seconds field. Defaults to "6".

class sqlalchemy.dialects.oracle.**NCLOB**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

Bases: sqlalchemy.types.Text

__init__(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

Create a string-holding type.

Parameters

- **length** – optional, a length for the column for use in DDL and CAST expressions. May be safely omitted if no CREATE TABLE will be issued. Certain databases may require a length for use in DDL, and will raise an exception when the CREATE TABLE DDL is issued if a VARCHAR with no length is included. Whether the value is interpreted as bytes or characters is database specific.
- **collation** – Optional, a column-level collation for use in DDL and CAST expressions. Renders using the COLLATE keyword supported by SQLite, MySQL, and Postgresql. E.g.:

```
>>> from sqlalchemy import cast, select, String
>>> print select([cast('some string', String(collation='utf8'))])
SELECT CAST(:param_1 AS VARCHAR COLLATE utf8) AS anon_1
```

New in version 0.8: Added support for COLLATE to all string types.

- **convert_unicode** – When set to `True`, the *String* type will assume that input is to be passed as Python `unicode` objects, and results returned as Python `unicode` objects. If the DBAPI in use does not support Python unicode (which is fewer and fewer these days), SQLAlchemy will encode/decode the value, using the value of the `encoding` parameter passed to *create_engine()* as the encoding.

  When using a DBAPI that natively supports Python unicode objects, this flag generally does not need to be set. For columns that are explicitly intended to store non-ASCII data, the *Unicode* or *UnicodeText* types should be used regardless, which feature the same behavior of `convert_unicode` but also indicate an underlying column type that directly supports unicode, such as `NVARCHAR`.

  For the extremely rare case that Python `unicode` is to be encoded/decoded by SQLAlchemy on a backend that does natively support Python `unicode`, the value `force` can be passed here which will cause SQLAlchemy's encode/decode services to be used unconditionally.

- **unicode_error** – Optional, a method to use to handle Unicode conversion errors. Behaves like the `errors` keyword argument to the standard library's `string.decode()` functions. This flag requires that `convert_unicode` is set to `force` - otherwise, SQLAlchemy is not guaranteed to handle the task of unicode conversion. Note that this flag adds significant performance overhead to row-fetching operations for backends that already return unicode objects natively (which most DBAPIs do). This flag should only be used as a last resort for reading strings from a column with varied or corrupted encodings.

class `sqlalchemy.dialects.oracle.`**NUMBER**(*precision=None*, *scale=None*, *asdecimal=None*)

Bases: *sqlalchemy.types.Numeric*, *sqlalchemy.types.Integer*

class `sqlalchemy.dialects.oracle.`**LONG**(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

Bases: *sqlalchemy.types.Text*

__init__(*length=None*, *collation=None*, *convert_unicode=False*, *unicode_error=None*, *_warn_on_bytestring=False*)

Create a string-holding type.

Parameters

- **length** – optional, a length for the column for use in DDL and CAST expressions. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a `length` for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued if a `VARCHAR` with no length is included. Whether the value is interpreted as bytes or characters is database specific.

- **collation** – Optional, a column-level collation for use in DDL and CAST expressions. Renders using the COLLATE keyword supported by SQLite, MySQL, and Postgresql. E.g.:

```
>>> from sqlalchemy import cast, select, String
>>> print select([cast('some string', String(collation='utf8'))])
SELECT CAST(:param_1 AS VARCHAR COLLATE utf8) AS anon_1
```

  New in version 0.8: Added support for COLLATE to all string types.

- **convert_unicode** – When set to `True`, the *String* type will assume that input is to be passed as Python `unicode` objects, and results returned as Python `unicode` objects. If the DBAPI in use does not support Python unicode (which

is fewer and fewer these days), SQLAlchemy will encode/decode the value, using the value of the `encoding` parameter passed to *create_engine()* as the encoding.

When using a DBAPI that natively supports Python unicode objects, this flag generally does not need to be set. For columns that are explicitly intended to store non-ASCII data, the *Unicode* or *UnicodeText* types should be used regardless, which feature the same behavior of `convert_unicode` but also indicate an underlying column type that directly supports unicode, such as `NVARCHAR`.

For the extremely rare case that Python `unicode` is to be encoded/decoded by SQLAlchemy on a backend that does natively support Python `unicode`, the value `force` can be passed here which will cause SQLAlchemy's encode/decode services to be used unconditionally.

- **`unicode_error`** – Optional, a method to use to handle Unicode conversion errors. Behaves like the `errors` keyword argument to the standard library's `string.decode()` functions. This flag requires that `convert_unicode` is set to `force` - otherwise, SQLAlchemy is not guaranteed to handle the task of unicode conversion. Note that this flag adds significant performance overhead to row-fetching operations for backends that already return unicode objects natively (which most DBAPIs do). This flag should only be used as a last resort for reading strings from a column with varied or corrupted encodings.

class `sqlalchemy.dialects.oracle.`**`RAW`**(*length=None*)

Bases: `sqlalchemy.types._Binary`

cx_Oracle

Support for the Oracle database via the cx-Oracle driver.

DBAPI

Documentation and download information (if applicable) for cx-Oracle is available at: http://cx-oracle.sourceforge.net/

Connecting

Connect String:

```
oracle+cx_oracle://user:pass@host:port/dbname[?key=value&key=value...]
```

Additional Connect Arguments

When connecting with `dbname` present, the host, port, and dbname tokens are converted to a TNS name using the cx_oracle `makedsn()` function. Otherwise, the host token is taken directly as a TNS name.

Additional arguments which may be specified either as query string arguments on the URL, or as keyword arguments to *create_engine()* are:

- `allow_twophase` - enable two-phase transactions. Defaults to `True`.
- `arraysize` - set the cx_oracle.arraysize value on cursors, defaulted to 50. This setting is significant with cx_Oracle as the contents of LOB objects are only readable within a "live" row (e.g. within a batch of 50 rows).

- `auto_convert_lobs` - defaults to True; See *LOB Objects*.
- `auto_setinputsizes` - the cx_oracle.setinputsizes() call is issued for all bind parameters. This is required for LOB datatypes but can be disabled to reduce overhead. Defaults to `True`. Specific types can be excluded from this process using the `exclude_setinputsizes` parameter.
- `coerce_to_unicode` - see *Unicode* for detail.
- `coerce_to_decimal` - see *Precision Numerics* for detail.
- `exclude_setinputsizes` - a tuple or list of string DBAPI type names to be excluded from the "auto setinputsizes" feature. The type names here must match DBAPI types that are found in the "cx_Oracle" module namespace, such as cx_Oracle.UNICODE, cx_Oracle.NCLOB, etc. Defaults to `(STRING, UNICODE)`.

  New in version 0.8: specific DBAPI types can be excluded from the auto_setinputsizes feature via the exclude_setinputsizes attribute.
- `mode` - This is given the string value of SYSDBA or SYSOPER, or alternatively an integer value. This value is only available as a URL query string argument.
- `threaded` - enable multithreaded access to cx_oracle connections. Defaults to `True`. Note that this is the opposite default of the cx_Oracle DBAPI itself.
- `service_name` - An option to use connection string (DSN) with `SERVICE_NAME` instead of `SID`. It can't be passed when a `database` part is given. E.g. `oracle+cx_oracle://scott:tiger@host:1521/?service_name=hr` is a valid url. This value is only available as a URL query string argument.

  New in version 1.0.0.

Unicode

The cx_Oracle DBAPI as of version 5 fully supports unicode, and has the ability to return string results as Python unicode objects natively.

When used in Python 3, cx_Oracle returns all strings as Python unicode objects (that is, plain `str` in Python 3). In Python 2, it will return as Python unicode those column values that are of type `NVARCHAR` or `NCLOB`. For column values that are of type `VARCHAR` or other non-unicode string types, it will return values as Python strings (e.g. bytestrings).

The cx_Oracle SQLAlchemy dialect presents two different options for the use case of returning `VARCHAR` column values as Python unicode objects under Python 2:

- the cx_Oracle DBAPI has the ability to coerce all string results to Python unicode objects unconditionally using output type handlers. This has the advantage that the unicode conversion is global to all statements at the cx_Oracle driver level, meaning it works with raw textual SQL statements that have no typing information associated. However, this system has been observed to incur signfiicant performance overhead, not only because it takes effect for all string values unconditionally, but also because cx_Oracle under Python 2 seems to use a pure-Python function call in order to do the decode operation, which under cPython can orders of magnitude slower than doing it using C functions alone.
- SQLAlchemy has unicode-decoding services built in, and when using SQLAlchemy's C extensions, these functions do not use any Python function calls and are very fast. The disadvantage to this approach is that the unicode conversion only takes effect for statements where the *`Unicode`* type or *`String`* type with `convert_unicode=True` is explicitly associated with the result column. This is the case for any ORM or Core query or SQL expression as well as for a *`text()`* construct that specifies output column types, so in the vast majority of cases this is not an issue. However, when sending a completely raw string to *`Connection.execute()`*, this typing information isn't present, unless the string is handled within a *`text()`* construct that adds typing information.

As of version 0.9.2 of SQLAlchemy, the default approach is to use SQLAlchemy's typing system. This keeps cx_Oracle's expensive Python 2 approach disabled unless the user explicitly wants it. Under Python 3, SQLAlchemy detects that cx_Oracle is returning unicode objects natively and cx_Oracle's system is used.

To re-enable cx_Oracle's output type handler under Python 2, the `coerce_to_unicode=True` flag (new in 0.9.4) can be passed to *create_engine()*:

```
engine = create_engine("oracle+cx_oracle://dsn", coerce_to_unicode=True)
```

Alternatively, to run a pure string SQL statement and get `VARCHAR` results as Python unicode under Python 2 without using cx_Oracle's native handlers, the *text()* feature can be used:

```
from sqlalchemy import text, Unicode
result = conn.execute(
    text("select username from user").columns(username=Unicode))
```

Changed in version 0.9.2: cx_Oracle's outputtypehandlers are no longer used for unicode results of non-unicode datatypes in Python 2, after they were identified as a major performance bottleneck. SQLAlchemy's own unicode facilities are used instead.

New in version 0.9.4: Added the `coerce_to_unicode` flag, to re-enable cx_Oracle's outputtypehandler and revert to pre-0.9.2 behavior.

RETURNING Support

The cx_oracle DBAPI supports a limited subset of Oracle's already limited RETURNING support. Typically, results can only be guaranteed for at most one column being returned; this is the typical case when SQLAlchemy uses RETURNING to get just the value of a primary-key-associated sequence value. Additional column expressions will cause problems in a non-determinative way, due to cx_oracle's lack of support for the OCI_DATA_AT_EXEC API which is required for more complex RETURNING scenarios.

For this reason, stability may be enhanced by disabling RETURNING support completely; SQLAlchemy otherwise will use RETURNING to fetch newly sequence-generated primary keys. As illustrated in *RETURNING Support*:

```
engine = create_engine("oracle://scott:tiger@dsn",
                       implicit_returning=False)
```

See also:

http://docs.oracle.com/cd/B10501_01/appdev.920/a96584/oci05bnd.htm#420693 - OCI documentation for RETURNING

http://sourceforge.net/mailarchive/message.php?msg_id=31338136 - cx_oracle developer commentary

LOB Objects

cx_oracle returns oracle LOBs using the cx_oracle.LOB object. SQLAlchemy converts these to strings so that the interface of the Binary type is consistent with that of other backends, and so that the linkage to a live cursor is not needed in scenarios like result.fetchmany() and result.fetchall(). This means that by default, LOB objects are fully fetched unconditionally by SQLAlchemy, and the linkage to a live cursor is broken.

To disable this processing, pass `auto_convert_lobs=False` to *create_engine()*.

Two Phase Transaction Support

Two Phase transactions are implemented using XA transactions, and are known to work in a rudimental fashion with recent versions of cx_Oracle as of SQLAlchemy 0.8.0b2, 0.7.10. However, the mechanism is not yet considered to be robust and should still be regarded as experimental.

In particular, the cx_Oracle DBAPI as recently as 5.1.2 has a bug regarding two phase which prevents a particular DBAPI connection from being consistently usable in both prepared transactions as well as traditional DBAPI usage patterns; therefore once a particular connection is used via `Connection.begin_prepared()`, all subsequent usages of the underlying DBAPI connection must be within the context of prepared transactions.

The default behavior of *Engine* is to maintain a pool of DBAPI connections. Therefore, due to the above glitch, a DBAPI connection that has been used in a two-phase operation, and is then returned to the pool, will not be usable in a non-two-phase context. To avoid this situation, the application can make one of several choices:

- Disable connection pooling using *NullPool*
- Ensure that the particular *Engine* in use is only used for two-phase operations. A *Engine* bound to an ORM *Session* which includes `twophase=True` will consistently use the two-phase transaction style.
- For ad-hoc two-phase operations without disabling pooling, the DBAPI connection in use can be evicted from the connection pool using the *Connection.detach()* method.

Changed in version 0.8.0b2,0.7.10: Support for cx_oracle prepared transactions has been implemented and tested.

Precision Numerics

The SQLAlchemy dialect goes through a lot of steps to ensure that decimal numbers are sent and received with full accuracy. An "outputtypehandler" callable is associated with each cx_oracle connection object which detects numeric types and receives them as string values, instead of receiving a Python `float` directly, which is then passed to the Python `Decimal` constructor. The *Numeric* and *Float* types under the cx_oracle dialect are aware of this behavior, and will coerce the `Decimal` to `float` if the `asdecimal` flag is `False` (default on *Float*, optional on *Numeric*).

Because the handler coerces to `Decimal` in all cases first, the feature can detract significantly from performance. If precision numerics aren't required, the decimal handling can be disabled by passing the flag `coerce_to_decimal=False` to *create_engine()*:

```
engine = create_engine("oracle+cx_oracle://dsn", coerce_to_decimal=False)
```

New in version 0.7.6: Add the `coerce_to_decimal` flag.

Another alternative to performance is to use the cdecimal library; see *Numeric* for additional notes.

The handler attempts to use the "precision" and "scale" attributes of the result set column to best determine if subsequent incoming values should be received as `Decimal` as opposed to int (in which case no processing is added). There are several scenarios where OCI does not provide unambiguous data as to the numeric type, including some situations where individual rows may return a combination of floating point and integer values. Certain values for "precision" and "scale" have been observed to determine this scenario. When it occurs, the outputtypehandler receives as string and then passes off to a processing function which detects, for each returned value, if a decimal point is present, and if so converts to `Decimal`, otherwise to int. The intention is that simple int-based statements like "SELECT my_seq.nextval() FROM DUAL" continue to return ints and not `Decimal` objects, and that any kind of floating point value is received as a string so that there is no floating point loss of precision.

The "decimal point is present" logic itself is also sensitive to locale. Under OCI, this is controlled by the NLS_LANG environment variable. Upon first connection, the dialect runs a test to determine the current "decimal" character, which can be a comma "," for European locales. From that point forward the outputtypehandler uses that character

to represent a decimal point. Note that cx_oracle 5.0.3 or greater is required when dealing with numerics with locale settings that don't use a period "." as the decimal character.

Changed in version 0.6.6: The outputtypehandler supports the case where the locale uses a comma "," character to represent a decimal point.

zxjdbc

Support for the Oracle database via the zxJDBC for Jython driver.

Note: Jython is not supported by current versions of SQLAlchemy. The zxjdbc dialect should be considered as experimental.

DBAPI

Drivers for this database are available at: http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html.

Connecting

Connect String:

```
oracle+zxjdbc://user:pass@host/dbname
```

4.1.5 PostgreSQL

Support for the PostgreSQL database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- *psycopg2*
- *pg8000*
- *psycopg2cffi*
- *py-postgresql*
- *zxJDBC for Jython*

Sequences/SERIAL

PostgreSQL supports sequences, and SQLAlchemy uses these as the default means of creating new primary key values for integer-based primary key columns. When creating tables, SQLAlchemy will issue the `SERIAL` datatype for integer-based primary key columns, which generates a sequence and server side default corresponding to the column.

To specify a specific named sequence to be used for primary key generation, use the `Sequence()` construct:

```
Table('sometable', metadata,
        Column('id', Integer, Sequence('some_id_seq'), primary_key=True)
    )
```

When SQLAlchemy issues a single INSERT statement, to fulfill the contract of having the "last insert identifier" available, a RETURNING clause is added to the INSERT statement which specifies the primary key columns should be returned after the statement completes. The RETURNING functionality only takes place if Postgresql 8.2 or later is in use. As a fallback approach, the sequence, whether specified explicitly or implicitly via `SERIAL`, is executed independently beforehand, the returned value to be used in the subsequent insert. Note that when an *insert()* construct is executed using "executemany" semantics, the "last inserted identifier" functionality does not apply; no RETURNING clause is emitted nor is the sequence pre-executed in this case.

To force the usage of RETURNING by default off, specify the flag `implicit_returning=False` to *create_engine()*.

Transaction Isolation Level

All Postgresql dialects support setting of transaction isolation level both via a dialect-specific parameter *create_engine.isolation_level* accepted by *create_engine()*, as well as the `isolation_level` argument as passed to *Connection.execution_options()*. When using a non-psycopg2 dialect, this feature works by issuing the command `SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL <level>` for each new connection.

To set isolation level using *create_engine()*:

```
engine = create_engine(
    "postgresql+pg8000://scott:tiger@localhost/test",
    isolation_level="READ UNCOMMITTED"
)
```

To set using per-connection execution options:

```
connection = engine.connect()
connection = connection.execution_options(
    isolation_level="READ COMMITTED"
)
```

Valid values for `isolation_level` include:

- `READ COMMITTED`
- `READ UNCOMMITTED`
- `REPEATABLE READ`
- `SERIALIZABLE`

The *psycopg2* and *pg8000* dialects also offer the special level `AUTOCOMMIT`.

See also:

Psycopg2 Transaction Isolation Level

pg8000 Transaction Isolation Level

Remote-Schema Table Introspection and Postgresql search_path

The Postgresql dialect can reflect tables from any schema. The *Table.schema* argument, or alternatively the *MetaData.reflect.schema* argument determines which schema will be searched for the table or tables. The reflected *Table* objects will in all cases retain this `.schema` attribute as was specified. However, with regards to tables which these *Table* objects refer to via foreign key constraint, a decision must be made as to how the

`.schema` is represented in those remote tables, in the case where that remote schema name is also a member of the current Postgresql search path.

By default, the Postgresql dialect mimics the behavior encouraged by Postgresql's own `pg_get_constraintdef()` builtin procedure. This function returns a sample definition for a particular foreign key constraint, omitting the referenced schema name from that definition when the name is also in the Postgresql schema search path. The interaction below illustrates this behavior:

```
test=> CREATE TABLE test_schema.referred(id INTEGER PRIMARY KEY);
CREATE TABLE
test=> CREATE TABLE referring(
test(>         id INTEGER PRIMARY KEY,
test(>         referred_id INTEGER REFERENCES test_schema.referred(id));
CREATE TABLE
test=> SET search_path TO public, test_schema;
test=> SELECT pg_catalog.pg_get_constraintdef(r.oid, true) FROM
test-> pg_catalog.pg_class c JOIN pg_catalog.pg_namespace n
test-> ON n.oid = c.relnamespace
test-> JOIN pg_catalog.pg_constraint r  ON c.oid = r.conrelid
test-> WHERE c.relname='referring' AND r.contype = 'f'
test-> ;
               pg_get_constraintdef
---------------------------------------------------
 FOREIGN KEY (referred_id) REFERENCES referred(id)
(1 row)
```

Above, we created a table `referred` as a member of the remote schema `test_schema`, however when we added `test_schema` to the PG `search_path` and then asked `pg_get_constraintdef()` for the `FOREIGN KEY` syntax, `test_schema` was not included in the output of the function.

On the other hand, if we set the search path back to the typical default of `public`:

```
test=> SET search_path TO public;
SET
```

The same query against `pg_get_constraintdef()` now returns the fully schema-qualified name for us:

```
test=> SELECT pg_catalog.pg_get_constraintdef(r.oid, true) FROM
test-> pg_catalog.pg_class c JOIN pg_catalog.pg_namespace n
test-> ON n.oid = c.relnamespace
test-> JOIN pg_catalog.pg_constraint r  ON c.oid = r.conrelid
test-> WHERE c.relname='referring' AND r.contype = 'f';
                     pg_get_constraintdef
---------------------------------------------------------------
 FOREIGN KEY (referred_id) REFERENCES test_schema.referred(id)
(1 row)
```

SQLAlchemy will by default use the return value of `pg_get_constraintdef()` in order to determine the remote schema name. That is, if our `search_path` were set to include `test_schema`, and we invoked a table reflection process as follows:

```
>>> from sqlalchemy import Table, MetaData, create_engine
>>> engine = create_engine("postgresql://scott:tiger@localhost/test")
>>> with engine.connect() as conn:
...     conn.execute("SET search_path TO test_schema, public")
...     meta = MetaData()
...     referring = Table('referring', meta,
...                       autoload=True, autoload_with=conn)
```

```
...
<sqlalchemy.engine.result.ResultProxy object at 0x101612ed0>
```

The above process would deliver to the *MetaData.tables* collection `referred` table named **without** the schema:

```
>>> meta.tables['referred'].schema is None
True
```

To alter the behavior of reflection such that the referred schema is maintained regardless of the `search_path` setting, use the `postgresql_ignore_search_path` option, which can be specified as a dialect-specific argument to both *Table* as well as *MetaData.reflect()*:

```
>>> with engine.connect() as conn:
...     conn.execute("SET search_path TO test_schema, public")
...     meta = MetaData()
...     referring = Table('referring', meta, autoload=True,
...                       autoload_with=conn,
...                       postgresql_ignore_search_path=True)
...
<sqlalchemy.engine.result.ResultProxy object at 0x1016126d0>
```

We will now have `test_schema.referred` stored as schema-qualified:

```
>>> meta.tables['test_schema.referred'].schema
'test_schema'
```

Best Practices for Postgresql Schema reflection

The description of Postgresql schema reflection behavior is complex, and is the product of many years of dealing with widely varied use cases and user preferences. But in fact, there's no need to understand any of it if you just stick to the simplest use pattern: leave the `search_path` set to its default of `public` only, never refer to the name `public` as an explicit schema name otherwise, and refer to all other schema names explicitly when building up a *Table* object. The options described here are only for those users who can't, or prefer not to, stay within these guidelines.

Note that **in all cases**, the "default" schema is always reflected as `None`. The "default" schema on Postgresql is that which is returned by the Postgresql `current_schema()` function. On a typical Postgresql installation, this is the name `public`. So a table that refers to another which is in the `public` (i.e. default) schema will always have the `.schema` attribute set to `None`.

New in version 0.9.2: Added the `postgresql_ignore_search_path` dialect-level option accepted by *Table* and *MetaData.reflect()*.

See also:

The Schema Search Path - on the Postgresql website.

INSERT/UPDATE...RETURNING

The dialect supports PG 8.2's `INSERT..RETURNING`, `UPDATE..RETURNING` and `DELETE..RETURNING` syntaxes. `INSERT..RETURNING` is used by default for single-row INSERT statements in order to fetch newly generated primary key identifiers. To specify an explicit `RETURNING` clause, use the `_UpdateBase.returning()` method on a per-statement basis:

```
# INSERT..RETURNING
result = table.insert().returning(table.c.col1, table.c.col2).\
    values(name='foo')
print result.fetchall()

# UPDATE..RETURNING
result = table.update().returning(table.c.col1, table.c.col2).\
    where(table.c.name=='foo').values(name='bar')
print result.fetchall()

# DELETE..RETURNING
result = table.delete().returning(table.c.col1, table.c.col2).\
    where(table.c.name=='foo')
print result.fetchall()
```

Full Text Search

SQLAlchemy makes available the Postgresql `@@` operator via the *ColumnElement.match()* method on any textual column expression. On a Postgresql dialect, an expression like the following:

```
select([sometable.c.text.match("search string")])
```

will emit to the database:

```
SELECT text @@ to_tsquery('search string') FROM table
```

The Postgresql text search functions such as `to_tsquery()` and `to_tsvector()` are available explicitly using the standard *func* construct. For example:

```
select([
    func.to_tsvector('fat cats ate rats').match('cat & rat')
])
```

Emits the equivalent of:

```
SELECT to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat')
```

The *postgresql.TSVECTOR* type can provide for explicit CAST:

```
from sqlalchemy.dialects.postgresql import TSVECTOR
from sqlalchemy import select, cast
select([cast("some text", TSVECTOR)])
```

produces a statement equivalent to:

```
SELECT CAST('some text' AS TSVECTOR) AS anon_1
```

Full Text Searches in Postgresql are influenced by a combination of: the PostgresSQL setting of `default_text_search_config`, the `regconfig` used to build the GIN/GiST indexes, and the `regconfig` optionally passed in during a query.

When performing a Full Text Search against a column that has a GIN or GiST index that is already pre-computed (which is common on full text searches) one may need to explicitly pass in a particular PostgresSQL `regconfig` value to ensure the query-planner utilizes the index and does not re-compute the column on demand.

In order to provide for this explicit query planning, or to use different search strategies, the `match` method accepts a `postgresql_regconfig` keyword argument:

```
select([mytable.c.id]).where(
    mytable.c.title.match('somestring', postgresql_regconfig='english')
)
```

Emits the equivalent of:

```
SELECT mytable.id FROM mytable
WHERE mytable.title @@ to_tsquery('english', 'somestring')
```

One can also specifically pass in a *'regconfig'* value to the `to_tsvector()` command as the initial argument:

```
select([mytable.c.id]).where(
        func.to_tsvector('english', mytable.c.title )            .match('somestring', postgresql_rego
    )
```

produces a statement equivalent to:

```
SELECT mytable.id FROM mytable
WHERE to_tsvector('english', mytable.title) @@
    to_tsquery('english', 'somestring')
```

It is recommended that you use the `EXPLAIN ANALYZE...` tool from PostgresSQL to ensure that you are generating queries with SQLAlchemy that take full advantage of any indexes you may have created for full text search.

FROM ONLY ...

The dialect supports PostgreSQL's ONLY keyword for targeting only a particular table in an inheritance hierarchy. This can be used to produce the `SELECT ... FROM ONLY`, `UPDATE ONLY ...`, and `DELETE FROM ONLY ...` syntaxes. It uses SQLAlchemy's hints mechanism:

```
# SELECT ... FROM ONLY ...
result = table.select().with_hint(table, 'ONLY', 'postgresql')
print result.fetchall()

# UPDATE ONLY ...
table.update(values=dict(foo='bar')).with_hint('ONLY',
                                               dialect_name='postgresql')

# DELETE FROM ONLY ...
table.delete().with_hint('ONLY', dialect_name='postgresql')
```

Postgresql-Specific Index Options

Several extensions to the *Index* construct are available, specific to the PostgreSQL dialect.

Partial Indexes

Partial indexes add criterion to the index definition so that the index is applied to a subset of rows. These can be specified on *Index* using the `postgresql_where` keyword argument:

```
Index('my_index', my_table.c.id, postgresql_where=tbl.c.value > 10)
```

Operator Classes

PostgreSQL allows the specification of an *operator class* for each column of an index (see http://www.postgresql.org/docs/8.3/interactive/indexes-opclass.html). The `Index` construct allows these to be specified via the `postgresql_ops` keyword argument:

```
Index('my_index', my_table.c.id, my_table.c.data,
                        postgresql_ops={
                            'data': 'text_pattern_ops',
                            'id': 'int4_ops'
                        })
```

New in version 0.7.2: `postgresql_ops` keyword argument to `Index` construct.

Note that the keys in the `postgresql_ops` dictionary are the "key" name of the `Column`, i.e. the name used to access it from the `.c` collection of `Table`, which can be configured to be different than the actual name of the column as expressed in the database.

Index Types

PostgreSQL provides several index types: B-Tree, Hash, GiST, and GIN, as well as the ability for users to create their own (see http://www.postgresql.org/docs/8.3/static/indexes-types.html). These can be specified on `Index` using the `postgresql_using` keyword argument:

```
Index('my_index', my_table.c.data, postgresql_using='gin')
```

The value passed to the keyword argument will be simply passed through to the underlying CREATE INDEX command, so it *must* be a valid index type for your version of PostgreSQL.

Index Storage Parameters

PostgreSQL allows storage parameters to be set on indexes. The storage parameters available depend on the index method used by the index. Storage parameters can be specified on `Index` using the `postgresql_with` keyword argument:

```
Index('my_index', my_table.c.data, postgresql_with={"fillfactor": 50})
```

New in version 1.0.6.

Indexes with CONCURRENTLY

The Postgresql index option CONCURRENTLY is supported by passing the flag `postgresql_concurrently` to the `Index` construct:

```
tbl = Table('testtbl', m, Column('data', Integer))

idx1 = Index('test_idx1', tbl.c.data, postgresql_concurrently=True)
```

The above index construct will render SQL as:

```
CREATE INDEX CONCURRENTLY test_idx1 ON testtbl (data)
```

New in version 0.9.9.

Postgresql Index Reflection

The Postgresql database creates a UNIQUE INDEX implicitly whenever the UNIQUE CONSTRAINT construct is used. When inspecting a table using *Inspector*, the *Inspector.get_indexes()* and the *Inspector.get_unique_constraints()* will report on these two constructs distinctly; in the case of the index, the key `duplicates_constraint` will be present in the index entry if it is detected as mirroring a constraint. When performing reflection using `Table(..., autoload=True)`, the UNIQUE INDEX is **not** returned in `Table.indexes` when it is detected as mirroring a *UniqueConstraint* in the `Table.constraints` collection.

Changed in version 1.0.0: - *Table* reflection now includes *UniqueConstraint* objects present in the `Table.constraints` collection; the Postgresql backend will no longer include a "mirrored" *Index* construct in `Table.indexes` if it is detected as corresponding to a unique constraint.

Special Reflection Options

The *Inspector* used for the Postgresql backend is an instance of *PGInspector*, which offers additional methods:

```
from sqlalchemy import create_engine, inspect

engine = create_engine("postgresql+psycopg2://localhost/test")
insp = inspect(engine)  # will be a PGInspector

print(insp.get_enums())
```

class `sqlalchemy.dialects.postgresql.base.`**`PGInspector`**(*conn*)

Bases: *sqlalchemy.engine.reflection.Inspector*

`get_enums`(*schema=None*)

Return a list of ENUM objects.

Each member is a dictionary containing these fields:

- name - name of the enum
- schema - the schema name for the enum.
- visible - boolean, whether or not this enum is visible in the default search path.
- labels - a list of string labels that apply to the enum.

Parameters **`schema`** – schema name. If None, the default schema (typically 'public') is used. May also be set to '*' to indicate load enums for all schemas.

New in version 1.0.0.

`get_foreign_table_names`(*schema=None*)

Return a list of FOREIGN TABLE names.

Behavior is similar to that of *Inspector.get_table_names()*, except that the list is limited to those tables tha report a `relkind` value of `f`.

New in version 1.0.0.

get_table_oid(*table_name*, *schema=None*)
Return the OID for the given table name.

PostgreSQL Table Options

Several options for CREATE TABLE are supported directly by the PostgreSQL dialect in conjunction with the *Table* construct:

- TABLESPACE:

```
Table("some_table", metadata, ..., postgresql_tablespace='some_tablespace')
```

- ON COMMIT:

```
Table("some_table", metadata, ..., postgresql_on_commit='PRESERVE ROWS')
```

- WITH OIDS:

```
Table("some_table", metadata, ..., postgresql_with_oids=True)
```

- WITHOUT OIDS:

```
Table("some_table", metadata, ..., postgresql_with_oids=False)
```

- INHERITS:

```
Table("some_table", metadata, ..., postgresql_inherits="some_supertable")

Table("some_table", metadata, ..., postgresql_inherits=("t1", "t2", ...))
```

New in version 1.0.0.

See also:

Postgresql CREATE TABLE options

ARRAY Types

The Postgresql dialect supports arrays, both as multidimensional column types as well as array literals:

- *postgresql.ARRAY* - ARRAY datatype
- *postgresql.array* - array literal
- *postgresql.array_agg()* - ARRAY_AGG SQL function
- *postgresql.aggregate_order_by* - helper for PG's ORDER BY aggregate function syntax.

JSON Types

The Postgresql dialect supports both JSON and JSONB datatypes, including psycopg2's native support and support for all of Postgresql's special operators:

- *postgresql.JSON*
- *postgresql.JSONB*

HSTORE Type

The Postgresql HSTORE type as well as hstore literals are supported:

- *postgresql.HSTORE* - HSTORE datatype
- *postgresql.hstore* - hstore literal

ENUM Types

Postgresql has an independently creatable TYPE structure which is used to implement an enumerated type. This approach introduces significant complexity on the SQLAlchemy side in terms of when this type should be CREATED and DROPPED. The type object is also an independently reflectable entity. The following sections should be consulted:

- *postgresql.ENUM* - DDL and typing support for ENUM.
- *PGInspector.get_enums()* - retrieve a listing of current ENUM types
- *postgresql.ENUM.create()* , *postgresql.ENUM.drop()* - individual CREATE and DROP commands for ENUM.

Using ENUM with ARRAY

The combination of ENUM and ARRAY is not directly supported by backend DBAPIs at this time. In order to send and receive an ARRAY of ENUM, use the following workaround type:

```
class ArrayOfEnum(ARRAY):

    def bind_expression(self, bindvalue):
        return sa.cast(bindvalue, self)

    def result_processor(self, dialect, coltype):
        super_rp = super(ArrayOfEnum, self).result_processor(
            dialect, coltype)

        def handle_raw_string(value):
            inner = re.match(r"^{(.*)}$", value).group(1)
            return inner.split(",")

        def process(value):
            if value is None:
                return None
            return super_rp(handle_raw_string(value))
        return process
```

E.g.:

```
Table(
    'mydata', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', ArrayOfEnum(ENUM('a', 'b, 'c', name='myenum')))

)
```

This type is not included as a built-in type as it would be incompatible with a DBAPI that suddenly decides to support ARRAY of ENUM directly in a new version.

PostgreSQL Data Types

As with all SQLAlchemy dialects, all UPPERCASE types that are known to be valid with Postgresql are importable from the top level dialect, whether they originate from *sqlalchemy.types* or from the local dialect:

```
from sqlalchemy.dialects.postgresql import \
    ARRAY, BIGINT, BIT, BOOLEAN, BYTEA, CHAR, CIDR, DATE, \
    DOUBLE_PRECISION, ENUM, FLOAT, HSTORE, INET, INTEGER, \
    INTERVAL, JSON, JSONB, MACADDR, NUMERIC, OID, REAL, SMALLINT, TEXT, \
    TIME, TIMESTAMP, UUID, VARCHAR, INT4RANGE, INT8RANGE, NUMRANGE, \
    DATERANGE, TSRANGE, TSTZRANGE, TSVECTOR
```

Types which are specific to PostgreSQL, or have PostgreSQL-specific construction arguments, are as follows:

class `sqlalchemy.dialects.postgresql.`**`aggregate_order_by`**(*target*, *order_by*)

Bases: *sqlalchemy.sql.expression.ColumnElement*

Represent a Postgresql aggregate order by expression.

E.g.:

```
from sqlalchemy.dialects.postgresql import aggregate_order_by
expr = func.array_agg(aggregate_order_by(table.c.a, table.c.b.desc()))
stmt = select([expr])
```

would represent the expression:

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

Similarly:

```
expr = func.string_agg(
    table.c.a,
    aggregate_order_by(literal_column("','"), table.c.a)
)
stmt = select([expr])
```

Would represent:

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

New in version 1.1.

See also:

array_agg

class sqlalchemy.dialects.postgresql.**array**(*clauses*, ***kw*)

Bases: *sqlalchemy.sql.expression.Tuple*

A Postgresql ARRAY literal.

This is used to produce ARRAY literals in SQL expressions, e.g.:

```
from sqlalchemy.dialects.postgresql import array
from sqlalchemy.dialects import postgresql
from sqlalchemy import select, func

stmt = select([
                array([1,2]) + array([3,4,5])
            ])

print stmt.compile(dialect=postgresql.dialect())
```

Produces the SQL:

```
SELECT ARRAY[%(param_1)s, %(param_2)s] ||
    ARRAY[%(param_3)s, %(param_4)s, %(param_5)s]) AS anon_1
```

An instance of *array* will always have the datatype *ARRAY*. The "inner" type of the array is inferred from the values present, unless the `type_` keyword argument is passed:

```
array(['foo', 'bar'], type_=CHAR)
```

New in version 0.8: Added the *array* literal type.

See also:

postgresql.ARRAY

class sqlalchemy.dialects.postgresql.**ARRAY**(*item_type*, *as_tuple=False*, *dimensions=None*, *zero_indexes=False*)

Bases: sqlalchemy.sql.expression.SchemaEventTarget, *sqlalchemy.types.ARRAY*

Postgresql ARRAY type.

Changed in version 1.1: The *postgresql.ARRAY* type is now a subclass of the core *types.ARRAY* type.

The *postgresql.ARRAY* type is constructed in the same way as the core *types.ARRAY* type; a member type is required, and a number of dimensions is recommended if the type is to be used for more than one dimension:

```
from sqlalchemy.dialects import postgresql

mytable = Table("mytable", metadata,
        Column("data", postgresql.ARRAY(Integer, dimensions=2))
    )
```

The *postgresql.ARRAY* type provides all operations defined on the core *types.ARRAY* type, including support for "dimensions", indexed access, and simple matching such as *types.ARRAY.Comparator.any()* and *types.ARRAY.Comparator.all()*. *postgresql.ARRAY* class also provides PostgreSQL-specific methods for containment operations, including *postgresql.ARRAY.Comparator.contains()* *postgresql.ARRAY.Comparator.contained_by()*, and *postgresql.ARRAY.Comparator.overlap()*, e.g.:

```
mytable.c.data.contains([1, 2])
```

The *postgresql.ARRAY* type may not be supported on all PostgreSQL DBAPIs; it is currently known to work on psycopg2 only.

Additionally, the *postgresql.ARRAY* type does not work directly in conjunction with the *ENUM* type. For a workaround, see the special type at *Using ENUM with ARRAY*.

See also:

types.ARRAY - base array type

postgresql.array - produces a literal array value.

class Comparator(*expr*)

Bases: `sqlalchemy.types.Comparator`

Define comparison operations for *ARRAY*.

Note that these operations are in addition to those provided by the base *types.ARRAY.Comparator* class, including *types.ARRAY.Comparator.any()* and *types.ARRAY.Comparator.all()*.

contained_by(*other*)

Boolean expression. Test if elements are a proper subset of the elements of the argument array expression.

contains(*other*, ***kwargs*)

Boolean expression. Test if elements are a superset of the elements of the argument array expression.

overlap(*other*)

Boolean expression. Test if array has elements in common with an argument array expression.

ARRAY.**__init__**(*item_type*, *as_tuple=False*, *dimensions=None*, *zero_indexes=False*)

Construct an ARRAY.

E.g.:

```
Column('myarray', ARRAY(Integer))
```

Arguments are:

Parameters

- **item_type** – The data type of items of this array. Note that dimensionality is irrelevant here, so multi-dimensional arrays like `INTEGER[][]`, are constructed as `ARRAY(Integer)`, not as `ARRAY(ARRAY(Integer))` or such.
- **as_tuple=False** – Specify whether return results should be converted to tuples from lists. DBAPIs such as psycopg2 return lists by default. When tuples are returned, the results are hashable.
- **dimensions** – if non-None, the ARRAY will assume a fixed number of dimensions. This will cause the DDL emitted for this ARRAY to include the exact number of bracket clauses `[]`, and will also optimize the performance of the type overall. Note that PG arrays are always implicitly "non-dimensioned", meaning they can store any number of dimensions no matter how they were declared.
- **zero_indexes=False** – when True, index values will be converted between Python zero-based and Postgresql one-based indexes, e.g. a value of one will be added to all index values before passing to the database.

New in version 0.9.5.

sqlalchemy.dialects.postgresql.**array_agg**(**arg*, ***kw*)

Postgresql-specific form of `array_agg`, ensures return type is `postgresql.ARRAY` and not the plain `types.ARRAY`.

New in version 1.1.

sqlalchemy.dialects.postgresql.**Any**(*other*, *arrexpr*, *operator=<built-in function eq>*)

A synonym for the `ARRAY.Comparator.any()` method.

This method is legacy and is here for backwards-compatiblity.

See also:

`expression.any_()`

sqlalchemy.dialects.postgresql.**All**(*other*, *arrexpr*, *operator=<built-in function eq>*)

A synonym for the `ARRAY.Comparator.all()` method.

This method is legacy and is here for backwards-compatiblity.

See also:

`expression.all_()`

class sqlalchemy.dialects.postgresql.**BIT**(*length=None*, *varying=False*)

Bases: `sqlalchemy.types.TypeEngine`

class sqlalchemy.dialects.postgresql.**BYTEA**(*length=None*)

Bases: `sqlalchemy.types.LargeBinary`

__init__(*length=None*)

Construct a LargeBinary type.

Parameters **length** – optional, a length for the column for use in DDL statements, for those BLOB types that accept a length (i.e. MySQL). It does *not* produce a *lengthed* BINARY/VARBINARY type - use the BINARY/VARBINARY types specifically for those. May be safely omitted if no `CREATE TABLE` will be issued. Certain databases may require a *length* for use in DDL, and will raise an exception when the `CREATE TABLE` DDL is issued.

class sqlalchemy.dialects.postgresql.**CIDR**

Bases: `sqlalchemy.types.TypeEngine`

__init__

x.__init__(...) initializes x; see help(type(x)) for signature

class sqlalchemy.dialects.postgresql.**DOUBLE_PRECISION**(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*, ***kwargs*)

Bases: `sqlalchemy.types.Float`

__init__(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*, ***kwargs*)

Construct a Float.

Parameters

- **precision** – the numeric precision for use in DDL `CREATE TABLE`.
- **asdecimal** – the same flag as that of `Numeric`, but defaults to `False`. Note that setting this flag to `True` results in floating point conversion.

- **decimal_return_scale** – Default scale to use when converting from floats to Python decimals. Floating point values will typically be much longer due to decimal inaccuracy, and most floating point database types don't have a notion of "scale", so by default the float type looks for the first ten decimal places when converting. Specfiying this value will override that length. Note that the MySQL float types, which do include "scale", will use "scale" as the default for decimal_return_scale, if not otherwise specified.

  New in version 0.9.0.

- ****kwargs** – deprecated. Additional arguments here are ignored by the default `Float` type. For database specific floats that support additional arguments, see that dialect's documentation for details, such as `sqlalchemy.dialects.mysql.FLOAT`.

class `sqlalchemy.dialects.postgresql.`**`ENUM`**(**enums*, ***kw*)

Bases: `sqlalchemy.types.Enum`

Postgresql ENUM type.

This is a subclass of `types.Enum` which includes support for PG's `CREATE TYPE` and `DROP TYPE`.

When the builtin type `types.Enum` is used and the `Enum.native_enum` flag is left at its default of True, the Postgresql backend will use a `postgresql.ENUM` type as the implementation, so the special create/drop rules will be used.

The create/drop behavior of ENUM is necessarily intricate, due to the awkward relationship the ENUM type has in relationship to the parent table, in that it may be "owned" by just a single table, or may be shared among many tables.

When using `types.Enum` or `postgresql.ENUM` in an "inline" fashion, the `CREATE TYPE` and `DROP TYPE` is emitted corresponding to when the `Table.create()` and `Table.drop()` methods are called:

```
table = Table('sometable', metadata,
    Column('some_enum', ENUM('a', 'b', 'c', name='myenum'))
)

table.create(engine)  # will emit CREATE ENUM and CREATE TABLE
table.drop(engine)  # will emit DROP TABLE and DROP ENUM
```

To use a common enumerated type between multiple tables, the best practice is to declare the `types.Enum` or `postgresql.ENUM` independently, and associate it with the `MetaData` object itself:

```
my_enum = ENUM('a', 'b', 'c', name='myenum', metadata=metadata)

t1 = Table('sometable_one', metadata,
    Column('some_enum', myenum)
)

t2 = Table('sometable_two', metadata,
    Column('some_enum', myenum)
)
```

When this pattern is used, care must still be taken at the level of individual table creates. Emitting CREATE TABLE without also specifying `checkfirst=True` will still cause issues:

```
t1.create(engine) # will fail: no such type 'myenum'
```

If we specify `checkfirst=True`, the individual table-level create operation will check for the `ENUM` and create if not exists:

```
# will check if enum exists, and emit CREATE TYPE if not
t1.create(engine, checkfirst=True)
```

When using a metadata-level ENUM type, the type will always be created and dropped if either the metadata-wide create/drop is called:

```
metadata.create_all(engine)  # will emit CREATE TYPE
metadata.drop_all(engine)  # will emit DROP TYPE
```

The type can also be created and dropped directly:

```
my_enum.create(engine)
my_enum.drop(engine)
```

Changed in version 1.0.0: The Postgresql *postgresql.ENUM* type now behaves more strictly with regards to CREATE/DROP. A metadata-level ENUM type will only be created and dropped at the metadata level, not the table level, with the exception of `table.create(checkfirst=True)`. The `table.drop()` call will now emit a DROP TYPE for a table-level enumerated type.

__init__(**enums*, ***kw*)

Construct an *ENUM*.

Arguments are the same as that of *types.Enum*, but also including the following parameters.

Parameters **create_type** – Defaults to True. Indicates that `CREATE TYPE` should be emitted, after optionally checking for the presence of the type, when the parent table is being created; and additionally that `DROP TYPE` is called when the table is dropped. When `False`, no check will be performed and no `CREATE TYPE` or `DROP TYPE` is emitted, unless *create()* or *drop()* are called directly. Setting to `False` is helpful when invoking a creation scheme to a SQL file without access to the actual database - the *create()* and *drop()* methods can be used to emit SQL to a target bind.

New in version 0.7.4.

create(*bind=None*, *checkfirst=True*)

Emit `CREATE TYPE` for this *ENUM*.

If the underlying dialect does not support Postgresql CREATE TYPE, no action is taken.

Parameters

- **bind** – a connectable *Engine*, *Connection*, or similar object to emit SQL.
- **checkfirst** – if `True`, a query against the PG catalog will be first performed to see if the type does not exist already before creating.

drop(*bind=None*, *checkfirst=True*)

Emit `DROP TYPE` for this *ENUM*.

If the underlying dialect does not support Postgresql DROP TYPE, no action is taken.

Parameters

- **bind** – a connectable *Engine*, *Connection*, or similar object to emit SQL.
- **checkfirst** – if `True`, a query against the PG catalog will be first performed to see if the type actually exists before dropping.

class sqlalchemy.dialects.postgresql.**HSTORE**(*text_type=None*)

Bases: *sqlalchemy.types.Indexable*, *sqlalchemy.types.Concatenable*, *sqlalchemy.types.TypeEngine*

Represent the Postgresql HSTORE type.

The *HSTORE* type stores dictionaries containing strings, e.g.:

```
data_table = Table('data_table', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', HSTORE)
)

with engine.connect() as conn:
    conn.execute(
        data_table.insert(),
        data = {"key1": "value1", "key2": "value2"}
    )
```

HSTORE provides for a wide range of operations, including:

- Index operations:

```
data_table.c.data['some key'] == 'some value'
```

- Containment operations:

```
data_table.c.data.has_key('some key')

data_table.c.data.has_all(['one', 'two', 'three'])
```

- Concatenation:

```
data_table.c.data + {"k1": "v1"}
```

For a full list of special methods see `HSTORE.comparator_factory`.

For usage with the SQLAlchemy ORM, it may be desirable to combine the usage of *HSTORE* with *MutableDict* dictionary now part of the *sqlalchemy.ext.mutable* extension. This extension will allow "in-place" changes to the dictionary, e.g. addition of new keys or replacement/removal of existing keys to/from the current dictionary, to produce events which will be detected by the unit of work:

```
from sqlalchemy.ext.mutable import MutableDict

class MyClass(Base):
    __tablename__ = 'data_table'

    id = Column(Integer, primary_key=True)
    data = Column(MutableDict.as_mutable(HSTORE))

my_object = session.query(MyClass).one()

# in-place mutation, requires Mutable extension
# in order for the ORM to detect
my_object.data['some_key'] = 'some value'

session.commit()
```

When the *sqlalchemy.ext.mutable* extension is not used, the ORM will not be alerted to any changes to the contents of an existing dictionary, unless that dictionary value is re-assigned to the HSTORE-attribute itself, thus generating a change event.

New in version 0.8.

See also:

hstore - render the Postgresql `hstore()` function.

class Comparator(*expr*)

Bases: `sqlalchemy.types.Comparator`, `sqlalchemy.types.Comparator`

Define comparison operations for *HSTORE*.

array()

Text array expression. Returns array of alternating keys and values.

contained_by(*other*)

Boolean expression. Test if keys are a proper subset of the keys of the argument jsonb expression.

contains(*other*, ***kwargs*)

Boolean expression. Test if keys (or array) are a superset of/contained the keys of the argument jsonb expression.

defined(*key*)

Boolean expression. Test for presence of a non-NULL value for the key. Note that the key may be a SQLA expression.

delete(*key*)

HStore expression. Returns the contents of this hstore with the given key deleted. Note that the key may be a SQLA expression.

has_all(*other*)

Boolean expression. Test for presence of all keys in jsonb

has_any(*other*)

Boolean expression. Test for presence of any key in jsonb

has_key(*other*)

Boolean expression. Test for presence of a key. Note that the key may be a SQLA expression.

keys()

Text array expression. Returns array of keys.

matrix()

Text array expression. Returns array of [key, value] pairs.

slice(*array*)

HStore expression. Returns a subset of an hstore defined by array of keys.

vals()

Text array expression. Returns array of values.

HSTORE.**__init__**(*text_type=None*)

Construct a new *HSTORE*.

Parameters **text_type** – the type that should be used for indexed values. Defaults to *types.Text*.

New in version 1.1.0.

HSTORE.**comparator_factory**

alias of *Comparator*

class sqlalchemy.dialects.postgresql.**hstore**(**args*, ***kwargs*)

Bases: *sqlalchemy.sql.functions.GenericFunction*

Construct an hstore value within a SQL expression using the Postgresql `hstore()` function.

The *hstore* function accepts one or two arguments as described in the Postgresql documentation.

E.g.:

```
from sqlalchemy.dialects.postgresql import array, hstore

select([hstore('key1', 'value1')])

select([
        hstore(
            array(['key1', 'key2', 'key3']),
            array(['value1', 'value2', 'value3'])
        )
    ])
```

New in version 0.8.

See also:

HSTORE - the Postgresql `HSTORE` datatype.

type

alias of *HSTORE*

class sqlalchemy.dialects.postgresql.**INET**

Bases: *sqlalchemy.types.TypeEngine*

__init__

x.__init__(...) initializes x; see help(type(x)) for signature

class sqlalchemy.dialects.postgresql.**INTERVAL**(*precision=None*)

Bases: *sqlalchemy.types.TypeEngine*

Postgresql INTERVAL type.

The INTERVAL type may not be supported on all DBAPIs. It is known to work on psycopg2 and not pg8000 or zxjdbc.

class sqlalchemy.dialects.postgresql.**JSON**(*none_as_null=False*, *astext_type=None*)

Bases: *sqlalchemy.types.JSON*

Represent the Postgresql JSON type.

This type is a specialization of the Core-level *types.JSON* type. Be sure to read the documentation for *types.JSON* for important tips regarding treatment of NULL values and ORM use.

Changed in version 1.1: *postgresql.JSON* is now a Postgresql- specific specialization of the new *types.JSON* type.

The operators provided by the Postgresql version of *JSON* include:

- Index operations (the `->` operator):

```
data_table.c.data['some key']

data_table.c.data[5]
```

- Index operations returning text (the `->>` operator):

```
data_table.c.data['some key'].astext == 'some value'
```

- Index operations with CAST (equivalent to `CAST(col ->> ['some key'] AS <type>)`):

```
data_table.c.data['some key'].astext.cast(Integer) == 5
```

- Path index operations (the `#>` operator):

```
data_table.c.data[('key_1', 'key_2', 5, ..., 'key_n')]
```

•Path index operations returning text (the #>> operator):

```
data_table.c.data[('key_1', 'key_2', 5, ..., 'key_n')].astext == 'some value'
```

Changed in version 1.1: The *ColumnElement.cast()* operator on JSON objects now requires that the *JSON.Comparator.astext* modifier be called explicitly, if the cast works only from a textual string.

Index operations return an expression object whose type defaults to *JSON* by default, so that further JSON-oriented instructions may be called upon the result type.

Custom serializers and deserializers are specified at the dialect level, that is using *create_engine()*. The reason for this is that when using psycopg2, the DBAPI only allows serializers at the per-cursor or per-connection level. E.g.:

```
engine = create_engine("postgresql://scott:tiger@localhost/test",
                       json_serializer=my_serialize_fn,
                       json_deserializer=my_deserialize_fn
                )
```

When using the psycopg2 dialect, the json_deserializer is registered against the database using `psycopg2.extras.register_default_json`.

See also:

types.JSON - Core level JSON type

JSONB

class Comparator(*expr*)

Bases: `sqlalchemy.types.Comparator`

Define comparison operations for *JSON*.

astext

On an indexed expression, use the "astext" (e.g. "->>") conversion when rendered in SQL.

E.g.:

```
select([data_table.c.data['some key'].astext])
```

See also:

ColumnElement.cast()

`JSON.`**__init__**(*none_as_null=False*, *astext_type=None*)

Construct a *JSON* type.

Parameters

- **none_as_null** – if True, persist the value `None` as a SQL NULL value, not the JSON encoding of `null`. Note that when this flag is False, the *null()* construct can still be used to persist a NULL value:

```
from sqlalchemy import null
conn.execute(table.insert(), data=null())
```

Changed in version 0.9.8: - Added `none_as_null`, and *null()* is now supported in order to persist a NULL value.

See also:

JSON.NULL

- **astext_type** – the type to use for the *JSON.Comparator.astext* accessor on indexed attributes. Defaults to *types.Text*.

  New in version 1.1.

JSON.**comparator_factory**
alias of *Comparator*

class sqlalchemy.dialects.postgresql.**JSONB**(*none_as_null=False*, *astext_type=None*)
Bases: sqlalchemy.dialects.postgresql.json.JSON

Represent the Postgresql JSONB type.

The *JSONB* type stores arbitrary JSONB format data, e.g.:

```
data_table = Table('data_table', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', JSONB)
)

with engine.connect() as conn:
    conn.execute(
        data_table.insert(),
        data = {"key1": "value1", "key2": "value2"}
    )
```

The *JSONB* type includes all operations provided by *JSON*, including the same behaviors for indexing operations. It also adds additional operators specific to JSONB, including *JSONB.Comparator.has_key()*, *JSONB.Comparator.has_all()*, *JSONB.Comparator.has_any()*, *JSONB.Comparator.contains()*, and *JSONB.Comparator.contained_by()*.

Like the *JSON* type, the *JSONB* type does not detect in-place changes when used with the ORM, unless the *sqlalchemy.ext.mutable* extension is used.

Custom serializers and deserializers are shared with the *JSON* class, using the `json_serializer` and `json_deserializer` keyword arguments. These must be specified at the dialect level using *create_engine()*. When using psycopg2, the serializers are associated with the jsonb type using `psycopg2.extras.register_default_jsonb` on a per-connection basis, in the same way that `psycopg2.extras.register_default_json` is used to register these handlers with the json type.

New in version 0.9.7.

See also:

JSON

class Comparator(*expr*)
Bases: sqlalchemy.dialects.postgresql.json.Comparator

Define comparison operations for *JSON*.

contained_by(*other*)
Boolean expression. Test if keys are a proper subset of the keys of the argument jsonb expression.

contains(*other*, ***kwargs*)
Boolean expression. Test if keys (or array) are a superset of/contained the keys of the argument jsonb expression.

has_all(*other*)
Boolean expression. Test for presence of all keys in jsonb

has_any(*other*)
Boolean expression. Test for presence of any key in jsonb

has_key(*other*)
Boolean expression. Test for presence of a key. Note that the key may be a SQLA expression.

JSONB.**comparator_factory**
alias of *Comparator*

class sqlalchemy.dialects.postgresql.**MACADDR**
Bases: *sqlalchemy.types.TypeEngine*

__init__
x.__init__(...) initializes x; see help(type(x)) for signature

class sqlalchemy.dialects.postgresql.**OID**
Bases: *sqlalchemy.types.TypeEngine*

Provide the Postgresql OID type.

New in version 0.9.5.

__init__
x.__init__(...) initializes x; see help(type(x)) for signature

class sqlalchemy.dialects.postgresql.**REAL**(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*, ***kwargs*)
Bases: *sqlalchemy.types.Float*

The SQL REAL type.

__init__(*precision=None*, *asdecimal=False*, *decimal_return_scale=None*, ***kwargs*)
Construct a Float.

Parameters

- **precision** – the numeric precision for use in DDL `CREATE TABLE`.
- **asdecimal** – the same flag as that of *Numeric*, but defaults to `False`. Note that setting this flag to `True` results in floating point conversion.
- **decimal_return_scale** – Default scale to use when converting from floats to Python decimals. Floating point values will typically be much longer due to decimal inaccuracy, and most floating point database types don't have a notion of "scale", so by default the float type looks for the first ten decimal places when converting. Specfiying this value will override that length. Note that the MySQL float types, which do include "scale", will use "scale" as the default for decimal_return_scale, if not otherwise specified.

  New in version 0.9.0.
- ****kwargs** – deprecated. Additional arguments here are ignored by the default *Float* type. For database specific floats that support additional arguments, see that dialect's documentation for details, such as *sqlalchemy.dialects.mysql.FLOAT*.

class sqlalchemy.dialects.postgresql.**TSVECTOR**
Bases: *sqlalchemy.types.TypeEngine*

The *postgresql.TSVECTOR* type implements the Postgresql text search type TSVECTOR.

It can be used to do full text queries on natural language documents.

New in version 0.9.0.

See also:

Full Text Search

__init__
x.__init__(...) initializes x; see help(type(x)) for signature

class sqlalchemy.dialects.postgresql.**UUID**(*as_uuid=False*)
Bases: *sqlalchemy.types.TypeEngine*

Postgresql UUID type.

Represents the UUID column type, interpreting data either as natively returned by the DBAPI or as Python uuid objects.

The UUID type may not be supported on all DBAPIs. It is known to work on psycopg2 and not pg8000.

__init__(*as_uuid=False*)
Construct a UUID type.

Parameters **as_uuid=False** – if True, values will be interpreted as Python uuid objects, converting to/from string via the DBAPI.

Range Types

The new range column types found in PostgreSQL 9.2 onwards are catered for by the following types:

class sqlalchemy.dialects.postgresql.**INT4RANGE**
Bases: *sqlalchemy.dialects.postgresql.ranges.RangeOperators*, *sqlalchemy.types.TypeEngine*

Represent the Postgresql INT4RANGE type.

New in version 0.8.2.

class sqlalchemy.dialects.postgresql.**INT8RANGE**
Bases: *sqlalchemy.dialects.postgresql.ranges.RangeOperators*, *sqlalchemy.types.TypeEngine*

Represent the Postgresql INT8RANGE type.

New in version 0.8.2.

class sqlalchemy.dialects.postgresql.**NUMRANGE**
Bases: *sqlalchemy.dialects.postgresql.ranges.RangeOperators*, *sqlalchemy.types.TypeEngine*

Represent the Postgresql NUMRANGE type.

New in version 0.8.2.

class sqlalchemy.dialects.postgresql.**DATERANGE**
Bases: *sqlalchemy.dialects.postgresql.ranges.RangeOperators*, *sqlalchemy.types.TypeEngine*

Represent the Postgresql DATERANGE type.

New in version 0.8.2.

class sqlalchemy.dialects.postgresql.**TSRANGE**
Bases: *sqlalchemy.dialects.postgresql.ranges.RangeOperators*, *sqlalchemy.types.TypeEngine*

Represent the Postgresql TSRANGE type.

New in version 0.8.2.

class sqlalchemy.dialects.postgresql.**TSTZRANGE**

Bases: *sqlalchemy.dialects.postgresql.ranges.RangeOperators*, *sqlalchemy.types.TypeEngine*

Represent the Postgresql TSTZRANGE type.

New in version 0.8.2.

The types above get most of their functionality from the following mixin:

class sqlalchemy.dialects.postgresql.ranges.**RangeOperators**

This mixin provides functionality for the Range Operators listed in Table 9-44 of the postgres documentation for Range Functions and Operators. It is used by all the range types provided in the `postgres` dialect and can likely be used for any range types you create yourself.

No extra support is provided for the Range Functions listed in Table 9-45 of the postgres documentation. For these, the normal *func()* object should be used.

New in version 0.8.2: Support for Postgresql RANGE operations.

class comparator_factory(*expr*)

Bases: `sqlalchemy.types.Comparator`

Define comparison operations for range types.

__ne__(*other*)

Boolean expression. Returns true if two ranges are not equal

adjacent_to(*other*)

Boolean expression. Returns true if the range in the column is adjacent to the range in the operand.

contained_by(*other*)

Boolean expression. Returns true if the column is contained within the right hand operand.

contains(*other*, ***kw*)

Boolean expression. Returns true if the right hand operand, which can be an element or a range, is contained within the column.

not_extend_left_of(*other*)

Boolean expression. Returns true if the range in the column does not extend left of the range in the operand.

not_extend_right_of(*other*)

Boolean expression. Returns true if the range in the column does not extend right of the range in the operand.

overlaps(*other*)

Boolean expression. Returns true if the column overlaps (has points in common with) the right hand operand.

strictly_left_of(*other*)

Boolean expression. Returns true if the column is strictly left of the right hand operand.

strictly_right_of(*other*)

Boolean expression. Returns true if the column is strictly right of the right hand operand.

Warning: The range type DDL support should work with any Postgres DBAPI driver, however the data types returned may vary. If you are using `psycopg2`, it's recommended to upgrade to version 2.5 or later before using these column types.

When instantiating models that use these column types, you should pass whatever data type is expected by the DBAPI driver you're using for the column type. For psycopg2 these are NumericRange, DateRange, DateTimeRange and DateTimeTZRange or the class you've registered with register_range().

For example:

```
from psycopg2.extras import DateTimeRange
from sqlalchemy.dialects.postgresql import TSRANGE

class RoomBooking(Base):

    __tablename__ = 'room_booking'

    room = Column(Integer(), primary_key=True)
    during = Column(TSRANGE())

booking = RoomBooking(
    room=101,
    during=DateTimeRange(datetime(2013, 3, 23), None)
)
```

PostgreSQL Constraint Types

SQLAlchemy supports Postgresql EXCLUDE constraints via the *ExcludeConstraint* class:

class sqlalchemy.dialects.postgresql.**ExcludeConstraint**(**elements*, ***kw*)

Bases: *sqlalchemy.schema.ColumnCollectionConstraint*

A table-level EXCLUDE constraint.

Defines an EXCLUDE constraint as described in the postgres documentation.

__init__(**elements*, ***kw*)

Parameters

- ***elements** – A sequence of two tuples of the form (column, operator) where column must be a column name or Column object and operator must be a string containing the operator to use.
- **name** – Optional, the in-database name of this constraint.
- **deferrable** – Optional bool. If set, emit DEFERRABLE or NOT DEFERRABLE when issuing DDL for this constraint.
- **initially** – Optional string. If set, emit INITIALLY <value> when issuing DDL for this constraint.
- **using** – Optional string. If set, emit USING <index_method> when issuing DDL for this constraint. Defaults to 'gist'.
- **where** – Optional string. If set, emit WHERE <predicate> when issuing DDL for this constraint.

For example:

```
from sqlalchemy.dialects.postgresql import ExcludeConstraint, TSRANGE

class RoomBooking(Base):
```

```
    __tablename__ = 'room_booking'

    room = Column(Integer(), primary_key=True)
    during = Column(TSRANGE())

    __table_args__ = (
        ExcludeConstraint(('room', '='), ('during', '&&')),
    )
```

psycopg2

Support for the PostgreSQL database via the psycopg2 driver.

DBAPI

Documentation and download information (if applicable) for psycopg2 is available at: http://pypi.python.org/pypi/psycopg2/

Connecting

Connect String:

```
postgresql+psycopg2://user:password@host:port/dbname[?key=value&key=value...]
```

psycopg2 Connect Arguments

psycopg2-specific keyword arguments which are accepted by *create_engine()* are:

- `server_side_cursors`: Enable the usage of "server side cursors" for SQL statements which support this feature. What this essentially means from a psycopg2 point of view is that the cursor is created using a name, e.g. `connection.cursor('some name')`, which has the effect that result rows are not immediately pre-fetched and buffered after statement execution, but are instead left on the server and only retrieved as needed. SQLAlchemy's *ResultProxy* uses special row-buffering behavior when this feature is enabled, such that groups of 100 rows at a time are fetched over the wire to reduce conversational overhead. Note that the `stream_results=True` execution option is a more targeted way of enabling this mode on a per-execution basis.
- `use_native_unicode`: Enable the usage of Psycopg2 "native unicode" mode per connection. True by default.

  See also:

  Disabling Native Unicode

- `isolation_level`: This option, available for all PostgreSQL dialects, includes the `AUTOCOMMIT` isolation level when using the psycopg2 dialect.

  See also:

  Psycopg2 Transaction Isolation Level

- `client_encoding`: sets the client encoding in a libpq-agnostic way, using psycopg2's `set_client_encoding()` method.

  See also:

Unicode with Psycopg2

Unix Domain Connections

psycopg2 supports connecting via Unix domain connections. When the `host` portion of the URL is omitted, SQLAlchemy passes `None` to psycopg2, which specifies Unix-domain communication rather than TCP/IP communication:

```
create_engine("postgresql+psycopg2://user:password@/dbname")
```

By default, the socket file used is to connect to a Unix-domain socket in `/tmp`, or whatever socket directory was specified when PostgreSQL was built. This value can be overridden by passing a pathname to psycopg2, using `host` as an additional keyword argument:

```
create_engine("postgresql+psycopg2://user:password@/dbname?host=/var/lib/postgresql")
```

See also:

PQconnectdbParams

Per-Statement/Connection Execution Options

The following DBAPI-specific options are respected when used with *`Connection.execution_options()`*, *`Executable.execution_options()`*, *`Query.execution_options()`*, in addition to those not specific to DBAPIs:

- `isolation_level` - Set the transaction isolation level for the lifespan of a *`Connection`* (can only be set on a connection, not a statement or query). See *Psycopg2 Transaction Isolation Level*.
- `stream_results` - Enable or disable usage of psycopg2 server side cursors - this feature makes use of "named" cursors in combination with special result handling methods so that result rows are not fully buffered. If `None` or not set, the `server_side_cursors` option of the *`Engine`* is used.
- `max_row_buffer` - when using `stream_results`, an integer value that specifies the maximum number of rows to buffer at a time. This is interpreted by the `BufferedRowResultProxy`, and if omitted the buffer will grow to ultimately store 1000 rows at a time.

  New in version 1.0.6.

Unicode with Psycopg2

By default, the psycopg2 driver uses the `psycopg2.extensions.UNICODE` extension, such that the DBAPI receives and returns all strings as Python Unicode objects directly - SQLAlchemy passes these values through without change. Psycopg2 here will encode/decode string values based on the current "client encoding" setting; by default this is the value in the `postgresql.conf` file, which often defaults to `SQL_ASCII`. Typically, this can be changed to `utf8`, as a more useful default:

```
# postgresql.conf file

# client_encoding = sql_ascii # actually, defaults to database
                              # encoding
client_encoding = utf8
```

A second way to affect the client encoding is to set it within Psycopg2 locally. SQLAlchemy will call psycopg2's `connection.set_client_encoding()` method on all new connections based on the value passed to *create_engine()* using the `client_encoding` parameter:

```
# set_client_encoding() setting;
# works for *all* Postgresql versions
engine = create_engine("postgresql://user:pass@host/dbname",
                       client_encoding='utf8')
```

This overrides the encoding specified in the Postgresql client configuration. When using the parameter in this way, the psycopg2 driver emits `SET client_encoding TO 'utf8'` on the connection explicitly, and works in all Postgresql versions.

Note that the `client_encoding` setting as passed to *create_engine()* is **not the same** as the more recently added `client_encoding` parameter now supported by libpq directly. This is enabled when `client_encoding` is passed directly to `psycopg2.connect()`, and from SQLAlchemy is passed using the *create_engine.connect_args* parameter:

```
# libpq direct parameter setting;
# only works for Postgresql **9.1 and above**
engine = create_engine("postgresql://user:pass@host/dbname",
                       connect_args={'client_encoding': 'utf8'})

# using the query string is equivalent
engine = create_engine("postgresql://user:pass@host/dbname?client_encoding=utf8")
```

The above parameter was only added to libpq as of version 9.1 of Postgresql, so using the previous method is better for cross-version support.

Disabling Native Unicode SQLAlchemy can also be instructed to skip the usage of the psycopg2 `UNICODE` extension and to instead utilize its own unicode encode/decode services, which are normally reserved only for those DBAPIs that don't fully support unicode directly. Passing `use_native_unicode=False` to *create_engine()* will disable usage of `psycopg2.extensions.UNICODE`. SQLAlchemy will instead encode data itself into Python bytestrings on the way in and coerce from bytes on the way back, using the value of the *create_engine()* `encoding` parameter, which defaults to `utf-8`. SQLAlchemy's own unicode encode/decode functionality is steadily becoming obsolete as most DBAPIs now support unicode fully.

Bound Parameter Styles

The default parameter style for the psycopg2 dialect is "pyformat", where SQL is rendered using `%(paramname)s` style. This format has the limitation that it does not accommodate the unusual case of parameter names that actually contain percent or parenthesis symbols; as SQLAlchemy in many cases generates bound parameter names based on the name of a column, the presence of these characters in a column name can lead to problems.

There are two solutions to the issue of a *schema.Column* that contains one of these characters in its name. One is to specify the *schema.Column.key* for columns that have such names:

```
measurement = Table('measurement', metadata,
    Column('Size (meters)', Integer, key='size_meters')
)
```

Above, an INSERT statement such as `measurement.insert()` will use `size_meters` as the parameter name, and a SQL expression such as `measurement.c.size_meters > 10` will derive the bound parameter name from the `size_meters` key as well.

Changed in version 1.0.0: - SQL expressions will use `Column.key` as the source of naming when anonymous bound parameters are created in SQL expressions; previously, this behavior only applied to *`Table.insert()`* and *`Table.update()`* parameter names.

The other solution is to use a positional format; psycopg2 allows use of the "format" paramstyle, which can be passed to *`create_engine.paramstyle`*:

```
engine = create_engine(
    'postgresql://scott:tiger@localhost:5432/test', paramstyle='format')
```

With the above engine, instead of a statement like:

```
INSERT INTO measurement ("Size (meters)") VALUES (%(Size (meters))s)
{'Size (meters)': 1}
```

we instead see:

```
INSERT INTO measurement ("Size (meters)") VALUES (%s)
(1, )
```

Where above, the dictionary style is converted into a tuple with positional style.

Transactions

The psycopg2 dialect fully supports SAVEPOINT and two-phase commit operations.

Psycopg2 Transaction Isolation Level

As discussed in *Transaction Isolation Level*, all Postgresql dialects support setting of transaction isolation level both via the `isolation_level` parameter passed to *`create_engine()`*, as well as the `isolation_level` argument used by *`Connection.execution_options()`*. When using the psycopg2 dialect, these options make use of psycopg2's `set_isolation_level()` connection method, rather than emitting a Postgresql directive; this is because psycopg2's API-level setting is always emitted at the start of each transaction in any case.

The psycopg2 dialect supports these constants for isolation level:

- `READ COMMITTED`
- `READ UNCOMMITTED`
- `REPEATABLE READ`
- `SERIALIZABLE`
- `AUTOCOMMIT`

New in version 0.8.2: support for AUTOCOMMIT isolation level when using psycopg2.

See also:

Transaction Isolation Level

pg8000 Transaction Isolation Level

NOTICE logging

The psycopg2 dialect will log Postgresql NOTICE messages via the `sqlalchemy.dialects.postgresql` logger:

```
import logging
logging.getLogger('sqlalchemy.dialects.postgresql').setLevel(logging.INFO)
```

HSTORE type

The `psycopg2` DBAPI includes an extension to natively handle marshalling of the HSTORE type. The SQLAlchemy psycopg2 dialect will enable this extension by default when psycopg2 version 2.4 or greater is used, and it is detected that the target database has the HSTORE type set up for use. In other words, when the dialect makes the first connection, a sequence like the following is performed:

1. Request the available HSTORE oids using `psycopg2.extras.HstoreAdapter.get_oids()`. If this function returns a list of HSTORE identifiers, we then determine that the `HSTORE` extension is present. This function is **skipped** if the version of psycopg2 installed is less than version 2.4.
2. If the `use_native_hstore` flag is at its default of `True`, and we've detected that `HSTORE` oids are available, the `psycopg2.extensions.register_hstore()` extension is invoked for all connections.

The `register_hstore()` extension has the effect of **all Python dictionaries being accepted as parameters regardless of the type of target column in SQL**. The dictionaries are converted by this extension into a textual HSTORE expression. If this behavior is not desired, disable the use of the hstore extension by setting `use_native_hstore` to `False` as follows:

```
engine = create_engine("postgresql+psycopg2://scott:tiger@localhost/test",
            use_native_hstore=False)
```

The `HSTORE` type is **still supported** when the `psycopg2.extensions.register_hstore()` extension is not used. It merely means that the coercion between Python dictionaries and the HSTORE string format, on both the parameter side and the result side, will take place within SQLAlchemy's own marshalling logic, and not that of `psycopg2` which may be more performant.

pg8000

Support for the PostgreSQL database via the pg8000 driver.

DBAPI

Documentation and download information (if applicable) for pg8000 is available at: https://pythonhosted.org/pg8000/

Connecting

Connect String:

```
postgresql+pg8000://user:password@host:port/dbname[?key=value&key=value...]
```

Unicode

pg8000 will encode / decode string values between it and the server using the PostgreSQL `client_encoding` parameter; by default this is the value in the `postgresql.conf` file, which often defaults to `SQL_ASCII`. Typically, this can be changed to `utf-8`, as a more useful default:

```
#client_encoding = sql_ascii # actually, defaults to database
                             # encoding
client_encoding = utf8
```

The `client_encoding` can be overriden for a session by executing the SQL:

SET CLIENT_ENCODING TO 'utf8';

SQLAlchemy will execute this SQL on all new connections based on the value passed to *`create_engine()`* using the `client_encoding` parameter:

```
engine = create_engine(
    "postgresql+pg8000://user:pass@host/dbname", client_encoding='utf8')
```

pg8000 Transaction Isolation Level

The pg8000 dialect offers the same isolation level settings as that of the *psycopg2* dialect:

- `READ COMMITTED`
- `READ UNCOMMITTED`
- `REPEATABLE READ`
- `SERIALIZABLE`
- `AUTOCOMMIT`

New in version 0.9.5: support for AUTOCOMMIT isolation level when using pg8000.

See also:

Transaction Isolation Level

Psycopg2 Transaction Isolation Level

psycopg2cffi

Support for the PostgreSQL database via the psycopg2cffi driver.

DBAPI

Documentation and download information (if applicable) for psycopg2cffi is available at: http://pypi.python.org/pypi/psycopg2cffi/

Connecting

Connect String:

```
postgresql+psycopg2cffi://user:password@host:port/dbname[?key=value&key=value...]
```

`psycopg2cffi` is an adaptation of `psycopg2`, using CFFI for the C layer. This makes it suitable for use in e.g. PyPy. Documentation is as per `psycopg2`.

New in version 1.0.0.

See also:

`sqlalchemy.dialects.postgresql.psycopg2`

py-postgresql

Support for the PostgreSQL database via the py-postgresql driver.

DBAPI

Documentation and download information (if applicable) for py-postgresql is available at: http://python.projects.pgfoundry.org/

Connecting

Connect String:

```
postgresql+pypostgresql://user:password@host:port/dbname[?key=value&key=value...]
```

zxjdbc

Support for the PostgreSQL database via the zxJDBC for Jython driver.

DBAPI

Drivers for this database are available at: http://jdbc.postgresql.org/

Connecting

Connect String:

```
postgresql+zxjdbc://scott:tiger@localhost/db
```

4.1.6 SQLite

Support for the SQLite database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- *pysqlite*
- *pysqlcipher*

Date and Time Types

SQLite does not have built-in DATE, TIME, or DATETIME types, and pysqlite does not provide out of the box functionality for translating values between Python *datetime* objects and a SQLite-supported format. SQLAlchemy's own `DateTime` and related types provide date formatting and parsing functionality when SQlite is used. The implementation classes are `DATETIME`, `DATE` and `TIME`. These types represent dates and times as ISO formatted strings, which also nicely support ordering. There's no reliance on typical "libc" internals for these functions so historical dates are fully supported.

Ensuring Text affinity

The DDL rendered for these types is the standard `DATE`, `TIME` and `DATETIME` indicators. However, custom storage formats can also be applied to these types. When the storage format is detected as containing no alpha characters, the DDL for these types is rendered as `DATE_CHAR`, `TIME_CHAR`, and `DATETIME_CHAR`, so that the column continues to have textual affinity.

See also:

Type Affinity - in the SQLite documentation

SQLite Auto Incrementing Behavior

Background on SQLite's autoincrement is at: http://sqlite.org/autoinc.html

Key concepts:

- SQLite has an implicit "auto increment" feature that takes place for any non-composite primary-key column that is specifically created using "INTEGER PRIMARY KEY" for the type + primary key.
- SQLite also has an explicit "AUTOINCREMENT" keyword, that is **not** equivalent to the implicit autoincrement feature; this keyword is not recommended for general use. SQLAlchemy does not render this keyword unless a special SQLite-specific directive is used (see below). However, it still requires that the column's type is named "INTEGER".

Using the AUTOINCREMENT Keyword

To specifically render the AUTOINCREMENT keyword on the primary key column when rendering DDL, add the flag `sqlite_autoincrement=True` to the Table construct:

```
Table('sometable', metadata,
        Column('id', Integer, primary_key=True),
        sqlite_autoincrement=True)
```

Allowing autoincrement behavior SQLAlchemy types other than Integer/INTEGER

SQLite's typing model is based on naming conventions. Among other things, this means that any type name which contains the substring `"INT"` will be determined to be of "integer affinity". A type named `"BIGINT"`, `"SPECIAL_INT"` or even `"XYZINTQPR"`, will be considered by SQLite to be of "integer" affinity. However, **the SQLite autoincrement feature, whether implicitly or explicitly enabled, requires that the name of the column's type is exactly the string "INTEGER"**. Therefore, if an application uses a type like *BigInteger* for a primary key, on SQLite this type will need to be rendered as the name `"INTEGER"` when emitting the initial `CREATE TABLE` statement in order for the autoincrement behavior to be available.

One approach to achieve this is to use *Integer* on SQLite only using *TypeEngine.with_variant()*:

```
table = Table(
    "my_table", metadata,
    Column("id", BigInteger().with_variant(Integer, "sqlite"), primary_key=True)
)
```

Another is to use a subclass of *BigInteger* that overrides its DDL name to be `INTEGER` when compiled against SQLite:

```
from sqlalchemy import BigInteger
from sqlalchemy.ext.compiler import compiles

class SLBigInteger(BigInteger):
    pass

@compiles(SLBigInteger, 'sqlite')
def bi_c(element, compiler, **kw):
    return "INTEGER"

@compiles(SLBigInteger)
def bi_c(element, compiler, **kw):
    return compiler.visit_BIGINT(element, **kw)


table = Table(
    "my_table", metadata,
    Column("id", SLBigInteger(), primary_key=True)
)
```

See also:

TypeEngine.with_variant()

Custom SQL Constructs and Compilation Extension

Datatypes In SQLite Version 3

Database Locking Behavior / Concurrency

SQLite is not designed for a high level of write concurrency. The database itself, being a file, is locked completely during write operations within transactions, meaning exactly one "connection" (in reality a file handle) has exclusive access to the database during this period - all other "connections" will be blocked during this time.

The Python DBAPI specification also calls for a connection model that is always in a transaction; there is no `connection.begin()` method, only `connection.commit()` and `connection.rollback()`, upon which a new transaction is to be begun immediately. This may seem to imply that the SQLite driver would in theory

allow only a single filehandle on a particular database file at any time; however, there are several factors both within SQlite itself as well as within the pysqlite driver which loosen this restriction significantly.

However, no matter what locking modes are used, SQLite will still always lock the database file once a transaction is started and DML (e.g. INSERT, UPDATE, DELETE) has at least been emitted, and this will block other transactions at least at the point that they also attempt to emit DML. By default, the length of time on this block is very short before it times out with an error.

This behavior becomes more critical when used in conjunction with the SQLAlchemy ORM. SQLAlchemy's `Session` object by default runs within a transaction, and with its autoflush model, may emit DML preceding any SELECT statement. This may lead to a SQLite database that locks more quickly than is expected. The locking mode of SQLite and the pysqlite driver can be manipulated to some degree, however it should be noted that achieving a high degree of write-concurrency with SQLite is a losing battle.

For more information on SQLite's lack of write concurrency by design, please see Situations Where Another RDBMS May Work Better - High Concurrency near the bottom of the page.

The following subsections introduce areas that are impacted by SQLite's file-based architecture and additionally will usually require workarounds to work when using the pysqlite driver.

Transaction Isolation Level

SQLite supports "transaction isolation" in a non-standard way, along two axes. One is that of the PRAGMA read_uncommitted instruction. This setting can essentially switch SQLite between its default mode of `SERIALIZABLE` isolation, and a "dirty read" isolation mode normally referred to as `READ UNCOMMITTED`.

SQLAlchemy ties into this PRAGMA statement using the `create_engine.isolation_level` parameter of `create_engine()`. Valid values for this parameter when used with SQLite are `"SERIALIZABLE"` and `"READ UNCOMMITTED"` corresponding to a value of 0 and 1, respectively. SQLite defaults to `SERIALIZABLE`, however its behavior is impacted by the pysqlite driver's default behavior.

The other axis along which SQLite's transactional locking is impacted is via the nature of the `BEGIN` statement used. The three varieties are "deferred", "immediate", and "exclusive", as described at BEGIN TRANSACTION. A straight `BEGIN` statement uses the "deferred" mode, where the the database file is not locked until the first read or write operation, and read access remains open to other transactions until the first write operation. But again, it is critical to note that the pysqlite driver interferes with this behavior by *not even emitting BEGIN* until the first write operation.

> **Warning:** SQLite's transactional scope is impacted by unresolved issues in the pysqlite driver, which defers BEGIN statements to a greater degree than is often feasible. See the section *Serializable isolation / Savepoints / Transactional DDL* for techniques to work around this behavior.

SAVEPOINT Support

SQLite supports SAVEPOINTs, which only function once a transaction is begun. SQLAlchemy's SAVEPOINT support is available using the `Connection.begin_nested()` method at the Core level, and `Session.begin_nested()` at the ORM level. However, SAVEPOINTs won't work at all with pysqlite unless workarounds are taken.

> **Warning:** SQLite's SAVEPOINT feature is impacted by unresolved issues in the pysqlite driver, which defers BEGIN statements to a greater degree than is often feasible. See the section *Serializable isolation / Savepoints / Transactional DDL* for techniques to work around this behavior.

Transactional DDL

The SQLite database supports transactional DDL as well. In this case, the pysqlite driver is not only failing to start transactions, it also is ending any existing transction when DDL is detected, so again, workarounds are required.

> **Warning:** SQLite's transactional DDL is impacted by unresolved issues in the pysqlite driver, which fails to emit BEGIN and additionally forces a COMMIT to cancel any transaction when DDL is encountered. See the section *Serializable isolation / Savepoints / Transactional DDL* for techniques to work around this behavior.

Foreign Key Support

SQLite supports FOREIGN KEY syntax when emitting CREATE statements for tables, however by default these constraints have no effect on the operation of the table.

Constraint checking on SQLite has three prerequisites:

- At least version 3.6.19 of SQLite must be in use
- The SQLite library must be compiled *without* the SQLITE_OMIT_FOREIGN_KEY or SQLITE_OMIT_TRIGGER symbols enabled.
- The `PRAGMA foreign_keys = ON` statement must be emitted on all connections before use.

SQLAlchemy allows for the `PRAGMA` statement to be emitted automatically for new connections through the usage of events:

```
from sqlalchemy.engine import Engine
from sqlalchemy import event

@event.listens_for(Engine, "connect")
def set_sqlite_pragma(dbapi_connection, connection_record):
    cursor = dbapi_connection.cursor()
    cursor.execute("PRAGMA foreign_keys=ON")
    cursor.close()
```

> **Warning:** When SQLite foreign keys are enabled, it is **not possible** to emit CREATE or DROP statements for tables that contain mutually-dependent foreign key constraints; to emit the DDL for these tables requires that ALTER TABLE be used to create or drop these constraints separately, for which SQLite has no support.

See also:

SQLite Foreign Key Support - on the SQLite web site.

Events - SQLAlchemy event API.

Creating/Dropping Foreign Key Constraints via ALTER **- more information on SQLAlchemy's facilities for handling** mutually-dependent foreign key constraints.

Type Reflection

SQLite types are unlike those of most other database backends, in that the string name of the type usually does not correspond to a "type" in a one-to-one fashion. Instead, SQLite links per-column typing behavior to one of five so-called "type affinities" based on a string matching pattern for the type.

SQLAlchemy's reflection process, when inspecting types, uses a simple lookup table to link the keywords returned to provided SQLAlchemy types. This lookup table is present within the SQLite dialect as it is for all other dialects.

However, the SQLite dialect has a different "fallback" routine for when a particular type name is not located in the lookup map; it instead implements the SQLite "type affinity" scheme located at http://www.sqlite.org/datatype3.html section 2.1.

The provided typemap will make direct associations from an exact string name match for the following types:

BIGINT, *BLOB*, *BOOLEAN*, *BOOLEAN*, *CHAR*, *DATE*, *DATETIME*, *FLOAT*, *DECIMAL*, *FLOAT*, *INTEGER*, *INTEGER*, *NUMERIC*, *REAL*, *SMALLINT*, *TEXT*, *TIME*, *TIMESTAMP*, *VARCHAR*, *NVARCHAR*, *NCHAR*

When a type name does not match one of the above types, the "type affinity" lookup is used instead:

- *INTEGER* is returned if the type name includes the string `INT`
- *TEXT* is returned if the type name includes the string `CHAR`, `CLOB` or `TEXT`
- *NullType* is returned if the type name includes the string `BLOB`
- *REAL* is returned if the type name includes the string `REAL`, `FLOA` or `DOUB`.
- Otherwise, the *NUMERIC* type is used.

New in version 0.9.3: Support for SQLite type affinity rules when reflecting columns.

Partial Indexes

A partial index, e.g. one which uses a WHERE clause, can be specified with the DDL system using the argument `sqlite_where`:

```
tbl = Table('testtbl', m, Column('data', Integer))
idx = Index('test_idx1', tbl.c.data,
            sqlite_where=and_(tbl.c.data > 5, tbl.c.data < 10))
```

The index will be rendered at create time as:

```
CREATE INDEX test_idx1 ON testtbl (data)
WHERE data > 5 AND data < 10
```

New in version 0.9.9.

Dotted Column Names

Using table or column names that explicitly have periods in them is **not recommended**. While this is generally a bad idea for relational databases in general, as the dot is a syntactically significant character, the SQLite driver has a bug which requires that SQLAlchemy filter out these dots in result sets.

The bug, entirely outside of SQLAlchemy, can be illustrated thusly:

```
import sqlite3

conn = sqlite3.connect(":memory:")
cursor = conn.cursor()

cursor.execute("create table x (a integer, b integer)")
cursor.execute("insert into x (a, b) values (1, 1)")
cursor.execute("insert into x (a, b) values (2, 2)")

cursor.execute("select x.a, x.b from x")
assert [c[0] for c in cursor.description] == ['a', 'b']
```

```
cursor.execute('''
    select x.a, x.b from x where a=1
    union
    select x.a, x.b from x where a=2
''')
assert [c[0] for c in cursor.description] == ['a', 'b'], \
    [c[0] for c in cursor.description]
```

The second assertion fails:

```
Traceback (most recent call last):
  File "test.py", line 19, in <module>
    [c[0] for c in cursor.description]
AssertionError: ['x.a', 'x.b']
```

Where above, the driver incorrectly reports the names of the columns including the name of the table, which is entirely inconsistent vs. when the UNION is not present.

SQLAlchemy relies upon column names being predictable in how they match to the original statement, so the SQLAlchemy dialect has no choice but to filter these out:

```
from sqlalchemy import create_engine

eng = create_engine("sqlite://")
conn = eng.connect()

conn.execute("create table x (a integer, b integer)")
conn.execute("insert into x (a, b) values (1, 1)")
conn.execute("insert into x (a, b) values (2, 2)")

result = conn.execute("select x.a, x.b from x")
assert result.keys() == ["a", "b"]

result = conn.execute('''
    select x.a, x.b from x where a=1
    union
    select x.a, x.b from x where a=2
''')
assert result.keys() == ["a", "b"]
```

Note that above, even though SQLAlchemy filters out the dots, *both names are still addressable*:

```
>>> row = result.first()
>>> row["a"]
1
>>> row["x.a"]
1
>>> row["b"]
1
>>> row["x.b"]
1
```

Therefore, the workaround applied by SQLAlchemy only impacts `ResultProxy.keys()` and `RowProxy.keys()` in the public API. In the very specific case where an application is forced to use column names that contain dots, and the functionality of `ResultProxy.keys()` and `RowProxy.keys()` is required to return these dotted names unmodified, the `sqlite_raw_colnames` execution option may be provided, either on a per-`Connection` basis:

```
result = conn.execution_options(sqlite_raw_colnames=True).execute('''
    select x.a, x.b from x where a=1
    union
    select x.a, x.b from x where a=2
''')
assert result.keys() == ["x.a", "x.b"]
```

or on a per-*Engine* basis:

```
engine = create_engine("sqlite://", execution_options={"sqlite_raw_colnames": True})
```

When using the per-*Engine* execution option, note that **Core and ORM queries that use UNION may not function properly**.

SQLite Data Types

As with all SQLAlchemy dialects, all UPPERCASE types that are known to be valid with SQLite are importable from the top level dialect, whether they originate from *sqlalchemy.types* or from the local dialect:

```
from sqlalchemy.dialects.sqlite import \
            BLOB, BOOLEAN, CHAR, DATE, DATETIME, DECIMAL, FLOAT, \
            INTEGER, NUMERIC, SMALLINT, TEXT, TIME, TIMESTAMP, \
            VARCHAR
```

class sqlalchemy.dialects.sqlite.**DATETIME**(**args*, ***kwargs*)

Bases: sqlalchemy.dialects.sqlite.base._DateTimeMixin, *sqlalchemy.types.DateTime*

Represent a Python datetime object in SQLite using a string.

The default string storage format is:

```
"%(year)04d-%(month)02d-%(day)02d %(hour)02d:%(min)02d:%(second)02d.%(microsecond)06d"
```

e.g.:

```
2011-03-15 12:05:57.10558
```

The storage format can be customized to some degree using the `storage_format` and `regexp` parameters, such as:

```
import re
from sqlalchemy.dialects.sqlite import DATETIME

dt = DATETIME(
    storage_format="%(year)04d/%(month)02d/%(day)02d %(hour)02d:%(min)02d:%(second)02d",
    regexp=r"(\d+)/(\d+)/(\d+) (\d+)-(\d+)-(\d+)"
)
```

Parameters

- **storage_format** – format string which will be applied to the dict with keys year, month, day, hour, minute, second, and microsecond.

- **regexp** – regular expression which will be applied to incoming result rows. If the regexp contains named groups, the resulting match dict is applied to the Python datetime() constructor as keyword arguments. Otherwise, if positional groups are used, the datetime() constructor is called with positional arguments via `*map(int, match_obj.groups(0))`.

class `sqlalchemy.dialects.sqlite.`**`DATE`**(*storage_format=None*, *regexp=None*, ***kw*)

Bases: `sqlalchemy.dialects.sqlite.base._DateTimeMixin`, *`sqlalchemy.types.Date`*

Represent a Python date object in SQLite using a string.

The default string storage format is:

```
"%(year)04d-%(month)02d-%(day)02d"
```

e.g.:

```
2011-03-15
```

The storage format can be customized to some degree using the `storage_format` and `regexp` parameters, such as:

```
import re
from sqlalchemy.dialects.sqlite import DATE

d = DATE(
        storage_format="%(month)02d/%(day)02d/%(year)04d",
        regexp=re.compile("(?P<month>\d+)/(?P<day>\d+)/(?P<year>\d+)")
    )
```

Parameters

- **storage_format** – format string which will be applied to the dict with keys year, month, and day.
- **regexp** – regular expression which will be applied to incoming result rows. If the regexp contains named groups, the resulting match dict is applied to the Python date() constructor as keyword arguments. Otherwise, if positional groups are used, the date() constructor is called with positional arguments via `*map(int, match_obj.groups(0))`.

class `sqlalchemy.dialects.sqlite.`**`TIME`**(**args*, ***kwargs*)

Bases: `sqlalchemy.dialects.sqlite.base._DateTimeMixin`, *`sqlalchemy.types.Time`*

Represent a Python time object in SQLite using a string.

The default string storage format is:

```
"%(hour)02d:%(minute)02d:%(second)02d.%(microsecond)06d"
```

e.g.:

```
12:05:57.10558
```

The storage format can be customized to some degree using the `storage_format` and `regexp` parameters, such as:

```
import re
from sqlalchemy.dialects.sqlite import TIME

t = TIME(
    storage_format="%(hour)02d-%(minute)02d-%(second)02d-%(microsecond)06d",
    regexp=re.compile("(\d+)-(\d+)-(\d+)-(?:-(\d+))?")
)
```

Parameters

- **storage_format** – format string which will be applied to the dict with keys hour, minute, second, and microsecond.
- **regexp** – regular expression which will be applied to incoming result rows. If the regexp contains named groups, the resulting match dict is applied to the Python time() constructor as keyword arguments. Otherwise, if positional groups are used, the time() constructor is called with positional arguments via `*map(int, match_obj.groups(0))`.

Pysqlite

Support for the SQLite database via the pysqlite driver. Note that `pysqlite` is the same driver as the `sqlite3` module included with the Python distribution.

DBAPI

Documentation and download information (if applicable) for pysqlite is available at: http://docs.python.org/library/sqlite3.html

Connecting

Connect String:

```
sqlite+pysqlite:///file_path
```

Driver

When using Python 2.5 and above, the built in `sqlite3` driver is already installed and no additional installation is needed. Otherwise, the `pysqlite2` driver needs to be present. This is the same driver as `sqlite3`, just with a different name.

The `pysqlite2` driver will be loaded first, and if not found, `sqlite3` is loaded. This allows an explicitly installed pysqlite driver to take precedence over the built in one. As with all dialects, a specific DBAPI module may be provided to *create_engine()* to control this explicitly:

```
from sqlite3 import dbapi2 as sqlite
e = create_engine('sqlite+pysqlite:///file.db', module=sqlite)
```

Connect Strings

The file specification for the SQLite database is taken as the "database" portion of the URL. Note that the format of a SQLAlchemy url is:

```
driver://user:pass@host/database
```

This means that the actual filename to be used starts with the characters to the **right** of the third slash. So connecting to a relative filepath looks like:

```
# relative path
e = create_engine('sqlite:///path/to/database.db')
```

An absolute path, which is denoted by starting with a slash, means you need **four** slashes:

```
# absolute path
e = create_engine('sqlite:////path/to/database.db')
```

To use a Windows path, regular drive specifications and backslashes can be used. Double backslashes are probably needed:

```
# absolute path on Windows
e = create_engine('sqlite:///C:\\path\\to\\database.db')
```

The sqlite `:memory:` identifier is the default if no filepath is present. Specify `sqlite://` and nothing else:

```
# in-memory database
e = create_engine('sqlite://')
```

Compatibility with sqlite3 "native" date and datetime types

The pysqlite driver includes the sqlite3.PARSE_DECLTYPES and sqlite3.PARSE_COLNAMES options, which have the effect of any column or expression explicitly cast as "date" or "timestamp" will be converted to a Python date or datetime object. The date and datetime types provided with the pysqlite dialect are not currently compatible with these options, since they render the ISO date/datetime including microseconds, which pysqlite's driver does not. Additionally, SQLAlchemy does not at this time automatically render the "cast" syntax required for the freestanding functions "current_timestamp" and "current_date" to return datetime/date types natively. Unfortunately, pysqlite does not provide the standard DBAPI types in `cursor.description`, leaving SQLAlchemy with no way to detect these types on the fly without expensive per-row type checks.

Keeping in mind that pysqlite's parsing option is not recommended, nor should be necessary, for use with SQLAlchemy, usage of PARSE_DECLTYPES can be forced if one configures "native_datetime=True" on create_engine():

```
engine = create_engine('sqlite://',
    connect_args={'detect_types':
        sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_COLNAMES},
    native_datetime=True
)
```

With this flag enabled, the DATE and TIMESTAMP types (but note - not the DATETIME or TIME types...confused yet ?) will not perform any bind parameter or result processing. Execution of "func.current_date()" will return a string. "func.current_timestamp()" is registered as returning a DATETIME type in SQLAlchemy, so this function still receives SQLAlchemy-level result processing.

Threading/Pooling Behavior

Pysqlite's default behavior is to prohibit the usage of a single connection in more than one thread. This is originally intended to work with older versions of SQLite that did not support multithreaded operation under various circumstances. In particular, older SQLite versions did not allow a `:memory:` database to be used in multiple threads under any circumstances.

Pysqlite does include a now-undocumented flag known as `check_same_thread` which will disable this check, however note that pysqlite connections are still not safe to use in concurrently in multiple threads. In particular, any statement execution calls would need to be externally mutexed, as Pysqlite does not provide for thread-safe propagation of error messages among other things. So while even `:memory:` databases can be shared among threads in modern SQLite, Pysqlite doesn't provide enough thread-safety to make this usage worth it.

SQLAlchemy sets up pooling to work with Pysqlite's default behavior:

- When a `:memory:` SQLite database is specified, the dialect by default will use *SingletonThreadPool*. This pool maintains a single connection per thread, so that all access to the engine within the current thread use the same `:memory:` database - other threads would access a different `:memory:` database.
- When a file-based database is specified, the dialect will use *NullPool* as the source of connections. This pool closes and discards connections which are returned to the pool immediately. SQLite file-based connections have extremely low overhead, so pooling is not necessary. The scheme also prevents a connection from being used again in a different thread and works best with SQLite's coarse-grained file locking.

  Changed in version 0.7: Default selection of *NullPool* for SQLite file-based databases. Previous versions select *SingletonThreadPool* by default for all SQLite databases.

Using a Memory Database in Multiple Threads To use a `:memory:` database in a multithreaded scenario, the same connection object must be shared among threads, since the database exists only within the scope of that connection. The *StaticPool* implementation will maintain a single connection globally, and the `check_same_thread` flag can be passed to Pysqlite as `False`:

```
from sqlalchemy.pool import StaticPool
engine = create_engine('sqlite://',
                       connect_args={'check_same_thread':False},
                       poolclass=StaticPool)
```

Note that using a `:memory:` database in multiple threads requires a recent version of SQLite.

Using Temporary Tables with SQLite Due to the way SQLite deals with temporary tables, if you wish to use a temporary table in a file-based SQLite database across multiple checkouts from the connection pool, such as when using an ORM *Session* where the temporary table should continue to remain after *Session.commit()* or *Session.rollback()* is called, a pool which maintains a single connection must be used. Use *SingletonThreadPool* if the scope is only needed within the current thread, or *StaticPool* is scope is needed within multiple threads for this case:

```
# maintain the same connection per thread
from sqlalchemy.pool import SingletonThreadPool
engine = create_engine('sqlite:///mydb.db',
                       poolclass=SingletonThreadPool)


# maintain the same connection across all threads
from sqlalchemy.pool import StaticPool
engine = create_engine('sqlite:///mydb.db',
                       poolclass=StaticPool)
```

Note that *SingletonThreadPool* should be configured for the number of threads that are to be used; beyond that number, connections will be closed out in a non deterministic way.

Unicode

The pysqlite driver only returns Python `unicode` objects in result sets, never plain strings, and accommodates `unicode` objects within bound parameter values in all cases. Regardless of the SQLAlchemy string type in use, string-based result values will by Python `unicode` in Python 2. The *Unicode* type should still be used to indicate those columns that require unicode, however, so that non-`unicode` values passed inadvertently will emit a warning. Pysqlite will emit an error if a non-`unicode` string is passed containing non-ASCII characters.

Serializable isolation / Savepoints / Transactional DDL

In the section *Database Locking Behavior / Concurrency*, we refer to the pysqlite driver's assortment of issues that prevent several features of SQLite from working correctly. The pysqlite DBAPI driver has several long-standing bugs which impact the correctness of its transactional behavior. In its default mode of operation, SQLite features such as SERIALIZABLE isolation, transactional DDL, and SAVEPOINT support are non-functional, and in order to use these features, workarounds must be taken.

The issue is essentially that the driver attempts to second-guess the user's intent, failing to start transactions and sometimes ending them prematurely, in an effort to minimize the SQLite databases's file locking behavior, even though SQLite itself uses "shared" locks for read-only activities.

SQLAlchemy chooses to not alter this behavior by default, as it is the long-expected behavior of the pysqlite driver; if and when the pysqlite driver attempts to repair these issues, that will be more of a driver towards defaults for SQLAlchemy.

The good news is that with a few events, we can implement transactional support fully, by disabling pysqlite's feature entirely and emitting BEGIN ourselves. This is achieved using two event listeners:

```
from sqlalchemy import create_engine, event

engine = create_engine("sqlite:///myfile.db")

@event.listens_for(engine, "connect")
def do_connect(dbapi_connection, connection_record):
    # disable pysqlite's emitting of the BEGIN statement entirely.
    # also stops it from emitting COMMIT before any DDL.
    dbapi_connection.isolation_level = None

@event.listens_for(engine, "begin")
def do_begin(conn):
    # emit our own BEGIN
    conn.execute("BEGIN")
```

Above, we intercept a new pysqlite connection and disable any transactional integration. Then, at the point at which SQLAlchemy knows that transaction scope is to begin, we emit `"BEGIN"` ourselves.

When we take control of `"BEGIN"`, we can also control directly SQLite's locking modes, introduced at BEGIN TRANSACTION, by adding the desired locking mode to our `"BEGIN"`:

```
@event.listens_for(engine, "begin")
def do_begin(conn):
    conn.execute("BEGIN EXCLUSIVE")
```

See also:

BEGIN TRANSACTION - on the SQLite site

sqlite3 SELECT does not BEGIN a transaction - on the Python bug tracker

sqlite3 module breaks transactions and potentially corrupts data - on the Python bug tracker

Pysqlcipher

Support for the SQLite database via the pysqlcipher driver. `pysqlcipher` is a fork of the standard `pysqlite` driver to make use of the SQLCipher backend.

New in version 0.9.9.

DBAPI

Documentation and download information (if applicable) for pysqlcipher is available at: https://pypi.python.org/pypi/pysqlcipher

Connecting

Connect String:

```
sqlite+pysqlcipher://:passphrase/file_path[?kdf_iter=<iter>]
```

Driver

The driver here is the pysqlcipher driver, which makes use of the SQLCipher engine. This system essentially introduces new PRAGMA commands to SQLite which allows the setting of a passphrase and other encryption parameters, allowing the database file to be encrypted.

Connect Strings

The format of the connect string is in every way the same as that of the *`pysqlite`* driver, except that the "password" field is now accepted, which should contain a passphrase:

```
e = create_engine('sqlite+pysqlcipher://:testing@/foo.db')
```

For an absolute file path, two leading slashes should be used for the database name:

```
e = create_engine('sqlite+pysqlcipher://:testing@//path/to/foo.db')
```

A selection of additional encryption-related pragmas supported by SQLCipher as documented at https://www.zetetic.net/sqlcipher/sqlcipher-api/ can be passed in the query string, and will result in that PRAGMA being called for each new connection. Currently, `cipher`, `kdf_iter` `cipher_page_size` and `cipher_use_hmac` are supported:

```
e = create_engine('sqlite+pysqlcipher://:testing@/foo.db?cipher=aes-256-cfb&kdf_iter=64000')
```

Pooling Behavior

The driver makes a change to the default pool behavior of pysqlite as described in *Threading/Pooling Behavior*. The pysqlcipher driver has been observed to be significantly slower on connection than the pysqlite driver, most likely due to the encryption overhead, so the dialect here defaults to using the `SingletonThreadPool` implementation, instead of the `NullPool` pool used by pysqlite. As always, the pool implementation is entirely configurable using the `create_engine.poolclass` parameter; the `StaticPool` may be more feasible for single-threaded use, or `NullPool` may be used to prevent unencrypted connections from being held open for long periods of time, at the expense of slower startup time for new connections.

4.1.7 Sybase

Support for the Sybase database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- *Python-Sybase*
- *PyODBC*
- *mxODBC*

Note: The Sybase dialect functions on current SQLAlchemy versions but is not regularly tested, and may have many issues and caveats not currently handled.

python-sybase

Support for the Sybase database via the Python-Sybase driver.

DBAPI

Documentation and download information (if applicable) for Python-Sybase is available at: http://python-sybase.sourceforge.net/

Connecting

Connect String:

```
sybase+pysybase://<username>:<password>@<dsn>/[database name]
```

Unicode Support

The python-sybase driver does not appear to support non-ASCII strings of any kind at this time.

pyodbc

Support for the Sybase database via the PyODBC driver.

DBAPI

Documentation and download information (if applicable) for PyODBC is available at: http://pypi.python.org/pypi/pyodbc/

Connecting

Connect String:

```
sybase+pyodbc://<username>:<password>@<dsnname>[/<database>]
```

Unicode Support

The pyodbc driver currently supports usage of these Sybase types with Unicode or multibyte strings:

```
CHAR
NCHAR
NVARCHAR
TEXT
VARCHAR
```

Currently *not* supported are:

```
UNICHAR
UNITEXT
UNIVARCHAR
```

mxodbc

Support for the Sybase database via the mxODBC driver.

DBAPI

Documentation and download information (if applicable) for mxODBC is available at: http://www.egenix.com/

Connecting

Connect String:

```
sybase+mxodbc://<username>:<password>@<dsnname>
```

Note: This dialect is a stub only and is likely non functional at this time.

4.2 External Dialects

Changed in version 0.8: As of SQLAlchemy 0.8, several dialects have been moved to external projects, and dialects for new databases will also be published as external projects. The rationale here is to keep the base SQLAlchemy install and test suite from growing inordinately large.

The "classic" dialects such as SQLite, MySQL, Postgresql, Oracle, SQL Server, and Firebird will remain in the Core for the time being.

Changed in version 1.0: The Drizzle dialect has been moved into the third party system.

Current external dialect projects for SQLAlchemy include:

4.2.1 Production Ready

- ibm_db_sa - driver for IBM DB2 and Informix, developed jointly by IBM and SQLAlchemy developers.
- sqlalchemy-redshift - driver for Amazon Redshift, adapts the existing Postgresql/psycopg2 driver.
- sqlalchemy_exasol - driver for EXASolution.
- sqlalchemy-sqlany - driver for SAP Sybase SQL Anywhere, developed by SAP.
- sqlalchemy-monetdb - driver for MonetDB.

4.2.2 Experimental / Incomplete

Dialects that are in an incomplete state or are considered somewhat experimental.

- CALCHIPAN - Adapts Pandas dataframes to SQLAlchemy.
- sqlalchemy-cubrid - driver for the CUBRID database.

4.2.3 Attic

Dialects in the "attic" are those that were contributed for SQLAlchemy long ago but have received little attention or demand since then, and are now moved out to their own repositories in at best a semi-working state. Community members interested in these dialects should feel free to pick up on their current codebase and fork off into working libraries.

- sqlalchemy-access - driver for Microsoft Access.
- sqlalchemy-drizzle - driver for the Drizzle MySQL variant.
- sqlalchemy-informixdb - driver for the informixdb DBAPI.
- sqlalchemy-maxdb - driver for the MaxDB database

CHAPTER 5

Frequently Asked Questions

The Frequently Asked Questions section is a growing collection of commonly observed questions to well-known issues.

5.1 Connections / Engines

- *How do I configure logging?*
- *How do I pool database connections? Are my connections pooled?*
- *How do I pass custom connect arguments to my database API?*
- *"MySQL Server has gone away"*
- *Why does SQLAlchemy issue so many ROLLBACKs?*
  - *I'm on MyISAM - how do I turn it off?*
  - *I'm on SQL Server - how do I turn those ROLLBACKs into COMMITs?*
- *I am using multiple connections with a SQLite database (typically to test transaction operation), and my test program is not working!*
- *How do I get at the raw DBAPI connection when using an Engine?*
- *How do I use engines / connections / sessions with Python multiprocessing, or os.fork()?*

5.1.1 How do I configure logging?

See *Configuring Logging*.

5.1.2 How do I pool database connections? Are my connections pooled?

SQLAlchemy performs application-level connection pooling automatically in most cases. With the exception of SQLite, a `Engine` object refers to a `QueuePool` as a source of connectivity.

For more detail, see *Engine Configuration* and *Connection Pooling*.

5.1.3 How do I pass custom connect arguments to my database API?

The `create_engine()` call accepts additional arguments either directly via the `connect_args` keyword argument:

```
e = create_engine("mysql://scott:tiger@localhost/test",
                  connect_args={"encoding": "utf8"})
```

Or for basic string and integer arguments, they can usually be specified in the query string of the URL:

```
e = create_engine("mysql://scott:tiger@localhost/test?encoding=utf8")
```

See also:

Custom DBAPI connect() arguments

5.1.4 "MySQL Server has gone away"

There are two major causes for this error:

1. The MySQL client closes connections which have been idle for a set period of time, defaulting to eight hours. This can be avoided by using the `pool_recycle` setting with `create_engine()`, described at *Connection Timeouts*.

2. Usage of the MySQLdb DBAPI, or a similar DBAPI, in a non-threadsafe manner, or in an otherwise inappropriate way. The MySQLdb connection object is not threadsafe - this expands out to any SQLAlchemy system that links to a single connection, which includes the ORM `Session`. For background on how `Session` should be used in a multithreaded environment, see *Is the session thread-safe?*.

5.1.5 Why does SQLAlchemy issue so many ROLLBACKs?

SQLAlchemy currently assumes DBAPI connections are in "non-autocommit" mode - this is the default behavior of the Python database API, meaning it must be assumed that a transaction is always in progress. The connection pool issues `connection.rollback()` when a connection is returned. This is so that any transactional resources remaining on the connection are released. On a database like Postgresql or MSSQL where table resources are aggressively locked, this is critical so that rows and tables don't remain locked within connections that are no longer in use. An application can otherwise hang. It's not just for locks, however, and is equally critical on any database that has any kind of transaction isolation, including MySQL with InnoDB. Any connection that is still inside an old transaction will return stale data, if that data was already queried on that connection within isolation. For background on why you might see stale data even on MySQL, see http://dev.mysql.com/doc/refman/5.1/en/innodb-transaction-model.html

I'm on MyISAM - how do I turn it off?

The behavior of the connection pool's connection return behavior can be configured using `reset_on_return`:

```
from sqlalchemy import create_engine
from sqlalchemy.pool import QueuePool

engine = create_engine('mysql://scott:tiger@localhost/myisam_database', pool=QueuePool(reset_on_retur
```

I'm on SQL Server - how do I turn those ROLLBACKs into COMMITs?

`reset_on_return` accepts the values `commit`, `rollback` in addition to `True`, `False`, and `None`. Setting to `commit` will cause a COMMIT as any connection is returned to the pool:

```
engine = create_engine('mssql://scott:tiger@mydsn', pool=QueuePool(reset_on_return='commit'))
```

5.1.6 I am using multiple connections with a SQLite database (typically to test transaction operation), and my test program is not working!

If using a SQLite `:memory:` database, or a version of SQLAlchemy prior to version 0.7, the default connection pool is the `SingletonThreadPool`, which maintains exactly one SQLite connection per thread. So two connections in use in the same thread will actually be the same SQLite connection. Make sure you're not using a :memory: database and use `NullPool`, which is the default for non-memory databases in current SQLAlchemy versions.

See also:

Threading/Pooling Behavior - info on PySQLite's behavior.

5.1.7 How do I get at the raw DBAPI connection when using an Engine?

With a regular SA engine-level Connection, you can get at a pool-proxied version of the DBAPI connection via the `Connection.connection` attribute on `Connection`, and for the really-real DBAPI connection you can call the `ConnectionFairy.connection` attribute on that - but there should never be any need to access the non-pool-proxied DBAPI connection, as all methods are proxied through:

```
engine = create_engine(...)
conn = engine.connect()
conn.connection.<do DBAPI things>
cursor = conn.connection.cursor(<DBAPI specific arguments..>)
```

You must ensure that you revert any isolation level settings or other operation-specific settings on the connection back to normal before returning it to the pool.

As an alternative to reverting settings, you can call the `Connection.detach()` method on either `Connection` or the proxied connection, which will de-associate the connection from the pool such that it will be closed and discarded when `Connection.close()` is called:

```
conn = engine.connect()
conn.detach()  # detaches the DBAPI connection from the connection pool
conn.connection.<go nuts>
conn.close()  # connection is closed for real, the pool replaces it with a new connection
```

5.1.8 How do I use engines / connections / sessions with Python multiprocessing, or os.fork()?

The key goal with multiple python processes is to prevent any database connections from being shared across processes. Depending on specifics of the driver and OS, the issues that arise here range from non-working connections to socket connections that are used by multiple processes concurrently, leading to broken messaging (the latter case is typically the most common).

The SQLAlchemy `Engine` object refers to a connection pool of existing database connections. So when this object is replicated to a child process, the goal is to ensure that no database connections are carried over. There are three general approaches to this:

1. Disable pooling using `NullPool`. This is the most simplistic, one shot system that prevents the `Engine` from using any connection more than once.
2. Call `Engine.dispose()` on any given `Engine` as soon one is within the new process. In Python multiprocessing, constructs such as `multiprocessing.Pool` include "initializer" hooks which are a place that this can be performed; otherwise at the top of where `os.fork()` or where the `Process` object begins the child fork, a single call to `Engine.dispose()` will ensure any remaining connections are flushed.

3. An event handler can be applied to the connection pool that tests for connections being shared across process boundaries, and invalidates them. This looks like the following:

```
import os
import warnings

from sqlalchemy import event
from sqlalchemy import exc

def add_engine_pidguard(engine):
    """Add multiprocessing guards.

    Forces a connection to be reconnected if it is detected
    as having been shared to a sub-process.

    """

    @event.listens_for(engine, "connect")
    def connect(dbapi_connection, connection_record):
        connection_record.info['pid'] = os.getpid()

    @event.listens_for(engine, "checkout")
    def checkout(dbapi_connection, connection_record, connection_proxy):
        pid = os.getpid()
        if connection_record.info['pid'] != pid:
            # substitute log.debug() or similar here as desired
            warnings.warn(
                "Parent process %(orig)s forked (%(newproc)s) with an open "
                "database connection, "
                "which is being discarded and recreated." %
                {"newproc": pid, "orig": connection_record.info['pid']})
            connection_record.connection = connection_proxy.connection = None
            raise exc.DisconnectionError(
                "Connection record belongs to pid %s, "
                "attempting to check out in pid %s" %
                (connection_record.info['pid'], pid)
            )
```

These events are applied to an *Engine* as soon as its created:

```
engine = create_engine("...")

add_engine_pidguard(engine)
```

The above strategies will accommodate the case of an *Engine* being shared among processes. However, for the case of a transaction-active *Session* or *Connection* being shared, there's no automatic fix for this; an application needs to ensure a new child process only initiate new *Connection* objects and transactions, as well as ORM *Session* objects. For a *Session* object, technically this is only needed if the session is currently transaction-bound, however the scope of a single *Session* is in any case intended to be kept within a single call stack in any case (e.g. not a global object, not shared between processes or threads).

5.2 MetaData / Schema

- *My program is hanging when I say `table.drop()` / `metadata.drop_all()`*
- *Does SQLAlchemy support ALTER TABLE, CREATE VIEW, CREATE TRIGGER, Schema Upgrade Functionality?*
- *How can I sort Table objects in order of their dependency?*
- *How can I get the CREATE TABLE/ DROP TABLE output as a string?*
- *How can I subclass Table/Column to provide certain behaviors/configurations?*

5.2.1 My program is hanging when I say `table.drop()` / `metadata.drop_all()`

This usually corresponds to two conditions: 1. using PostgreSQL, which is really strict about table locks, and 2. you have a connection still open which contains locks on the table and is distinct from the connection being used for the DROP statement. Heres the most minimal version of the pattern:

```
connection = engine.connect()
result = connection.execute(mytable.select())

mytable.drop(engine)
```

Above, a connection pool connection is still checked out; furthermore, the result object above also maintains a link to this connection. If "implicit execution" is used, the result will hold this connection opened until the result object is closed or all rows are exhausted.

The call to `mytable.drop(engine)` attempts to emit DROP TABLE on a second connection procured from the `Engine` which will lock.

The solution is to close out all connections before emitting DROP TABLE:

```
connection = engine.connect()
result = connection.execute(mytable.select())

# fully read result sets
result.fetchall()

# close connections
connection.close()

# now locks are removed
mytable.drop(engine)
```

5.2.2 Does SQLAlchemy support ALTER TABLE, CREATE VIEW, CREATE TRIGGER, Schema Upgrade Functionality?

General ALTER support isn't present in SQLAlchemy directly. For special DDL on an ad-hoc basis, the `DDL` and related constructs can be used. See `core/ddl` for a discussion on this subject.

A more comprehensive option is to use schema migration tools, such as Alembic or SQLAlchemy-Migrate; see *Altering Schemas through Migrations* for discussion on this.

5.2.3 How can I sort Table objects in order of their dependency?

This is available via the `MetaData.sorted_tables` function:

```
metadata = MetaData()
# ... add Table objects to metadata
ti = metadata.sorted_tables:
for t in ti:
    print t
```

5.2.4 How can I get the CREATE TABLE/ DROP TABLE output as a string?

Modern SQLAlchemy has clause constructs which represent DDL operations. These can be rendered to strings like any other SQL expression:

```
from sqlalchemy.schema import CreateTable

print CreateTable(mytable)
```

To get the string specific to a certain engine:

```
print CreateTable(mytable).compile(engine)
```

There's also a special form of `Engine` that can let you dump an entire metadata creation sequence, using this recipe:

```
def dump(sql, *multiparams, **params):
    print sql.compile(dialect=engine.dialect)
engine = create_engine('postgresql://', strategy='mock', executor=dump)
metadata.create_all(engine, checkfirst=False)
```

The Alembic tool also supports an "offline" SQL generation mode that renders database migrations as SQL scripts.

5.2.5 How can I subclass Table/Column to provide certain behaviors/configurations?

`Table` and `Column` are not good targets for direct subclassing. However, there are simple ways to get on-construction behaviors using creation functions, and behaviors related to the linkages between schema objects such as constraint conventions or naming conventions using attachment events. An example of many of these techniques can be seen at Naming Conventions.

5.3 SQL Expressions

- *How do I render SQL expressions as strings, possibly with bound parameters inlined?*
- *Why does `.col.in_([])` Produce `col != col`? Why not `1=0`?*

5.3.1 How do I render SQL expressions as strings, possibly with bound parameters inlined?

The "stringification" of a SQLAlchemy statement or Query in the vast majority of cases is as simple as:

```
print(str(statement))
```

this applies both to an ORM *Query* as well as any *select()* or other statement. Additionally, to get the statement as compiled to a specific dialect or engine, if the statement itself is not already bound to one you can pass this in to *ClauseElement.compile()*:

```
print(statement.compile(someengine))
```

or without an *Engine*:

```
from sqlalchemy.dialects import postgresql
print(statement.compile(dialect=postgresql.dialect()))
```

When given an ORM *Query* object, in order to get at the *ClauseElement.compile()* method we only need access the *statement* accessor first:

```
statement = query.statement
print(statement.compile(someengine))
```

The above forms will render the SQL statement as it is passed to the Python DBAPI, which includes that bound parameters are not rendered inline. SQLAlchemy normally does not stringify bound parameters, as this is handled appropriately by the Python DBAPI, not to mention bypassing bound parameters is probably the most widely exploited security hole in modern web applications. SQLAlchemy has limited ability to do this stringification in certain circumstances such as that of emitting DDL. In order to access this functionality one can use the `literal_binds` flag, passed to `compile_kwargs`:

```
from sqlalchemy.sql import table, column, select

t = table('t', column('x'))

s = select([t]).where(t.c.x == 5)

print(s.compile(compile_kwargs={"literal_binds": True}))
```

the above approach has the caveats that it is only supported for basic types, such as ints and strings, and furthermore if a *bindparam()* without a pre-set value is used directly, it won't be able to stringify that either.

To support inline literal rendering for types not supported, implement a *TypeDecorator* for the target type which includes a *TypeDecorator.process_literal_param()* method:

```
from sqlalchemy import TypeDecorator, Integer


class MyFancyType(TypeDecorator):
    impl = Integer

    def process_literal_param(self, value, dialect):
        return "my_fancy_formatting(%s)" % value

from sqlalchemy import Table, Column, MetaData

tab = Table('mytable', MetaData(), Column('x', MyFancyType()))

print(
    tab.select().where(tab.c.x > 5).compile(
```

```
        compile_kwargs={"literal_binds": True})
)
```

producing output like:

```
SELECT mytable.x
FROM mytable
WHERE mytable.x > my_fancy_formatting(5)
```

5.3.2 Why does `.col.in_([])` Produce `col != col`? Why not `1=0`?

A little introduction to the issue. The IN operator in SQL, given a list of elements to compare against a column, generally does not accept an empty list, that is while it is valid to say:

```
column IN (1, 2, 3)
```

it's not valid to say:

```
column IN ()
```

SQLAlchemy's `Operators.in_()` operator, when given an empty list, produces this expression:

```
column != column
```

As of version 0.6, it also produces a warning stating that a less efficient comparison operation will be rendered. This expression is the only one that is both database agnostic and produces correct results.

For example, the naive approach of "just evaluate to false, by comparing 1=0 or 1!=1", does not handle nulls properly. An expression like:

```
NOT column != column
```

will not return a row when "column" is null, but an expression which does not take the column into account:

```
NOT 1=0
```

will.

Closer to the mark is the following CASE expression:

```
CASE WHEN column IS NOT NULL THEN 1=0 ELSE NULL END
```

We don't use this expression due to its verbosity, and its also not typically accepted by Oracle within a WHERE clause - depending on how you phrase it, you'll either get "ORA-00905: missing keyword" or "ORA-00920: invalid relational operator". It's also still less efficient than just rendering SQL without the clause altogether (or not issuing the SQL at all, if the statement is just a simple search).

The best approach therefore is to avoid the usage of IN given an argument list of zero length. Instead, don't emit the Query in the first place, if no rows should be returned. The warning is best promoted to a full error condition using the Python warnings filter (see http://docs.python.org/library/warnings.html).

5.4 ORM Configuration

- *How do I map a table that has no primary key?*
- *How do I configure a Column that is a Python reserved word or similar?*
- *How do I get a list of all columns, relationships, mapped attributes, etc. given a mapped class?*
- *I'm getting a warning or error about "Implicitly combining column X under attribute Y"*
- *I'm using Declarative and setting primaryjoin/secondaryjoin using an `and_()` or `or_()`, and I am getting an error message about foreign keys.*
- *Why is `ORDER BY` required with `LIMIT` (especially with `subqueryload()`)?*

5.4.1 How do I map a table that has no primary key?

The SQLAlchemy ORM, in order to map to a particular table, needs there to be at least one column denoted as a primary key column; multiple-column, i.e. composite, primary keys are of course entirely feasible as well. These columns do **not** need to be actually known to the database as primary key columns, though it's a good idea that they are. It's only necessary that the columns *behave* as a primary key does, e.g. as a unique and not nullable identifier for a row.

Most ORMs require that objects have some kind of primary key defined because the object in memory must correspond to a uniquely identifiable row in the database table; at the very least, this allows the object can be targeted for UPDATE and DELETE statements which will affect only that object's row and no other. However, the importance of the primary key goes far beyond that. In SQLAlchemy, all ORM-mapped objects are at all times linked uniquely within a `Session` to their specific database row using a pattern called the identity map, a pattern that's central to the unit of work system employed by SQLAlchemy, and is also key to the most common (and not-so-common) patterns of ORM usage.

Note: It's important to note that we're only talking about the SQLAlchemy ORM; an application which builds on Core and deals only with `Table` objects, `select()` constructs and the like, **does not** need any primary key to be present on or associated with a table in any way (though again, in SQL, all tables should really have some kind of primary key, lest you need to actually update or delete specific rows).

In almost all cases, a table does have a so-called candidate key, which is a column or series of columns that uniquely identify a row. If a table truly doesn't have this, and has actual fully duplicate rows, the table is not corresponding to first normal form and cannot be mapped. Otherwise, whatever columns comprise the best candidate key can be applied directly to the mapper:

```
class SomeClass(Base):
    __table__ = some_table_with_no_pk
    __mapper_args__ = {
        'primary_key':[some_table_with_no_pk.c.uid, some_table_with_no_pk.c.bar]
    }
```

Better yet is when using fully declared table metadata, use the `primary_key=True` flag on those columns:

```
class SomeClass(Base):
    __tablename__ = "some_table_with_no_pk"

    uid = Column(Integer, primary_key=True)
    bar = Column(String, primary_key=True)
```

All tables in a relational database should have primary keys. Even a many-to-many association table - the primary key would be the composite of the two association columns:

```
CREATE TABLE my_association (
  user_id INTEGER REFERENCES user(id),
  account_id INTEGER REFERENCES account(id),
  PRIMARY KEY (user_id, account_id)
)
```

5.4.2 How do I configure a Column that is a Python reserved word or similar?

Column-based attributes can be given any name desired in the mapping. See *Naming Columns Distinctly from Attribute Names*.

5.4.3 How do I get a list of all columns, relationships, mapped attributes, etc. given a mapped class?

This information is all available from the `Mapper` object.

To get at the `Mapper` for a particular mapped class, call the `inspect()` function on it:

```
from sqlalchemy import inspect

mapper = inspect(MyClass)
```

From there, all information about the class can be acquired using such methods as:

- `Mapper.attrs` - a namespace of all mapped attributes. The attributes themselves are instances of `MapperProperty`, which contain additional attributes that can lead to the mapped SQL expression or column, if applicable.
- `Mapper.column_attrs` - the mapped attribute namespace limited to column and SQL expression attributes. You might want to use `Mapper.columns` to get at the `Column` objects directly.
- `Mapper.relationships` - namespace of all `RelationshipProperty` attributes.
- `Mapper.all_orm_descriptors` - namespace of all mapped attributes, plus user-defined attributes defined using systems such as `hybrid_property`, `AssociationProxy` and others.
- `Mapper.columns` - A namespace of `Column` objects and other named SQL expressions associated with the mapping.
- `Mapper.mapped_table` - The `Table` or other selectable to which this mapper is mapped.
- `Mapper.local_table` - The `Table` that is "local" to this mapper; this differs from `Mapper.mapped_table` in the case of a mapper mapped using inheritance to a composed selectable.

5.4.4 I'm getting a warning or error about "Implicitly combining column X under attribute Y"

This condition refers to when a mapping contains two columns that are being mapped under the same attribute name due to their name, but there's no indication that this is intentional. A mapped class needs to have explicit names for every attribute that is to store an independent value; when two columns have the same name and aren't disambiguated, they fall under the same attribute and the effect is that the value from one column is **copied** into the other, based on which column was assigned to the attribute first.

This behavior is often desirable and is allowed without warning in the case where the two columns are linked together via a foreign key relationship within an inheritance mapping. When the warning or exception occurs, the issue can be

resolved by either assigning the columns to differently-named attributes, or if combining them together is desired, by using *column_property()* to make this explicit.

Given the example as follows:

```
from sqlalchemy import Integer, Column, ForeignKey
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class A(Base):
    __tablename__ = 'a'

    id = Column(Integer, primary_key=True)

class B(A):
    __tablename__ = 'b'

    id = Column(Integer, primary_key=True)
    a_id = Column(Integer, ForeignKey('a.id'))
```

As of SQLAlchemy version 0.9.5, the above condition is detected, and will warn that the `id` column of `A` and `B` is being combined under the same-named attribute `id`, which above is a serious issue since it means that a `B` object's primary key will always mirror that of its `A`.

A mapping which resolves this is as follows:

```
class A(Base):
    __tablename__ = 'a'

    id = Column(Integer, primary_key=True)

class B(A):
    __tablename__ = 'b'

    b_id = Column('id', Integer, primary_key=True)
    a_id = Column(Integer, ForeignKey('a.id'))
```

Suppose we did want `A.id` and `B.id` to be mirrors of each other, despite the fact that `B.a_id` is where `A.id` is related. We could combine them together using *column_property()*:

```
class A(Base):
    __tablename__ = 'a'

    id = Column(Integer, primary_key=True)

class B(A):
    __tablename__ = 'b'

    # probably not what you want, but this is a demonstration
    id = column_property(Column(Integer, primary_key=True), A.id)
    a_id = Column(Integer, ForeignKey('a.id'))
```

5.4.5 I'm using Declarative and setting primaryjoin/secondaryjoin using an `and_()` or `or_()`, and I am getting an error message about foreign keys.

Are you doing this?:

```
class MyClass(Base):
    # ....

    foo = relationship("Dest", primaryjoin=and_("MyClass.id==Dest.foo_id", "MyClass.foo==Dest.bar"))
```

That's an `and_()` of two string expressions, which SQLAlchemy cannot apply any mapping towards. Declarative allows *relationship()* arguments to be specified as strings, which are converted into expression objects using `eval()`. But this doesn't occur inside of an `and_()` expression - it's a special operation declarative applies only to the *entirety* of what's passed to primaryjoin or other arguments as a string:

```
class MyClass(Base):
    # ....

    foo = relationship("Dest", primaryjoin="and_(MyClass.id==Dest.foo_id, MyClass.foo==Dest.bar)")
```

Or if the objects you need are already available, skip the strings:

```
class MyClass(Base):
    # ....

    foo = relationship(Dest, primaryjoin=and_(MyClass.id==Dest.foo_id, MyClass.foo==Dest.bar))
```

The same idea applies to all the other arguments, such as `foreign_keys`:

```
# wrong !
foo = relationship(Dest, foreign_keys=["Dest.foo_id", "Dest.bar_id"])

# correct !
foo = relationship(Dest, foreign_keys="[Dest.foo_id, Dest.bar_id]")

# also correct !
foo = relationship(Dest, foreign_keys=[Dest.foo_id, Dest.bar_id])

# if you're using columns from the class that you're inside of, just use the column objects !
class MyClass(Base):
    foo_id = Column(...)
    bar_id = Column(...)
    # ...

    foo = relationship(Dest, foreign_keys=[foo_id, bar_id])
```

5.4.6 Why is `ORDER BY` required with `LIMIT` (especially with `subqueryload()`)?

A relational database can return rows in any arbitrary order, when an explicit ordering is not set. While this ordering very often corresponds to the natural order of rows within a table, this is not the case for all databases and all queries. The consequence of this is that any query that limits rows using `LIMIT` or `OFFSET` should **always** specify an `ORDER BY`. Otherwise, it is not deterministic which rows will actually be returned.

When we use a SQLAlchemy method like *Query.first()*, we are in fact applying a `LIMIT` of one to the query, so without an explicit ordering it is not deterministic what row we actually get back. While we may not notice this for simple queries on databases that usually returns rows in their natural order, it becomes much more of an issue if we also use *orm.subqueryload()* to load related collections, and we may not be loading the collections as intended.

SQLAlchemy implements *orm.subqueryload()* by issuing a separate query, the results of which are matched up to the results from the first query. We see two queries emitted like this:

```
>>> session.query(User).options(subqueryload(User.addresses)).all()
-- the "main" query
SELECT users.id AS users_id
FROM users

-- the "load" query issued by subqueryload
SELECT addresses.id AS addresses_id,
       addresses.user_id AS addresses_user_id,
       anon_1.users_id AS anon_1_users_id
FROM (SELECT users.id AS users_id FROM users) AS anon_1
JOIN addresses ON anon_1.users_id = addresses.user_id
ORDER BY anon_1.users_id
```

The second query embeds the first query as a source of rows. When the inner query uses `OFFSET` and/or `LIMIT` without ordering, the two queries may not see the same results:

```
>>> user = session.query(User).options(subqueryload(User.addresses)).first()
-- the "main" query
SELECT users.id AS users_id
FROM users
 LIMIT 1

-- the "load" query issued by subqueryload
SELECT addresses.id AS addresses_id,
       addresses.user_id AS addresses_user_id,
       anon_1.users_id AS anon_1_users_id
FROM (SELECT users.id AS users_id FROM users LIMIT 1) AS anon_1
JOIN addresses ON anon_1.users_id = addresses.user_id
ORDER BY anon_1.users_id
```

Depending on database specifics, there is a chance we may get the a result like the following for the two queries:

```
-- query #1
+--------+
|users_id|
+--------+
|       1|
+--------+

-- query #2
+------------+-----------------+---------------+
|addresses_id|addresses_user_id|anon_1_users_id|
+------------+-----------------+---------------+
|           3|                2|              2|
+------------+-----------------+---------------+
|           4|                2|              2|
+------------+-----------------+---------------+
```

Above, we receive two `addresses` rows for `user.id` of 2, and none for 1. We've wasted two rows and failed to actually load the collection. This is an insidious error because without looking at the SQL and the results, the ORM will not show that there's any issue; if we access the `addresses` for the `User` we have, it will emit a lazy load for the collection and we won't see that anything actually went wrong.

The solution to this problem is to always specify a deterministic sort order, so that the main query always returns the same set of rows. This generally means that you should *`Query.order_by()`* on a unique column on the table. The primary key is a good choice for this:

```
session.query(User).options(subqueryload(User.addresses)).order_by(User.id).first()
```

Note that *joinedload()* does not suffer from the same problem because only one query is ever issued, so the load query cannot be different from the main query.

See also:

The Importance of Ordering

5.5 Performance

- *How can I profile a SQLAlchemy powered application?*
  - *Query Profiling*
  - *Code Profiling*
  - *Execution Slowness*
  - *Result Fetching Slowness - Core*
  - *Result Fetching Slowness - ORM*
- *I'm inserting 400,000 rows with the ORM and it's really slow!*

5.5.1 How can I profile a SQLAlchemy powered application?

Looking for performance issues typically involves two stratgies. One is query profiling, and the other is code profiling.

Query Profiling

Sometimes just plain SQL logging (enabled via python's logging module or via the `echo=True` argument on *create_engine()*) can give an idea how long things are taking. For example, if you log something right after a SQL operation, you'd see something like this in your log:

```
17:37:48,325 INFO  [sqlalchemy.engine.base.Engine.0x...048c] SELECT ...
17:37:48,326 INFO  [sqlalchemy.engine.base.Engine.0x...048c] {<params>}
17:37:48,660 DEBUG [myapp.somemessage]
```

if you logged `myapp.somemessage` right after the operation, you know it took 334ms to complete the SQL part of things.

Logging SQL will also illustrate if dozens/hundreds of queries are being issued which could be better organized into much fewer queries. When using the SQLAlchemy ORM, the "eager loading" feature is provided to partially (*contains_eager()*) or fully (*joinedload()*, *subqueryload()*) automate this activity, but without the ORM "eager loading" typically means to use joins so that results across multiple tables can be loaded in one result set instead of multiplying numbers of queries as more depth is added (i.e. `r + r*r2 + r*r2*r3` ...)

For more long-term profiling of queries, or to implement an application-side "slow query" monitor, events can be used to intercept cursor executions, using a recipe like the following:

```
from sqlalchemy import event
from sqlalchemy.engine import Engine
import time
import logging
```

```
logging.basicConfig()
logger = logging.getLogger("myapp.sqltime")
logger.setLevel(logging.DEBUG)

@event.listens_for(Engine, "before_cursor_execute")
def before_cursor_execute(conn, cursor, statement,
                          parameters, context, executemany):
    conn.info.setdefault('query_start_time', []).append(time.time())
    logger.debug("Start Query: %s", statement)

@event.listens_for(Engine, "after_cursor_execute")
def after_cursor_execute(conn, cursor, statement,
                          parameters, context, executemany):
    total = time.time() - conn.info['query_start_time'].pop(-1)
    logger.debug("Query Complete!")
    logger.debug("Total Time: %f", total)
```

Above, we use the *ConnectionEvents.before_cursor_execute()* and *ConnectionEvents.after_cursor_execute()* events to establish an interception point around when a statement is executed. We attach a timer onto the connection using the `_ConnectionRecord.info` dictionary; we use a stack here for the occasional case where the cursor execute events may be nested.

Code Profiling

If logging reveals that individual queries are taking too long, you'd need a breakdown of how much time was spent within the database processing the query, sending results over the network, being handled by the DBAPI, and finally being received by SQLAlchemy's result set and/or ORM layer. Each of these stages can present their own individual bottlenecks, depending on specifics.

For that you need to use the Python Profiling Module. Below is a simple recipe which works profiling into a context manager:

```
import cProfile
import StringIO
import pstats
import contextlib

@contextlib.contextmanager
def profiled():
    pr = cProfile.Profile()
    pr.enable()
    yield
    pr.disable()
    s = StringIO.StringIO()
    ps = pstats.Stats(pr, stream=s).sort_stats('cumulative')
    ps.print_stats()
    # uncomment this to see who's calling what
    # ps.print_callers()
    print s.getvalue()
```

To profile a section of code:

```
with profiled():
    Session.query(FooClass).filter(FooClass.somevalue==8).all()
```

The output of profiling can be used to give an idea where time is being spent. A section of profiling output looks like this:

```
13726 function calls (13042 primitive calls) in 0.014 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
222/21    0.001    0.000    0.011    0.001 lib/sqlalchemy/orm/loading.py:26(instances)
220/20    0.002    0.000    0.010    0.001 lib/sqlalchemy/orm/loading.py:327(_instance)
220/20    0.000    0.000    0.010    0.000 lib/sqlalchemy/orm/loading.py:284(populate_state)
   20    0.000    0.000    0.010    0.000 lib/sqlalchemy/orm/strategies.py:987(load_collection_from_s
   20    0.000    0.000    0.009    0.000 lib/sqlalchemy/orm/strategies.py:935(get)
    1    0.000    0.000    0.009    0.009 lib/sqlalchemy/orm/strategies.py:940(_load)
   21    0.000    0.000    0.008    0.000 lib/sqlalchemy/orm/strategies.py:942(<genexpr>)
    2    0.000    0.000    0.004    0.002 lib/sqlalchemy/orm/query.py:2400(__iter__)
    2    0.000    0.000    0.002    0.001 lib/sqlalchemy/orm/query.py:2414(_execute_and_instances)
    2    0.000    0.000    0.002    0.001 lib/sqlalchemy/engine/base.py:659(execute)
    2    0.000    0.000    0.002    0.001 lib/sqlalchemy/sql/elements.py:321(_execute_on_connection)
    2    0.000    0.000    0.002    0.001 lib/sqlalchemy/engine/base.py:788(_execute_clauseelement)

...
```

Above, we can see that the `instances()` SQLAlchemy function was called 222 times (recursively, and 21 times from the outside), taking a total of .011 seconds for all calls combined.

Execution Slowness

The specifics of these calls can tell us where the time is being spent. If for example, you see time being spent within `cursor.execute()`, e.g. against the DBAPI:

```
2    0.102    0.102    0.204    0.102 {method 'execute' of 'sqlite3.Cursor' objects}
```

this would indicate that the database is taking a long time to start returning results, and it means your query should be optimized, either by adding indexes or restructuring the query and/or underlying schema. For that task, analysis of the query plan is warranted, using a system such as EXPLAIN, SHOW PLAN, etc. as is provided by the database backend.

Result Fetching Slowness - Core

If on the other hand you see many thousands of calls related to fetching rows, or very long calls to `fetchall()`, it may mean your query is returning more rows than expected, or that the fetching of rows itself is slow. The ORM itself typically uses `fetchall()` to fetch rows (or `fetchmany()` if the *`Query.yield_per()`* option is used).

An inordinately large number of rows would be indicated by a very slow call to `fetchall()` at the DBAPI level:

```
2    0.300    0.600    0.300    0.600 {method 'fetchall' of 'sqlite3.Cursor' objects}
```

An unexpectedly large number of rows, even if the ultimate result doesn't seem to have many rows, can be the result of a cartesian product - when multiple sets of rows are combined together without appropriately joining the tables together. It's often easy to produce this behavior with SQLAlchemy Core or ORM query if the wrong *`Column`* objects are used in a complex query, pulling in additional FROM clauses that are unexpected.

On the other hand, a fast call to `fetchall()` at the DBAPI level, but then slowness when SQLAlchemy's *`ResultProxy`* is asked to do a `fetchall()`, may indicate slowness in processing of datatypes, such as unicode conversions and similar:

```
# the DBAPI cursor is fast...
2    0.020    0.040    0.020    0.040 {method 'fetchall' of 'sqlite3.Cursor' objects}

...

# but SQLAlchemy's result proxy is slow, this is type-level processing
2    0.100    0.200    0.100    0.200 lib/sqlalchemy/engine/result.py:778(fetchall)
```

In some cases, a backend might be doing type-level processing that isn't needed. More specifically, seeing calls within the type API that are slow are better indicators - below is what it looks like when we use a type like this:

```
from sqlalchemy import TypeDecorator
import time

class Foo(TypeDecorator):
    impl = String

    def process_result_value(self, value, thing):
        # intentionally add slowness for illustration purposes
        time.sleep(.001)
        return value
```

the profiling output of this intentionally slow operation can be seen like this:

```
200    0.001    0.000    0.237    0.001 lib/sqlalchemy/sql/type_api.py:911(process)
200    0.001    0.000    0.236    0.001 test.py:28(process_result_value)
200    0.235    0.001    0.235    0.001 {time.sleep}
```

that is, we see many expensive calls within the `type_api` system, and the actual time consuming thing is the `time.sleep()` call.

Make sure to check the `Dialect documentation` for notes on known performance tuning suggestions at this level, especially for databases like Oracle. There may be systems related to ensuring numeric accuracy or string processing that may not be needed in all cases.

There also may be even more low-level points at which row-fetching performance is suffering; for example, if time spent seems to focus on a call like `socket.receive()`, that could indicate that everything is fast except for the actual network connection, and too much time is spent with data moving over the network.

Result Fetching Slowness - ORM

To detect slowness in ORM fetching of rows (which is the most common area of performance concern), calls like `populate_state()` and `_instance()` will illustrate individual ORM object populations:

```
# the ORM calls _instance for each ORM-loaded row it sees, and
# populate_state for each ORM-loaded row that results in the population
# of an object's attributes
220/20    0.001    0.000    0.010    0.000 lib/sqlalchemy/orm/loading.py:327(_instance)
220/20    0.000    0.000    0.009    0.000 lib/sqlalchemy/orm/loading.py:284(populate_state)
```

The ORM's slowness in turning rows into ORM-mapped objects is a product of the complexity of this operation combined with the overhead of cPython. Common strategies to mitigate this include:

- fetch individual columns instead of full entities, that is:

```
session.query(User.id, User.name)
```

instead of:

```
session.query(User)
```

- Use `Bundle` objects to organize column-based results:

```
u_b = Bundle('user', User.id, User.name)
a_b = Bundle('address', Address.id, Address.email)

for user, address in session.query(u_b, a_b).join(User.addresses):
    # ...
```

- Use result caching - see *Dogpile Caching* for an in-depth example of this.
- Consider a faster interpreter like that of Pypy.

The output of a profile can be a little daunting but after some practice they are very easy to read.

See also:

Performance - a suite of performance demonstrations with bundled profiling capabilities.

5.5.2 I'm inserting 400,000 rows with the ORM and it's really slow!

The SQLAlchemy ORM uses the unit of work pattern when synchronizing changes to the database. This pattern goes far beyond simple "inserts" of data. It includes that attributes which are assigned on objects are received using an attribute instrumentation system which tracks changes on objects as they are made, includes that all rows inserted are tracked in an identity map which has the effect that for each row SQLAlchemy must retrieve its "last inserted id" if not already given, and also involves that rows to be inserted are scanned and sorted for dependencies as needed. Objects are also subject to a fair degree of bookkeeping in order to keep all of this running, which for a very large number of rows at once can create an inordinate amount of time spent with large data structures, hence it's best to chunk these.

Basically, unit of work is a large degree of automation in order to automate the task of persisting a complex object graph into a relational database with no explicit persistence code, and this automation has a price.

ORMs are basically not intended for high-performance bulk inserts - this is the whole reason SQLAlchemy offers the Core in addition to the ORM as a first-class component.

For the use case of fast bulk inserts, the SQL generation and execution system that the ORM builds on top of is part of the `Core`. Using this system directly, we can produce an INSERT that is competitive with using the raw database API directly.

Alternatively, the SQLAlchemy ORM offers the *Bulk Operations* suite of methods, which provide hooks into subsections of the unit of work process in order to emit Core-level INSERT and UPDATE constructs with a small degree of ORM-based automation.

The example below illustrates time-based tests for several different methods of inserting rows, going from the most automated to the least. With cPython 2.7, runtimes observed:

```
classics-MacBook-Pro:sqlalchemy classic$ python test.py
SQLAlchemy ORM: Total time for 100000 records 12.0471920967 secs
SQLAlchemy ORM pk given: Total time for 100000 records 7.06283402443 secs
SQLAlchemy ORM bulk_save_objects(): Total time for 100000 records 0.856323003769 secs
SQLAlchemy Core: Total time for 100000 records 0.485800027847 secs
sqlite3: Total time for 100000 records 0.487842082977 sec
```

We can reduce the time by a factor of three using recent versions of Pypy:

```
classics-MacBook-Pro:sqlalchemy classic$ /usr/local/src/pypy-2.1-beta2-osx64/bin/pypy test.py
SQLAlchemy ORM: Total time for 100000 records 5.88369488716 secs
SQLAlchemy ORM pk given: Total time for 100000 records 3.52294301987 secs
SQLAlchemy Core: Total time for 100000 records 0.613556146622 secs
sqlite3: Total time for 100000 records 0.442467927933 sec
```

Script:

```
import time
import sqlite3

from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String,  create_engine
from sqlalchemy.orm import scoped_session, sessionmaker

Base = declarative_base()
DBSession = scoped_session(sessionmaker())
engine = None


class Customer(Base):
    __tablename__ = "customer"
    id = Column(Integer, primary_key=True)
    name = Column(String(255))


def init_sqlalchemy(dbname='sqlite:///sqlalchemy.db'):
    global engine
    engine = create_engine(dbname, echo=False)
    DBSession.remove()
    DBSession.configure(bind=engine, autoflush=False, expire_on_commit=False)
    Base.metadata.drop_all(engine)
    Base.metadata.create_all(engine)


def test_sqlalchemy_orm(n=100000):
    init_sqlalchemy()
    t0 = time.time()
    for i in xrange(n):
        customer = Customer()
        customer.name = 'NAME ' + str(i)
        DBSession.add(customer)
        if i % 1000 == 0:
            DBSession.flush()
    DBSession.commit()
    print(
        "SQLAlchemy ORM: Total time for " + str(n) +
        " records " + str(time.time() - t0) + " secs")


def test_sqlalchemy_orm_pk_given(n=100000):
    init_sqlalchemy()
    t0 = time.time()
    for i in xrange(n):
        customer = Customer(id=i+1, name="NAME " + str(i))
        DBSession.add(customer)
```

```
        if i % 1000 == 0:
            DBSession.flush()
    DBSession.commit()
    print(
        "SQLAlchemy ORM pk given: Total time for " + str(n) +
        " records " + str(time.time() - t0) + " secs")


def test_sqlalchemy_orm_bulk_insert(n=100000):
    init_sqlalchemy()
    t0 = time.time()
    n1 = n
    while n1 > 0:
        n1 = n1 - 10000
        DBSession.bulk_insert_mappings(
            Customer,
            [
                dict(name="NAME " + str(i))
                for i in xrange(min(10000, n1))
            ]
        )
    DBSession.commit()
    print(
        "SQLAlchemy ORM bulk_save_objects(): Total time for " + str(n) +
        " records " + str(time.time() - t0) + " secs")


def test_sqlalchemy_core(n=100000):
    init_sqlalchemy()
    t0 = time.time()
    engine.execute(
        Customer.__table__.insert(),
        [{"name": 'NAME ' + str(i)} for i in xrange(n)]
    )
    print(
        "SQLAlchemy Core: Total time for " + str(n) +
        " records " + str(time.time() - t0) + " secs")


def init_sqlite3(dbname):
    conn = sqlite3.connect(dbname)
    c = conn.cursor()
    c.execute("DROP TABLE IF EXISTS customer")
    c.execute(
        "CREATE TABLE customer (id INTEGER NOT NULL, "
        "name VARCHAR(255), PRIMARY KEY(id))")
    conn.commit()
    return conn


def test_sqlite3(n=100000, dbname='sqlite3.db'):
    conn = init_sqlite3(dbname)
    c = conn.cursor()
    t0 = time.time()
    for i in xrange(n):
        row = ('NAME ' + str(i),)
        c.execute("INSERT INTO customer (name) VALUES (?)", row)
    conn.commit()
```

```
    print(
        "sqlite3: Total time for " + str(n) +
        " records " + str(time.time() - t0) + " sec")

if __name__ == '__main__':
    test_sqlalchemy_orm(100000)
    test_sqlalchemy_orm_pk_given(100000)
    test_sqlalchemy_orm_bulk_insert(100000)
    test_sqlalchemy_core(100000)
    test_sqlite3(100000)
```

5.6 Sessions / Queries

- *I'm re-loading data with my Session but it isn't seeing changes that I committed elsewhere*
- *"This Session's transaction has been rolled back due to a previous exception during flush." (or similar)*
  - *But why does flush() insist on issuing a ROLLBACK?*
  - *But why isn't the one automatic call to ROLLBACK enough? Why must I ROLLBACK again?*
- *How do I make a Query that always adds a certain filter to every query?*
- *I've created a mapping against an Outer Join, and while the query returns rows, no objects are returned. Why not?*
- *I'm using `joinedload()` or `lazy=False` to create a JOIN/OUTER JOIN and SQLAlchemy is not constructing the correct query when I try to add a WHERE, ORDER BY, LIMIT, etc. (which relies upon the (OUTER) JOIN)*
- *Query has no `__len__()`, why not?*
- *How Do I use Textual SQL with ORM Queries?*
- *I'm calling `Session.delete(myobject)` and it isn't removed from the parent collection!*
- *why isn't my `__init__()` called when I load objects?*
- *how do I use ON DELETE CASCADE with SA's ORM?*
- *I set the "foo_id" attribute on my instance to "7", but the "foo" attribute is still `None` - shouldn't it have loaded Foo with id #7?*
- *How do I walk all objects that are related to a given object?*
- *Is there a way to automagically have only unique keywords (or other kinds of objects) without doing a query for the keyword and getting a reference to the row containing that keyword?*

5.6.1 I'm re-loading data with my Session but it isn't seeing changes that I committed elsewhere

The main issue regarding this behavior is that the session acts as though the transaction is in the *serializable* isolation state, even if it's not (and it usually is not). In practical terms, this means that the session does not alter any data that it's already read within the scope of a transaction.

If the term "isolation level" is unfamiliar, then you first need to read this link:

Isolation Level

In short, serializable isolation level generally means that once you SELECT a series of rows in a transaction, you will get *the identical data* back each time you re-emit that SELECT. If you are in the next-lower isolation level, "repeatable read", you'll see newly added rows (and no longer see deleted rows), but for rows that you've *already* loaded, you won't see any change. Only if you are in a lower isolation level, e.g. "read committed", does it become possible to see a row of data change its value.

For information on controlling the isolation level when using the SQLAlchemy ORM, see *Setting Transaction Isolation Levels*.

To simplify things dramatically, the `Session` itself works in terms of a completely isolated transaction, and doesn't overwrite any mapped attributes it's already read unless you tell it to. The use case of trying to re-read data you've already loaded in an ongoing transaction is an *uncommon* use case that in many cases has no effect, so this is considered to be the exception, not the norm; to work within this exception, several methods are provided to allow specific data to be reloaded within the context of an ongoing transaction.

To understand what we mean by "the transaction" when we talk about the `Session`, your `Session` is intended to only work within a transaction. An overview of this is at *Managing Transactions*.

Once we've figured out what our isolation level is, and we think that our isolation level is set at a low enough level so that if we re-SELECT a row, we should see new data in our `Session`, how do we see it?

Three ways, from most common to least:

1. We simply end our transaction and start a new one on next access with our `Session` by calling `Session.commit()` (note that if the `Session` is in the lesser-used "autocommit" mode, there would be a call to `Session.begin()` as well). The vast majority of applications and use cases do not have any issues with not being able to "see" data in other transactions because they stick to this pattern, which is at the core of the best practice of **short lived transactions**. See *When do I construct a Session, when do I commit it, and when do I close it?* for some thoughts on this.
2. We tell our `Session` to re-read rows that it has already read, either when we next query for them using `Session.expire_all()` or `Session.expire()`, or immediately on an object using `Session.refresh`. See *Refreshing / Expiring* for detail on this.
3. We can run whole queries while setting them to definitely overwrite already-loaded objects as they read rows by using `Query.populate_existing()`.

But remember, **the ORM cannot see changes in rows if our isolation level is repeatable read or higher, unless we start a new transaction**.

5.6.2 "This Session's transaction has been rolled back due to a previous exception during flush." (or similar)

This is an error that occurs when a `Session.flush()` raises an exception, rolls back the transaction, but further commands upon the *Session* are called without an explicit call to `Session.rollback()` or `Session.close()`.

It usually corresponds to an application that catches an exception upon `Session.flush()` or `Session.commit()` and does not properly handle the exception. For example:

```
from sqlalchemy import create_engine, Column, Integer
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base(create_engine('sqlite://'))

class Foo(Base):
    __tablename__ = 'foo'
    id = Column(Integer, primary_key=True)

Base.metadata.create_all()

session = sessionmaker()()

# constraint violation
```

```
session.add_all([Foo(id=1), Foo(id=1)])

try:
    session.commit()
except:
    # ignore error
    pass

# continue using session without rolling back
session.commit()
```

The usage of the `Session` should fit within a structure similar to this:

```
try:
    <use session>
    session.commit()
except:
   session.rollback()
   raise
finally:
   session.close()  # optional, depends on use case
```

Many things can cause a failure within the try/except besides flushes. You should always have some kind of "framing" of your session operations so that connection and transaction resources have a definitive boundary, otherwise your application doesn't really have its usage of resources under control. This is not to say that you need to put try/except blocks all throughout your application - on the contrary, this would be a terrible idea. You should architect your application such that there is one (or few) point(s) of "framing" around session operations.

For a detailed discussion on how to organize usage of the `Session`, please see *When do I construct a Session, when do I commit it, and when do I close it?*.

But why does flush() insist on issuing a ROLLBACK?

It would be great if `Session.flush()` could partially complete and then not roll back, however this is beyond its current capabilities since its internal bookkeeping would have to be modified such that it can be halted at any time and be exactly consistent with what's been flushed to the database. While this is theoretically possible, the usefulness of the enhancement is greatly decreased by the fact that many database operations require a ROLLBACK in any case. Postgres in particular has operations which, once failed, the transaction is not allowed to continue:

```
test=> create table foo(id integer primary key);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "foo_pkey" for table "foo"
CREATE TABLE
test=> begin;
BEGIN
test=> insert into foo values(1);
INSERT 0 1
test=> commit;
COMMIT
test=> begin;
BEGIN
test=> insert into foo values(1);
ERROR:  duplicate key value violates unique constraint "foo_pkey"
test=> insert into foo values(2);
ERROR:  current transaction is aborted, commands ignored until end of transaction block
```

What SQLAlchemy offers that solves both issues is support of SAVEPOINT, via *Session.begin_nested()*. Using *Session.begin_nested()*, you can frame an operation that may potentially fail within a transaction, and then "roll back" to the point before its failure while maintaining the enclosing transaction.

But why isn't the one automatic call to ROLLBACK enough? Why must I ROLLBACK again?

This is again a matter of the *Session* providing a consistent interface and refusing to guess about what context its being used. For example, the *Session* supports "framing" above within multiple levels. Such as, suppose you had a decorator `@with_session()`, which did this:

```
def with_session(fn):
   def go(*args, **kw):
       session.begin(subtransactions=True)
       try:
           ret = fn(*args, **kw)
           session.commit()
           return ret
       except:
           session.rollback()
           raise
   return go
```

The above decorator begins a transaction if one does not exist already, and then commits it, if it were the creator. The "subtransactions" flag means that if *Session.begin()* were already called by an enclosing function, nothing happens except a counter is incremented - this counter is decremented when *Session.commit()* is called and only when it goes back to zero does the actual COMMIT happen. It allows this usage pattern:

```
@with_session
def one():
   # do stuff
   two()


@with_session
def two():
   # etc.

one()

two()
```

`one()` can call `two()`, or `two()` can be called by itself, and the `@with_session` decorator ensures the appropriate "framing" - the transaction boundaries stay on the outermost call level. As you can see, if `two()` calls `flush()` which throws an exception and then issues a `rollback()`, there will *always* be a second `rollback()` performed by the decorator, and possibly a third corresponding to two levels of decorator. If the `flush()` pushed the `rollback()` all the way out to the top of the stack, and then we said that all remaining `rollback()` calls are moot, there is some silent behavior going on there. A poorly written enclosing method might suppress the exception, and then call `commit()` assuming nothing is wrong, and then you have a silent failure condition. The main reason people get this error in fact is because they didn't write clean "framing" code and they would have had other problems down the road.

If you think the above use case is a little exotic, the same kind of thing comes into play if you want to SAVEPOINT- you might call `begin_nested()` several times, and the `commit()`/`rollback()` calls each resolve the most recent `begin_nested()`. The meaning of `rollback()` or `commit()` is dependent upon which enclosing block it is called, and you might have any sequence of `rollback()`/`commit()` in any order, and its the level of nesting that determines their behavior.

In both of the above cases, if `flush()` broke the nesting of transaction blocks, the behavior is, depending on scenario, anywhere from "magic" to silent failure to blatant interruption of code flow.

`flush()` makes its own "subtransaction", so that a transaction is started up regardless of the external transactional state, and when complete it calls `commit()`, or `rollback()` upon failure - but that `rollback()` corresponds to its own subtransaction - it doesn't want to guess how you'd like to handle the external "framing" of the transaction, which could be nested many levels with any combination of subtransactions and real SAVEPOINTs. The job of starting/ending the "frame" is kept consistently with the code external to the `flush()`, and we made a decision that this was the most consistent approach.

5.6.3 How do I make a Query that always adds a certain filter to every query?

See the recipe at PreFilteredQuery.

5.6.4 I've created a mapping against an Outer Join, and while the query returns rows, no objects are returned. Why not?

Rows returned by an outer join may contain NULL for part of the primary key, as the primary key is the composite of both tables. The *`Query`* object ignores incoming rows that don't have an acceptable primary key. Based on the setting of the `allow_partial_pks` flag on *`mapper()`*, a primary key is accepted if the value has at least one non-NULL value, or alternatively if the value has no NULL values. See `allow_partial_pks` at *`mapper()`*.

5.6.5 I'm using `joinedload()` or `lazy=False` to create a JOIN/OUTER JOIN and SQLAlchemy is not constructing the correct query when I try to add a WHERE, ORDER BY, LIMIT, etc. (which relies upon the (OUTER) JOIN)

The joins generated by joined eager loading are only used to fully load related collections, and are designed to have no impact on the primary results of the query. Since they are anonymously aliased, they cannot be referenced directly.

For detail on this beahvior, see *The Zen of Eager Loading*.

5.6.6 Query has no `__len__()`, why not?

The Python `__len__()` magic method applied to an object allows the `len()` builtin to be used to determine the length of the collection. It's intuitive that a SQL query object would link `__len__()` to the *`Query.count()`* method, which emits a *SELECT COUNT*. The reason this is not possible is because evaluating the query as a list would incur two SQL calls instead of one:

```
class Iterates(object):
    def __len__(self):
        print "LEN!"
        return 5

    def __iter__(self):
        print "ITER!"
        return iter([1, 2, 3, 4, 5])

list(Iterates())
```

output:

```
ITER!
LEN!
```

5.6.7 How Do I use Textual SQL with ORM Queries?

See:

- *Using Textual SQL* - Ad-hoc textual blocks with `Query`
- *Using SQL Expressions with Sessions* - Using `Session` with textual SQL directly.

5.6.8 I'm calling `Session.delete(myobject)` and it isn't removed from the parent collection!

See *Deleting from Collections* for a description of this behavior.

5.6.9 why isn't my `__init__()` called when I load objects?

See *Constructors and Object Initialization* for a description of this behavior.

5.6.10 how do I use ON DELETE CASCADE with SA's ORM?

SQLAlchemy will always issue UPDATE or DELETE statements for dependent rows which are currently loaded in the `Session`. For rows which are not loaded, it will by default issue SELECT statements to load those rows and udpate/delete those as well; in other words it assumes there is no ON DELETE CASCADE configured. To configure SQLAlchemy to cooperate with ON DELETE CASCADE, see *Using Passive Deletes*.

5.6.11 I set the "foo_id" attribute on my instance to "7", but the "foo" attribute is still `None` - shouldn't it have loaded Foo with id #7?

The ORM is not constructed in such a way as to support immediate population of relationships driven from foreign key attribute changes - instead, it is designed to work the other way around - foreign key attributes are handled by the ORM behind the scenes, the end user sets up object relationships naturally. Therefore, the recommended way to set `o.foo` is to do just that - set it!:

```
foo = Session.query(Foo).get(7)
o.foo = foo
Session.commit()
```

Manipulation of foreign key attributes is of course entirely legal. However, setting a foreign-key attribute to a new value currently does not trigger an "expire" event of the `relationship()` in which it's involved. This means that for the following sequence:

```
o = Session.query(SomeClass).first()
assert o.foo is None  # accessing an un-set attribute sets it to None
o.foo_id = 7
```

`o.foo` is initialized to `None` when we first accessed it. Setting `o.foo_id = 7` will have the value of "7" as pending, but no flush has occurred - so `o.foo` is still `None`:

```
# attribute is already set to None, has not been
# reconciled with o.foo_id = 7 yet
assert o.foo is None
```

For `o.foo` to load based on the foreign key mutation is usually achieved naturally after the commit, which both flushes the new foreign key value and expires all state:

```
Session.commit()  # expires all attributes

foo_7 = Session.query(Foo).get(7)

assert o.foo is foo_7  # o.foo lazyloads on access
```

A more minimal operation is to expire the attribute individually - this can be performed for any persistent object using *Session.expire()*:

```
o = Session.query(SomeClass).first()
o.foo_id = 7
Session.expire(o, ['foo'])  # object must be persistent for this

foo_7 = Session.query(Foo).get(7)

assert o.foo is foo_7  # o.foo lazyloads on access
```

Note that if the object is not persistent but present in the *Session*, it's known as pending. This means the row for the object has not been INSERTed into the database yet. For such an object, setting `foo_id` does not have meaning until the row is inserted; otherwise there is no row yet:

```
new_obj = SomeClass()
new_obj.foo_id = 7

Session.add(new_obj)

# accessing an un-set attribute sets it to None
assert new_obj.foo is None

Session.flush()  # emits INSERT

# expire this because we already set .foo to None
Session.expire(o, ['foo'])

assert new_obj.foo is foo_7  # now it loads
```

Attribute loading for non-persistent objects

One variant on the "pending" behavior above is if we use the flag `load_on_pending` on *relationship()*. When this flag is set, the lazy loader will emit for `new_obj.foo` before the INSERT proceeds; another variant of this is to use the *Session.enable_relationship_loading()* method, which can "attach" an object to a *Session* in such a way that many-to-one relationships load as according to foreign key attributes regardless of the object being in any particular state. Both techniques are **not recommended for general use**; they were added to suit specific programming scenarios encountered by users which involve the repurposing of the ORM's usual object states.

The recipe ExpireRelationshipOnFKChange features an example using SQLAlchemy events in order to coordinate the setting of foreign key attributes with many-to-one relationships.

5.6.12 How do I walk all objects that are related to a given object?

An object that has other objects related to it will correspond to the *relationship()* constructs set up between mappers. This code fragment will iterate all the objects, correcting for cycles as well:

```
from sqlalchemy import inspect


def walk(obj):
    deque = [obj]

    seen = set()

    while deque:
        obj = deque.pop(0)
        if obj in seen:
            continue
        else:
            seen.add(obj)
            yield obj
        insp = inspect(obj)
        for relationship in insp.mapper.relationships:
            related = getattr(obj, relationship.key)
            if relationship.uselist:
                deque.extend(related)
            elif related is not None:
                deque.append(related)
```

The function can be demonstrated as follows:

```
Base = declarative_base()


class A(Base):
    __tablename__ = 'a'
    id = Column(Integer, primary_key=True)
    bs = relationship("B", backref="a")


class B(Base):
    __tablename__ = 'b'
    id = Column(Integer, primary_key=True)
    a_id = Column(ForeignKey('a.id'))
    c_id = Column(ForeignKey('c.id'))
    c = relationship("C", backref="bs")


class C(Base):
    __tablename__ = 'c'
    id = Column(Integer, primary_key=True)


a1 = A(bs=[B(), B(c=C())])


for obj in walk(a1):
    print obj
```

Output:

```
<__main__.A object at 0x10303b190>
<__main__.B object at 0x103025210>
<__main__.B object at 0x10303b0d0>
<__main__.C object at 0x103025490>
```

5.6.13 Is there a way to automagically have only unique keywords (or other kinds of objects) without doing a query for the keyword and getting a reference to the row containing that keyword?

When people read the many-to-many example in the docs, they get hit with the fact that if you create the same `Keyword` twice, it gets put in the DB twice. Which is somewhat inconvenient.

This UniqueObject recipe was created to address this issue.

CHAPTER 6

Changes and Migration

SQLAlchemy changelogs and migration guides are now integrated within the main documentation.

6.1 Current Migration Guide

6.1.1 What's New in SQLAlchemy 1.1?

About this Document

This document describes changes between SQLAlchemy version 1.0, at the moment the current release series of SQLAlchemy, and SQLAlchemy version 1.1, which is the current development series of SQLAlchemy.

As the 1.1 series is under development, issues that are targeted at this series can be seen under the 1.1 milestone. Please note that the set of issues within the milestone is not fixed; some issues may be moved to later milestones in order to allow for a timely release.

Document last updated: December 4, 2015

Introduction

This guide introduces what's new in SQLAlchemy version 1.1, and also documents changes which affect users migrating their applications from the 1.0 series of SQLAlchemy to 1.1.

Please carefully review the sections on behavioral changes for potentially backwards-incompatible changes in behavior.

Platform / Installer Changes

Setuptools is now required for install

SQLAlchemy's `setup.py` file has for many years supported operation both with Setuptools installed and without; supporting a "fallback" mode that uses straight Distutils. As a Setuptools-less Python environment is now unheard of, and in order to support the featureset of Setuptools more fully, in particular to support py.test's integration with it, `setup.py` now depends on Setuptools fully.

See also:

Installation Guide

#3489

Enabling / Disabling C Extension builds is only via environment variable

The C Extensions build by default during install as long as it is possible. To disable C extension builds, the `DISABLE_SQLALCHEMY_CEXT` environment variable was made available as of SQLAlchemy 0.8.6 / 0.9.4. The previous approach of using the `--without-cextensions` argument has been removed, as it relies on deprecated features of setuptools.

See also:

Installing the C Extensions

#3500

New Features and Improvements - ORM

New Session lifecycle events

The `Session` has long supported events that allow some degree of tracking of state changes to objects, including `SessionEvents.before_attach()`, `SessionEvents.after_attach()`, and `SessionEvents.before_flush()`. The Session documentation also documents major object states at *Quickie Intro to Object States*. However, there has never been system of tracking objects specifically as they pass through these transitions. Additionally, the status of "deleted" objects has historically been murky as the objects act somewhere between the "persistent" and "detached" states.

To clean up this area and allow the realm of session state transition to be fully transparent, a new series of events have been added that are intended to cover every possible way that an object might transition between states, and additionally the "deleted" status has been given its own official state name within the realm of session object states.

New State Transition Events Transitions between all states of an object such as persistent, pending and others can now be intercepted in terms of a session-level event intended to cover a specific transition. Transitions as objects move into a `Session`, move out of a `Session`, and even all the transitions which occur when the transaction is rolled back using `Session.rollback()` are explicitly present in the interface of `SessionEvents`.

In total, there are **ten new events**. A summary of these events is in a newly written documentation section *Object Lifecycle Events*.

New Object State "deleted" is added, deleted objects no longer "persistent" The persistent state of an object in the `Session` has always been documented as an object that has a valid database identity; however in the case of objects that were deleted within a flush, they have always been in a grey area where they are not really "detached" from the `Session` yet, because they can still be restored within a rollback, but are not really "persistent" because their database identity has been deleted and they aren't present in the identity map.

To resolve this grey area given the new events, a new object state deleted is introduced. This state exists between the "persistent" and "detached" states. An object that is marked for deletion via `Session.delete()` remains in the "persistent" state until a flush proceeds; at that point, it is removed from the identity map, moves to the "deleted" state, and the `SessionEvents.persistent_to_deleted()` hook is invoked. If the `Session` object's transaction is rolled back, the object is restored as persistent; the `SessionEvents.deleted_to_persistent()` transition is called. Otherwise if the `Session` object's transaction is committed, the `SessionEvents.deleted_to_detached()` transition is invoked.

Additionally, the `InstanceState.persistent` accessor **no longer returns True** for an object that is in the new "deleted" state; instead, the `InstanceState.deleted` accessor has been enhanced to reliably report on this new state. When the object is detached, the `InstanceState.deleted` returns False and the `InstanceState.detached` accessor is True instead. To determine if an object was deleted either in the current transaction or in a previous transaction, use the `InstanceState.was_deleted` accessor.

Strong Identity Map is Deprecated One of the inspirations for the new series of transition events was to enable leak-proof tracking of objects as they move in and out of the identity map, so that a "strong reference" may be maintained mirroring the object moving in and out of this map. With this new capability, there is no longer any need for the `Session.weak_identity_map` parameter and the corresponding `StrongIdentityMap` object. This option has remained in SQLAlchemy for many years as the "strong-referencing" behavior used to be the only behavior available, and many applications were written to assume this behavior. It has long been recommended that strong-reference tracking of objects not be an intrinsic job of the `Session` and instead be an application-level construct built as needed by the application; the new event model allows even the exact behavior of the strong identity map to be replicated. See *Session Referencing Behavior* for a new recipe illustrating how to replace the strong identity map.

#2677

Changes regarding "unhashable" types

The `Query` object has a well-known behavior of "deduping" returned rows that contain at least one ORM-mapped entity (e.g., a full mapped object, as opposed to individual column values). The primary purpose of this is so that the handling of entities works smoothly in conjunction with the identity map, including to accommodate for the duplicate entities normally represented within joined eager loading, as well as when joins are used for the purposes of filtering on additional columns.

This deduplication relies upon the hashability of the elements within the row. With the introduction of Postgresql's special types like `postgresql.ARRAY`, `postgresql.HSTORE` and `postgresql.JSON`, the experience of types within rows being unhashable and encountering problems here is more prevalent than it was previously.

In fact, SQLAlchemy has since version 0.8 included a flag on datatypes that are noted as "unhashable", however this flag was not used consistently on built in types. As described in *ARRAY and JSON types now correctly specify "unhashable"*, this flag is now set consistently for all of Postgresql's "structural" types.

The "unhashable" flag is also set on the `NullType` type, as `NullType` is used to refer to any expression of unknown type.

Additionally, the treatment of a so-called "unhashable" type is slightly different than its been in previous releases; internally we are using the `id()` function to get a "hash value" from these structures, just as we would any ordinary mapped object. This replaces the previous approach which applied a counter to the object.

#3499

Specific checks added for passing mapped classes, instances as SQL literals

The typing system now has specific checks for passing of SQLAlchemy "inspectable" objects in contexts where they would otherwise be handled as literal values. Any SQLAlchemy built-in object that is legal to pass as a SQL value includes a method `__clause_element__()` which provides a valid SQL expression for that object. For SQLAlchemy objects that don't provide this, such as mapped classes, mappers, and mapped instances, a more informative error message is emitted rather than allowing the DBAPI to receive the object and fail later. An example is illustrated below, where a string-based attribute `User.name` is compared to a full instance of `User()`, rather than against a string value:

```
>>> some_user = User()
>>> q = s.query(User).filter(User.name == some_user)
...
sqlalchemy.exc.ArgumentError: Object <__main__.User object at 0x103167e90> is not legal as a SQL lite
```

The exception is now immediate when the comparison is made between `User.name == some_user`. Previously, a comparison like the above would produce a SQL expression that would only fail once resolved into a DBAPI execution call; the mapped `User` object would ultimately become a bound parameter that would be rejected by the DBAPI.

Note that in the above example, the expression fails because `User.name` is a string-based (e.g. column oriented) attribute. The change does *not* impact the usual case of comparing a many-to-one relationship attribute to an object, which is handled distinctly:

```
>>> # Address.user refers to the User mapper, so
>>> # this is of course still OK!
>>> q = s.query(Address).filter(Address.user == some_user)
```

#3321

New options allowing explicit persistence of NULL over a default

Related to the new JSON-NULL support added to Postgresql as part of *JSON "null" is inserted as expected with ORM operations, regardless of column default present*, the base `TypeEngine` class now supports a method `TypeEngine.evaluates_none()` which allows a positive set of the `None` value on an attribute to be persisted as NULL, rather than omitting the column from the INSERT statement, which has the effect of using the column-level default. This allows a mapper-level configuration of the existing object-level technique of assigning `sql.null()` to the attribute.

See also:

Forcing NULL on a column with a default

#3250

Further Fixes to single-table inheritance querying

Continuing from 1.0's *Change to single-table-inheritance criteria when using from_self(), count()*, the `Query` should no longer inappropriately add the "single inheritance" criteria when the query is against a subquery expression such as an exists:

```
class Widget(Base):
    __tablename__ = 'widget'
    id = Column(Integer, primary_key=True)
    type = Column(String)
    data = Column(String)
    __mapper_args__ = {'polymorphic_on': type}


class FooWidget(Widget):
    __mapper_args__ = {'polymorphic_identity': 'foo'}

q = session.query(FooWidget).filter(FooWidget.data == 'bar').exists()

session.query(q).all()
```

Produces:

```
SELECT EXISTS (SELECT 1
FROM widget
WHERE widget.data = :data_1 AND widget.type IN (:type_1)) AS anon_1
```

The IN clause on the inside is appropriate, in order to limit to FooWidget objects, however previously the IN clause would also be generated a second time on the outside of the subquery.

#3582

Session.merge resolves pending conflicts the same as persistent

The *Session.merge()* method will now track the identities of objects given within a graph to maintain primary key uniqueness before emitting an INSERT. When duplicate objects of the same identity are encountered, non-primary-key attributes are **overwritten** as the objects are encountered, which is essentially non-deterministic. This behavior matches that of how persistent objects, that is objects that are already located in the database via primary key, are already treated, so this behavior is more internally consistent.

Given:

```
u1 = User(id=7, name='x')
u1.orders = [
    Order(description='o1', address=Address(id=1, email_address='a')),
    Order(description='o2', address=Address(id=1, email_address='b')),
    Order(description='o3', address=Address(id=1, email_address='c'))
]

sess = Session()
sess.merge(u1)
```

Above, we merge a `User` object with three new `Order` objects, each referring to a distinct `Address` object, however each is given the same primary key. The current behavior of *Session.merge()* is to look in the identity map for this `Address` object, and use that as the target. If the object is present, meaning that the database already has a row for `Address` with primary key "1", we can see that the `email_address` field of the `Address` will be overwritten three times, in this case with the values a, b and finally c.

However, if the `Address` row for primary key "1" were not present, *Session.merge()* would instead create three separate `Address` instances, and we'd then get a primary key conflict upon INSERT. The new behavior is that the proposed primary key for these `Address` objects are tracked in a separate dictionary so that we merge the state of the three proposed `Address` objects onto one `Address` object to be inserted.

It may have been preferable if the original case emitted some kind of warning that conflicting data were present in a single merge-tree, however the non-deterministic merging of values has been the behavior for many years for the persistent case; it now matches for the pending case. A feature that warns for conflicting values could still be feasible for both cases but would add considerable performance overhead as each column value would have to be compared during the merge.

#3601

New Features and Improvements - Core

The `.autoincrement` directive is no longer implicitly enabled for a composite primary key column

SQLAlchemy has always had the convenience feature of enabling the backend database's "autoincrement" feature for a single-column integer primary key; by "autoincrement" we mean that the database column will include whatever

DDL directives the database provides in order to indicate an auto-incrementing integer identifier, such as the SERIAL keyword on Postgresql or AUTO_INCREMENT on MySQL, and additionally that the dialect will recieve these generated values from the execution of a `Table.insert()` construct using techniques appropriate to that backend.

What's changed is that this feature no longer turns on automatically for a *composite* primary key; previously, a table definition such as:

```
Table(
    'some_table', metadata,
    Column('x', Integer, primary_key=True),
    Column('y', Integer, primary_key=True)
)
```

Would have "autoincrement" semantics applied to the `'x'` column, only because it's first in the list of primary key columns. In order to disable this, one would have to turn off `autoincrement` on all columns:

```
# old way
Table(
    'some_table', metadata,
    Column('x', Integer, primary_key=True, autoincrement=False),
    Column('y', Integer, primary_key=True, autoincrement=False)
)
```

With the new behavior, the composite primary key will not have autoincrement semantics unless a column is marked explcitly with `autoincrement=True`:

```
# column 'y' will be SERIAL/AUTO_INCREMENT/ auto-generating
Table(
    'some_table', metadata,
    Column('x', Integer, primary_key=True),
    Column('y', Integer, primary_key=True, autoincrement=True)
)
```

In order to anticipate some potential backwards-incompatible scenarios, the `Table.insert()` construct will perform more thorough checks for missing primary key values on composite primary key columns that don't have autoincrement set up; given a table such as:

```
Table(
    'b', metadata,
    Column('x', Integer, primary_key=True),
    Column('y', Integer, primary_key=True)
)
```

An INSERT emitted with no values for this table will produce the exception:

```
CompileError: Column 'b.x' is marked as a member of the primary
key for table 'b', but has no Python-side or server-side default
generator indicated, nor does it indicate 'autoincrement=True',
and no explicit value is passed.  Primary key columns may not
store NULL. Note that as of SQLAlchemy 1.1, 'autoincrement=True'
must be indicated explicitly for composite (e.g. multicolumn)
primary keys if AUTO_INCREMENT/SERIAL/IDENTITY behavior is
expected for one of the columns in the primary key. CREATE TABLE
statements are impacted by this change as well on most backends.
```

For a column that is receiving primary key values from a server-side default or something less common such as a trigger, the presence of a value generator can be indicated using `FetchedValue`:

```
Table(
    'b', metadata,
    Column('x', Integer, primary_key=True, server_default=FetchedValue()),
    Column('y', Integer, primary_key=True, server_default=FetchedValue())
)
```

For the very unlikely case where a composite primary key is actually intended to store NULL in one or more of its columns (only supported on SQLite and MySQL), specify the column with `nullable=True`:

```
Table(
    'b', metadata,
    Column('x', Integer, primary_key=True),
    Column('y', Integer, primary_key=True, nullable=True)
)
```

In a related change, the `autoincrement` flag may be set to True on a column that has a client-side or server-side default. This typically will not have much impact on the behavior of the column during an INSERT.

See also:

No more generation of an implicit KEY for composite primary key w/ AUTO_INCREMENT

#3216

A UNION or similar of SELECTs with LIMIT/OFFSET/ORDER BY now parenthesizes the embedded selects

An issue that, like others, was long driven by SQLite's lack of capabilities has now been enhanced to work on all supporting backends. We refer to a query that is a UNION of SELECT statements that themselves contain row-limiting or ordering features which include LIMIT, OFFSET, and/or ORDER BY:

```
(SELECT x FROM table1 ORDER BY y LIMIT 1) UNION
(SELECT x FROM table2 ORDER BY y LIMIT 2)
```

The above query requires parenthesis within each sub-select in order to group the sub-results correctly. Production of the above statement in SQLAlchemy Core looks like:

```
stmt1 = select([table1.c.x]).order_by(table1.c.y).limit(1)
stmt2 = select([table1.c.x]).order_by(table2.c.y).limit(2)

stmt = union(stmt1, stmt2)
```

Previously, the above construct would not produce parenthesization for the inner SELECT statements, producing a query that fails on all backends.

The above formats will **continue to fail on SQLite**; additionally, the format that includes ORDER BY but no LIMIT/SELECT will **continue to fail on Oracle**. This is not a backwards-incompatible change, because the queries fail without the parentheses as well; with the fix, the queries at least work on all other databases.

In all cases, in order to produce a UNION of limited SELECT statements that also works on SQLite and in all cases on Oracle, the subqueries must be a SELECT of an ALIAS:

```
stmt1 = select([table1.c.x]).order_by(table1.c.y).limit(1).alias().select()
stmt2 = select([table2.c.x]).order_by(table2.c.y).limit(2).alias().select()

stmt = union(stmt1, stmt2)
```

This workaround works on all SQLAlchemy versions. In the ORM, it looks like:

```
stmt1 = session.query(Model1).order_by(Model1.y).limit(1).subquery().select()
stmt2 = session.query(Model2).order_by(Model2.y).limit(1).subquery().select()

stmt = session.query(Model1).from_statement(stmt1.union(stmt2))
```

The behavior here has many parallels to the "join rewriting" behavior introduced in SQLAlchemy 0.9 in *Many JOIN and LEFT OUTER JOIN expressions will no longer be wrapped in (SELECT * FROM ..) AS ANON_1*; however in this case we have opted not to add new rewriting behavior to accommodate this case for SQLite. The existing rewriting behavior is very complicated already, and the case of UNIONs with parenthesized SELECT statements is much less common than the "right-nested-join" use case of that feature.

#2528

JSON support added to Core

As MySQL now has a JSON datatype in addition to the Postgresql JSON datatype, the core now gains a `sqlalchemy.types.JSON` datatype that is the basis for both of these. Using this type allows access to the "getitem" operator as well as the "getpath" operator in a way that is agnostic across Postgresql and MySQL.

The new datatype also has a series of improvements to the handling of NULL values as well as expression handling.

See also:

MySQL JSON Support

`types.JSON`

`postgresql.JSON`

`mysql.JSON`

#3619

JSON "null" is inserted as expected with ORM operations, regardless of column default present The `types.JSON` type and its descendant types `postgresql.JSON` and `mysql.JSON` have a flag `types.JSON.none_as_null` which when set to True indicates that the Python value `None` should translate into a SQL NULL rather than a JSON NULL value. This flag defaults to False, which means that the column should *never* insert SQL NULL or fall back to a default unless the `null()` constant were used. However, this would fail in the ORM under two circumstances; one is when the column also contained a default or server_default value, a positive value of `None` on the mapped attribute would still result in the column-level default being triggered, replacing the `None` value:

```
obj = MyObject(json_value=None)
session.add(obj)
session.commit()   # would fire off default / server_default, not encode "'none'"
```

The other is when the `Session.bulk_insert_mappings()` method were used, `None` would be ignored in all cases:

```
session.bulk_insert_mappings(
    MyObject,
    [{"json_value": None}])  # would insert SQL NULL and/or trigger defaults
```

The *types.JSON* type now implements the *TypeEngine.should_evaluate_none* flag, indicating that None should not be ignored here; it is configured automatically based on the value of *types.JSON.none_as_null*. Thanks to #3061, we can differentiate when the value None is actively set by the user versus when it was never set at all.

If the attribute is not set at all, then column level defaults *will* fire off and/or SQL NULL will be inserted as expected, as was the behavior previously. Below, the two variants are illustrated:

```
obj = MyObject(json_value=None)
session.add(obj)
session.commit()   # *will not* fire off column defaults, will insert JSON 'null'

obj = MyObject()
session.add(obj)
session.commit()   # *will* fire off column defaults, and/or insert SQL NULL
```

The feature applies as well to the new base *types.JSON* type and its descendant types.

#3514

New JSON.NULL Constant Added To ensure that an application can always have full control at the value level of whether a *types.JSON*, *postgresql.JSON*, *mysql.JSON*, or *postgresql.JSONB* column should receive a SQL NULL or JSON "null" value, the constant *types.JSON.NULL* has been added, which in conjunction with *null()* can be used to determine fully between SQL NULL and JSON "null", regardless of what *types.JSON.none_as_null* is set to:

```
from sqlalchemy import null
from sqlalchemy.dialects.postgresql import JSON

obj1 = MyObject(json_value=null())  # will *always* insert SQL NULL
obj2 = MyObject(json_value=JSON.NULL)  # will *always* insert JSON string "null"

session.add_all([obj1, obj2])
session.commit()
```

The feature applies as well to the new base *types.JSON* type and its descendant types.

#3514

Array support added to Core; new ANY and ALL operators

Along with the enhancements made to the Postgresql *postgresql.ARRAY* type described in *Correct SQL Types are Established from Indexed Access of ARRAY, JSON, HSTORE*, the base class of *postgresql.ARRAY* itself has been moved to Core in a new class *types.ARRAY*.

Arrays are part of the SQL standard, as are several array-oriented functions such as array_agg() and unnest(). In support of these constructs for not just PostgreSQL but also potentially for other array-capable backends in the future such as DB2, the majority of array logic for SQL expressions is now in Core. The *types.ARRAY* type still **only works on Postgresql**, however it can be used directly, supporting special array use cases such as indexed access, as well as support for the ANY and ALL:

```
mytable = Table("mytable", metadata,
        Column("data", ARRAY(Integer, dimensions=2))
    )

expr = mytable.c.data[5][6]
```

```
expr = mytable.c.data[5].any(12)
```

In support of ANY and ALL, the *types.ARRAY* type retains the same *types.ARRAY.Comparator.any()* and *types.ARRAY.Comparator.all()* methods from the PostgreSQL type, but also exports these operations to new standalone operator functions *sql.expression.any_()* and *sql.expression.all_()*. These two functions work in more of the traditional SQL way, allowing a right-side expression form such as:

```
from sqlalchemy import any_, all_

select([mytable]).where(12 == any_(mytable.c.data[5]))
```

For the PostgreSQL-specific operators "contains", "contained_by", and "overlaps", one should continue to use the *postgresql.ARRAY* type directly, which provides all functionality of the *types.ARRAY* type as well.

The *sql.expression.any_()* and *sql.expression.all_()* operators are open-ended at the Core level, however their interpretation by backend databases is limited. On the Postgresql backend, the two operators **only accept array values**. Whereas on the MySQL backend, they **only accept subquery values**. On MySQL, one can use an expression such as:

```
from sqlalchemy import any_, all_

subq = select([mytable.c.value])
select([mytable]).where(12 > any_(subq))
```

#3516

New Function features, "WITHIN GROUP", array_agg and set aggregate functions

With the new *types.ARRAY* type we can also implement a pre-typed function for the `array_agg()` SQL function that returns an array, which is now available using *array_agg*:

```
from sqlalchemy import func
stmt = select([func.array_agg(table.c.value)])
```

A Postgresql element for an aggregate ORDER BY is also added via *postgresql.aggregate_order_by*:

```
from sqlalchemy.dialects.postgresql import aggregate_order_by
expr = func.array_agg(aggregate_order_by(table.c.a, table.c.b.desc()))
stmt = select([expr])
```

Producing:

```
SELECT array_agg(table1.a ORDER BY table1.b DESC) AS array_agg_1 FROM table1
```

The PG dialect itself also provides an *postgresql.array_agg()* wrapper to ensure the *postgresql.ARRAY* type:

```
from sqlalchemy.dialects.postgresql import array_agg
stmt = select([array_agg(table.c.value).contains('foo')])
```

Additionally, functions like `percentile_cont()`, `percentile_disc()`, `rank()`, `dense_rank()` and others that require an ordering via `WITHIN GROUP (ORDER BY <expr>)` are now available via the *FunctionElement.within_group()* modifier:

```
from sqlalchemy import func
stmt = select([
    department.c.id,
    func.percentile_cont(0.5).within_group(
        department.c.salary.desc()
    )
])
```

The above statement would produce SQL similar to:

```
SELECT department.id, percentile_cont(0.5)
WITHIN GROUP (ORDER BY department.salary DESC)
```

Placeholders with correct return types are now provided for these functions, and include *percentile_cont*, *percentile_disc*, *rank*, *dense_rank*, *mode*, *percent_rank*, and *cume_dist*.

#3132 #1370

TypeDecorator now works with Enum, Boolean, "schema" types automatically

The *SchemaType* types include types such as *Enum* and *Boolean* which, in addition to corresponding to a database type, also generate either a CHECK constraint or in the case of Postgresql ENUM a new CREATE TYPE statement, will now work automatically with *TypeDecorator* recipes. Previously, a *TypeDecorator* for an *postgresql.ENUM* had to look like this:

```
# old way
class MyEnum(TypeDecorator, SchemaType):
    impl = postgresql.ENUM('one', 'two', 'three', name='myenum')

    def _set_table(self, table):
        self.impl._set_table(table)
```

The *TypeDecorator* now propagates those additional events so it can be done like any other type:

```
# new way
class MyEnum(TypeDecorator):
    impl = postgresql.ENUM('one', 'two', 'three', name='myenum')
```

#2919

Multi-Tenancy Schema Translation for Table objects

To support the use case of an application that uses the same set of *Table* objects in many schemas, such as schema-per-user, a new execution option *Connection.execution_options.schema_translate_map* is added. Using this mapping, a set of *Table* objects can be made on a per-connection basis to refer to any set of schemas instead of the *Table.schema* to which they were assigned. The translation works for DDL and SQL generation, as well as with the ORM.

For example, if the `User` class were assigned the schema "per_user":

```
class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
```

```
    __table_args__ = {'schema': 'per_user'}
```

On each request, the `Session` can be set up to refer to a different schema each time:

```
session = Session()
session.connection(execution_options={
    "schema_translate_map": {"per_user": "account_one"}})

# will query from the ``account_one.user`` table
session.query(User).get(5)
```

See also:

Translation of Schema Names

#2685

The type_coerce function is now a persistent SQL element

The `expression.type_coerce()` function previously would return an object either of type `BindParameter` or `Label`, depending on the input. An effect this would have was that in the case where expression transformations were used, such as the conversion of an element from a `Column` to a `BindParameter` that's critical to ORM-level lazy loading, the type coercion information would not be used since it would have been lost already.

To improve this behavior, the function now returns a persistent `TypeCoerce` container around the given expression, which itself remains unaffected; this construct is evaluated explicitly by the SQL compiler. This allows for the coercion of the inner expression to be maintained no matter how the statement is modified, including if the contained element is replaced with a different one, as is common within the ORM's lazy loading feature.

The test case illustrating the effect makes use of a heterogeneous primaryjoin condition in conjunction with custom types and lazy loading. Given a custom type that applies a CAST as a "bind expression":

```
class StringAsInt(TypeDecorator):
    impl = String

    def column_expression(self, col):
        return cast(col, Integer)

    def bind_expression(self, value):
        return cast(value, String)
```

Then, a mapping where we are equating a string "id" column on one table to an integer "id" column on the other:

```
class Person(Base):
    __tablename__ = 'person'
    id = Column(StringAsInt, primary_key=True)

    pets = relationship(
        'Pets',
        primaryjoin=(
            'foreign(Pets.person_id)'
            '==cast(type_coerce(Person.id, Integer), Integer)'
        )
    )

class Pets(Base):
```

```
    __tablename__ = 'pets'
    id = Column('id', Integer, primary_key=True)
    person_id = Column('person_id', Integer)
```

Above, in the *relationship.primaryjoin* expression, we are using *type_coerce()* to handle bound parameters passed via lazyloading as integers, since we already know these will come from our `StringAsInt` type which maintains the value as an integer in Python. We are then using *cast()* so that as a SQL expression, the VARCHAR "id" column will be CAST to an integer for a regular non- converted join as with *Query.join()* or *orm.joinedload()*. That is, a joinedload of `.pets` looks like:

```
SELECT person.id AS person_id, pets_1.id AS pets_1_id,
       pets_1.person_id AS pets_1_person_id
FROM person
LEFT OUTER JOIN pets AS pets_1
ON pets_1.person_id = CAST(person.id AS INTEGER)
```

Without the CAST in the ON clause of the join, strongly-typed databases such as Postgresql will refuse to implicitly compare the integer and fail.

The lazyload case of `.pets` relies upon replacing the `Person.id` column at load time with a bound parameter, which receives a Python-loaded value. This replacement is specifically where the intent of our *type_coerce()* function would be lost. Prior to the change, this lazy load comes out as:

```
SELECT pets.id AS pets_id, pets.person_id AS pets_person_id
FROM pets
WHERE pets.person_id = CAST(CAST(%(param_1)s AS VARCHAR) AS INTEGER)
{'param_1': 5}
```

Where above, we see that our in-Python value of `5` is CAST first to a VARCHAR, then back to an INTEGER in SQL; a double CAST which works, but is nevertheless not what we asked for.

With the change, the *type_coerce()* function maintains a wrapper even after the column is swapped out for a bound parameter, and the query now looks like:

```
SELECT pets.id AS pets_id, pets.person_id AS pets_person_id
FROM pets
WHERE pets.person_id = CAST(%(param_1)s AS INTEGER)
{'param_1': 5}
```

Where our outer CAST that's in our primaryjoin still takes effect, but the needless CAST that's in part of the `StringAsInt` custom type is removed as intended by the *type_coerce()* function.

#3531

Key Behavioral Changes - ORM

Key Behavioral Changes - Core

Dialect Improvements and Changes - Postgresql

ARRAY and JSON types now correctly specify "unhashable"

As described in *Changes regarding "unhashable" types*, the ORM relies upon being able to produce a hash function for column values when a query's selected entities mixes full ORM entities with column expressions. The `hashable=False` flag is now correctly set on all of PG's "data structure" types, including *postgresql.ARRAY*

and `postgresql.JSON`. The `JSONB` and `HSTORE` types already included this flag. For `postgresql.ARRAY`, this is conditional based on the `postgresql.ARRAY.as_tuple` flag, however it should no longer be necessary to set this flag in order to have an array value present in a composed ORM row.

See also:

Changes regarding "unhashable" types

Correct SQL Types are Established from Indexed Access of ARRAY, JSON, HSTORE

#3499

Correct SQL Types are Established from Indexed Access of ARRAY, JSON, HSTORE

For all three of `ARRAY`, `JSON` and `HSTORE`, the SQL type assigned to the expression returned by indexed access, e.g. `col[someindex]`, should be correct in all cases.

This includes:

- The SQL type assigned to indexed access of an `ARRAY` takes into account the number of dimensions configured. An `ARRAY` with three dimensions will return a SQL expression with a type of `ARRAY` of one less dimension. Given a column with type `ARRAY(Integer, dimensions=3)`, we can now perform this expression:

```
int_expr = col[5][6][7]   # returns an Integer expression object
```

  Previously, the indexed access to `col[5]` would return an expression of type `Integer` where we could no longer perform indexed access for the remaining dimensions, unless we used `cast()` or `type_coerce()`.

- The `JSON` and `JSONB` types now mirror what Postgresql itself does for indexed access. This means that all indexed access for a `JSON` or `JSONB` type returns an expression that itself is *always* `JSON` or `JSONB` itself, unless the `astext` modifier is used. This means that whether the indexed access of the JSON structure ultimately refers to a string, list, number, or other JSON structure, Postgresql always considers it to be JSON itself unless it is explicitly cast differently. Like the `ARRAY` type, this means that it is now straightforward to produce JSON expressions with multiple levels of indexed access:

```
json_expr = json_col['key1']['attr1'][5]
```

- The "textual" type that is returned by indexed access of `HSTORE` as well as the "textual" type that is returned by indexed access of `JSON` and `JSONB` in conjunction with the `astext` modifier is now configurable; it defaults to `Text` in both cases but can be set to a user-defined type using the `postgresql.JSON.astext_type` or `postgresql.HSTORE.text_type` parameters.

See also:

The JSON cast() operation now requires .astext is called explicitly

#3499 #3487

The JSON cast() operation now requires `.astext` is called explicitly

As part of the changes in *Correct SQL Types are Established from Indexed Access of ARRAY, JSON, HSTORE*, the workings of the `ColumnElement.cast()` operator on `postgresql.JSON` and `postgresql.JSONB` no longer implictly invoke the `postgresql.JSON.Comparator.astext` modifier; Postgresql's JSON/JSONB types support CAST operations to each other without the "astext" aspect.

This means that in most cases, an application that was doing this:

```
expr = json_col['somekey'].cast(Integer)
```

Will now need to change to this:

```
expr = json_col['somekey'].astext.cast(Integer)
```

ARRAY with ENUM will now emit CREATE TYPE for the ENUM

A table definition like the following will now emit CREATE TYPE as expected:

```
enum = Enum(
    'manager', 'place_admin', 'carwash_admin',
    'parking_admin', 'service_admin', 'tire_admin',
    'mechanic', 'carwasher', 'tire_mechanic', name="work_place_roles")

class WorkPlacement(Base):
    __tablename__ = 'work_placement'
    id = Column(Integer, primary_key=True)
    roles = Column(ARRAY(enum))


e = create_engine("postgresql://scott:tiger@localhost/test", echo=True)
Base.metadata.create_all(e)
```

emits:

```
CREATE TYPE work_place_roles AS ENUM (
    'manager', 'place_admin', 'carwash_admin', 'parking_admin',
    'service_admin', 'tire_admin', 'mechanic', 'carwasher',
    'tire_mechanic')

CREATE TABLE work_placement (
    id SERIAL NOT NULL,
    roles work_place_roles[],
    PRIMARY KEY (id)
)
```

#2729

Dialect Improvements and Changes - MySQL

MySQL JSON Support

A new type *mysql.JSON* is added to the MySQL dialect supporting the JSON type newly added to MySQL 5.7. This type provides both persistence of JSON as well as rudimentary indexed-access using the `JSON_EXTRACT` function internally. An indexable JSON column that works across MySQL and Postgresql can be achieved by using the *types.JSON* datatype common to both MySQL and Postgresql.

See also:

JSON support added to Core

#3547

No more generation of an implicit KEY for composite primary key w/ AUTO_INCREMENT

The MySQL dialect had the behavior such that if a composite primary key on an InnoDB table featured AUTO_INCREMENT on one of its columns which was not the first column, e.g.:

```
t = Table(
    'some_table', metadata,
    Column('x', Integer, primary_key=True, autoincrement=False),
    Column('y', Integer, primary_key=True, autoincrement=True),
    mysql_engine='InnoDB'
)
```

DDL such as the following would be generated:

```
CREATE TABLE some_table (
    x INTEGER NOT NULL,
    y INTEGER NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (x, y),
    KEY idx_autoinc_y (y)
)ENGINE=InnoDB
```

Note the above "KEY" with an auto-generated name; this is a change that found its way into the dialect many years ago in response to the issue that the AUTO_INCREMENT would otherwise fail on InnoDB without this additional KEY.

This workaround has been removed and replaced with the much better system of just stating the AUTO_INCREMENT column *first* within the primary key:

```
CREATE TABLE some_table (
    x INTEGER NOT NULL,
    y INTEGER NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (y, x)
)ENGINE=InnoDB
```

Along with the change *The .autoincrement directive is no longer implicitly enabled for a composite primary key column*, composite primary keys with or without auto increment are now easier to specify; `Column.autoincrement` now defaults to the value `"auto"` and the `autoincrement=False` directives are no longer needed:

```
t = Table(
    'some_table', metadata,
    Column('x', Integer, primary_key=True),
    Column('y', Integer, primary_key=True, autoincrement=True),
    mysql_engine='InnoDB'
)
```

Dialect Improvements and Changes - SQLite

Improved Support for Remote Schemas

The SQLite dialect now implements `Inspector.get_schema_names()` and additionally has improved support for tables and indexes that are created and reflected from a remote schema, which in SQLite is a database that is assigned a name via the `ATTACH` statement; previously, the `CREATE INDEX` DDL didn't work correctly for a schema-bound table and the `Inspector.get_foreign_keys()` method will now indicate the given schema in the results. Cross-schema foreign keys aren't supported.

Dialect Improvements and Changes - SQL Server

String / varlength types no longer represent "max" explicitly on reflection

When reflecting a type such as *String*, *Text*, etc. which includes a length, an "un-lengthed" type under SQL Server would copy the "length" parameter as the value "max":

```
>>> from sqlalchemy import create_engine, inspect
>>> engine = create_engine('mssql+pyodbc://scott:tiger@ms_2008', echo=True)
>>> engine.execute("create table s (x varchar(max), y varbinary(max))")
>>> insp = inspect(engine)
>>> for col in insp.get_columns("s"):
...     print col['type'].__class__, col['type'].length
...
<class 'sqlalchemy.sql.sqltypes.VARCHAR'> max
<class 'sqlalchemy.dialects.mssql.base.VARBINARY'> max
```

The "length" parameter in the base types is expected to be an integer value or None only; None indicates unbounded length which the SQL Server dialect interprets as "max". The fix then is so that these lengths come out as None, so that the type objects work in non-SQL Server contexts:

```
>>> for col in insp.get_columns("s"):
...     print col['type'].__class__, col['type'].length
...
<class 'sqlalchemy.sql.sqltypes.VARCHAR'> None
<class 'sqlalchemy.dialects.mssql.base.VARBINARY'> None
```

Applications which may have been relying on a direct comparison of the "length" value to the string "max" should consider the value of None to mean the same thing.

#3504

The legacy_schema_aliasing flag is now set to False

SQLAlchemy 1.0.5 introduced the legacy_schema_aliasing flag to the MSSQL dialect, allowing so-called "legacy mode" aliasing to be turned off. This aliasing attempts to turn schema-qualified tables into aliases; given a table such as:

```
account_table = Table(
    'account', metadata,
    Column('id', Integer, primary_key=True),
    Column('info', String(100)),
    schema="customer_schema"
)
```

The legacy mode of behavior will attempt to turn a schema-qualified table name into an alias:

```
>>> eng = create_engine("mssql+pymssql://mydsn", legacy_schema_aliasing=True)
>>> print(account_table.select().compile(eng))
SELECT account_1.id, account_1.info
FROM customer_schema.account AS account_1
```

However, this aliasing has been shown to be unnecessary and in many cases produces incorrect SQL.

In SQLAlchemy 1.1, the `legacy_schema_aliasing` flag now defaults to False, disabling this mode of behavior and allowing the MSSQL dialect to behave normally with schema-qualified tables. For applications which may rely on this behavior, set the flag back to True.

#3434

Dialect Improvements and Changes - Oracle

6.2 Change logs

6.2.1 1.1 Changelog

1.1.0b1

no release date

orm

- **[orm] [feature]** The `SessionEvents` suite now includes events to allow unambiguous tracking of all object lifecycle state transitions in terms of the `Session` itself, e.g. pending, transient, persistent, detached. The state of the object within each event is also defined.

  See also:

  New Session lifecycle events

  ¶ References: #2677

- **[orm] [feature]** Added a new session lifecycle state deleted. This new state represents an object that has been deleted from the persistent state and will move to the detached state once the transaction is committed. This resolves the long-standing issue that objects which were deleted existed in a gray area between persistent and detached. The `InstanceState.persistent` accessor will **no longer** report on a deleted object as persistent; the `InstanceState.deleted` accessor will instead be True for these objects, until they become detached.

  See also:

  New Session lifecycle events

  ¶ References: #2677

- **[orm] [feature]** Added new checks for the common error case of passing mapped classes or mapped instances into contexts where they are interpreted as SQL bound parameters; a new exception is raised for this.

  See also:

  Specific checks added for passing mapped classes, instances as SQL literals

  ¶ References: #3321

- **[orm] [bug]** The `Session.merge()` method now tracks pending objects by primary key before emitting an INSERT, and merges distinct objects with duplicate primary keys together as they are encountered, which is essentially semi-deterministic at best. This behavior matches what happens already with persistent objects.

  See also:

  Session.merge resolves pending conflicts the same as persistent

  ¶ References: #3601

- **[orm] [bug]** Fixed bug where the "single table inheritance" criteria would be added onto the end of a query in some inappropriate situations, such as when querying from an exists() of a single-inheritance subclass.

  See also:

  Further Fixes to single-table inheritance querying

  ¶ References: #3582

- **[orm] [bug]** Added a new type-level modifier `TypeEngine.evaluates_none()` which indicates to the ORM that a positive set of None should be persisted as the value NULL, instead of omitting the column from the INSERT statement. This feature is used both as part of the implementation for #3514 as well as a standalone feature available on any type.

  See also:

  New options allowing explicit persistence of NULL over a default

  ¶ References: #3250

- **[orm] [bug]** Internal calls to "bookkeeping" functions within `Session.bulk_save_objects()` and related bulk methods have been scaled back to the extent that this functionality is not currently used, e.g. checks for column default values to be fetched after an INSERT or UPDATE statement. ¶ References: #3526

- **[orm] [bug] [postgresql]** Additional fixes have been made regarding the value of `None` in conjunction with the Postgresql `postgresql.JSON` type. When the `JSON.none_as_null` flag is left at its default value of `False`, the ORM will now correctly insert the Json "'null"' string into the column whenever the value on the ORM object is set to the value `None` or when the value `None` is used with `Session.bulk_insert_mappings()`, **including** if the column has a default or server default on it.

  See also:

  JSON "null" is inserted as expected with ORM operations, regardless of column default present

  New options allowing explicit persistence of NULL over a default

  ¶ References: #3514

- **[orm] [change]** The `Session.weak_identity_map` parameter is deprecated. See the new recipe at *Session Referencing Behavior* for an event-based approach to maintaining strong identity map behavior.

  See also:

  New Session lifecycle events

  ¶ References: #2677

engine

- **[engine] [feature]** Multi-tenancy schema translation for `Table` objects is added. This supports the use case of an application that uses the same set of `Table` objects in many schemas, such as schema-per-user. A new execution option `Connection.execution_options.schema_translate_map` is added.

  See also:

  Multi-Tenancy Schema Translation for Table objects

  ¶ References: #2685

- **[engine] [feature]** Added a new entrypoint system to the engine to allow "plugins" to be stated in the query string for a URL. Custom plugins can be written which will be given the chance up front to alter and/or consume the engine's URL and keyword arguments, and then at engine create time will be given the engine itself to allow additional modifications or event registration. Plugins are written as a subclass of `CreateEnginePlugin`; see that class for details. ¶ References: #3536

sql

- **[sql] [feature]** Added a new type to core `types.JSON`. This is the base of the PostgreSQL `postgresql.JSON` type as well as that of the new `mysql.JSON` type, so that a PG/MySQL-agnostic JSON column may be used. The type features basic index and path searching support.

  See also:

  JSON support added to Core

  ¶ References: #3619

- **[sql] [feature]** Added support for "set-aggregate" functions of the form `<function> WITHIN GROUP (ORDER BY <criteria>)`, using the method `FunctionElement.within_group()`. A series of common set-aggregate functions with return types derived from the set have been added. This includes functions like `percentile_cont`, `dense_rank` and others.

  See also:

  New Function features, "WITHIN GROUP", array_agg and set aggregate functions

  ¶ References: #1370

- **[sql] [feature] [postgresql]** Added support for the SQL-standard function `array_agg`, which automatically returns an `postgresql.ARRAY` of the correct type and supports index / slice operations, as well as `postgresql.array_agg()`, which returns a `postgresql.ARRAY` with additional comparison features. As arrays are only supported on Postgresql at the moment, only actually works on Postgresql. Also added a new construct `postgresql.aggregate_order_by` in support of PG's "ORDER BY" extension.

  See also:

  New Function features, "WITHIN GROUP", array_agg and set aggregate functions

  ¶ References: #3132

- **[sql] [feature]** Added a new type to core `types.ARRAY`. This is the base of the PostgreSQL `postgresql.ARRAY` type, and is now part of Core to begin supporting various SQL-standard array-supporting features including some functions and eventual support for native arrays on other databases that have an "array" concept, such as DB2 or Oracle. Additionally, new operators `expression.any_()` and `expression.all_()` have been added. These support not just array constructs on Postgresql, but also subqueries that are usable on MySQL (but sadly not on Postgresql).

  See also:

  Array support added to Core; new ANY and ALL operators

  ¶ References: #3516

- **[sql] [bug]** Fixed an assertion that would raise somewhat inappropriately if a `Index` were associated with a `Column` that is associated with a lower-case-t `TableClause`; the association should be ignored for the purposes of associating the index with a `Table`. ¶ References: #3616

- **[sql] [bug]** The `type_coerce()` construct is now a fully fledged Core expression element which is late-evaluated at compile time. Previously, the function was only a conversion function which would handle different expression inputs by returning either a `Label` of a column-oriented expression or a copy of a given `BindParameter` object, which in particular prevented the operation from being logically maintained when an ORM-level expression transformation would convert a column to a bound parameter (e.g. for lazy loading).

  See also:

  The type_coerce function is now a persistent SQL element

  ¶ References: #3531

- **[sql] [bug]** The `TypeDecorator` type extender will now work in conjunction with a `SchemaType` implementation, typically `Enum` or `Boolean` with regards to ensuring that the per-table events are propagated from the implementation type to the outer type. These events are used to ensure that the constraints or Postgresql types (e.g. ENUM) are correctly created (and possibly dropped) along with the parent table.

  See also:

  TypeDecorator now works with Enum, Boolean, "schema" types automatically

  ¶ References: #2919

- **[sql] [bug]** The behavior of the `union()` construct and related constructs such as `Query.union()` now handle the case where the embedded SELECT statements need to be parenthesized due to the fact that they include LIMIT, OFFSET and/or ORDER BY. These queries **do not work on SQLite**, and will fail on that backend as they did before, but should now work on all other backends.

  See also:

  A UNION or similar of SELECTs with LIMIT/OFFSET/ORDER BY now parenthesizes the embedded selects

  ¶ References: #2528

- **[sql] [mysql] [change]** The system by which a `Column` considers itself to be an "auto increment" column has been changed, such that autoincrement is no longer implicitly enabled for a `Table` that has a composite primary key. In order to accommodate being able to enable autoincrement for a composite PK member column while at the same time maintaining SQLAlchemy's long standing behavior of enabling implicit autoincrement for a single integer primary key, a third state has been added to the `Column.autoincrement` parameter `"auto"`, which is now the default.

  See also:

  The .autoincrement directive is no longer implicitly enabled for a composite primary key column

  No more generation of an implicit KEY for composite primary key w/ AUTO_INCREMENT

  ¶ References: #3216

schema

- **[schema] [enhancement]** The default generation functions passed to `Column` objects are now run through "update_wrapper", or an equivalent function if a callable non-function is passed, so that introspection tools preserve the name and docstring of the wrapped function. Pull request courtesy hsum. ¶ References: pull request github:204

postgresql

- **[postgresql] [feature]** Added a new constant `postgresql.JSON.NULL`, indicating that the JSON NULL value should be used for a value regardless of other settings.

  See also:

  New JSON.NULL Constant Added

  ¶ References: #3514

- **[postgresql] [bug]** Added support for reflecting the source of materialized views to the Postgresql version of the `Inspector.get_view_definition()` method. ¶ References: #3587

- **[postgresql] [bug]** The use of a `postgresql.ARRAY` object that refers to a `types.Enum` or `postgresql.ENUM` subtype will now emit the expected "CREATE TYPE" and "DROP TYPE" DDL when the type is used within a "CREATE TABLE" or "DROP TABLE".

See also:

ARRAY with ENUM will now emit CREATE TYPE for the ENUM

¶ References: #2729

- **[postgresql] [bug]** The "hashable" flag on special datatypes such as `postgresql.ARRAY`, `postgresql.JSON` and `postgresql.HSTORE` is now set to False, which allows these types to be fetchable in ORM queries that include entities within the row.

  See also:

  Changes regarding "unhashable" types

  ARRAY and JSON types now correctly specify "unhashable"

  ¶ References: #3499

- **[postgresql] [bug]** The Postgresql `postgresql.ARRAY` type now supports multidimensional indexed access, e.g. expressions such as `somecol[5][6]` without any need for explicit casts or type coercions, provided that the `postgresql.ARRAY.dimensions` parameter is set to the desired number of dimensions.

  See also:

  Correct SQL Types are Established from Indexed Access of ARRAY, JSON, HSTORE

  ¶ References: #3487

- **[postgresql] [bug]** The return type for the `postgresql.JSON` and `postgresql.JSONB` when using indexed access has been fixed to work like Postgresql itself, and returns an expression that itself is of type `postgresql.JSON` or `postgresql.JSONB`. Previously, the accessor would return `NullType` which disallowed subsequent JSON-like operators to be used.

  See also:

  Correct SQL Types are Established from Indexed Access of ARRAY, JSON, HSTORE

  ¶ References: #3503

- **[postgresql] [bug]** The `postgresql.JSON`, `postgresql.JSONB` and `postgresql.HSTORE` datatypes now allow full control over the return type from an indexed textual access operation, either `column[someindex].astext` for a JSON type or `column[someindex]` for an HSTORE type, via the `postgresql.JSON.astext_type` and `postgresql.HSTORE.text_type` parameters.

  See also:

  Correct SQL Types are Established from Indexed Access of ARRAY, JSON, HSTORE

  ¶ References: #3503

- **[postgresql] [bug]** The `postgresql.JSON.Comparator.astext` modifier no longer calls upon `ColumnElement.cast()` implicitly, as PG's JSON/JSONB types allow cross-casting between each other as well. Code that makes use of `ColumnElement.cast()` on JSON indexed access, e.g. `col[someindex].cast(Integer)`, will need to be changed to call `postgresql.JSON.Comparator.astext` explicitly.

  See also:

  The JSON cast() operation now requires .astext is called explicitly

  ¶ References: #3503

mysql

- **[mysql] [feature]** Added `mysql.JSON` for MySQL 5.7. The JSON type provides persistence of JSON values in MySQL as well as basic operator support of "getitem" and "getpath", making use of the `JSON_EXTRACT` function in order to refer to individual paths in a JSON structure.

  See also:

  MySQL JSON Support

  ¶ References: #3547

- **[mysql] [change]** The MySQL dialect no longer generates an extra "KEY" directive when generating CREATE TABLE DDL for a table using InnoDB with a composite primary key with AUTO_INCREMENT on a column that isn't the first column; to overcome InnoDB's limitation here, the PRIMARY KEY constraint is now generated with the AUTO_INCREMENT column placed first in the list of columns.

  See also:

  No more generation of an implicit KEY for composite primary key w/ AUTO_INCREMENT

  The .autoincrement directive is no longer implicitly enabled for a composite primary key column

  ¶ References: #3216

sqlite

- **[sqlite] [change]** Added support to the SQLite dialect for the `Inspector.get_schema_names()` method to work with SQLite; pull request courtesy Brian Van Klaveren. Also repaired support for creation of indexes with schemas as well as reflection of foreign key constraints in schema-bound tables.

  See also:

  Improved Support for Remote Schemas

  ¶ References: pull request github:198

mssql

- **[mssql] [bug]** Fixed issue where the SQL Server dialect would reflect a string- or other variable-length column type with unbounded length by assigning the token `"max"` to the length attribute of the string. While using the `"max"` token explicitly is supported by the SQL Server dialect, it isn't part of the normal contract of the base string types, and instead the length should just be left as None. The dialect now assigns the length to None on reflection of the type so that the type behaves normally in other contexts.

  See also:

  String / varlength types no longer represent "max" explicitly on reflection

  ¶ References: #3504

- **[mssql] [change]** The `legacy_schema_aliasing` flag, introduced in version 1.0.5 as part of #3424 to allow disabling of the MSSQL dialect's attempts to create aliases for schema-qualified tables, now defaults to False; the old behavior is now disabled unless explicitly turned on.

  See also:

  The legacy_schema_aliasing flag is now set to False

  ¶ References: #3434

6.2.2 1.0 Changelog

1.0.12

no release date

orm

- **[orm] [bug]** Fixed regression since 0.9 where the 0.9 style loader options system failed to accommodate for multiple `undefer_group()` loader options in a single query. Multiple `undefer_group()` options will now be taken into account even against the same entity. ¶ References: #3623

mssql

- **[mssql] [bug]** Fixed the syntax of the `extract()` function when used on MSSQL against a datetime value; the quotes around the keyword are removed. Pull request courtesy Guillaume Doumenc. ¶ References: #3624, pull request bitbucket:70
- **[mssql] [bug] [firebird]** Fixed 1.0 regression where the eager fetch of cursor.rowcount was no longer called for an UPDATE or DELETE statement emitted via plain text or via the `text()` construct, affecting those drivers that erase cursor.rowcount once the cursor is closed such as SQL Server ODBC and Firebird drivers. ¶ References: #3622

1.0.11

Released: December 22, 2015

orm

- **[orm] [bug]** Fixed regression caused in 1.0.10 by the fix for #3593 where the check added for a polymorphic joinedload from a poly_subclass->class->poly_baseclass connection would fail for the scenario of class->poly_subclass->class. ¶ References: #3611
- **[orm] [bug]** Fixed bug where `Session.bulk_update_mappings()` and related would not bump a version id counter when in use. The experience here is still a little rough as the original version id is required in the given dictionaries and there's not clean error reporting on that yet. ¶ References: #3610
- **[orm] [bug]** Major fixes to the `Mapper.eager_defaults` flag, this flag would not be honored correctly in the case that multiple UPDATE statements were to be emitted, either as part of a flush or a bulk update operation. Additionally, RETURNING would be emitted unnecessarily within update statements. ¶ References: #3609
- **[orm] [bug]** Fixed bug where use of the `Query.select_from()` method would cause a subsequent call to the `Query.with_parent()` method to fail. ¶ References: #3606

sql

- **[sql] [bug]** Fixed bug in `Update.return_defaults()` which would cause all insert-default holding columns not otherwise included in the SET clause (such as primary key cols) to get rendered into the RETURNING even though this is an UPDATE. ¶ References: #3609

mysql

- **[mysql] [bug]** An adjustment to the regular expression used to parse MySQL views, such that we no longer assume the "ALGORITHM" keyword is present in the reflected view source, as some users have reported this not being present in some Amazon RDS environments. ¶ References: #3613
- **[mysql] [bug]** Added new reserved words for MySQL 5.7 to the MySQL dialect, including 'generated', 'optimizer_costs', 'stored', 'virtual'. Pull request courtesy Hanno Schlichting. ¶ References: pull request github:222

misc

- **[bug] [ext]** Further fixes to #3605, pop method on `MutableDict`, where the "default" argument was not included. ¶ References: #3605
- **[bug] [ext]** Fixed bug in baked loader system where the systemwide monkeypatch for setting up baked lazy loaders would interfere with other loader strategies that rely on lazy loading as a fallback, e.g. joined and subquery eager loaders, leading to `IndexError` exceptions at mapper configuration time. ¶ References: #3612

1.0.10

Released: December 11, 2015

orm

- **[orm] [bug]** Fixed issue where post_update on a many-to-one relationship would fail to emit an UPDATE in the case where the attribute were set to None and not previously loaded. ¶ References: #3599
- **[orm] [bug]** Fixed bug which is actually a regression that occurred between versions 0.8.0 and 0.8.1, due #2714. The case where joined eager loading needs to join out over a subclass-bound relationship when "with_polymorphic" were also used would fail to join from the correct entity. ¶ References: #3593
- **[orm] [bug]** Fixed joinedload bug which would occur when a. the query includes limit/offset criteria that forces a subquery b. the relationship uses "secondary" c. the primaryjoin of the relationship refers to a column that is either not part of the primary key, or is a PK col in a joined-inheritance subclass table that is under a different attribute name than the parent table's primary key column d. the query defers the columns that are present in the primaryjoin, typically via not being included in load_only(); the necessary column(s) would not be present in the subquery and produce invalid SQL. ¶ References: #3592
- **[orm] [bug]** A rare case which occurs when a `Session.rollback()` fails in the scope of a `Session.flush()` operation that's raising an exception, as has been observed in some MySQL SAVEPOINT cases, prevents the original database exception from being observed when it was emitted during flush, but only on Py2K because Py2K does not support exception chaining; on Py3K the originating exception is chained. As a workaround, a warning is emitted in this specific case showing at least the string message of the original database error before we proceed to raise the rollback-originating exception. ¶ References: #2696

orm declarative

- **[bug] [orm] [declarative]** Fixed bug where in Py2K a unicode literal would not be accepted as the string name of a class or other argument within declarative using `backref()` on `relationship()`. Pull request courtesy Nils Philippsen. ¶ References: pull request github:212

sql

- **[sql] [feature]** Added support for parameter-ordered SET clauses in an UPDATE statement. This feature is available by passing the `preserve_parameter_order` flag either to the core `Update` construct or alternatively adding it to the `Query.update.update_args` dictionary at the ORM-level, also passing the parameters themselves as a list of 2-tuples. Thanks to Gorka Eguileor for implementation and tests.

  See also:

  Parameter-Ordered Updates

  ¶ References: pull request github:200

- **[sql] [bug]** Fixed issue within the `Insert.from_select()` construct whereby the `Select` construct would have its `._raw_columns` collection mutated in-place when compiling the `Insert` construct, when the target `Table` has Python-side defaults. The `Select` construct would compile standalone with the erroneous column present subsequent to compilation of the `Insert`, and the the `Insert` statement itself would fail on a second compile attempt due to duplicate bound parameters. ¶ References: #3603

- **[sql] [bug] [postgresql]** Fixed bug where CREATE TABLE with a no-column table, but a constraint such as a CHECK constraint would render an erroneous comma in the definition; this scenario can occur such as with a Postgresql INHERITS table that has no columns of its own. ¶ References: #3598

postgresql

- **[postgresql] [bug]** Fixed issue where the "FOR UPDATE OF" Postgresql-specific SELECT modifier would fail if the referred table had a schema qualifier; PG needs the schema name to be omitted. Pull request courtesy Diana Clarke. ¶ References: #3573, pull request github:216

- **[postgresql] [bug]** Fixed bug where some varieties of SQL expression passed to the "where" clause of `postgresql.ExcludeConstraint` would fail to be accepted correctly. Pull request courtesy aisch. ¶ References: pull request github:215

- **[postgresql] [bug]** Fixed the `.python_type` attribute of `postgresql.INTERVAL` to return `datetime.timedelta` in the same way as that of `types.Interval.python_type`, rather than raising `NotImplementedError`. ¶ References: #3571

mysql

- **[mysql] [bug]** Fixed bug in MySQL reflection where the "fractional sections portion" of the `mysql.DATETIME`, `mysql.TIMESTAMP` and `mysql.TIME` types would be incorrectly placed into the `timezone` attribute, which is unused by MySQL, instead of the `fsp` attribute. ¶ References: #3602

mssql

- **[mssql] [bug]** Added the error "20006: Write to the server failed" to the list of disconnect errors for the pymssql driver, as this has been observed to render a connection unusable. ¶ References: #3585

- **[mssql] [bug]** A descriptive ValueError is now raised in the event that SQL server returns an invalid date or time format from a DATE or TIME column, rather than failing with a NoneType error. Pull request courtesy Ed Avis. ¶ References: pull request github:206

- **[mssql] [bug]** Fixed issue where DDL generated for the MSSQL types DATETIME2, TIME and DATETIMEOFFSET with a precision of "zero" would not generate the precision field. Pull request courtesy Jacobo de Vera. ¶ References: pull request github:213

misc

- **[bug] [ext]** Added support for the `dict.pop()` and `dict.popitem()` methods to the `mutable.MutableDict` class. ¶ References: #3605
- **[bug] [py3k]** Updates to internal getargspec() calls, some py36-related fixture updates, and alterations to two iterators to "return" instead of raising StopIteration, to allow tests to pass without errors or warnings on Py3.5, Py3.6, pull requests courtesy Jacob MacDonald, Luri de Silvio, and Phil Jones. ¶ References: pull request github:210, pull request github:218, pull request github:211
- **[bug] [ext]** Fixed an issue in baked queries where the .get() method, used either directly or within lazy loads, didn't consider the mapper's "get clause" as part of the cache key, causing bound parameter mismatches if the clause got re-generated. This clause is cached by mappers on the fly but in highly concurrent scenarios may be generated more than once when first accessed. ¶ References: #3597
- **[tests] [change]** The ORM and Core tutorials, which have always been in doctest format, are now exercised within the normal unit test suite in both Python 2 and Python 3. ¶

1.0.9

Released: October 20, 2015

orm

- **[orm] [feature]** Added new method `Query.one_or_none()`; same as `Query.one()` but returns None if no row found. Pull request courtesy esiegerman. ¶ References: pull request github:201
- **[orm] [bug] [postgresql]** Fixed regression in 1.0 where new feature of using "executemany" for UPDATE statements in the ORM (e.g. *UPDATE statements are now batched with executemany() in a flush*) would break on Postgresql and other RETURNING backends when using server-side version generation schemes, as the server side value is retrieved via RETURNING which is not supported with executemany. ¶ References: #3556
- **[orm] [bug]** Fixed rare TypeError which could occur when stringifying certain kinds of internal column loader options within internal logging. ¶ References: #3539
- **[orm] [bug]** Fixed bug in `Session.bulk_save_objects()` where a mapped column that had some kind of "fetch on update" value and was not locally present in the given object would cause an AttributeError within the operation. ¶ References: #3525
- **[orm] [bug]** Fixed 1.0 regression where the "noload" loader strategy would fail to function for a many-to-one relationship. The loader used an API to place "None" into the dictionary which no longer actually writes a value; this is a side effect of #3061. ¶ References: #3510

sql

- **[sql] [bug]** Fixed regression in 1.0-released default-processor for multi-VALUES insert statement, #3288, where the column type for the default-holding column would not be propagated to the compiled statement in the case where the default was being used, leading to bind-level type handlers not being invoked. ¶ References: #3520

postgresql

- **[postgresql] [bug]** An adjustment to the new Postgresql feature of reflecting storage options and USING of #3455 released in 1.0.6, to disable the feature for Postgresql versions < 8.2 where the `reloptions` column is

not provided; this allows Amazon Redshift to again work as it is based on an 8.0.x version of Postgresql. Fix courtesy Pete Hollobon. ¶ References: pull request github:190

oracle

- **[oracle] [bug] [py3k]** Fixed support for cx_Oracle version 5.2, which was tripping up SQLAlchemy's version detection under Python 3 and inadvertently not using the correct unicode mode for Python 3. This would cause issues such as bound variables mis-interpreted as NULL and rows silently not being returned. ¶ This change is also **backported** to: 0.9.11

  References: #3491

- **[oracle] [bug]** Fixed bug in Oracle dialect where reflection of tables and other symbols with names quoted to force all-lower-case would not be identified properly in reflection queries. The `quoted_name` construct is now applied to incoming symbol names that detect as forced into all-lower-case within the "name normalize" process. ¶ References: #3548

misc

- **[feature] [ext]** Added the `AssociationProxy.info` parameter to the `AssociationProxy` constructor, to suit the `AssociationProxy.info` accessor that was added in #2971. This is possible because `AssociationProxy` is constructed explicitly, unlike a hybrid which is constructed implicitly via the decorator syntax. ¶ References: #3551

- **[bug] [examples]** Fixed two issues in the "history_meta" example where history tracking could encounter empty history, and where a column keyed to an alternate attribute name would fail to track properly. Fixes courtesy Alex Fraser. ¶

- **[bug] [sybase]** Fixed two issues regarding Sybase reflection, allowing tables without primary keys to be reflected as well as ensured that a SQL statement involved in foreign key detection is pre-fetched up front to avoid driver issues upon nested queries. Fixes here courtesy Eugene Zapolsky; note that we cannot currently test Sybase to locally verify these changes. ¶ References: #3508, #3509

1.0.8

Released: July 22, 2015

engine

- **[engine] [bug]** Fixed critical issue whereby the pool "checkout" event handler may be called against a stale connection without the "connect" event handler having been called, in the case where the pool attempted to reconnect after being invalidated and failed; the stale connection would remain present and would be used on a subsequent attempt. This issue has a greater impact in the 1.0 series subsequent to 1.0.2, as it also delivers a blanked-out `.info` dictionary to the event handler; prior to 1.0.2 the `.info` dictionary is still the previous one. ¶ This change is also **backported** to: 0.9.11

  References: #3497

sqlite

- **[sqlite] [bug]** Fixed bug in SQLite dialect where reflection of UNIQUE constraints that included non-alphabetic characters in the names, like dots or spaces, would not be reflected with their name. ¶ This change is also

backported to: 0.9.10

References: #3495

misc

- **[misc] [bug]** Fixed an issue where a particular base class within utils didn't implement `__slots__`, and therefore meant all subclasses of that class didn't either, negating the rationale for `__slots__` to be in use. Didn't cause any issue except on IronPython which apparently does not implement `__slots__` behavior compatibly with cPython. ¶ References: #3494

1.0.7

Released: July 20, 2015

orm

- **[orm] [bug]** Fixed 1.0 regression where value objects that override `__eq__()` to return a non-boolean-capable object, such as some geoalchemy types as well as numpy types, were being tested for `bool()` during a unit of work update operation, where in 0.9 the return value of `__eq__()` was tested against "is True" to guard against this. ¶ References: #3469
- **[orm] [bug]** Fixed 1.0 regression where a "deferred" attribute would not populate correctly if it were loaded within the "optimized inheritance load", which is a special SELECT emitted in the case of joined table inheritance used to populate expired or unloaded attributes against a joined table without loading the base table. This is related to the fact that SQLA 1.0 no longer guesses about loading deferred columns and must be directed explicitly. ¶ References: #3468
- **[orm] [bug]** Fixed 1.0 regression where the "parent entity" of a synonym- mapped attribute on top of an `aliased()` object would resolve to the original mapper, not the `aliased()` version of it, thereby causing problems for a `Query` that relies on this attribute (e.g. it's the only representative attribute given in the constructor) to figure out the correct FROM clause for the query. ¶ References: #3466

orm declarative

- **[bug] [orm] [declarative]** Fixed bug in `AbstractConcreteBase` extension where a column setup on the ABC base which had a different attribute name vs. column name would not be correctly mapped on the final base class. The failure on 0.9 would be silent whereas on 1.0 it raised an ArgumentError, so may not have been noticed prior to 1.0. ¶ References: #3480

engine

- **[engine] [bug]** Fixed regression where new methods on `ResultProxy` used by the ORM `Query` object (part of the performance enhancements of #3175) would not raise the "this result does not return rows" exception in the case where the driver (typically MySQL) fails to generate cursor.description correctly; an AttributeError against NoneType would be raised instead. ¶ References: #3481
- **[engine] [bug]** Fixed regression where `ResultProxy.keys()` would return un-adjusted internal symbol names for "anonymous" labels, which are the "foo_1" types of labels we see generated for SQL functions without labels and similar. This was a side effect of the performance enhancements implemented as part of #918. ¶ References: #3483

sql

- **[sql] [feature]** Added a `ColumnElement.cast()` method which performs the same purpose as the standalone `cast()` function. Pull request courtesy Sebastian Bank. ¶ References: #3459, pull request bitbucket:56
- **[sql] [bug]** Fixed bug where coersion of literal `True` or `False` constant in conjunction with `and_()` or `or_()` would fail with an AttributeError. ¶ References: #3490
- **[sql] [bug]** Fixed potential issue where a custom subclass of `FunctionElement` or other column element that incorrectly states 'None' or any other invalid object as the `.type` attribute will report this exception instead of recursion overflow. ¶ References: #3485
- **[sql] [bug]** Fixed bug where the modulus SQL operator wouldn't work in reverse due to a missing `__rmod__` method. Pull request courtesy dan-gittik. ¶ References: pull request github:188

schema

- **[schema] [feature]** Added support for the MINVALUE, MAXVALUE, NO MINVALUE, NO MAXVALUE, and CYCLE arguments for CREATE SEQUENCE as supported by Postgresql and Oracle. Pull request courtesy jakeogh. ¶ References: pull request github:186

1.0.6

Released: June 25, 2015

orm

- **[orm] [bug]** Fixed a major regression in the 1.0 series where the version_id_counter feature would cause an object's version counter to be incremented when there was no net change to the object's row, but instead an object related to it via relationship (e.g. typically many-to-one) were associated or de-associated with it, resulting in an UPDATE statement that updates the object's version counter and nothing else. In the use case where the relatively recent "server side" and/or "programmatic/conditional" version counter feature were used (e.g. setting version_id_generator to False), the bug could cause an UPDATE without a valid SET clause to be emitted. ¶ References: #3465
- **[orm] [bug]** Fixed 1.0 regression where the enhanced behavior of single-inheritance joins of #3222 takes place inappropriately for a JOIN along explicit join criteria with a single-inheritance subclass that does not make use of any discriminator, resulting in an additional "AND NULL" clause. ¶ References: #3462
- **[orm] [bug]** Fixed bug in new `Session.bulk_update_mappings()` feature where the primary key columns used in the WHERE clause to locate the row would also be included in the SET clause, setting their value to themselves unnecessarily. Pull request courtesy Patrick Hayes. ¶ References: #3451, pull request github:181
- **[orm] [bug]** Fixed an unexpected-use regression whereby custom `Comparator` objects that made use of the `__clause_element__()` method and returned an object that was an ORM-mapped `InstrumentedAttribute` and not explicitly a `ColumnElement` would fail to be correctly handled when passed as an expression to `Session.query()`. The logic in 0.9 happened to succeed on this, so this use case is now supported. ¶ References: #3448

sql

- **[sql] [bug]** Fixed a bug where clause adaption as applied to a *Label* object would fail to accommodate the labeled SQL expression in all cases, such that any SQL operation that made use of `Label.self_group()` would use the original unadapted expression. One effect of this would be that an ORM *aliased()* construct would not fully accommodate attributes mapped by *column_property*, such that the un-aliased table could leak out when the property were used in some kinds of SQL comparisons. ¶ References: #3445

postgresql

- **[postgresql] [feature]** Added support for storage parameters under CREATE INDEX, using a new keyword argument `postgresql_with`. Also added support for reflection to support both the `postgresql_with` flag as well as the `postgresql_using` flag, which will now be set on *Index* objects that are reflected, as well present in a new "dialect_options" dictionary in the result of *Inspector.get_indexes()*. Pull request courtesy Pete Hollobon.

  See also:

  Index Storage Parameters

  ¶ References: #3455, pull request github:179

- **[postgresql] [feature]** Added new execution option `max_row_buffer` which is interpreted by the psycopg2 dialect when the `stream_results` option is used, which sets a limit on the size of the row buffer that may be allocated. This value is also provided based on the integer value sent to *Query.yield_per()*. Pull request courtesy mcclurem. ¶ References: pull request github:182

- **[postgresql] [bug] [pypy]** Re-fixed this issue first released in 1.0.5 to fix psycopg2cffi JSONB support once again, as they suddenly switched on unconditional decoding of JSONB types in version 2.7.1. Version detection now specifies 2.7.1 as where we should expect the DBAPI to do json encoding for us. ¶ References: #3439

- **[postgresql] [bug]** Repaired the *ExcludeConstraint* construct to support common features that other objects like *Index* now do, that the column expression may be specified as an arbitrary SQL expression such as *cast* or *text*. ¶ References: #3454

mssql

- **[mssql] [bug]** Fixed issue when using *VARBINARY* type in conjunction with an INSERT of NULL + pyodbc; pyodbc requires a special object be passed in order to persist NULL. As the *VARBINARY* type is now usually the default for *LargeBinary* due to #3039, this issue is partially a regression in 1.0. The pymssql driver appears to be unaffected. ¶ References: #3464

misc

- **[bug] [documentation]** Fixed an internal "memoization" routine for method types such that a Python descriptor is no longer used; repairs inspectability of these methods including support for Sphinx documentation. ¶ References: #2077

1.0.5

Released: June 7, 2015

orm

- **[orm] [feature]** Added new event `InstanceEvents.refresh_flush()`, invoked when an INSERT or UPDATE level default value fetched via RETURNING or Python-side default is invoked within the flush process. This is to provide a hook that is no longer present as a result of #3167, where attribute and validation events are no longer called within the flush process. ¶ References: #3427
- **[orm] [bug]** The "lightweight named tuple" used when a `Query` returns rows failed to implement `__slots__` correctly such that it still had a `__dict__`. This is resolved, but in the extremely unlikely case someone was assigning values to the returned tuples, that will no longer work. ¶ References: #3420

engine

- **[engine] [feature]** Added new engine event `ConnectionEvents.engine_disposed()`. Called after the `Engine.dispose()` method is called. ¶
- **[engine] [feature]** Adjustments to the engine plugin hook, such that the `URL.get_dialect()` method will continue to return the ultimate `Dialect` object when a dialect plugin is used, without the need for the caller to be aware of the `Dialect.get_dialect_cls()` method. ¶ References: #3379
- **[engine] [bug]** Fixed bug where known boolean values used by `engine_from_config()` were not being parsed correctly; these included `pool_threadlocal` and the psycopg2 argument `use_native_unicode`. ¶ References: #3435
- **[engine] [bug]** Added support for the case of the misbehaving DBAPI that has pep-249 exception names linked to exception classes of an entirely different name, preventing SQLAlchemy's own exception wrapping from wrapping the error appropriately. The SQLAlchemy dialect in use needs to implement a new accessor `DefaultDialect.dbapi_exception_translation_map` to support this feature; this is implemented now for the py-postgresql dialect. ¶ References: #3421
- **[engine] [bug]** Fixed bug involving the case when pool checkout event handlers are used and connection attempts are made in the handler itself which fail, the owning connection record would not be freed until the stack trace of the connect error itself were freed. For the case where a test pool of only a single connection were used, this means the pool would be fully checked out until that stack trace were freed. This mostly impacts very specific debugging scenarios and is unlikely to have been noticable in any production application. The fix applies an explicit checkin of the record before re-raising the caught exception. ¶ References: #3419

sql

- **[sql] [feature]** Added official support for a CTE used by the SELECT present inside of `Insert.from_select()`. This behavior worked accidentally up until 0.9.9, when it no longer worked due to unrelated changes as part of #3248. Note that this is the rendering of the WITH clause after the INSERT, before the SELECT; the full functionality of CTEs rendered at the top level of INSERT, UPDATE, DELETE is a new feature targeted for a later release. ¶ This change is also **backported** to: 0.9.10

  References: #3418

postgresql

- **[postgresql] [bug] [pypy]** Repaired some typing and test issues related to the pypy psycopg2cffi dialect, in particular that the current 2.7.0 version does not have native support for the JSONB type. The version detection for psycopg2 features has been tuned into a specific sub-version for psycopg2cffi. Additionally, test coverage has been enabled for the full series of psycopg2 features under psycopg2cffi. ¶ References: #3439

mssql

- **[mssql] [bug]** Added a new dialect flag to the MSSQL dialect `legacy_schema_aliasing` which when set to False will disable a very old and obsolete behavior, that of the compiler's attempt to turn all schema-qualified table names into alias names, to work around old and no longer locatable issues where SQL server could not parse a multi-part identifier name in all circumstances. The behavior prevented more sophisticated statements from working correctly, including those which use hints, as well as CRUD statements that embed correlated SELECT statements. Rather than continue to repair the feature to work with more complex statements, it's better to just disable it as it should no longer be needed for any modern SQL server version. The flag defaults to True for the 1.0.x series, leaving current behavior unchanged for this version series. In the 1.1 series, it will default to False. For the 1.0 series, when not set to either value explicitly, a warning is emitted when a schema-qualified table is first used in a statement, which suggests that the flag be set to False for all modern SQL Server versions.

  See also:

  Legacy Schema Mode

  ¶ References: #3430, #3424

misc

- **[feature] [ext]** Added support for `*args` to be passed to the baked query initial callable, in the same way that `*args` are supported for the *`BakedQuery.add_criteria()`* and *`BakedQuery.with_criteria()`* methods. Initial PR courtesy Naoki INADA. ¶ References: pull request bitbucket:54
- **[feature] [ext]** Added a new semi-public method to *`MutableBase`* `MutableBase._get_listen_keys()`. Overriding this method is needed in the case where a *`MutableBase`* subclass needs events to propagate for attribute keys other than the key to which the mutable type is associated with, when intercepting the *`InstanceEvents.refresh()`* or *`InstanceEvents.refresh_flush()`* events. The current example of this is composites using *`MutableComposite`*. ¶ References: #3427
- **[bug] [ext]** Fixed regression in the *`sqlalchemy.ext.mutable`* extension as a result of the bugfix for #3167, where attribute and validation events are no longer called within the flush process. The mutable extension was relying upon this behavior in the case where a column level Python-side default were responsible for generating the new value on INSERT or UPDATE, or when a value were fetched from the RETURNING clause for "eager defaults" mode. The new value would not be subject to any event when populated and the mutable extension could not establish proper coercion or history listening. A new event *`InstanceEvents.refresh_flush()`* is added which the mutable extension now makes use of for this use case. ¶ References: #3427

1.0.4

Released: May 7, 2015

orm

- **[orm] [bug]** Fixed unexpected-use regression where in the odd case that the primaryjoin of a relationship involved comparison to an unhashable type such as an HSTORE, lazy loads would fail due to a hash-oriented check on the statement parameters, modified in 1.0 as a result of #3061 to use hashing and modified in #3368 to occur in cases more common than "load on pending". The values are now checked for the `__hash__` attribute beforehand. ¶ References: #3416

- **[orm] [bug]** Liberalized an assertion that was added as part of #3347 to protect against unknown conditions when splicing inner joins together within joined eager loads with `innerjoin=True`; if some of the joins use a "secondary" table, the assertion needs to unwrap further joins in order to pass. ¶ References: #3412, #3347
- **[orm] [bug]** Repaired / added to tests yet more expressions that were reported as failing with the new 'entity' key value added to `Query.column_descriptions`, the logic to discover the "from" clause is again reworked to accommodate columns from aliased classes, as well as to report the correct value for the "aliased" flag in these cases. ¶ References: #3409, #3320

schema

- **[schema] [bug]** Fixed bug in enhanced constraint-attachment logic introduced in #3341 where in the unusual case of a constraint that refers to a mixture of `Column` objects and string column names at the same time, the auto-attach-on-column-attach logic will be skipped; for the constraint to be auto-attached in this case, all columns must be assembled on the target table up front. Added a new section to the migration document regarding the original feature as well as this change.

  See also:

  Constraints referring to unattached Columns can auto-attach to the Table when their referred columns are attached

  ¶ References: #3411

misc

- **[bug] [ext]** Fixed bug where when using extended attribute instrumentation system, the correct exception would not be raised when `class_mapper()` were called with an invalid input that also happened to not be weak referencable, such as an integer. ¶ This change is also **backported** to: 0.9.10

  References: #3408
- **[bug] [tests] [pypy]** Fixed an import that prevented "pypy setup.py test" from working correctly. ¶ This change is also **backported** to: 0.9.10

  References: #3406

1.0.3

Released: April 30, 2015

orm

- **[orm] [bug] [pypy]** Fixed regression from 0.9.10 prior to release due to #3349 where the check for query state on `Query.update()` or `Query.delete()` compared the empty tuple to itself using `is`, which fails on Pypy to produce `True` in this case; this would erronously emit a warning in 0.9 and raise an exception in 1.0. ¶ References: #3405
- **[orm] [bug]** Fixed regression from 0.9.10 prior to release where the new addition of `entity` to the `Query.column_descriptions` accessor would fail if the target entity was produced from a core selectable such as a `Table` or `CTE` object. ¶ References: #3320, #3403
- **[orm] [bug]** Fixed regression within the flush process when an attribute were set to a SQL expression for an UPDATE, and the SQL expression when compared to the previous value of the attribute would produce a SQL

comparison other than == or !=, the exception "Boolean value of this clause is not defined" would raise. The fix ensures that the unit of work will not interpret the SQL expression in this way. ¶ References: #3402

- **[orm] [bug]** Fixed unexpected use regression due to #2992 where textual elements placed into the *Query.order_by()* clause in conjunction with joined eager loading would be added to the columns clause of the inner query in such a way that they were assumed to be table-bound column names, in the case where the joined eager load needs to wrap the query in a subquery to accommodate for a limit/offset.

  Originally, the behavior here was intentional, in that a query such as `query(User).order_by('name').limit(1)` would order by `user.name` even if the query was modified by joined eager loading to be within a subquery, as `'name'` would be interpreted as a symbol to be located within the FROM clauses, in this case `User.name`, which would then be copied into the columns clause to ensure it were present for ORDER BY. However, the feature fails to anticipate the case where `order_by("name")` refers to a specific label name present in the local columns clause already and not a name bound to a selectable in the FROM clause.

  Beyond that, the feature also fails for deprecated cases such as `order_by("name desc")`, which, while it emits a warning that *text()* should be used here (note that the issue does not impact cases where *text()* is used explicitly), still produces a different query than previously where the "name desc" expression is copied into the columns clause inappropriately. The resolution is such that the "joined eager loading" aspect of the feature will skip over these so-called "label reference" expressions when augmenting the inner columns clause, as though they were *text()* constructs already. ¶ References: #3392

- **[orm] [bug]** Fixed a regression regarding the *MapperEvents.instrument_class()* event where its invocation was moved to be after the class manager's instrumentation of the class, which is the opposite of what the documentation for the event explicitly states. The rationale for the switch was due to Declarative taking the step of setting up the full "instrumentation manager" for a class before it was mapped for the purpose of the new `@declared_attr` features described in *Improvements to declarative mixins, @declared_attr and related features*, but the change was also made against the classical use of *mapper()* for consistency. However, SQLSoup relies upon the instrumentation event happening before any instrumentation under classical mapping. The behavior is reverted in the case of classical and declarative mapping, the latter implemented by using a simple memoization without using class manager. ¶ References: #3388

- **[orm] [bug]** Fixed issue in new *QueryEvents.before_compile()* event where changes made to the *Query* object's collection of entities to load within the event would render in the SQL, but would not be reflected during the loading process. ¶ References: #3387

engine

- **[engine] [feature]** New features added to support engine/pool plugins with advanced functionality. Added a new "soft invalidate" feature to the connection pool at the level of the checked out connection wrapper as well as the *_ConnectionRecord*. This works similarly to a modern pool invalidation in that connections aren't actively closed, but are recycled only on next checkout; this is essentially a per-connection version of that feature. A new event `PoolEvents.soft_invalidate` is added to complement it.

  Also added new flag *ExceptionContext.invalidate_pool_on_disconnect*. Allows an error handler within *ConnectionEvents.handle_error()* to maintain a "disconnect" condition, but to handle calling invalidate on individual connections in a specific manner within the event. ¶ References: #3379

- **[engine] [feature]** Added new event `DialectEvents.do_connect`, which allows interception / replacement of when the *Dialect.connect()* hook is called to create a DBAPI connection. Also added dialect plugin hooks *Dialect.get_dialect_cls()* and *Dialect.engine_created()* which allow external plugins to add events to existing dialects using entry points. ¶ References: #3355

sql

- **[sql] [feature]** Added a placeholder method `TypeEngine.compare_against_backend()` which is now consumed by Alembic migrations as of 0.7.6. User-defined types can implement this method to assist in the comparison of a type against one reflected from the database. ¶
- **[sql] [bug]** Fixed bug where the truncation of long labels in SQL could produce a label that overlapped another label that is not truncated; this because the length threshhold for truncation was greater than the portion of the label that remains after truncation. These two values have now been made the same; label_length - 6. The effect here is that shorter column labels will be "truncated" where they would not have been truncated before. ¶ References: #3396
- **[sql] [bug]** Fixed regression due to #3282 where the `tables` collection passed as a keyword argument to the `DDLEvents.before_create()`, `DDLEvents.after_create()`, `DDLEvents.before_drop()`, and `DDLEvents.after_drop()` events would no longer be a list of tables, but instead a list of tuples which contained a second entry with foreign keys to be added or dropped. As the `tables` collection, while documented as not necessarily stable, has come to be relied upon, this change is considered a regression. Additionally, in some cases for "drop", this collection would be an iterator that would cause the operation to fail if prematurely iterated. The collection is now a list of table objects in all cases and test coverage for the format of this collection is now added. ¶ References: #3391

misc

- **[bug] [ext]** Fixed bug in association proxy where an any()/has() on an relationship->scalar non-object attribute comparison would fail, e.g. `filter(Parent.some_collection_to_attribute.any(Child.attr == 'foo'))` ¶ References: #3397

1.0.2

Released: April 24, 2015

orm declarative

- **[bug] [orm] [declarative]** Fixed unexpected use regression regarding the declarative `__declare_first__` and `__declare_last__` accessors where these would no longer be called on the superclass of the declarative base. ¶ References: #3383

sql

- **[sql] [bug]** Fixed a regression that was incorrectly fixed in 1.0.0b4 (hence becoming two regressions); reports that SELECT statements would GROUP BY a label name and fail was misconstrued that certain backends such as SQL Server should not be emitting ORDER BY or GROUP BY on a simple label name at all; when in fact, we had forgotten that 0.9 was already emitting ORDER BY on a simple label name for all backends, as described in *Label constructs can now render as their name alone in an ORDER BY*, even though 1.0 includes a rewrite of this logic as part of #2992. As far as emitting GROUP BY against a simple label, even Postgresql has cases where it will raise an error even though the label to group on should be apparent, so it is clear that GROUP BY should never be rendered in this way automatically.

  In 1.0.2, SQL Server, Firebird and others will again emit ORDER BY on a simple label name when passed a `Label` construct that is also present in the columns clause. Additionally, no backend will emit GROUP BY against the simple label name only when passed a `Label` construct. ¶ References: #3338, #3385

1.0.1

Released: April 23, 2015

orm

- **[orm] [bug]** Fixed issue where a query of the form `query(B).filter(B.a != A(id=7))` would render the `NEVER_SET` symbol, when given a transient object. For a persistent object, it would always use the persisted database value and not the currently set value. Assuming autoflush is turned on, this usually would not be apparent for persistent values, as any pending changes would be flushed first in any case. However, this is inconsistent vs. the logic used for the non-negated comparison, `query(B).filter(B.a == A(id=7))`, which does use the current value and additionally allows comparisons to transient objects. The comparison now uses the current value and not the database-persisted value.

  Unlike the other `NEVER_SET` issues that are repaired as regressions caused by #3061 in this release, this particular issue is present at least as far back as 0.8 and possibly earlier, however it was discovered as a result of repairing the related `NEVER_SET` issues.

  See also:

  A "negated contains or equals" relationship comparison will use the current value of attributes, not the database value

  ¶ References: #3374

- **[orm] [bug]** Fixed unexpected use regression cause by #3061 where the NEVER_SET symbol could leak into relationship-oriented queries, including `filter()` and `with_parent()` queries. The `None` symbol is returned in all cases, however many of these queries have never been correctly supported in any case, and produce comparisons to NULL without using the IS operator. For this reason, a warning is also added to that subset of relationship queries that don't currently provide for `IS NULL`.

  See also:

  Warnings emitted when comparing objects with None values to relationships

  ¶ References: #3371

- **[orm] [bug]** Fixed a regression caused by #3061 where the NEVER_SET symbol could leak into a lazyload query, subsequent to the flush of a pending object. This would occur typically for a many-to-one relationship that does not use a simple "get" strategy. The good news is that the fix improves efficiency vs. 0.9, because we can now skip the SELECT statement entirely when we detect NEVER_SET symbols present in the parameters; prior to #3061, we couldn't discern if the None here were set or not. ¶ References: #3368

engine

- **[engine] [bug]** Added the string value `"none"` to those accepted by the *`Pool.reset_on_return`* parameter as a synonym for `None`, so that string values can be used for all settings, allowing utilities like *`engine_from_config()`* to be usable without issue. ¶ This change is also **backported** to: 0.9.10

  References: #3375

sql

- **[sql] [bug]** Fixed issue where a straight SELECT EXISTS query would fail to assign the proper result type of Boolean to the result mapping, and instead would leak column types from within the query into the result map. This issue exists in 0.9 and earlier as well, however has less of an impact in those versions. In 1.0, due

to #918 this becomes a regression in that we now rely upon the result mapping to be very accurate, else we can assign result-type processors to the wrong column. In all versions, this issue also has the effect that a simple EXISTS will not apply the Boolean type handler, leading to simple 1/0 values for backends without native boolean instead of True/False. The fix includes that an EXISTS columns argument will be anon-labeled like other column expressions; a similar fix is implemented for pure-boolean expressions like `not_(True())`. ¶ References: #3372

sqlite

- **[sqlite] [bug]** Fixed a regression due to #3282, where due to the fact that we attempt to assume the availability of ALTER when creating/dropping schemas, in the case of SQLite we simply said to not worry about foreign keys at all, since ALTER is not available, when creating and dropping tables. This meant that the sorting of tables was basically skipped in the case of SQLite, and for the vast majority of SQLite use cases, this is not an issue.

  However, users who were doing DROPs on SQLite with tables that contained data and with referential integrity turned on would then experience errors, as the dependency sorting *does* matter in the case of DROP with enforced constraints, when those tables have data (SQLite will still happily let you create foreign keys to nonexistent tables and drop tables referring to existing ones with constraints enabled, as long as there's no data being referenced).

  In order to maintain the new feature of #3282 while still allowing a SQLite DROP operation to maintain ordering, we now do the sort with full FKs taken under consideration, and if we encounter an unresolvable cycle, only *then* do we forego attempting to sort the tables; we instead emit a warning and go with the unsorted list. If an environment needs both ordered DROPs *and* has foreign key cycles, then the warning notes they will need to restore the `use_alter` flag to their *`ForeignKey`* and *`ForeignKeyConstraint`* objects so that just those objects will be omitted from the dependency sort.

  See also:

  The use_alter flag on ForeignKeyConstraint is (usually) no longer needed - contains an updated note about SQLite.

  ¶ References: #3378

firebird

- **[firebird] [bug]** Fixed a regression due to #3034 where limit/offset clauses were not properly interpreted by the Firebird dialect. Pull request courtesy effem-git. ¶ References: #3380, pull request github:168
- **[firebird] [bug]** Fixed support for "literal_binds" mode when using limit/offset with Firebird, so that the values are again rendered inline when this is selected. Related to #3034. ¶ References: #3381

1.0.0

Released: April 16, 2015

orm

- **[orm] [feature]** Added new argument *`Query.update.update_args`* which allows kw arguments such as `mysql_limit` to be passed to the underlying *`Update`* construct. Pull request courtesy Amir Sadoughi. ¶ References: pull request github:164

- **[orm] [bug]** Identified an inconsistency when handling `Query.join()` to the same target more than once; it implicitly dedupes only in the case of a relationship join, and due to #3233, in 1.0 a join to the same table twice behaves differently than 0.9 in that it no longer erroneously aliases. To help document this change, the verbiage regarding #3233 in the migration notes has been generalized, and a warning has been added when `Query.join()` is called against the same target relationship more than once. ¶ References: #3367
- **[orm] [bug]** Made a small improvement to the heuristics of relationship when determining remote side with semi-self-referential (e.g. two joined inh subclasses referring to each other), non-simple join conditions such that the parententity is taken into account and can reduce the need for using the `remote()` annotation; this can restore some cases that might have worked without the annotation prior to 0.9.4 via #2948. ¶ References: #3364

sql

- **[sql] [feature]** The topological sorting used to sort `Table` objects and available via the `MetaData.sorted_tables` collection will now produce a **deterministic** ordering; that is, the same ordering each time given a set of tables with particular names and dependencies. This is to help with comparison of DDL scripts and other use cases. The tables are sent to the topological sort sorted by name, and the topological sort itself will process the incoming data in an ordered fashion. Pull request courtesy Sebastian Bank.

  See also:

  MetaData.sorted_tables accessor is "deterministic"

  ¶ References: #3084, pull request bitbucket:47
- **[sql] [bug]** Fixed issue where a `MetaData` object that used a naming convention would not properly work with pickle. The attribute was skipped leading to inconsistencies and failures if the unpickled `MetaData` object were used to base additional tables from. ¶ This change is also **backported** to: 0.9.10

  References: #3362

postgresql

- **[postgresql] [bug]** Fixed a long-standing bug where the `Enum` type as used with the psycopg2 dialect in conjunction with non-ascii values and `native_enum=False` would fail to decode return results properly. This stemmed from when the PG `postgresql.ENUM` type used to be a standalone type without a "non native" option. ¶ This change is also **backported** to: 0.9.10

  References: #3354

mssql

- **[mssql] [bug]** Fixed a regression where the "last inserted id" mechanics would fail to store the correct value for MSSQL on an INSERT where the primary key value was present in the insert params before execution, as well as in the case where an INSERT from SELECT would state the target columns as column objects, instead of string keys. ¶ References: #3360
- **[mssql] [bug]** Using the `Binary` constructor now present in pymssql rather than patching one in. Pull request courtesy Ramiro Morales. ¶ References: pull request github:166

misc

- **[bug] [tests]** Fixed the pathing used when tests run; for sqla_nose.py and py.test, the "./lib" prefix is again inserted at the head of sys.path but only if sys.flags.no_user_site isn't set; this makes it act just like the way Python puts "." in the current path by default. For tox, we are setting the PYTHONNOUSERSITE flag now. ¶ References: #3356

1.0.0b5

Released: April 3, 2015

orm

- **[orm] [bug]** Fixed bug where the state tracking within multiple, nested `Session.begin_nested()` operations would fail to propagate the "dirty" flag for an object that had been updated within the inner savepoint, such that if the enclosing savepoint were rolled back, the object would not be part of the state that was expired and therefore reverted to its database state. ¶ This change is also **backported** to: 0.9.10

  References: #3352

- **[orm] [bug]** `Query` doesn't support joins, subselects, or special FROM clauses when using the `Query.update()` or `Query.delete()` methods; instead of silently ignoring these fields if methods like `Query.join()` or `Query.select_from()` has been called, an error is raised. In 0.9.10 this only emits a warning. ¶ References: #3349

- **[orm] [bug]** Added a list() call around a weak dictionary used within the commit phase of the session, which without it could cause a "dictionary changed size during iter" error if garbage collection interacted within the process. Change was introduced by #3139. ¶

- **[orm] [bug]** Fixed a bug related to "nested" inner join eager loading, which exists in 0.9 as well but is more of a regression in 1.0 due to #3008 which turns on "nested" by default, such that a joined eager load that travels across sibling paths from a common ancestor using innerjoin=True will correctly splice each "innerjoin" sibling into the appropriate part of the join, when a series of inner/outer joins are mixed together. ¶ References: #3347

sql

- **[sql] [bug]** The warning emitted by the unicode type for a non-unicode type has been liberalized to warn for values that aren't even string values, such as integers; previously, the updated warning system of 1.0 made use of string formatting operations which would raise an internal TypeError. While these cases should ideally raise totally, some backends like SQLite and MySQL do accept them and are potentially in use by legacy code, not to mention that they will always pass through if unicode conversion is turned off for the target backend. ¶ References: #3346

postgresql

- **[postgresql] [bug]** Fixed bug where updated PG index reflection as a result of #3184 would cause index operations to fail on Postgresql versions 8.4 and earlier. The enhancements are now disabled when using an older version of Postgresql. ¶ References: #3343

1.0.0b4

Released: March 29, 2015

sql

- **[sql] [bug]** Fixed bug in new "label resolution" feature of #2992 where a label that was anonymous, then labeled again with a name, would fail to be locatable via a textual label. This situation occurs naturally when a mapped `column_property()` is given an explicit label in a query. ¶ References: #3340
- **[sql] [bug]** Fixed bug in new "label resolution" feature of #2992 where the string label placed in the order_by() or group_by() of a statement would place higher priority on the name as found inside the FROM clause instead of a more locally available name inside the columns clause. ¶ References: #3335

schema

- **[schema] [feature]** The "auto-attach" feature of constraints such as `UniqueConstraint` and `CheckConstraint` has been further enhanced such that when the constraint is associated with non-table-bound `Column` objects, the constraint will set up event listeners with the columns themselves such that the constraint auto attaches at the same time the columns are associated with the table. This in particular helps in some edge cases in declarative but is also of general use.

  See also:

  Constraints referring to unattached Columns can auto-attach to the Table when their referred columns are attached

  ¶ References: #3341

mysql

- **[mysql] [bug] [pymysql]** Fixed unicode support for PyMySQL when using an "executemany" operation with unicode parameters. SQLAlchemy now passes both the statement as well as the bound parameters as unicode objects, as PyMySQL generally uses string interpolation internally to produce the final statement, and in the case of executemany does the "encode" step only on the final statement. ¶ This change is also **backported** to: 0.9.10

  References: #3337

mssql

- **[mssql] [bug] [sybase] [firebird] [oracle]** Turned off the "simple order by" flag on the MSSQL, Oracle dialects; this is the flag that per #2992 causes an order by or group by an expression that's also in the columns clause to be copied by label, even if referenced as the expression object. The behavior for MSSQL is now the old behavior that copies the whole expression in by default, as MSSQL can be picky on these particularly in GROUP BY expressions. The flag is also turned off defensively for the Firebird and Sybase dialects.

  Note: this resolution was incorrect, please see version 1.0.2 for a rework of this resolution.

  ¶ References: #3338

1.0.0b3

Released: March 20, 2015

mysql

- **[mysql] [bug]** Repaired the commit for issue #2771 which was inadvertently commented out. ¶ References: #2771

1.0.0b2

Released: March 20, 2015

orm

- **[orm] [bug]** Fixed unexpected use regression from pullreq github:137 where Py2K unicode literals (e.g. `u""`) would not be accepted by the `relationship.cascade` option. Pull request courtesy Julien Castets. ¶ References: #3327, pull request github:160

orm declarative

- **[orm] [declarative] [change]** Loosened some restrictions that were added to `@declared_attr` objects, such that they were prevented from being called outside of the declarative process; this is related to the enhancements of #3150 which allow `@declared_attr` to return a value that is cached based on the current class as it's being configured. The exception raise has been removed, and the behavior changed so that outside of the declarative process, the function decorated by `@declared_attr` is called every time just like a regular `@property`, without using any caching, as none is available at this stage. ¶ References: #3331

engine

- **[engine] [bug]** The "auto close" for `ResultProxy` is now a "soft" close. That is, after exhausing all rows using the fetch methods, the DBAPI cursor is released as before and the object may be safely discarded, but the fetch methods may continue to be called for which they will return an end-of-result object (None for fetchone, empty list for fetchmany and fetchall). Only if `ResultProxy.close()` is called explicitly will these methods raise the "result is closed" error.

  See also:

  ResultProxy "auto close" is now a "soft" close

  ¶ References: #3329, #3330

mysql

- **[mysql] [bug] [py3k]** Fixed the `mysql.BIT` type on Py3K which was not using the `ord()` function correctly. Pull request courtesy David Marin. ¶ This change is also **backported** to: 0.9.10

  References: #3333, pull request github:158

- **[mysql] [bug]** Fixes to fully support using the `'utf8mb4'` MySQL-specific charset with MySQL dialects, in particular MySQL-Python and PyMySQL. In addition, MySQL databases that report more unusual charsets such as 'koi8u' or 'eucjpms' will also work correctly. Pull request courtesy Thomas Grainger. ¶ References: #2771, pull request bitbucket:49

1.0.0b1

Released: March 13, 2015

Version 1.0.0b1 is the first release of the 1.0 series. Many changes described here are also present in the 0.9 and sometimes the 0.8 series as well. For changes that are specific to 1.0 with an emphasis on compatibility concerns, see What's New in SQLAlchemy 1.0?.

general

- **[general] [feature]** Structural memory use has been improved via much more significant use of `__slots__` for many internal objects. This optimization is particularly geared towards the base memory size of large applications that have lots of tables and columns, and greatly reduces memory size for a variety of high-volume objects including event listening internals, comparator objects and parts of the ORM attribute and loader strategy system.

  See also:

  Significant Improvements in Structural Memory Use

  ¶

- **[general] [bug]** The `__module__` attribute is now set for all those SQL and ORM functions that are derived as "public factory" symbols, which should assist with documentation tools being able to report on the target module. ¶ References: #3218

orm

- **[orm] [feature]** Added a new entry `"entity"` to the dictionaries returned by `Query.column_descriptions`. This refers to the primary ORM mapped class or aliased class that is referred to by the expression. Compared to the existing entry for `"type"`, it will always be a mapped entity, even if extracted from a column expression, or None if the given expression is a pure core expression. See also #3403 which repaired a regression in this feature which was unreleased in 0.9.10 but was released in the 1.0 version. ¶ This change is also **backported** to: 0.9.10

  References: #3320

- **[orm] [feature]** Added new parameter `Session.connection.execution_options` which may be used to set up execution options on a `Connection` when it is first checked out, before the transaction has begun. This is used to set up options such as isolation level on the connection before the transaction starts.

  See also:

  Setting Transaction Isolation Levels - new documentation section detailing best practices for setting transaction isolation with sessions.

  ¶ This change is also **backported** to: 0.9.9

  References: #3296

- **[orm] [feature]** Added new method `Session.invalidate()`, functions similarly to `Session.close()`, except also calls `Connection.invalidate()` on all connections, guaranteeing that they will not be returned to the connection pool. This is useful in situations e.g. dealing with gevent timeouts when it is not safe to use the connection further, even for rollbacks. ¶ This change is also **backported** to: 0.9.9

- **[orm] [feature]** The "primaryjoin" model has been stretched a bit further to allow a join condition that is strictly from a single column to itself, translated through some kind of SQL function or expression. This is kind of experimental, but the first proof of concept is a "materialized path" join condition where a path string is

compared to itself using "like". The `ColumnOperators.like()` operator has also been added to the list of valid operators to use in a primaryjoin condition. ¶ This change is also **backported** to: 0.9.5

References: #3029

- **[orm] [feature]** Added new utility function `make_transient_to_detached()` which can be used to manufacture objects that behave as though they were loaded from a session, then detached. Attributes that aren't present are marked as expired, and the object can be added to a Session where it will act like a persistent one. ¶ This change is also **backported** to: 0.9.5

  References: #3017

- **[orm] [feature]** Added a new event suite `QueryEvents`. The `QueryEvents.before_compile()` event allows the creation of functions which may place additional modifications to `Query` objects before the construction of the SELECT statement. It is hoped that this event be made much more useful via the advent of a new inspection system that will allow for detailed modifications to be made against `Query` objects in an automated fashion.

  See also:

  `QueryEvents`

  ¶ References: #3317

- **[orm] [feature]** The subquery wrapping which occurs when joined eager loading is used with a one-to-many query that also features LIMIT, OFFSET, or DISTINCT has been disabled in the case of a one-to-one relationship, that is a one-to-many with `relationship.uselist` set to False. This will produce more efficient queries in these cases.

  See also:

  Subqueries no longer applied to uselist=False joined eager loads

  ¶ References: #3249

- **[orm] [feature]** Mapped state internals have been reworked to allow for a 50% reduction in callcounts specific to the "expiration" of objects, as in the "auto expire" feature of `Session.commit()` and for `Session.expire_all()`, as well as in the "cleanup" step which occurs when object states are garbage collected. ¶ References: #3307

- **[orm] [feature]** A warning is emitted when the same polymorphic identity is assigned to two different mappers in the same hierarchy. This is typically a user error and means that the two different mapping types cannot be correctly distinguished at load time. Pull request courtesy Sebastian Bank. ¶ References: #3262, pull request bitbucket:38

- **[orm] [feature]** A new series of `Session` methods which provide hooks directly into the unit of work's facility for emitting INSERT and UPDATE statements has been created. When used correctly, this expert-oriented system can allow ORM-mappings to be used to generate bulk insert and update statements batched into executemany groups, allowing the statements to proceed at speeds that rival direct use of the Core.

  See also:

  Bulk Operations

  ¶ References: #3100

- **[orm] [feature]** Added a parameter `Query.join.isouter` which is synonymous with calling `Query.outerjoin()`; this flag is to provide a more consistent interface compared to Core `FromClause.join()`. Pull request courtesy Jonathan Vanasco. ¶ References: #3217

- **[orm] [feature]** Added new event handlers `AttributeEvents.init_collection()` and `AttributeEvents.dispose_collection()`, which track when a collection is first associated

with an instance and when it is replaced. These handlers supersede the `collection.linker()` annotation. The old hook remains supported through an event adapter. ¶

- **[orm] [feature]** The `Query` will raise an exception when `Query.yield_per()` is used with mappings or options where either subquery eager loading, or joined eager loading with collections, would take place. These loading strategies are not currently compatible with yield_per, so by raising this error, the method is safer to use. Eager loads can be disabled with the `lazyload('*')` option or `Query.enable_eagerloads()`.

  See also:

  Joined/Subquery eager loading explicitly disallowed with yield_per

  ¶

- **[orm] [feature]** A new implementation for `KeyedTuple` used by the `Query` object offers dramatic speed improvements when fetching large numbers of column-oriented rows.

  See also:

  New KeyedTuple implementation dramatically faster

  ¶ References: #3176

- **[orm] [feature]** The behavior of `joinedload.innerjoin` as well as `relationship.innerjoin` is now to use "nested" inner joins, that is, right-nested, as the default behavior when an inner join joined eager load is chained to an outer join eager load.

  See also:

  Right inner join nesting now the default for joinedload with innerjoin=True

  ¶ References: #3008

- **[orm] [feature]** UPDATE statements can now be batched within an ORM flush into more performant executemany() call, similarly to how INSERT statements can be batched; this will be invoked within flush to the degree that subsequent UPDATE statements for the same mapping and table involve the identical columns within the VALUES clause, that no SET-level SQL expressions are embedded, and that the versioning requirements for the mapping are compatible with the backend dialect's ability to return a correct rowcount for an executemany operation. ¶

- **[orm] [feature]** The `info` parameter has been added to the constructor for `SynonymProperty` and `ComparableProperty`. ¶ References: #2963

- **[orm] [feature]** The `InspectionAttr.info` collection is now moved down to `InspectionAttr`, where in addition to being available on all `MapperProperty` objects, it is also now available on hybrid properties, association proxies, when accessed via `Mapper.all_orm_descriptors`. ¶ References: #2971

- **[orm] [changed]** The `proc()` callable passed to the `create_row_processor()` method of custom `Bundle` classes now accepts only a single "row" argument.

  See also:

  API Change for new Bundle feature when custom row loaders are used

  ¶

- **[orm] [changed]** Deprecated event hooks removed: `populate_instance`, `create_instance`, `translate_row`, `append_result`

  See also:

  Deprecated ORM Event Hooks Removed

  ¶

- **[orm] [bug]** Fixed bugs in ORM object comparisons where comparison of many-to-one `!= None` would fail if the source were an aliased class, or if the query needed to apply special aliasing to the expression due to aliased joins or polymorphic querying; also fixed bug in the case where comparing a many-to-one to an object state would fail if the query needed to apply special aliasing due to aliased joins or polymorphic querying. ¶ This change is also **backported** to: 0.9.9

  References: #3310

- **[orm] [bug]** Fixed bug where internal assertion would fail in the case where an `after_rollback()` handler for a *Session* incorrectly adds state to that *Session* within the handler, and the task to warn and remove this state (established by #2389) attempts to proceed. ¶ This change is also **backported** to: 0.9.9

  References: #3309

- **[orm] [bug]** Fixed bug where TypeError raised when *Query.join()* called with unknown kw arguments would raise its own TypeError due to broken formatting. Pull request courtesy Malthe Borch. ¶ This change is also **backported** to: 0.9.9

  References: pull request github:147

- **[orm] [bug]** Fixed bug in lazy loading SQL construction whereby a complex primaryjoin that referred to the same "local" column multiple times in the "column that points to itself" style of self-referential join would not be substituted in all cases. The logic to determine substitutions here has been reworked to be more open-ended. ¶ This change is also **backported** to: 0.9.9

  References: #3300

- **[orm] [bug]** The "wildcard" loader options, in particular the one set up by the *orm.load_only()* option to cover all attributes not explicitly mentioned, now takes into account the superclasses of a given entity, if that entity is mapped with inheritance mapping, so that attribute names within the superclasses are also omitted from the load. Additionally, the polymorphic discriminator column is unconditionally included in the list, just in the same way that primary key columns are, so that even with load_only() set up, polymorphic loading of subtypes continues to function correctly. ¶ This change is also **backported** to: 0.9.9

  References: #3287

- **[orm] [bug] [pypy]** Fixed bug where if an exception were thrown at the start of a *Query* before it fetched results, particularly when row processors can't be formed, the cursor would stay open with results pending and not actually be closed. This is typically only an issue on an interpreter like Pypy where the cursor isn't immediately GC'ed, and can in some circumstances lead to transactions/ locks being open longer than is desirable. ¶ This change is also **backported** to: 0.9.9

  References: #3285

- **[orm] [bug]** Fixed a leak which would occur in the unsupported and highly non-recommended use case of replacing a relationship on a fixed mapped class many times, referring to an arbitrarily growing number of target mappers. A warning is emitted when the old relationship is replaced, however if the mapping were already used for querying, the old relationship would still be referenced within some registries. ¶ This change is also **backported** to: 0.9.9

  References: #3251

- **[orm] [bug] [sqlite]** Fixed bug regarding expression mutations which could express itself as a "Could not locate column" error when using *Query* to select from multiple, anonymous column entities when querying against SQLite, as a side effect of the "join rewriting" feature used by the SQLite dialect. ¶ This change is also **backported** to: 0.9.9

  References: #3241

- **[orm] [bug]** Fixed bug where the ON clause for *Query.join()*, and *Query.outerjoin()* to a single-inheritance subclass using `of_type()` would not render the "single table criteria" in the ON clause if the `from_joinpoint=True` flag were set. ¶ This change is also **backported** to: 0.9.9

References: #3232

- **[orm] [bug] [engine]** Fixed bug that affected generally the same classes of event as that of #3199, when the `named=True` parameter would be used. Some events would fail to register, and others would not invoke the event arguments correctly, generally in the case of when an event was "wrapped" for adaption in some other way. The "named" mechanics have been rearranged to not interfere with the argument signature expected by internal wrapper functions. ¶ This change is also **backported** to: 0.9.8

  References: #3197

- **[orm] [bug]** Fixed bug that affected many classes of event, particularly ORM events but also engine events, where the usual logic of "de duplicating" a redundant call to `event.listen()` with the same arguments would fail, for those events where the listener function is wrapped. An assertion would be hit within registry.py. This assertion has now been integrated into the deduplication check, with the added bonus of a simpler means of checking deduplication across the board. ¶ This change is also **backported** to: 0.9.8

  References: #3199

- **[orm] [bug]** Fixed warning that would emit when a complex self-referential primaryjoin contained functions, while at the same time remote_side was specified; the warning would suggest setting "remote side". It now only emits if remote_side isn't present. ¶ This change is also **backported** to: 0.9.8

  References: #3194

- **[orm] [bug] [eagerloading]** Fixed a regression caused by #2976 released in 0.9.4 where the "outer join" propagation along a chain of joined eager loads would incorrectly convert an "inner join" along a sibling join path into an outer join as well, when only descendant paths should be receiving the "outer join" propagation; additionally, fixed related issue where "nested" join propagation would take place inappropriately between two sibling join paths. ¶ This change is also **backported** to: 0.9.7

  References: #3131

- **[orm] [bug]** Fixed a regression from 0.9.0 due to #2736 where the `Query.select_from()` method no longer set up the "from entity" of the `Query` object correctly, so that subsequent `Query.filter_by()` or `Query.join()` calls would fail to check the appropriate "from" entity when searching for attributes by string name. ¶ This change is also **backported** to: 0.9.7

  References: #2736, #3083

- **[orm] [bug]** Fixed bug where items that were persisted, deleted, or had a primary key change within a savepoint block would not participate in being restored to their former state (not in session, in session, previous PK) after the outer transaction were rolled back. ¶ This change is also **backported** to: 0.9.7

  References: #3108

- **[orm] [bug]** Fixed bug in subquery eager loading in conjunction with `with_polymorphic()`, the targeting of entities and columns in the subquery load has been made more accurate with respect to this type of entity and others. ¶ This change is also **backported** to: 0.9.7

  References: #3106

- **[orm] [bug]** Additional checks have been added for the case where an inheriting mapper is implicitly combining one of its column-based attributes with that of the parent, where those columns normally don't necessarily share the same value. This is an extension of an existing check that was added via #1892; however this new check emits only a warning, instead of an exception, to allow for applications that may be relying upon the existing behavior.

  See also:

  I'm getting a warning or error about "Implicitly combining column X under attribute Y"

  ¶ This change is also **backported** to: 0.9.5

References: #3042

- **[orm] [bug]** Modified the behavior of `orm.load_only()` such that primary key columns are always added to the list of columns to be "undeferred"; otherwise, the ORM can't load the row's identity. Apparently, one can defer the mapped primary keys and the ORM will fail, that hasn't been changed. But as load_only is essentially saying "defer all but X", it's more critical that PK cols not be part of this deferral. ¶ This change is also **backported** to: 0.9.5

  References: #3080

- **[orm] [bug]** Fixed a few edge cases which arise in the so-called "row switch" scenario, where an INSERT/DELETE can be turned into an UPDATE. In this situation, a many-to-one relationship set to None, or in some cases a scalar attribute set to None, may not be detected as a net change in value, and therefore the UPDATE would not reset what was on the previous row. This is due to some as-yet unresovled side effects of the way attribute history works in terms of implicitly assuming None isn't really a "change" for a previously un-set attribute. See also #3061.

  Note: This change has been **REVERTED** in 0.9.6. The full fix will be in version 1.0 of SQLAlchemy.

  ¶ This change is also **backported** to: 0.9.5

  References: #3060

- **[orm] [bug]** Related to #3060, an adjustment has been made to the unit of work such that loading for related many-to-one objects is slightly more aggressive, in the case of a graph of self-referential objects that are to be deleted; the load of related objects is to help determine the correct order for deletion if passive_deletes is not set. ¶ This change is also **backported** to: 0.9.5

- **[orm] [bug]** Fixed bug in SQLite join rewriting where anonymized column names due to repeats would not correctly be rewritten in subqueries. This would affect SELECT queries with any kind of subquery + join. ¶ This change is also **backported** to: 0.9.5

  References: #3057

- **[orm] [bug] [sql]** Fixes to the newly enhanced boolean coercion in #2804 where the new rules for "where" and "having" woudn't take effect for the "whereclause" and "having" kw arguments of the `select()` construct, which is also what `Query` uses so wasn't working in the ORM either. ¶ This change is also **backported** to: 0.9.5

  References: #3013

- **[orm] [bug]** Fixed bug in subquery eager loading where a long chain of eager loads across a polymorphic-subclass boundary in conjunction with polymorphic loading would fail to locate the subclass-link in the chain, erroring out with a missing property name on an `AliasedClass`. ¶ This change is also **backported** to: 0.9.5, 0.8.7

  References: #3055

- **[orm] [bug]** Fixed ORM bug where the `class_mapper()` function would mask AttributeErrors or KeyErrors that should raise during mapper configuration due to user errors. The catch for attribute/keyerror has been made more specific to not include the configuration step. ¶ This change is also **backported** to: 0.9.5, 0.8.7

  References: #3047

- **[orm] [bug]** Fixed bug where the session attachment error "object is already attached to session X" would fail to prevent the object from also being attached to the new session, in the case that execution continued after the error raise occurred. ¶ References: #3301

- **[orm] [bug]** The primary `Mapper` of a `Query` is now passed to the `Session.get_bind()` method when calling upon `Query.count()`, `Query.update()`, `Query.delete()`, as well as queries against mapped columns, `column_property` objects, and SQL functions and expressions derived from mapped columns.

This allows sessions that rely upon either customized `Session.get_bind()` schemes or "bound" metadata to work in all relevant cases.

See also:

Session.get_bind() will receive the Mapper in all relevant Query cases

¶ References: #3242, #3227, #1326

- **[orm] [bug]** The `PropComparator.of_type()` modifier has been improved in conjunction with loader directives such as `joinedload()` and `contains_eager()` such that if two `PropComparator.of_type()` modifiers of the same base type/path are encountered, they will be joined together into a single "polymorphic" entity, rather than replacing the entity of type A with the one of type B. E.g. a joinedload of `A.b.of_type(BSub1)->BSub1.c` combined with joinedload of `A.b.of_type(BSub2)->BSub2.c` will create a single joinedload of `A.b.of_type((BSub1, BSub2)) -> BSub1.c, BSub2.c`, without the need for the `with_polymorphic` to be explicit in the query.

  See also:

  Eager Loading of Specific or Polymorphic Subtypes - contains an updated example illustrating the new format.

  ¶ References: #3256

- **[orm] [bug]** Repaired support of the `copy.deepcopy()` call when used by the `orm.util.CascadeOptions` argument, which occurs if `copy.deepcopy()` is being used with `relationship()` (not an officially supported use case). Pull request courtesy duesenfranz. ¶ References: pull request github:137

- **[orm] [bug]** Fixed bug where `Session.expunge()` would not fully detach the given object if the object had been subject to a delete operation that was flushed, but not committed. This would also affect related operations like `make_transient()`.

  See also:

  session.expunge() will fully detach an object that's been deleted

  ¶ References: #3139

- **[orm] [bug]** A warning is emitted in the case of multiple relationships that ultimately will populate a foreign key column in conflict with another, where the relationships are attempting to copy values from different source columns. This occurs in the case where composite foreign keys with overlapping columns are mapped to relationships that each refer to a different referenced column. A new documentation section illustrates the example as well as how to overcome the issue by specifying "foreign" columns specifically on a per-relationship basis.

  See also:

  Overlapping Foreign Keys

  ¶ References: #3230

- **[orm] [bug]** The `Query.update()` method will now convert string key names in the given dictionary of values into mapped attribute names against the mapped class being updated. Previously, string names were taken in directly and passed to the core update statement without any means to resolve against the mapped entity. Support for synonyms and hybrid attributes as the subject attributes of `Query.update()` are also supported.

  See also:

  query.update() now resolves string names into mapped attribute names

  ¶ References: #3228

- **[orm] [bug]** Improvements to the mechanism used by `Session` to locate "binds" (e.g. engines to use), such engines can be associated with mixin classes, concrete subclasses, as well as a wider variety of table metadata such as joined inheritance tables.

  See also:

  Session.get_bind() handles a wider variety of inheritance scenarios

  ¶ References: #3035

- **[orm] [bug]** Fixed bug in single table inheritance where a chain of joins that included the same single inh entity more than once (normally this should raise an error) could, in some cases depending on what was being joined "from", implicitly alias the second case of the single inh entity, producing a query that "worked". But as this implicit aliasing is not intended in the case of single table inheritance, it didn't really "work" fully and was very misleading, since it wouldn't always appear.

  See also:

  Changes and fixes in handling of duplicate join targets

  ¶ References: #3233

- **[orm] [bug]** The ON clause rendered when using `Query.join()`, `Query.outerjoin()`, or the standalone `orm.join()` / `orm.outerjoin()` functions to a single-inheritance subclass will now include the "single table criteria" in the ON clause even if the ON clause is otherwise hand-rolled; it is now added to the criteria using AND, the same way as if joining to a single-table target using relationship or similar.

  This is sort of in-between feature and bug.

  See also:

  single-table-inheritance criteria added to all ON clauses unconditionally

  ¶ References: #3222

- **[orm] [bug]** A major rework to the behavior of expression labels, most specifically when used with ColumnProperty constructs with custom SQL expressions and in conjunction with the "order by labels" logic first introduced in 0.9. Fixes include that an `order_by(Entity.some_col_prop)` will now make use of "order by label" rules even if Entity has been subject to aliasing, either via inheritance rendering or via the use of the `aliased()` construct; rendering of the same column property multiple times with aliasing (e.g. `query(Entity.some_prop, entity_alias.some_prop)`) will label each occurrence of the entity with a distinct label, and additionally "order by label" rules will work for both (e.g. `order_by(Entity.some_prop, entity_alias.some_prop)`). Additional issues that could prevent the "order by label" logic from working in 0.9, most notably that the state of a Label could change such that "order by label" would stop working depending on how things were called, has been fixed.

  See also:

  ColumnProperty constructs work a lot better with aliases, order_by

  ¶ References: #3148, #3188

- **[orm] [bug]** Changed the approach by which the "single inheritance criterion" is applied, when using `Query.from_self()`, or its common user `Query.count()`. The criteria to limit rows to those with a certain type is now indicated on the inside subquery, not the outside one, so that even if the "type" column is not available in the columns clause, we can filter on it on the "inner" query.

  See also:

  Change to single-table-inheritance criteria when using from_self(), count()

  ¶ References: #3177

- **[orm] [bug]** Made a small adjustment to the mechanics of lazy loading, such that it has less chance of interfering with a joinload() in the very rare circumstance that an object points to itself; in this scenario, the object refers to itself while loading its attributes which can cause a mixup between loaders. The use case of "object points to itself" is not fully supported, but the fix also removes some overhead so for now is part of testing. ¶ References: #3145
- **[orm] [bug]** The "resurrect" ORM event has been removed. This event hook had no purpose since the old "mutable attribute" system was removed in 0.8. ¶ References: #3171
- **[orm] [bug]** Fixed bug where attribute "set" events or columns with `@validates` would have events triggered within the flush process, when those columns were the targets of a "fetch and populate" operation, such as an autoincremented primary key, a Python side default, or a server-side default "eagerly" fetched via RETURNING. ¶ References: #3167
- **[orm] [bug] [py3k]** The *`IdentityMap`* exposed from *`Session.identity_map`* now returns lists for `items()` and `values()` in Py3K. Early porting to Py3K here had these returning iterators, when they technically should be "iterable views"..for now, lists are OK. ¶
- **[orm] [bug]** The "evaluator" for query.update()/delete() won't work with multi-table updates, and needs to be set to *synchronize_session=False* or *synchronize_session='fetch'*; this now raises an exception, with a message to change the synchronize setting. This is upgraded from a warning emitted as of 0.9.7. ¶ References: #3117
- **[orm] [change]** Mapped attributes marked as deferred without explicit undeferral will now remain "deferred" even if their column is otherwise present in the result set in some way. This is a performance enhancement in that an ORM load no longer spends time searching for each deferred column when the result set is obtained. However, for an application that has been relying upon this, an explicit *`undefer()`* or similar option should now be used. ¶
- **[orm] [enhancement]** Adjustment to attribute mechanics concerning when a value is implicitly initialized to None via first access; this action, which has always resulted in a population of the attribute, no longer does so; the None value is returned but the underlying attribute receives no set event. This is consistent with how collections work and allows attribute mechanics to behave more consistently; in particular, getting an attribute with no value does not squash the event that should proceed if the value is actually set to None.

  See also:

  Changes to attribute events and other operations regarding attributes that have no pre-existing value

  where bound parameters are rendered inline as strings based on a compile-time option. Work on this feature is courtesy of Dobes Vandermeer.

  > **See also:**
  >
  > *Select/Query LIMIT / OFFSET may be specified as an arbitrary SQL expression*.

  ¶ References: #3061

orm declarative

- **[feature] [orm] [declarative]** The *`declared_attr`* construct has newly improved behaviors and features in conjunction with declarative. The decorated function will now have access to the final column copies present on the local mixin when invoked, and will also be invoked exactly once for each mapped class, the returned result being memoized. A new modifier *`declared_attr.cascading`* is added as well.

  See also:

  Improvements to declarative mixins, @declared_attr and related features

  ¶ References: #3150

- **[bug] [declarative] [orm]** Fixed "'NoneType' object has no attribute 'concrete'" error when using `AbstractConcreteBase` in conjunction with a subclass that declares `__abstract__`. ¶ This change is also **backported** to: 0.9.8

  References: #3185

- **[bug] [orm] [declarative]** Fixed bug where using an `__abstract__` mixin in the middle of a declarative inheritance hierarchy would prevent attributes and configuration being correctly propagated from the base class to the inheriting class. ¶ References: #3240, #3219

- **[bug] [orm] [declarative]** A relationship set up with `declared_attr` on a `AbstractConcreteBase` base class will now be configured on the abstract base mapping automatically, in addition to being set up on descendant concrete classes as usual.

  See also:

  Improvements to declarative mixins, @declared_attr and related features

  ¶ References: #2670

engine

- **[engine] [feature]** Added new user-space accessors for viewing transaction isolation levels; `Connection.get_isolation_level()`, `Connection.default_isolation_level`. ¶ This change is also **backported** to: 0.9.9

- **[engine] [feature]** Added new event `ConnectionEvents.handle_error()`, a more fully featured and comprehensive replacement for `ConnectionEvents.dbapi_error()`. ¶ This change is also **backported** to: 0.9.7

  References: #3076

- **[engine] [feature]** A new style of warning can be emitted which will "filter" up to N occurrences of a parameterized string. This allows parameterized warnings that can refer to their arguments to be delivered a fixed number of times until allowing Python warning filters to squelch them, and prevents memory from growing unbounded within Python's warning registries.

  See also:

  Session.get_bind() handles a wider variety of inheritance scenarios

  ¶ References: #3178

- **[engine] [bug]** Fixed bug in `Connection` and pool where the `Connection.invalidate()` method, or an invalidation due to a database disconnect, would fail if the `isolation_level` parameter had been used with `Connection.execution_options()`; the "finalizer" that resets the isolation level would be called on the no longer opened connection. ¶ This change is also **backported** to: 0.9.9

  References: #3302

- **[engine] [bug]** A warning is emitted if the `isolation_level` parameter is used with `Connection.execution_options()` when a `Transaction` is in play; DBAPIs and/or SQLAlchemy dialects such as psycopg2, MySQLdb may implicitly rollback or commit the transaction, or not change the setting til next transaction, so this is never safe. ¶ This change is also **backported** to: 0.9.9

  References: #3296

- **[engine] [bug]** The execution options passed to an `Engine` either via `create_engine.execution_options` or `Engine.update_execution_options()` are not passed to the special `Connection` used to initialize the dialect within the "first connect" event; dialects will usually perform their own queries in this phase, and none of the current available options should be applied

here. In particular, the "autocommit" option was causing an attempt to autocommit within this initial connect which would fail with an AttributeError due to the non-standard state of the `Connection`. ¶ This change is also **backported** to: 0.9.8

References: #3200

- **[engine] [bug]** The string keys that are used to determine the columns impacted for an INSERT or UPDATE are now sorted when they contribute towards the "compiled cache" cache key. These keys were previously not deterministically ordered, meaning the same statement could be cached multiple times on equivalent keys, costing both in terms of memory as well as performance. ¶ This change is also **backported** to: 0.9.8

  References: #3165

- **[engine] [bug]** Fixed bug which would occur if a DBAPI exception occurs when the engine first connects and does its initial checks, and the exception is not a disconnect exception, yet the cursor raises an error when we try to close it. In this case the real exception would be quashed as we tried to log the cursor close exception via the connection pool and failed, as we were trying to access the pool's logger in a way that is inappropriate in this very specific scenario. ¶ This change is also **backported** to: 0.9.5

  References: #3063

- **[engine] [bug]** Fixed some "double invalidate" situations were detected where a connection invalidation could occur within an already critical section like a connection.close(); ultimately, these conditions are caused by the change in #2907, in that the "reset on return" feature calls out to the Connection/Transaction in order to handle it, where "disconnect detection" might be caught. However, it's possible that the more recent change in #2985 made it more likely for this to be seen as the "connection invalidate" operation is much quicker, as the issue is more reproducible on 0.9.4 than 0.9.3.

  Checks are now added within any section that an invalidate might occur to halt further disallowed operations on the invalidated connection. This includes two fixes both at the engine level and at the pool level. While the issue was observed with highly concurrent gevent cases, it could in theory occur in any kind of scenario where a disconnect occurs within the connection close operation. ¶ This change is also **backported** to: 0.9.5

  References: #3043

- **[engine] [bug]** The engine-level error handling and wrapping routines will now take effect in all engine connection use cases, including when user-custom connect routines are used via the `create_engine.creator` parameter, as well as when the `Connection` encounters a connection error on revalidation.

  See also:

  DBAPI exception wrapping and handle_error() event improvements

  ¶ References: #3266

- **[engine] [bug]** Removing (or adding) an event listener at the same time that the event is being run itself, either from inside the listener or from a concurrent thread, now raises a RuntimeError, as the collection used is now an instance of `colletions.deque()` and does not support changes while being iterated. Previously, a plain Python list was used where removal from inside the event itself would produce silent failures. ¶ References: #3163

sql

- **[sql] [feature]** Liberalized the contract for `Index` a bit in that you can specify a `text()` expression as the target; the index no longer needs to have a table-bound column present if the index is to be manually added to the table, either via inline declaration or via `Table.append_constraint()`. ¶ This change is also **backported** to: 0.9.5

  References: #3028

- **[sql] [feature]** Added new flag `expression.between.symmetric`, when set to True renders "BETWEEN SYMMETRIC". Also added a new negation operator "notbetween_op", which now allows an expression like `~col.between(x, y)` to render as "col NOT BETWEEN x AND y", rather than a parentheiszed NOT string. ¶ This change is also **backported** to: 0.9.5

  References: #2990

- **[sql] [feature]** The SQL compiler now generates the mapping of expected columns such that they are matched to the received result set positionally, rather than by name. Originally, this was seen as a way to handle cases where we had columns returned with difficult-to-predict names, though in modern use that issue has been overcome by anonymous labeling. In this version, the approach basically reduces function call count per-result by a few dozen calls, or more for larger sets of result columns. The approach still degrades into a modern version of the old approach if any discrepancy in size exists between the compiled set of columns versus what was received, so there's no issue for partially or fully textual compilation scenarios where these lists might not line up. ¶ References: #918

- **[sql] [feature]** Literal values within a `DefaultClause`, which is invoked when using the `Column.server_default` parameter, will now be rendered using the "inline" compiler, so that they are rendered as-is, rather than as bound parameters.

  See also:

  Column server defaults now render literal values

  ¶ References: #3087

- **[sql] [feature]** The type of expression is reported when an object passed to a SQL expression unit can't be interpreted as a SQL fragment; pull request courtesy Ryan P. Kelly. ¶ References: pull request github:150

- **[sql] [feature]** Added a new parameter `Table.tometadata.name` to the `Table.tometadata()` method. Similar to `Table.tometadata.schema`, this argument causes the newly copied `Table` to take on the new name instead of the existing one. An interesting capability this adds is that of copying a `Table` object to the *same* `MetaData` target with a new name. Pull request courtesy n.d. parker. ¶ References: pull request github:139

- **[sql] [feature]** Exception messages have been spiffed up a bit. The SQL statement and parameters are not displayed if None, reducing confusion for error messages that weren't related to a statement. The full module and classname for the DBAPI-level exception is displayed, making it clear that this is a wrapped DBAPI exception. The statement and parameters themselves are bounded within a bracketed sections to better isolate them from the error message and from each other. ¶ References: #3172

- **[sql] [feature]** `Insert.from_select()` now includes Python and SQL-expression defaults if otherwise unspecified; the limitation where non- server column defaults aren't included in an INSERT FROM SELECT is now lifted and these expressions are rendered as constants into the SELECT statement.

  See also:

  INSERT FROM SELECT now includes Python and SQL-expression defaults

  ¶

- **[sql] [feature]** The `UniqueConstraint` construct is now included when reflecting a `Table` object, for databases where this is applicable. In order to achieve this with sufficient accuracy, MySQL and Postgresql now contain features that correct for the duplication of indexes and unique constraints when reflecting tables, indexes, and constraints. In the case of MySQL, there is not actually a "unique constraint" concept independent of a "unique index", so for this backend `UniqueConstraint` continues to remain non-present for a reflected `Table`. For Postgresql, the query used to detect indexes against `pg_index` has been improved to check for the same construct in `pg_constraint`, and the implicitly constructed unique index is not included with a reflected `Table`.

In both cases, the `Inspector.get_indexes()` and the `Inspector.get_unique_constraints()` methods return both constructs individually, but include a new token `duplicates_constraint` in the case of Postgresql or `duplicates_index` in the case of MySQL to indicate when this condition is detected. Pull request courtesy Johannes Erdfelt.

See also:

UniqueConstraint is now part of the Table reflection process

¶ References: #3184, pull request bitbucket:30

- **[sql] [feature]** Added new method `Select.with_statement_hint()` and ORM method `Query.with_statement_hint()` to support statement-level hints that are not specific to a table. ¶ References: #3206
- **[sql] [feature]** The `info` parameter has been added as a constructor argument to all schema constructs including `MetaData`, `Index`, `ForeignKey`, `ForeignKeyConstraint`, `UniqueConstraint`, `PrimaryKeyConstraint`, `CheckConstraint`. ¶ References: #2963
- **[sql] [feature]** The `Table.autoload_with` flag now implies that `Table.autoload` should be `True`. Pull request courtesy Malik Diarra. ¶ References: #3027
- **[sql] [feature]** The `Select.limit()` and `Select.offset()` methods now accept any SQL expression, in addition to integer values, as arguments. Typically this is used to allow a bound parameter to be passed, which can be substituted with a value later thus allowing Python-side caching of the SQL query. The implementation here is fully backwards compatible with existing third party dialects, however those dialects which implement special LIMIT/OFFSET systems will need modification in order to take advantage of the new capabilities. Limit and offset also support "literal_binds" mode, ¶ References: #3034
- **[sql] [changed]** The `column()` and `table()` constructs are now importable from the "from sqlalchemy" namespace, just like every other Core construct. ¶
- **[sql] [changed]** The implicit conversion of strings to `text()` constructs when passed to most builder methods of `select()` as well as `Query` now emits a warning with just the plain string sent. The textual conversion still proceeds normally, however. The only method that accepts a string without a warning are the "label reference" methods like order_by(), group_by(); these functions will now at compile time attempt to resolve a single string argument to a column or label expression present in the selectable; if none is located, the expression still renders, but you get the warning again. The rationale here is that the implicit conversion from string to text is more unexpected than not these days, and it is better that the user send more direction to the Core / ORM when passing a raw string as to what direction should be taken. Core/ORM tutorials have been updated to go more in depth as to how text is handled.

  See also:

  Warnings emitted when coercing full SQL fragments into text()

  ¶ References: #2992
- **[sql] [bug]** Added the `native_enum` flag to the `__repr__()` output of `Enum`, which is mostly important when using it with Alembic autogenerate. Pull request courtesy Dimitris Theodorou. ¶ This change is also **backported** to: 0.9.9

  References: pull request bitbucket:41
- **[sql] [bug]** Fixed bug where using a `TypeDecorator` that implemented a type that was also a `TypeDecorator` would fail with Python's "Cannot create a consistent method resolution order (MRO)" error, when any kind of SQL comparison expression were used against an object using this type. ¶ This change is also **backported** to: 0.9.9

  References: #3278

- **[sql] [bug]** Fixed issue where the columns from a SELECT embedded in an INSERT, either through the values clause or as a "from select", would pollute the column types used in the result set produced by the RETURNING clause when columns from both statements shared the same name, leading to potential errors or mis-adaptation when retrieving the returning rows. ¶ This change is also **backported** to: 0.9.9

  References: #3248

- **[sql] [bug]** Fixed bug where a fair number of SQL elements within the sql package would fail to `__repr__()` successfully, due to a missing `description` attribute that would then invoke a recursion overflow when an internal AttributeError would then re-invoke `__repr__()`. ¶ This change is also **backported** to: 0.9.8

  References: #3195

- **[sql] [bug]** An adjustment to table/index reflection such that if an index reports a column that isn't found to be present in the table, a warning is emitted and the column is skipped. This can occur for some special system column situations as has been observed with Oracle. ¶ This change is also **backported** to: 0.9.8

  References: #3180

- **[sql] [bug]** Fixed bug in CTE where `literal_binds` compiler argument would not be always be correctly propagated when one CTE referred to another aliased CTE in a statement. ¶ This change is also **backported** to: 0.9.8

  References: #3154

- **[sql] [bug]** Fixed 0.9.7 regression caused by #3067 in conjunction with a mis-named unit test such that so-called "schema" types like *Boolean* and *Enum* could no longer be pickled. ¶ This change is also **backported** to: 0.9.8

  References: #3144, #3067

- **[sql] [bug]** Fix bug in naming convention feature where using a check constraint convention that includes `constraint_name` would then force all *Boolean* and *Enum* types to require names as well, as these implicitly create a constraint, even if the ultimate target backend were one that does not require generation of the constraint such as Postgresql. The mechanics of naming conventions for these particular constraints has been reorganized such that the naming determination is done at DDL compile time, rather than at constraint/table construction time. ¶ This change is also **backported** to: 0.9.7

  References: #3067

- **[sql] [bug]** Fixed bug in common table expressions whereby positional bound parameters could be expressed in the wrong final order when CTEs were nested in certain ways. ¶ This change is also **backported** to: 0.9.7

  References: #3090

- **[sql] [bug]** Fixed bug where multi-valued *Insert* construct would fail to check subsequent values entries beyond the first one given for literal SQL expressions. ¶ This change is also **backported** to: 0.9.7

  References: #3069

- **[sql] [bug]** Added a "str()" step to the dialect_kwargs iteration for Python version < 2.6.5, working around the "no unicode keyword arg" bug as these args are passed along as keyword args within some reflection processes. ¶ This change is also **backported** to: 0.9.7

  References: #3123

- **[sql] [bug]** The *TypeEngine.with_variant()* method will now accept a type class as an argument which is internally converted to an instance, using the same convention long established by other constructs such as *Column*. ¶ This change is also **backported** to: 0.9.7

  References: #3122

- **[sql] [bug]** The *Column.nullable* flag is implicitly set to `False` when that *Column* is referred to in an explicit *PrimaryKeyConstraint* for that table. This behavior now matches that of when the *Column* itself

has the `Column.primary_key` flag set to `True`, which is intended to be an exactly equivalent case. ¶ This change is also **backported** to: 0.9.5

References: #3023

- **[sql] [bug]** Fixed bug where the `Operators.__and__()`, `Operators.__or__()` and `Operators.__invert__()` operator overload methods could not be overridden within a custom `TypeEngine.Comparator` implementation. ¶ This change is also **backported** to: 0.9.5

  References: #3012

- **[sql] [bug]** Fixed bug in new `DialectKWArgs.argument_for()` method where adding an argument for a construct not previously included for any special arguments would fail. ¶ This change is also **backported** to: 0.9.5

  References: #3024

- **[sql] [bug]** Fixed regression introduced in 0.9 where new "ORDER BY <labelname>" feature from #1068 would not apply quoting rules to the label name as rendered in the ORDER BY. ¶ This change is also **backported** to: 0.9.5

  References: #1068, #3020

- **[sql] [bug]** Restored the import for `Function` to the `sqlalchemy.sql.expression` import namespace, which was removed at the beginning of 0.9. ¶ This change is also **backported** to: 0.9.5

- **[sql] [bug]** Fixed bug in `Enum` and other `SchemaType` subclasses where direct association of the type with a `MetaData` would lead to a hang when events (like create events) were emitted on the `MetaData`. ¶ This change is also **backported** to: 0.9.7, 0.8.7

  References: #3124

- **[sql] [bug]** Fixed a bug within the custom operator plus `TypeEngine.with_variant()` system, whereby using a `TypeDecorator` in conjunction with variant would fail with an MRO error when a comparison operator was used. ¶ This change is also **backported** to: 0.9.7, 0.8.7

  References: #3102

- **[sql] [bug]** Fixed bug in INSERT..FROM SELECT construct where selecting from a UNION would wrap the union in an anonymous (e.g. unlabled) subquery. ¶ This change is also **backported** to: 0.9.5, 0.8.7

  References: #3044

- **[sql] [bug]** Fixed bug where `Table.update()` and `Table.delete()` would produce an empty WHERE clause when an empty `and_()` or `or_()` or other blank expression were applied. This is now consistent with that of `select()`. ¶ This change is also **backported** to: 0.9.5, 0.8.7

  References: #3045

- **[sql] [bug]** The multi-values version of `Insert.values()` has been repaired to work more usefully with tables that have Python- side default values and/or functions, as well as server-side defaults. The feature will now work with a dialect that uses "positional" parameters; a Python callable will also be invoked individually for each row just as is the case with an "executemany" style invocation; a server- side default column will no longer implicitly receive the value explicitly specified for the first row, instead refusing to invoke without an explicit value.

  See also:

  Python-side defaults invoked for each row invidually when using a multivalued insert

  ¶ References: #3288

- **[sql] [bug]** Fixed bug in `Table.tometadata()` method where the `CheckConstraint` associated with a `Boolean` or `Enum` type object would be doubled in the target table. The copy process now tracks the production of this constraint object as local to a type object. ¶ References: #3260
- **[sql] [bug]** The behavioral contract of the `ForeignKeyConstraint.columns` collection has been made consistent; this attribute is now a `ColumnCollection` like that of all other constraints and is initialized at the point when the constraint is associated with a `Table`.

  See also:

  ForeignKeyConstraint.columns is now a ColumnCollection

  ¶ References: #3243
- **[sql] [bug]** The `Column.key` attribute is now used as the source of anonymous bound parameter names within expressions, to match the existing use of this value as the key when rendered in an INSERT or UPDATE statement. This allows `Column.key` to be used as a "substitute" string to work around a difficult column name that doesn't translate well into a bound parameter name. Note that the paramstyle is configurable on `create_engine()` in any case, and most DBAPIs today support a named and positional style. ¶ References: #3245
- **[sql] [bug]** Fixed the name of the `PoolEvents.reset.dbapi_connection` parameter as passed to this event; in particular this affects usage of the "named" argument style for this event. Pull request courtesy Jason Goldberger. ¶ References: pull request github:146
- **[sql] [bug]** Reversing a change that was made in 0.9, the "singleton" nature of the "constants" `null()`, `true()`, and `false()` has been reverted. These functions returning a "singleton" object had the effect that different instances would be treated as the same regardless of lexical use, which in particular would impact the rendering of the columns clause of a SELECT statement.

  See also:

  null(), false() and true() constants are no longer singletons

  ¶ References: #3170
- **[sql] [bug] [engine]** Fixed bug where a "branched" connection, that is the kind you get when you call `Connection.connect()`, would not share invalidation status with the parent. The architecture of branching has been tweaked a bit so that the branched connection defers to the parent for all invalidation status and operations. ¶ References: #3215
- **[sql] [bug] [engine]** Fixed bug where a "branched" connection, that is the kind you get when you call `Connection.connect()`, would not share transaction status with the parent. The architecture of branching has been tweaked a bit so that the branched connection defers to the parent for all transactional status and operations. ¶ References: #3190
- **[sql] [bug]** Using `Insert.from_select()` now implies `inline=True` on `insert()`. This helps to fix a bug where an INSERT...FROM SELECT construct would inadvertently be compiled as "implicit returning" on supporting backends, which would cause breakage in the case of an INSERT that inserts zero rows (as implicit returning expects a row), as well as arbitrary return data in the case of an INSERT that inserts multiple rows (e.g. only the first row of many). A similar change is also applied to an INSERT..VALUES with multiple parameter sets; implicit RETURNING will no longer emit for this statement either. As both of these constructs deal with varible numbers of rows, the `ResultProxy.inserted_primary_key` accessor does not apply. Previously, there was a documentation note that one may prefer `inline=True` with INSERT..FROM SELECT as some databases don't support returning and therefore can't do "implicit" returning, but there's no reason an INSERT...FROM SELECT needs implicit returning in any case. Regular explicit `Insert.returning()` should be used to return variable numbers of result rows if inserted data is needed. ¶ References: #3169
- **[sql] [enhancement]** Custom dialects that implement `GenericTypeCompiler` can now be constructed such that the visit methods receive an indication of the owning expression object, if any. Any visit method that accepts keyword arguments (e.g. `**kw`) will in most cases receive a keyword argument `type_expression`,

referring to the expression object that the type is contained within. For columns in DDL, the dialect's compiler class may need to alter its `get_column_specification()` method to support this as well. The `UserDefinedType.get_col_spec()` method will also receive `type_expression` if it provides `**kw` in its argument signature. ¶ References: #3074

schema

- **[schema] [feature]** The DDL generation system of *`MetaData.create_all()`* and *`MetaData.drop_all()`* has been enhanced to in most cases automatically handle the case of mutually dependent foreign key constraints; the need for the *`ForeignKeyConstraint.use_alter`* flag is greatly reduced. The system also works for constraints which aren't given a name up front; only in the case of DROP is a name required for at least one of the constraints involved in the cycle.

  See also:

  The use_alter flag on ForeignKeyConstraint is (usually) no longer needed

  ¶ References: #3282

- **[schema] [feature]** Added a new accessor *`Table.foreign_key_constraints`* to complement the *`Table.foreign_keys`* collection, as well as *`ForeignKeyConstraint.referred_table`*. ¶

- **[schema] [bug]** The *`CheckConstraint`* construct now supports naming conventions that include the token `%(column_0_name)s`; the constraint expression is scanned for columns. Additionally, naming conventions for check constraints that don't include the `%(constraint_name)s` token will now work for *`SchemaType`*-generated constraints, such as those of *`Boolean`* and *`Enum`*; this stopped working in 0.9.7 due to #3067.

  See also:

  Naming CHECK Constraints

  Configuring Naming for Boolean, Enum, and other schema types

  ¶ References: #3067, #3299

postgresql

- **[postgresql] [feature]** Added support for the `CONCURRENTLY` keyword with Postgresql indexes, established using `postgresql_concurrently`. Pull request courtesy Iuri de Silvio.

  See also:

  Indexes with CONCURRENTLY

  ¶ This change is also **backported** to: 0.9.9

  References: pull request bitbucket:45

- **[postgresql] [feature] [pg8000]** Support is added for "sane multi row count" with the pg8000 driver, which applies mostly to when using versioning with the ORM. The feature is version-detected based on pg8000 1.9.14 or greater in use. Pull request courtesy Tony Locke. ¶ This change is also **backported** to: 0.9.8

  References: pull request github:125

- **[postgresql] [feature]** Added kw argument `postgresql_regconfig` to the *`ColumnOperators.match()`* operator, allows the "reg config" argument to be specified to the `to_tsquery()` function emitted. Pull request courtesy Jonathan Vanasco. ¶ This change is also **backported** to: 0.9.7

  References: #3078, pull request bitbucket:22

- **[postgresql] [feature]** Added support for Postgresql JSONB via `JSONB`. Pull request courtesy Damian Dimmich. ¶ This change is also **backported** to: 0.9.7

  References: pull request github:101

- **[postgresql] [feature]** Added support for AUTOCOMMIT isolation level when using the pg8000 DBAPI. Pull request courtesy Tony Locke. ¶ This change is also **backported** to: 0.9.5

  References: pull request github:88

- **[postgresql] [feature]** Added a new flag `ARRAY.zero_indexes` to the Postgresql `ARRAY` type. When set to `True`, a value of one will be added to all array index values before passing to the database, allowing better interoperability between Python style zero-based indexes and Postgresql one-based indexes. Pull request courtesy Alexey Terentev. ¶ This change is also **backported** to: 0.9.5

  References: #2785, pull request bitbucket:18

- **[postgresql] [feature]** The PG8000 dialect now supports the `create_engine.encoding` parameter, by setting up the client encoding on the connection which is then intercepted by pg8000. Pull request courtesy Tony Locke. ¶ References: pull request github:132

- **[postgresql] [feature]** Added support for PG8000's native JSONB feature. Pull request courtesy Tony Locke. ¶ References: pull request github:132

- **[postgresql] [feature] [pypy]** Added support for the psycopg2cffi DBAPI on pypy. Pull request courtesy shauns.

  See also:

  `sqlalchemy.dialects.postgresql.psycopg2cffi`

  ¶ References: #3052, pull request bitbucket:34

- **[postgresql] [feature]** Added support for the FILTER keyword as applied to aggregate functions, supported by Postgresql 9.4. Pull request courtesy Ilja Everilä.

  See also:

  Postgresql FILTER keyword

  ¶ References: pull request github:134

- **[postgresql] [feature]** Support has been added for reflection of materialized views and foreign tables, as well as support for materialized views within `Inspector.get_view_names()`, and a new method `PGInspector.get_foreign_table_names()` available on the Postgresql version of `Inspector`. Pull request courtesy Rodrigo Menezes.

  See also:

  Postgresql Dialect reflects Materialized Views, Foreign Tables

  ¶ References: #2891, pull request github:128

- **[postgresql] [feature]** Added support for PG table options TABLESPACE, ON COMMIT, WITH(OUT) OIDS, and INHERITS, when rendering DDL via the `Table` construct. Pull request courtesy malikdiarra.

  See also:

  PostgreSQL Table Options

  ¶ References: #2051

- **[postgresql] [feature]** Added new method `PGInspector.get_enums()`, when using the inspector for Postgresql will provide a list of ENUM types. Pull request courtesy Ilya Pekelny. ¶ References: pull request github:126

- **[postgresql] [bug]** Repaired support for Postgresql UUID types in conjunction with the ARRAY type when using psycopg2. The psycopg2 dialect now employs use of the psycopg2.extras.register_uuid() hook so that UUID values are always passed to/from the DBAPI as UUID() objects. The `UUID.as_uuid` flag is still honored, except with psycopg2 we need to convert returned UUID objects back into strings when this is disabled. ¶ This change is also **backported** to: 0.9.9

  References: #2940

- **[postgresql] [bug]** Added support for the `postgresql.JSONB` datatype when using psycopg2 2.5.4 or greater, which features native conversion of JSONB data so that SQLAlchemy's converters must be disabled; additionally, the newly added psycopg2 extension `extras.register_default_jsonb` is used to establish a JSON deserializer passed to the dialect via the `json_deserializer` argument. Also repaired the Postgresql integration tests which weren't actually round-tripping the JSONB type as opposed to the JSON type. Pull request courtesy Mateusz Susik. ¶ This change is also **backported** to: 0.9.9

  References: pull request github:145

- **[postgresql] [bug]** Repaired the use of the "array_oid" flag when registering the HSTORE type with older psycopg2 versions < 2.4.3, which does not support this flag, as well as use of the native json serializer hook "register_default_json" with user-defined `json_deserializer` on psycopg2 versions < 2.5, which does not include native json. ¶ This change is also **backported** to: 0.9.9

- **[postgresql] [bug]** Fixed bug where Postgresql dialect would fail to render an expression in an `Index` that did not correspond directly to a table-bound column; typically when a `text()` construct was one of the expressions within the index; or could misinterpret the list of expressions if one or more of them were such an expression. ¶ This change is also **backported** to: 0.9.9

  References: #3174

- **[postgresql] [bug]** A revisit to this issue first patched in 0.9.5, apparently psycopg2's `.closed` accessor is not as reliable as we assumed, so we have added an explicit check for the exception messages "SSL SYSCALL error: Bad file descriptor" and "SSL SYSCALL error: EOF detected" when detecting an is-disconnect scenario. We will continue to consult psycopg2's connection.closed as a first check. ¶ This change is also **backported** to: 0.9.8

  References: #3021

- **[postgresql] [bug]** Fixed bug where Postgresql JSON type was not able to persist or otherwise render a SQL NULL column value, rather than a JSON-encoded `'null'`. To support this case, changes are as follows:

  - The value `null()` can now be specified, which will always result in a NULL value resulting in the statement.
  - A new parameter `JSON.none_as_null` is added, which when True indicates that the Python `None` value should be peristed as SQL NULL, rather than JSON-encoded `'null'`.

  Retrival of NULL as None is also repaired for DBAPIs other than psycopg2, namely pg8000. ¶ This change is also **backported** to: 0.9.8

  References: #3159

- **[postgresql] [bug]** The exception wrapping system for DBAPI errors can now accommodate non-standard DBAPI exceptions, such as the psycopg2 TransactionRollbackError. These exceptions will now be raised using the closest available subclass in `sqlalchemy.exc`, in the case of TransactionRollbackError, `sqlalchemy.exc.OperationalError`. ¶ This change is also **backported** to: 0.9.8

  References: #3075

- **[postgresql] [bug]** Fixed bug in `postgresql.array` object where comparison to a plain Python list would fail to use the correct array constructor. Pull request courtesy Andrew. ¶ This change is also **backported** to: 0.9.8

References: #3141, pull request github:124

- **[postgresql] [bug]** Added a supported `FunctionElement.alias()` method to functions, e.g. the `func` construct. Previously, behavior for this method was undefined. The current behavior mimics that of pre-0.9.4, which is that the function is turned into a single-column FROM clause with the given alias name, where the column itself is anonymously named. ¶ This change is also **backported** to: 0.9.8

  References: #3137

- **[postgresql] [bug] [pg8000]** Fixed bug introduced in 0.9.5 by new pg8000 isolation level feature where engine-level isolation level parameter would raise an error on connect. ¶ This change is also **backported** to: 0.9.7

  References: #3134

- **[postgresql] [bug]** The psycopg2 `.closed` accessor is now consulted when determining if an exception is a "disconnect" error; ideally, this should remove the need for any other inspection of the exception message to detect disconnect, however we will leave those existing messages in place as a fallback. This should be able to handle newer cases like "SSL EOF" conditions. Pull request courtesy Dirk Mueller. ¶ This change is also **backported** to: 0.9.5

  References: #3021, pull request github:87

- **[postgresql] [bug]** Added the `hashable=False` flag to the PG `HSTORE` type, which is needed to allow the ORM to skip over trying to "hash" an ORM-mapped HSTORE column when requesting it in a mixed column/entity list. Patch courtesy Gunnlaugur Þór Briem. ¶ This change is also **backported** to: 0.9.5, 0.8.7

  References: #3053

- **[postgresql] [bug]** Added a new "disconnect" message "connection has been closed unexpectedly". This appears to be related to newer versions of SSL. Pull request courtesy Antti Haapala. ¶ This change is also **backported** to: 0.9.5, 0.8.7

  References: pull request bitbucket:13

- **[postgresql] [bug]** The Postgresql `postgresql.ENUM` type will emit a DROP TYPE instruction when a plain `table.drop()` is called, assuming the object is not associated directly with a `MetaData` object. In order to accomodate the use case of an enumerated type shared between multiple tables, the type should be associated directly with the `MetaData` object; in this case the type will only be created at the metadata level, or if created directly. The rules for create/drop of Postgresql enumerated types have been highly reworked in general.

  See also:

  Overhaul of ENUM type create/drop rules

  ¶ References: #3319

- **[postgresql] [bug]** The `PGDialect.has_table()` method will now query against `pg_catalog.pg_table_is_visible(c.oid)`, rather than testing for an exact schema match, when the schema name is None; this so that the method will also illustrate that temporary tables are present. Note that this is a behavioral change, as Postgresql allows a non-temporary table to silently overwrite an existing temporary table of the same name, so this changes the behavior of `checkfirst` in that unusual scenario.

  See also:

  Postgresql has_table() now works for temporary tables

  ¶ References: #3264

- **[postgresql] [enhancement]** Added a new type `postgresql.OID` to the Postgresql dialect. While "oid" is generally a private type within PG that is not exposed in modern versions, there are some PG use cases such as

large object support where these types might be exposed, as well as within some user-reported schema reflection use cases. ¶ This change is also **backported** to: 0.9.5

References: #3002

mysql

- **[mysql] [feature]** The MySQL dialect now renders TIMESTAMP with NULL / NOT NULL in all cases, so that MySQL 5.6.6 with the `explicit_defaults_for_timestamp` flag enabled will will allow TIMESTAMP to continue to work as expected when `nullable=False`. Existing applications are unaffected as SQLAlchemy has always emitted NULL for a TIMESTAMP column that is `nullable=True`.

  See also:

  MySQL TIMESTAMP Type now renders NULL / NOT NULL in all cases

  TIMESTAMP Columns and NULL

  ¶ References: #3155

- **[mysql] [feature]** Updated the "supports_unicode_statements" flag to True for MySQLdb and Pymysql under Python 2. This refers to the SQL statements themselves, not the parameters, and affects issues such as table and column names using non-ASCII characters. These drivers both appear to support Python 2 Unicode objects without issue in modern versions. ¶ References: #3121

- **[mysql] [bug]** Added a version check to the MySQLdb dialect surrounding the check for 'utf8_bin' collation, as this fails on MySQL server < 5.0. ¶ This change is also **backported** to: 0.9.9

  References: #3274

- **[mysql] [bug] [mysqlconnector]** Mysqlconnector as of version 2.0, probably as a side effect of the python 3 merge, now does not expect percent signs (e.g. as used as the modulus operator and others) to be doubled, even when using the "pyformat" bound parameter format (this change is not documented by Mysqlconnector). The dialect now checks for py2k and for mysqlconnector less than version 2.0 when detecting if the modulus operator should be rendered as `%%` or `%`. ¶ This change is also **backported** to: 0.9.8

- **[mysql] [bug] [mysqlconnector]** Unicode SQL is now passed for MySQLconnector version 2.0 and above; for Py2k and MySQL < 2.0, strings are encoded. ¶ This change is also **backported** to: 0.9.8

- **[mysql] [bug]** MySQL error 2014 "commands out of sync" appears to be raised as a ProgrammingError, not OperationalError, in modern MySQL-Python versions; all MySQL error codes that are tested for "is disconnect" are now checked within OperationalError and ProgrammingError regardless. ¶ This change is also **backported** to: 0.9.7, 0.8.7

  References: #3101

- **[mysql] [bug]** Fixed bug where column names added to `mysql_length` parameter on an index needed to have the same quoting for quoted names in order to be recognized. The fix makes the quotes optional but also provides the old behavior for backwards compatibility with those using the workaround. ¶ This change is also **backported** to: 0.9.5, 0.8.7

  References: #3085

- **[mysql] [bug]** Added support for reflecting tables where an index includes KEY_BLOCK_SIZE using an equal sign. Pull request courtesy Sean McGivern. ¶ This change is also **backported** to: 0.9.5, 0.8.7

  References: pull request bitbucket:15

- **[mysql] [bug]** The MySQL dialect now supports CAST on types that are constructed as `TypeDecorator` objects. ¶

- **[mysql] [bug]** A warning is emitted when `cast()` is used with the MySQL dialect on a type where MySQL does not support CAST; MySQL only supports CAST on a subset of datatypes. SQLAlchemy has for a long time just omitted the CAST for unsupported types in the case of MySQL. While we don't want to change this now, we emit a warning to show that it's taken place. A warning is also emitted when a CAST is used with an older MySQL version (< 4) that doesn't support CAST at all, it's skipped in this case as well. ¶ References: #3237

- **[mysql] [bug]** The `mysql.SET` type has been overhauled to no longer assume that the empty string, or a set with a single empty string value, is in fact a set with a single empty string; instead, this is by default treated as the empty set. In order to handle persistence of a `mysql.SET` that actually wants to include the blank value `''` as a legitimate value, a new bitwise operational mode is added which is enabled by the `mysql.SET.retrieve_as_bitwise` flag, which will persist and retrieve values unambiguously using their bitflag positioning. Storage and retrieval of unicode values for driver configurations that aren't converting unicode natively is also repaired.

  See also:

  MySQL SET Type Overhauled to support empty sets, unicode, blank value handling

  ¶ References: #3283

- **[mysql] [bug]** The `ColumnOperators.match()` operator is now handled such that the return type is not strictly assumed to be boolean; it now returns a `Boolean` subclass called `MatchType`. The type will still produce boolean behavior when used in Python expressions, however the dialect can override its behavior at result time. In the case of MySQL, while the MATCH operator is typically used in a boolean context within an expression, if one actually queries for the value of a match expression, a floating point value is returned; this value is not compatible with SQLAlchemy's C-based boolean processor, so MySQL's result-set behavior now follows that of the `Float` type. A new operator object `notmatch_op` is also added to better allow dialects to define the negation of a match operation.

  See also:

  The match() operator now returns an agnostic MatchType compatible with MySQL's floating point return value

  ¶ References: #3263

- **[mysql] [bug]** MySQL boolean symbols "true", "false" work again. 0.9's change in #2682 disallowed the MySQL dialect from making use of the "true" and "false" symbols in the context of "IS" / "IS NOT", but MySQL supports this syntax even though it has no boolean type. MySQL remains "non native boolean", but the `true()` and `false()` symbols again produce the keywords "true" and "false", so that an expression like `column.is_(true())` again works on MySQL.

  See also:

  MySQL boolean symbols "true", "false" work again

  ¶ References: #3186

- **[mysql] [bug]** The MySQL dialect will now disable `ConnectionEvents.handle_error()` events from firing for those statements which it uses internally to detect if a table exists or not. This is achieved using an execution option `skip_user_error_events` that disables the handle error event for the scope of that execution. In this way, user code that rewrites exceptions doesn't need to worry about the MySQL dialect or other dialects that occasionally need to catch SQLAlchemy specific exceptions. ¶

- **[mysql] [bug]** Changed the default value of "raise_on_warnings" to False for MySQLconnector. This was set at True for some reason. The "buffered" flag unfortunately must stay at True as MySQLconnector does not allow a cursor to be closed unless all results are fully fetched. ¶ References: #2515

- **[mysql] [change]** The `gaerdbms` dialect is no longer necessary, and emits a deprecation warning. Google now recommends using the MySQLdb dialect directly. ¶ This change is also **backported** to: 0.9.9

  References: #3275

sqlite

- **[sqlite] [feature]** Added support for partial indexes (e.g. with a WHERE clause) on SQLite. Pull request courtesy Kai Groner.

  See also:

  Partial Indexes

  ¶ This change is also **backported** to: 0.9.9

  References: pull request bitbucket:42

- **[sqlite] [feature]** Added a new SQLite backend for the SQLCipher backend. This backend provides for encrypted SQLite databases using the pysqlcipher Python driver, which is very similar to the pysqlite driver.

  See also:

  `pysqlcipher`

  ¶ This change is also **backported** to: 0.9.9

- **[sqlite] [bug]** When selecting from a UNION using an attached database file, the pysqlite driver reports column names in cursor.description as 'dbname.tablename.colname', instead of 'tablename.colname' as it normally does for a UNION (note that it's supposed to just be 'colname' for both, but we work around it). The column translation logic here has been adjusted to retrieve the rightmost token, rather than the second token, so it works in both cases. Workaround courtesy Tony Roberts. ¶ This change is also **backported** to: 0.9.8

  References: #3211

- **[sqlite] [bug]** Fixed a SQLite join rewriting issue where a subquery that is embedded as a scalar subquery such as within an IN would receive inappropriate substitutions from the enclosing query, if the same table were present inside the subquery as were in the enclosing query such as in a joined inheritance scenario. ¶ This change is also **backported** to: 0.9.7

  References: #3130

- **[sqlite] [bug]** UNIQUE and FOREIGN KEY constraints are now fully reflected on SQLite both with and without names. Previously, foreign key names were ignored and unnamed unique constraints were skipped. Thanks to Jon Nelson for assistance with this. ¶ References: #3261, #3244

- **[sqlite] [bug]** The SQLite dialect, when using the `sqlite.DATE`, `sqlite.TIME`, or `sqlite.DATETIME` types, and given a `storage_format` that only renders numbers, will render the types in DDL as `DATE_CHAR`, `TIME_CHAR`, and `DATETIME_CHAR`, so that despite the lack of alpha characters in the values, the column will still deliver the "text affinity". Normally this is not needed, as the textual values within the default storage formats already imply text.

  See also:

  Date and Time Types

  ¶ References: #3257

- **[sqlite] [bug]** SQLite now supports reflection of unique constraints from temp tables; previously, this would fail with a TypeError. Pull request courtesy Johannes Erdfelt.

  See also:

  SQLite/Oracle have distinct methods for temporary table/view name reporting - changes regarding SQLite temporary table and view reflection.

  ¶ References: #3203, pull request bitbucket:31

- **[sqlite] [bug]** Added `Inspector.get_temp_table_names()` and `Inspector.get_temp_view_names()`; currently, only the SQLite and Oracle dialects support these methods. The return of temporary table and view names has been **removed** from SQLite and Oracle's version of `Inspector.get_table_names()` and `Inspector.get_view_names()`; other database backends cannot support this information (such as MySQL), and the scope of operation is different in that the tables can be local to a session and typically aren't supported in remote schemas.

  See also:

  SQLite/Oracle have distinct methods for temporary table/view name reporting

  ¶ References: #3204

mssql

- **[mssql] [feature]** Enabled "multivalues insert" for SQL Server 2008. Pull request courtesy Albert Cervin. Also expanded the checks for "IDENTITY INSERT" mode to include when the identity key is present in the VALUEs clause of the statement. ¶ This change is also **backported** to: 0.9.7

  References: pull request github:98

- **[mssql] [feature]** SQL Server 2012 now recommends VARCHAR(max), NVARCHAR(max), VARBINARY(max) for large text/binary types. The MSSQL dialect will now respect this based on version detection, as well as the new `deprecate_large_types` flag.

  See also:

  Large Text/Binary Type Deprecation

  ¶ References: #3039

- **[mssql] [changed]** The hostname-based connection format for SQL Server when using pyodbc will no longer specify a default "driver name", and a warning is emitted if this is missing. The optimal driver name for SQL Server changes frequently and is per-platform, so hostname based connections need to specify this. DSN-based connections are preferred.

  See also:

  PyODBC driver name is required with hostname-based SQL Server connections

  ¶ References: #3182

- **[mssql] [bug]** Fixed the version string detection in the pymssql dialect to work with Microsoft SQL Azure, which changes the word "SQL Server" to "SQL Azure". ¶ This change is also **backported** to: 0.9.8

  References: #3151

- **[mssql] [bug]** Revised the query used to determine the current default schema name to use the `database_principal_id()` function in conjunction with the `sys.database_principals` view so that we can determine the default schema independently of the type of login in progress (e.g., SQL Server, Windows, etc). ¶ This change is also **backported** to: 0.9.5

  References: #3025

- **[mssql] [bug]** Added statement encoding to the "SET IDENTITY_INSERT" statements which operate when an explicit INSERT is being interjected into an IDENTITY column, to support non-ascii table identifiers on drivers such as pyodbc + unix + py2k that don't support unicode statements. ¶ This change is also **backported** to: 0.9.7, 0.8.7

- **[mssql] [bug]** In the SQL Server pyodbc dialect, repaired the implementation for the `description_encoding` dialect parameter, which when not explicitly set was preventing cursor.description

from being parsed correctly in the case of result sets that contained names in alternate encodings. This parameter shouldn't be needed going forward. ¶ This change is also **backported** to: 0.9.7, 0.8.7

References: #3091

oracle

- **[oracle] [feature]** Added support for cx_oracle connections to a specific service name, as opposed to a tns name, by passing `?service_name=<name>` to the URL. Pull request courtesy Sławomir Ehlert. ¶ References: pull request github:152
- **[oracle] [feature]** New Oracle DDL features for tables, indexes: COMPRESS, BITMAP. Patch courtesy Gabor Gombas. ¶
- **[oracle] [feature]** Added support for CTEs under Oracle. This includes some tweaks to the aliasing syntax, as well as a new CTE feature `CTE.suffix_with()`, which is useful for adding in special Oracle-specific directives to the CTE.

  See also:

  Improved support for CTEs in Oracle

  ¶ References: #3220
- **[oracle] [feature]** Added support for the Oracle table option ON COMMIT. ¶
- **[oracle] [bug]** Fixed long-standing bug in Oracle dialect where bound parameter names that started with numbers would not be quoted, as Oracle doesn't like numerics in bound parameter names. ¶ This change is also **backported** to: 0.9.8

  References: #2138
- **[oracle] [bug] [tests]** Fixed bug in oracle dialect test suite where in one test, 'username' was assumed to be in the database URL, even though this might not be the case. ¶ This change is also **backported** to: 0.9.7

  References: #3128
- **[oracle] [bug]** An alias name will be properly quoted when referred to using the `%(name)s` token inside the `Select.with_hint()` method. Previously, the Oracle backend hadn't implemented this quoting. ¶

misc

- **[feature] [examples]** Added a new example illustrating materialized paths, using the latest relationship features. Example courtesy Jack Zhou. ¶ This change is also **backported** to: 0.9.5

  References: pull request bitbucket:21
- **[feature] [ext]** Added a new extension suite `sqlalchemy.ext.baked`. This simple but unusual system allows for a dramatic savings in Python overhead for the construction and processing of orm `Query` objects, from query construction up through rendering of a string SQL statement.

  See also:

  Baked Queries

  ¶ References: #3054
- **[feature] [examples]** A new suite of examples dedicated to providing a detailed study into performance of SQLAlchemy ORM and Core, as well as the DBAPI, from multiple perspectives. The suite runs within a container that provides built in profiling displays both through console output as well as graphically via the RunSnake tool.

See also:

Performance

¶

- **[feature] [ext]** The `sqlalchemy.ext.automap` extension will now set `cascade="all, delete-orphan"` automatically on a one-to-many relationship/backref where the foreign key is detected as containing one or more non-nullable columns. This argument is present in the keywords passed to `automap.generate_relationship()` in this case and can still be overridden. Additionally, if the `ForeignKeyConstraint` specifies `ondelete="CASCADE"` for a non-nullable or `ondelete="SET NULL"` for a nullable set of columns, the argument `passive_deletes=True` is also added to the relationship. Note that not all backends support reflection of ondelete, but backends that do include Postgresql and MySQL. ¶ References: #3210

- **[removed]** The Drizzle dialect has been removed from the Core; it is now available as sqlalchemy-drizzle, an independent, third party dialect. The dialect is still based almost entirely off of the MySQL dialect present in SQLAlchemy.

  See also:

  Drizzle Dialect is now an External Dialect

  ¶

- **[bug] [ext] [py3k]** Fixed bug where the association proxy list class would not interpret slices correctly under Py3K. Pull request courtesy Gilles Dartiguelongue. ¶ This change is also **backported** to: 0.9.9

  References: pull request github:154

- **[bug] [examples]** Updated the *Versioning with a History Table* example such that mapped columns are remapped to match column names as well as grouping of columns; in particular, this allows columns that are explicitly grouped in a same-column-named joined inheritance scenario to be mapped in the same way in the history mappings, avoiding warnings added in the 0.9 series regarding this pattern and allowing the same view of attribute keys. ¶ This change is also **backported** to: 0.9.9

- **[bug] [examples]** Fixed a bug in the examples/generic_assocaitions/discriminator_on_association.py example, where the subclasses of AddressAssociation were not being mapped as "single table inheritance", leading to problems when trying to use the mappings further. ¶ This change is also **backported** to: 0.9.9

- **[bug] [declarative]** Fixed an unlikely race condition observed in some exotic end-user setups, where the attempt to check for "duplicate class name" in declarative would hit upon a not-totally-cleaned-up weak reference related to some other class being removed; the check here now ensures the weakref still references an object before calling upon it further. ¶ This change is also **backported** to: 0.9.8

  References: #3208

- **[bug] [ext]** Fixed bug in ordering list where the order of items would be thrown off during a collection replace event, if the reorder_on_append flag were set to True. The fix ensures that the ordering list only impacts the list that is explicitly associated with the object. ¶ This change is also **backported** to: 0.9.8

  References: #3191

- **[bug] [ext]** Fixed bug where `ext.mutable.MutableDict` failed to implement the `update()` dictionary method, thus not catching changes. Pull request courtesy Matt Chisholm. ¶ This change is also **backported** to: 0.9.8

- **[bug] [ext]** Fixed bug where a custom subclass of `ext.mutable.MutableDict` would not show up in a "coerce" operation, and would instead return a plain `ext.mutable.MutableDict`. Pull request courtesy Matt Chisholm. ¶ This change is also **backported** to: 0.9.8

- **[bug] [pool]** Fixed bug in connection pool logging where the "connection checked out" debug logging message would not emit if the logging were set up using `logging.setLevel()`, rather than using the `echo_pool`

flag. Tests to assert this logging have been added. This is a regression that was introduced in 0.9.0. ¶ This change is also **backported** to: 0.9.8

References: #3168

- **[bug] [tests]** Fixed bug where "python setup.py test" wasn't calling into distutils appropriately, and errors would be emitted at the end of the test suite. ¶ This change is also **backported** to: 0.9.7

- **[bug] [declarative]** Fixed bug when the declarative `__abstract__` flag was not being distinguished for when it was actually the value `False`. The `__abstract__` flag needs to acutally evaluate to a True value at the level being tested. ¶ This change is also **backported** to: 0.9.7

  References: #3097

- **[bug] [testsuite]** In public test suite, shanged to use of `String(40)` from less-supported `Text` in `StringTest.test_literal_backslashes`. Pullreq courtesy Jan. ¶ This change is also **backported** to: 0.9.5

  References: pull request github:95

- **[bug] [tests] [py3k]** Corrected for some deprecation warnings involving the `imp` module and Python 3.3 or greater, when running tests. Pull request courtesy Matt Chisholm. ¶ This change is also **backported** to: 0.9.5

  References: #2830, pull request bitbucket:2830

- **[bug] [declarative]** The `__mapper_args__` dictionary is copied from a declarative mixin or abstract class when accessed, so that modifications made to this dictionary by declarative itself won't conflict with that of other mappings. The dictionary is modified regarding the `version_id_col` and `polymorphic_on` arguments, replacing the column within with the one that is officially mapped to the local class/table. ¶ This change is also **backported** to: 0.9.5, 0.8.7

  References: #3062

- **[bug] [ext]** Fixed bug in mutable extension where *`MutableDict`* did not report change events for the `setdefault()` dictionary operation. ¶ This change is also **backported** to: 0.9.5, 0.8.7

  References: #3093, #3051

- **[bug] [ext]** Fixed bug where *`MutableDict.setdefault()`* didn't return the existing or new value (this bug was not released in any 0.8 version). Pull request courtesy Thomas Hervé. ¶ This change is also **backported** to: 0.9.5, 0.8.7

  References: #3093, #3051, pull request bitbucket:24

6.2.3 0.9 Changelog

0.9.11

no release date

engine

- **[engine] [bug]** Fixed critical issue whereby the pool "checkout" event handler may be called against a stale connection without the "connect" event handler having been called, in the case where the pool attempted to reconnect after being invalidated and failed; the stale connection would remain present and would be used on a subsequent attempt. This issue has a greater impact in the 1.0 series subsequent to 1.0.2, as it also delivers a blanked-out `.info` dictionary to the event handler; prior to 1.0.2 the `.info` dictionary is still the previous one. ¶ References: #3497

oracle

- **[oracle] [bug] [py3k]** Fixed support for cx_Oracle version 5.2, which was tripping up SQLAlchemy's version detection under Python 3 and inadvertently not using the correct unicode mode for Python 3. This would cause issues such as bound variables mis-interpreted as NULL and rows silently not being returned. ¶ References: #3491

0.9.10

Released: July 22, 2015

orm

- **[orm] [feature]** Added a new entry `"entity"` to the dictionaries returned by `Query.column_descriptions`. This refers to the primary ORM mapped class or aliased class that is referred to by the expression. Compared to the existing entry for `"type"`, it will always be a mapped entity, even if extracted from a column expression, or None if the given expression is a pure core expression. See also #3403 which repaired a regression in this feature which was unreleased in 0.9.10 but was released in the 1.0 version. ¶ References: #3320
- **[orm] [bug]** `Query` doesn't support joins, subselects, or special FROM clauses when using the `Query.update()` or `Query.delete()` methods; instead of silently ignoring these fields if methods like `Query.join()` or `Query.select_from()` has been called, a warning is emitted. As of 1.0.0b5 this will raise an error. ¶ References: #3349
- **[orm] [bug]** Fixed bug where the state tracking within multiple, nested `Session.begin_nested()` operations would fail to propagate the "dirty" flag for an object that had been updated within the inner savepoint, such that if the enclosing savepoint were rolled back, the object would not be part of the state that was expired and therefore reverted to its database state. ¶ References: #3352

engine

- **[engine] [bug]** Added the string value `"none"` to those accepted by the `Pool.reset_on_return` parameter as a synonym for `None`, so that string values can be used for all settings, allowing utilities like `engine_from_config()` to be usable without issue. ¶ References: #3375

sql

- **[sql] [feature]** Added official support for a CTE used by the SELECT present inside of `Insert.from_select()`. This behavior worked accidentally up until 0.9.9, when it no longer worked due to unrelated changes as part of #3248. Note that this is the rendering of the WITH clause after the INSERT, before the SELECT; the full functionality of CTEs rendered at the top level of INSERT, UPDATE, DELETE is a new feature targeted for a later release. ¶ References: #3418
- **[sql] [bug]** Fixed issue where a `MetaData` object that used a naming convention would not properly work with pickle. The attribute was skipped leading to inconsistencies and failures if the unpickled `MetaData` object were used to base additional tables from. ¶ References: #3362

postgresql

- **[postgresql] [bug]** Fixed a long-standing bug where the *Enum* type as used with the psycopg2 dialect in conjunction with non-ascii values and `native_enum=False` would fail to decode return results properly. This stemmed from when the PG *postgresql.ENUM* type used to be a standalone type without a "non native" option. ¶ References: #3354

mysql

- **[mysql] [bug] [pymysql]** Fixed unicode support for PyMySQL when using an "executemany" operation with unicode parameters. SQLAlchemy now passes both the statement as well as the bound parameters as unicode objects, as PyMySQL generally uses string interpolation internally to produce the final statement, and in the case of executemany does the "encode" step only on the final statement. ¶ References: #3337
- **[mysql] [bug] [py3k]** Fixed the *mysql.BIT* type on Py3K which was not using the `ord()` function correctly. Pull request courtesy David Marin. ¶ References: #3333, pull request github:158

sqlite

- **[sqlite] [bug]** Fixed bug in SQLite dialect where reflection of UNIQUE constraints that included non-alphabetic characters in the names, like dots or spaces, would not be reflected with their name. ¶ References: #3495

misc

- **[bug] [ext]** Fixed bug where when using extended attribute instrumentation system, the correct exception would not be raised when *class_mapper()* were called with an invalid input that also happened to not be weak referencable, such as an integer. ¶ References: #3408
- **[bug] [tests] [pypy]** Fixed an import that prevented "pypy setup.py test" from working correctly. ¶ References: #3406
- **[bug] [ext]** Fixed regression from 0.9.9 where the *as_declarative()* symbol was removed from the `sqlalchemy.ext.declarative` namespace. ¶ References: #3324

0.9.9

Released: March 10, 2015

orm

- **[orm] [feature]** Added new parameter *Session.connection.execution_options* which may be used to set up execution options on a *Connection* when it is first checked out, before the transaction has begun. This is used to set up options such as isolation level on the connection before the transaction starts.

  See also:

  Setting Transaction Isolation Levels - new documentation section detailing best practices for setting transaction isolation with sessions.

  ¶ References: #3296

- **[orm] [feature]** Added new method `Session.invalidate()`, functions similarly to `Session.close()`, except also calls `Connection.invalidate()` on all connections, guaranteeing that they will not be returned to the connection pool. This is useful in situations e.g. dealing with gevent timeouts when it is not safe to use the connection further, even for rollbacks. ¶
- **[orm] [bug]** Fixed bugs in ORM object comparisons where comparison of many-to-one `!= None` would fail if the source were an aliased class, or if the query needed to apply special aliasing to the expression due to aliased joins or polymorphic querying; also fixed bug in the case where comparing a many-to-one to an object state would fail if the query needed to apply special aliasing due to aliased joins or polymorphic querying. ¶ References: #3310
- **[orm] [bug]** Fixed bug where internal assertion would fail in the case where an `after_rollback()` handler for a `Session` incorrectly adds state to that `Session` within the handler, and the task to warn and remove this state (established by #2389) attempts to proceed. ¶ References: #3309
- **[orm] [bug]** Fixed bug where TypeError raised when `Query.join()` called with unknown kw arguments would raise its own TypeError due to broken formatting. Pull request courtesy Malthe Borch. ¶ References: pull request github:147
- **[orm] [bug]** Fixed bug in lazy loading SQL construction whereby a complex primaryjoin that referred to the same "local" column multiple times in the "column that points to itself" style of self-referential join would not be substituted in all cases. The logic to determine substitutions here has been reworked to be more open-ended. ¶ References: #3300
- **[orm] [bug]** The "wildcard" loader options, in particular the one set up by the `orm.load_only()` option to cover all attributes not explicitly mentioned, now takes into account the superclasses of a given entity, if that entity is mapped with inheritance mapping, so that attribute names within the superclasses are also omitted from the load. Additionally, the polymorphic discriminator column is unconditionally included in the list, just in the same way that primary key columns are, so that even with load_only() set up, polymorphic loading of subtypes continues to function correctly. ¶ References: #3287
- **[orm] [bug] [pypy]** Fixed bug where if an exception were thrown at the start of a `Query` before it fetched results, particularly when row processors can't be formed, the cursor would stay open with results pending and not actually be closed. This is typically only an issue on an interpreter like Pypy where the cursor isn't immediately GC'ed, and can in some circumstances lead to transactions/ locks being open longer than is desirable. ¶ References: #3285
- **[orm] [bug]** Fixed a leak which would occur in the unsupported and highly non-recommended use case of replacing a relationship on a fixed mapped class many times, referring to an arbitrarily growing number of target mappers. A warning is emitted when the old relationship is replaced, however if the mapping were already used for querying, the old relationship would still be referenced within some registries. ¶ References: #3251
- **[orm] [bug] [sqlite]** Fixed bug regarding expression mutations which could express itself as a "Could not locate column" error when using `Query` to select from multiple, anonymous column entities when querying against SQLite, as a side effect of the "join rewriting" feature used by the SQLite dialect. ¶ References: #3241
- **[orm] [bug]** Fixed bug where the ON clause for `Query.join()`, and `Query.outerjoin()` to a single-inheritance subclass using `of_type()` would not render the "single table criteria" in the ON clause if the `from_joinpoint=True` flag were set. ¶ References: #3232

engine

- **[engine] [feature]** Added new user-space accessors for viewing transaction isolation levels; `Connection.get_isolation_level()`, `Connection.default_isolation_level`. ¶
- **[engine] [bug]** Fixed bug in `Connection` and pool where the `Connection.invalidate()` method, or an invalidation due to a database disconnect, would fail if the `isolation_level` parameter had been used

with `Connection.execution_options()`; the "finalizer" that resets the isolation level would be called on the no longer opened connection. ¶ References: #3302

- **[engine] [bug]** A warning is emitted if the `isolation_level` parameter is used with `Connection.execution_options()` when a `Transaction` is in play; DBAPIs and/or SQLAlchemy dialects such as psycopg2, MySQLdb may implicitly rollback or commit the transaction, or not change the setting til next transaction, so this is never safe. ¶ References: #3296

sql

- **[sql] [bug]** Added the `native_enum` flag to the `__repr__()` output of `Enum`, which is mostly important when using it with Alembic autogenerate. Pull request courtesy Dimitris Theodorou. ¶ References: pull request bitbucket:41
- **[sql] [bug]** Fixed bug where using a `TypeDecorator` that implemented a type that was also a `TypeDecorator` would fail with Python's "Cannot create a consistent method resolution order (MRO)" error, when any kind of SQL comparison expression were used against an object using this type. ¶ References: #3278
- **[sql] [bug]** Fixed issue where the columns from a SELECT embedded in an INSERT, either through the values clause or as a "from select", would pollute the column types used in the result set produced by the RETURNING clause when columns from both statements shared the same name, leading to potential errors or mis-adaptation when retrieving the returning rows. ¶ References: #3248

schema

- **[schema] [bug]** Fixed bug in 0.9's foreign key setup system, such that the logic used to link a `ForeignKey` to its parent could fail when the foreign key used "link_to_name=True" in conjunction with a target `Table` that would not receive its parent column until later, such as within a reflection + "useexisting" scenario, if the target column in fact had a key value different from its name, as would occur in reflection if column reflect events were used to alter the .key of reflected `Column` objects so that the link_to_name becomes significant. Also repaired support for column type via FK transmission in a similar way when target columns had a different key and were referenced using link_to_name. ¶ References: #3298, #1765

postgresql

- **[postgresql] [feature]** Added support for the `CONCURRENTLY` keyword with Postgresql indexes, established using `postgresql_concurrently`. Pull request courtesy Iuri de Silvio.

  See also:

  Indexes with CONCURRENTLY

  ¶ References: pull request bitbucket:45
- **[postgresql] [bug]** Repaired support for Postgresql UUID types in conjunction with the ARRAY type when using psycopg2. The psycopg2 dialect now employs use of the psycopg2.extras.register_uuid() hook so that UUID values are always passed to/from the DBAPI as UUID() objects. The `UUID.as_uuid` flag is still honored, except with psycopg2 we need to convert returned UUID objects back into strings when this is disabled. ¶ References: #2940
- **[postgresql] [bug]** Added support for the `postgresql.JSONB` datatype when using psycopg2 2.5.4 or greater, which features native conversion of JSONB data so that SQLAlchemy's converters must be disabled; additionally, the newly added psycopg2 extension `extras.register_default_jsonb` is used to establish a JSON deserializer passed to the dialect via the `json_deserializer` argument. Also repaired the

Postgresql integration tests which weren't actually round-tripping the JSONB type as opposed to the JSON type. Pull request courtesy Mateusz Susik. ¶ References: pull request github:145

- **[postgresql] [bug]** Repaired the use of the "array_oid" flag when registering the HSTORE type with older psycopg2 versions < 2.4.3, which does not support this flag, as well as use of the native json serializer hook "register_default_json" with user-defined `json_deserializer` on psycopg2 versions < 2.5, which does not include native json. ¶
- **[postgresql] [bug]** Fixed bug where Postgresql dialect would fail to render an expression in an `Index` that did not correspond directly to a table-bound column; typically when a `text()` construct was one of the expressions within the index; or could misinterpret the list of expressions if one or more of them were such an expression. ¶ References: #3174

mysql

- **[mysql] [bug]** Added a version check to the MySQLdb dialect surrounding the check for 'utf8_bin' collation, as this fails on MySQL server < 5.0. ¶ References: #3274
- **[mysql] [change]** The `gaerdbms` dialect is no longer necessary, and emits a deprecation warning. Google now recommends using the MySQLdb dialect directly. ¶ References: #3275

sqlite

- **[sqlite] [feature]** Added support for partial indexes (e.g. with a WHERE clause) on SQLite. Pull request courtesy Kai Groner.

  See also:

  Partial Indexes

  ¶ References: pull request bitbucket:42
- **[sqlite] [feature]** Added a new SQLite backend for the SQLCipher backend. This backend provides for encrypted SQLite databases using the pysqlcipher Python driver, which is very similar to the pysqlite driver.

  See also:

  `pysqlcipher`

  ¶

misc

- **[bug] [ext] [py3k]** Fixed bug where the association proxy list class would not interpret slices correctly under Py3K. Pull request courtesy Gilles Dartiguelongue. ¶ References: pull request github:154
- **[bug] [examples]** Updated the *Versioning with a History Table* example such that mapped columns are remapped to match column names as well as grouping of columns; in particular, this allows columns that are explicitly grouped in a same-column-named joined inheritance scenario to be mapped in the same way in the history mappings, avoiding warnings added in the 0.9 series regarding this pattern and allowing the same view of attribute keys. ¶
- **[bug] [examples]** Fixed a bug in the examples/generic_assocaitions/discriminator_on_association.py example, where the subclasses of AddressAssociation were not being mapped as "single table inheritance", leading to problems when trying to use the mappings further. ¶

0.9.8

Released: October 13, 2014

orm

- **[orm] [bug] [engine]** Fixed bug that affected generally the same classes of event as that of #3199, when the `named=True` parameter would be used. Some events would fail to register, and others would not invoke the event arguments correctly, generally in the case of when an event was "wrapped" for adaption in some other way. The "named" mechanics have been rearranged to not interfere with the argument signature expected by internal wrapper functions. ¶ References: #3197
- **[orm] [bug]** Fixed bug that affected many classes of event, particularly ORM events but also engine events, where the usual logic of "de duplicating" a redundant call to `event.listen()` with the same arguments would fail, for those events where the listener function is wrapped. An assertion would be hit within registry.py. This assertion has now been integrated into the deduplication check, with the added bonus of a simpler means of checking deduplication across the board. ¶ References: #3199
- **[orm] [bug]** Fixed warning that would emit when a complex self-referential primaryjoin contained functions, while at the same time remote_side was specified; the warning would suggest setting "remote side". It now only emits if remote_side isn't present. ¶ References: #3194

orm declarative

- **[bug] [declarative] [orm]** Fixed "'NoneType' object has no attribute 'concrete'" error when using `AbstractConcreteBase` in conjunction with a subclass that declares `__abstract__`. ¶ References: #3185

engine

- **[engine] [bug]** The execution options passed to an `Engine` either via `create_engine.execution_options` or `Engine.update_execution_options()` are not passed to the special `Connection` used to initialize the dialect within the "first connect" event; dialects will usually perform their own queries in this phase, and none of the current available options should be applied here. In particular, the "autocommit" option was causing an attempt to autocommit within this initial connect which would fail with an AttributeError due to the non-standard state of the `Connection`. ¶ References: #3200
- **[engine] [bug]** The string keys that are used to determine the columns impacted for an INSERT or UPDATE are now sorted when they contribute towards the "compiled cache" cache key. These keys were previously not deterministically ordered, meaning the same statement could be cached multiple times on equivalent keys, costing both in terms of memory as well as performance. ¶ References: #3165

sql

- **[sql] [bug]** Fixed bug where a fair number of SQL elements within the sql package would fail to `__repr__()` successfully, due to a missing `description` attribute that would then invoke a recursion overflow when an internal AttributeError would then re-invoke `__repr__()`. ¶ References: #3195
- **[sql] [bug]** An adjustment to table/index reflection such that if an index reports a column that isn't found to be present in the table, a warning is emitted and the column is skipped. This can occur for some special system column situations as has been observed with Oracle. ¶ References: #3180

- **[sql] [bug]** Fixed bug in CTE where `literal_binds` compiler argument would not be always be correctly propagated when one CTE referred to another aliased CTE in a statement. ¶ References: #3154
- **[sql] [bug]** Fixed 0.9.7 regression caused by #3067 in conjunction with a mis-named unit test such that so-called "schema" types like *Boolean* and *Enum* could no longer be pickled. ¶ References: #3144, #3067

postgresql

- **[postgresql] [feature] [pg8000]** Support is added for "sane multi row count" with the pg8000 driver, which applies mostly to when using versioning with the ORM. The feature is version-detected based on pg8000 1.9.14 or greater in use. Pull request courtesy Tony Locke. ¶ References: pull request github:125
- **[postgresql] [bug]** A revisit to this issue first patched in 0.9.5, apparently psycopg2's `.closed` accessor is not as reliable as we assumed, so we have added an explicit check for the exception messages "SSL SYSCALL error: Bad file descriptor" and "SSL SYSCALL error: EOF detected" when detecting an is-disconnect scenario. We will continue to consult psycopg2's connection.closed as a first check. ¶ References: #3021
- **[postgresql] [bug]** Fixed bug where Postgresql JSON type was not able to persist or otherwise render a SQL NULL column value, rather than a JSON-encoded `'null'`. To support this case, changes are as follows:
  - The value *null()* can now be specified, which will always result in a NULL value resulting in the statement.
  - A new parameter *JSON.none_as_null* is added, which when True indicates that the Python `None` value should be peristed as SQL NULL, rather than JSON-encoded `'null'`.

  Retrival of NULL as None is also repaired for DBAPIs other than psycopg2, namely pg8000. ¶ References: #3159
- **[postgresql] [bug]** The exception wrapping system for DBAPI errors can now accommodate non-standard DBAPI exceptions, such as the psycopg2 TransactionRollbackError. These exceptions will now be raised using the closest available subclass in `sqlalchemy.exc`, in the case of TransactionRollbackError, `sqlalchemy.exc.OperationalError`. ¶ References: #3075
- **[postgresql] [bug]** Fixed bug in *postgresql.array* object where comparison to a plain Python list would fail to use the correct array constructor. Pull request courtesy Andrew. ¶ References: #3141, pull request github:124
- **[postgresql] [bug]** Added a supported *FunctionElement.alias()* method to functions, e.g. the `func` construct. Previously, behavior for this method was undefined. The current behavior mimics that of pre-0.9.4, which is that the function is turned into a single-column FROM clause with the given alias name, where the column itself is anonymously named. ¶ References: #3137

mysql

- **[mysql] [bug] [mysqlconnector]** Mysqlconnector as of version 2.0, probably as a side effect of the python 3 merge, now does not expect percent signs (e.g. as used as the modulus operator and others) to be doubled, even when using the "pyformat" bound parameter format (this change is not documented by Mysqlconnector). The dialect now checks for py2k and for mysqlconnector less than version 2.0 when detecting if the modulus operator should be rendered as `%%` or `%`. ¶
- **[mysql] [bug] [mysqlconnector]** Unicode SQL is now passed for MySQLconnector version 2.0 and above; for Py2k and MySQL < 2.0, strings are encoded. ¶

sqlite

- **[sqlite] [bug]** When selecting from a UNION using an attached database file, the pysqlite driver reports column names in cursor.description as 'dbname.tablename.colname', instead of 'tablename.colname' as it normally does for a UNION (note that it's supposed to just be 'colname' for both, but we work around it). The column translation logic here has been adjusted to retrieve the rightmost token, rather than the second token, so it works in both cases. Workaround courtesy Tony Roberts. ¶ References: #3211

mssql

- **[mssql] [bug]** Fixed the version string detection in the pymssql dialect to work with Microsoft SQL Azure, which changes the word "SQL Server" to "SQL Azure". ¶ References: #3151

oracle

- **[oracle] [bug]** Fixed long-standing bug in Oracle dialect where bound parameter names that started with numbers would not be quoted, as Oracle doesn't like numerics in bound parameter names. ¶ References: #2138

misc

- **[bug] [declarative]** Fixed an unlikely race condition observed in some exotic end-user setups, where the attempt to check for "duplicate class name" in declarative would hit upon a not-totally-cleaned-up weak reference related to some other class being removed; the check here now ensures the weakref still references an object before calling upon it further. ¶ References: #3208
- **[bug] [ext]** Fixed bug in ordering list where the order of items would be thrown off during a collection replace event, if the reorder_on_append flag were set to True. The fix ensures that the ordering list only impacts the list that is explicitly associated with the object. ¶ References: #3191
- **[bug] [ext]** Fixed bug where *ext.mutable.MutableDict* failed to implement the `update()` dictionary method, thus not catching changes. Pull request courtesy Matt Chisholm. ¶
- **[bug] [ext]** Fixed bug where a custom subclass of *ext.mutable.MutableDict* would not show up in a "coerce" operation, and would instead return a plain *ext.mutable.MutableDict*. Pull request courtesy Matt Chisholm. ¶
- **[bug] [pool]** Fixed bug in connection pool logging where the "connection checked out" debug logging message would not emit if the logging were set up using `logging.setLevel()`, rather than using the `echo_pool` flag. Tests to assert this logging have been added. This is a regression that was introduced in 0.9.0. ¶ References: #3168

0.9.7

Released: July 22, 2014

orm

- **[orm] [bug] [eagerloading]** Fixed a regression caused by #2976 released in 0.9.4 where the "outer join" propagation along a chain of joined eager loads would incorrectly convert an "inner join" along a sibling join path into an outer join as well, when only descendant paths should be receiving the "outer join" propagation;

additionally, fixed related issue where "nested" join propagation would take place inappropriately between two sibling join paths. ¶ References: #3131

- **[orm] [bug]** Fixed a regression from 0.9.0 due to #2736 where the `Query.select_from()` method no longer set up the "from entity" of the `Query` object correctly, so that subsequent `Query.filter_by()` or `Query.join()` calls would fail to check the appropriate "from" entity when searching for attributes by string name. ¶ References: #2736, #3083
- **[orm] [bug]** The "evaluator" for query.update()/delete() won't work with multi-table updates, and needs to be set to *synchronize_session=False* or *synchronize_session='fetch'*; a warning is now emitted. In 1.0 this will be promoted to a full exception. ¶ References: #3117
- **[orm] [bug]** Fixed bug where items that were persisted, deleted, or had a primary key change within a savepoint block would not participate in being restored to their former state (not in session, in session, previous PK) after the outer transaction were rolled back. ¶ References: #3108
- **[orm] [bug]** Fixed bug in subquery eager loading in conjunction with `with_polymorphic()`, the targeting of entities and columns in the subquery load has been made more accurate with respect to this type of entity and others. ¶ References: #3106
- **[orm] [bug]** Fixed bug involving dynamic attributes, that was again a regression of #3060 from version 0.9.5. A self-referential relationship with lazy='dynamic' would raise a TypeError within a flush operation. ¶ References: #3099

engine

- **[engine] [feature]** Added new event `ConnectionEvents.handle_error()`, a more fully featured and comprehensive replacement for `ConnectionEvents.dbapi_error()`. ¶ References: #3076

sql

- **[sql] [bug]** Fixed bug in `Enum` and other `SchemaType` subclasses where direct association of the type with a `MetaData` would lead to a hang when events (like create events) were emitted on the `MetaData`. ¶ This change is also **backported** to: 0.8.7

  References: #3124
- **[sql] [bug]** Fixed a bug within the custom operator plus `TypeEngine.with_variant()` system, whereby using a `TypeDecorator` in conjunction with variant would fail with an MRO error when a comparison operator was used. ¶ This change is also **backported** to: 0.8.7

  References: #3102
- **[sql] [bug]** Fix bug in naming convention feature where using a check constraint convention that includes `constraint_name` would then force all `Boolean` and `Enum` types to require names as well, as these implicitly create a constraint, even if the ultimate target backend were one that does not require generation of the constraint such as Postgresql. The mechanics of naming conventions for these particular constraints has been reorganized such that the naming determination is done at DDL compile time, rather than at constraint/table construction time. ¶ References: #3067
- **[sql] [bug]** Fixed bug in common table expressions whereby positional bound parameters could be expressed in the wrong final order when CTEs were nested in certain ways. ¶ References: #3090
- **[sql] [bug]** Fixed bug where multi-valued `Insert` construct would fail to check subsequent values entries beyond the first one given for literal SQL expressions. ¶ References: #3069

- **[sql] [bug]** Added a "str()" step to the dialect_kwargs iteration for Python version < 2.6.5, working around the "no unicode keyword arg" bug as these args are passed along as keyword args within some reflection processes. ¶ References: #3123
- **[sql] [bug]** The `TypeEngine.with_variant()` method will now accept a type class as an argument which is internally converted to an instance, using the same convention long established by other constructs such as `Column`. ¶ References: #3122

postgresql

- **[postgresql] [feature]** Added kw argument `postgresql_regconfig` to the `ColumnOperators.match()` operator, allows the "reg config" argument to be specified to the `to_tsquery()` function emitted. Pull request courtesy Jonathan Vanasco. ¶ References: #3078, pull request bitbucket:22
- **[postgresql] [feature]** Added support for Postgresql JSONB via `JSONB`. Pull request courtesy Damian Dimmich. ¶ References: pull request github:101
- **[postgresql] [bug] [pg8000]** Fixed bug introduced in 0.9.5 by new pg8000 isolation level feature where engine-level isolation level parameter would raise an error on connect. ¶ References: #3134

mysql

- **[mysql] [bug]** MySQL error 2014 "commands out of sync" appears to be raised as a ProgrammingError, not OperationalError, in modern MySQL-Python versions; all MySQL error codes that are tested for "is disconnect" are now checked within OperationalError and ProgrammingError regardless. ¶ This change is also **backported** to: 0.8.7

  References: #3101

sqlite

- **[sqlite] [bug]** Fixed a SQLite join rewriting issue where a subquery that is embedded as a scalar subquery such as within an IN would receive inappropriate substitutions from the enclosing query, if the same table were present inside the subquery as were in the enclosing query such as in a joined inheritance scenario. ¶ References: #3130

mssql

- **[mssql] [feature]** Enabled "multivalues insert" for SQL Server 2008. Pull request courtesy Albert Cervin. Also expanded the checks for "IDENTITY INSERT" mode to include when the identity key is present in the VALUEs clause of the statement. ¶ References: pull request github:98
- **[mssql] [bug]** Added statement encoding to the "SET IDENTITY_INSERT" statements which operate when an explicit INSERT is being interjected into an IDENTITY column, to support non-ascii table identifiers on drivers such as pyodbc + unix + py2k that don't support unicode statements. ¶ This change is also **backported** to: 0.8.7
- **[mssql] [bug]** In the SQL Server pyodbc dialect, repaired the implementation for the `description_encoding` dialect parameter, which when not explicitly set was preventing cursor.description from being parsed correctly in the case of result sets that contained names in alternate encodings. This parameter shouldn't be needed going forward. ¶ This change is also **backported** to: 0.8.7

  References: #3091

- **[mssql] [bug]** Fixed a regression from 0.9.5 caused by #3025 where the query used to determine "default schema" is invalid in SQL Server 2000. For SQL Server 2000 we go back to defaulting to the "schema name" parameter of the dialect, which is configurable but defaults to 'dbo'. ¶ References: #3025

oracle

- **[oracle] [bug] [tests]** Fixed bug in oracle dialect test suite where in one test, 'username' was assumed to be in the database URL, even though this might not be the case. ¶ References: #3128

misc

- **[bug] [tests]** Fixed bug where "python setup.py test" wasn't calling into distutils appropriately, and errors would be emitted at the end of the test suite. ¶
- **[bug] [declarative]** Fixed bug when the declarative `__abstract__` flag was not being distinguished for when it was actually the value `False`. The `__abstract__` flag needs to acutally evaluate to a True value at the level being tested. ¶ References: #3097

0.9.6

Released: June 23, 2014

orm

- **[orm] [bug]** Reverted the change for #3060 - this is a unit of work fix that is updated more comprehensively in 1.0 via #3061. The fix in #3060 unfortunately produces a new issue whereby an eager load of a many-to-one attribute can produce an event that is interpreted into an attribute change. ¶ References: #3060

0.9.5

Released: June 23, 2014

orm

- **[orm] [feature]** The "primaryjoin" model has been stretched a bit further to allow a join condition that is strictly from a single column to itself, translated through some kind of SQL function or expression. This is kind of experimental, but the first proof of concept is a "materialized path" join condition where a path string is compared to itself using "like". The `ColumnOperators.like()` operator has also been added to the list of valid operators to use in a primaryjoin condition. ¶ References: #3029
- **[orm] [feature]** Added new utility function `make_transient_to_detached()` which can be used to manufacture objects that behave as though they were loaded from a session, then detached. Attributes that aren't present are marked as expired, and the object can be added to a Session where it will act like a persistent one. ¶ References: #3017
- **[orm] [bug]** Fixed bug in subquery eager loading where a long chain of eager loads across a polymorphic-subclass boundary in conjunction with polymorphic loading would fail to locate the subclass-link in the chain, erroring out with a missing property name on an `AliasedClass`. ¶ This change is also **backported** to: 0.8.7

  References: #3055

- **[orm] [bug]** Fixed ORM bug where the `class_mapper()` function would mask AttributeErrors or KeyErrors that should raise during mapper configuration due to user errors. The catch for attribute/keyerror has been made more specific to not include the configuration step. ¶ This change is also **backported** to: 0.8.7

  References: #3047

- **[orm] [bug]** Additional checks have been added for the case where an inheriting mapper is implicitly combining one of its column-based attributes with that of the parent, where those columns normally don't necessarily share the same value. This is an extension of an existing check that was added via #1892; however this new check emits only a warning, instead of an exception, to allow for applications that may be relying upon the existing behavior.

  See also:

  I'm getting a warning or error about "Implicitly combining column X under attribute Y"

  ¶ References: #3042

- **[orm] [bug]** Modified the behavior of `orm.load_only()` such that primary key columns are always added to the list of columns to be "undeferred"; otherwise, the ORM can't load the row's identity. Apparently, one can defer the mapped primary keys and the ORM will fail, that hasn't been changed. But as load_only is essentially saying "defer all but X", it's more critical that PK cols not be part of this deferral. ¶ References: #3080

- **[orm] [bug]** Fixed a few edge cases which arise in the so-called "row switch" scenario, where an INSERT/DELETE can be turned into an UPDATE. In this situation, a many-to-one relationship set to None, or in some cases a scalar attribute set to None, may not be detected as a net change in value, and therefore the UPDATE would not reset what was on the previous row. This is due to some as-yet unresovled side effects of the way attribute history works in terms of implicitly assuming None isn't really a "change" for a previously un-set attribute. See also #3061.

  Note: This change has been **REVERTED** in 0.9.6. The full fix will be in version 1.0 of SQLAlchemy.

  ¶ References: #3060

- **[orm] [bug]** Related to #3060, an adjustment has been made to the unit of work such that loading for related many-to-one objects is slightly more aggressive, in the case of a graph of self-referential objects that are to be deleted; the load of related objects is to help determine the correct order for deletion if passive_deletes is not set. ¶

- **[orm] [bug]** Fixed bug in SQLite join rewriting where anonymized column names due to repeats would not correctly be rewritten in subqueries. This would affect SELECT queries with any kind of subquery + join. ¶ References: #3057

- **[orm] [bug] [sql]** Fixes to the newly enhanced boolean coercion in #2804 where the new rules for "where" and "having" woudn't take effect for the "whereclause" and "having" kw arguments of the `select()` construct, which is also what `Query` uses so wasn't working in the ORM either. ¶ References: #3013

engine

- **[engine] [bug]** Fixed bug which would occur if a DBAPI exception occurs when the engine first connects and does its initial checks, and the exception is not a disconnect exception, yet the cursor raises an error when we try to close it. In this case the real exception would be quashed as we tried to log the cursor close exception via the connection pool and failed, as we were trying to access the pool's logger in a way that is inappropriate in this very specific scenario. ¶ References: #3063

- **[engine] [bug]** Fixed some "double invalidate" situations were detected where a connection invalidation could occur within an already critical section like a connection.close(); ultimately, these conditions are caused by the change in #2907, in that the "reset on return" feature calls out to the Connection/Transaction in order to handle

it, where "disconnect detection" might be caught. However, it's possible that the more recent change in #2985 made it more likely for this to be seen as the "connection invalidate" operation is much quicker, as the issue is more reproducible on 0.9.4 than 0.9.3.

Checks are now added within any section that an invalidate might occur to halt further disallowed operations on the invalidated connection. This includes two fixes both at the engine level and at the pool level. While the issue was observed with highly concurrent gevent cases, it could in theory occur in any kind of scenario where a disconnect occurs within the connection close operation. ¶ References: #3043

sql

- **[sql] [feature]** Liberalized the contract for `Index` a bit in that you can specify a `text()` expression as the target; the index no longer needs to have a table-bound column present if the index is to be manually added to the table, either via inline declaration or via `Table.append_constraint()`. ¶ References: #3028
- **[sql] [feature]** Added new flag `expression.between.symmetric`, when set to True renders "BETWEEN SYMMETRIC". Also added a new negation operator "notbetween_op", which now allows an expression like `~col.between(x, y)` to render as "col NOT BETWEEN x AND y", rather than a parentheiszed NOT string. ¶ References: #2990
- **[sql] [bug]** Fixed bug in INSERT..FROM SELECT construct where selecting from a UNION would wrap the union in an anonymous (e.g. unlabled) subquery. ¶ This change is also **backported** to: 0.8.7

  References: #3044
- **[sql] [bug]** Fixed bug where `Table.update()` and `Table.delete()` would produce an empty WHERE clause when an empty `and_()` or `or_()` or other blank expression were applied. This is now consistent with that of `select()`. ¶ This change is also **backported** to: 0.8.7

  References: #3045
- **[sql] [bug]** The `Column.nullable` flag is implicitly set to `False` when that `Column` is referred to in an explicit `PrimaryKeyConstraint` for that table. This behavior now matches that of when the `Column` itself has the `Column.primary_key` flag set to `True`, which is intended to be an exactly equivalent case. ¶ References: #3023
- **[sql] [bug]** Fixed bug where the `Operators.__and__()`, `Operators.__or__()` and `Operators.__invert__()` operator overload methods could not be overridden within a custom `TypeEngine.Comparator` implementation. ¶ References: #3012
- **[sql] [bug]** Fixed bug in new `DialectKWArgs.argument_for()` method where adding an argument for a construct not previously included for any special arguments would fail. ¶ References: #3024
- **[sql] [bug]** Fixed regression introduced in 0.9 where new "ORDER BY <labelname>" feature from #1068 would not apply quoting rules to the label name as rendered in the ORDER BY. ¶ References: #1068, #3020
- **[sql] [bug]** Restored the import for `Function` to the `sqlalchemy.sql.expression` import namespace, which was removed at the beginning of 0.9. ¶

postgresql

- **[postgresql] [feature]** Added support for AUTOCOMMIT isolation level when using the pg8000 DBAPI. Pull request courtesy Tony Locke. ¶ References: pull request github:88
- **[postgresql] [feature]** Added a new flag `ARRAY.zero_indexes` to the Postgresql `ARRAY` type. When set to `True`, a value of one will be added to all array index values before passing to the database, allowing better interoperability between Python style zero-based indexes and Postgresql one-based indexes. Pull request courtesy Alexey Terentev. ¶ References: #2785, pull request bitbucket:18

- **[postgresql] [bug]** Added the `hashable=False` flag to the PG *HSTORE* type, which is needed to allow the ORM to skip over trying to "hash" an ORM-mapped HSTORE column when requesting it in a mixed column/entity list. Patch courtesy Gunnlaugur Þór Briem. ¶ This change is also **backported** to: 0.8.7

  References: #3053

- **[postgresql] [bug]** Added a new "disconnect" message "connection has been closed unexpectedly". This appears to be related to newer versions of SSL. Pull request courtesy Antti Haapala. ¶ This change is also **backported** to: 0.8.7

  References: pull request bitbucket:13

- **[postgresql] [bug]** The psycopg2 `.closed` accessor is now consulted when determining if an exception is a "disconnect" error; ideally, this should remove the need for any other inspection of the exception message to detect disconnect, however we will leave those existing messages in place as a fallback. This should be able to handle newer cases like "SSL EOF" conditions. Pull request courtesy Dirk Mueller. ¶ References: #3021, pull request github:87

- **[postgresql] [enhancement]** Added a new type *postgresql.OID* to the Postgresql dialect. While "oid" is generally a private type within PG that is not exposed in modern versions, there are some PG use cases such as large object support where these types might be exposed, as well as within some user-reported schema reflection use cases. ¶ References: #3002

mysql

- **[mysql] [bug]** Fixed bug where column names added to `mysql_length` parameter on an index needed to have the same quoting for quoted names in order to be recognized. The fix makes the quotes optional but also provides the old behavior for backwards compatibility with those using the workaround. ¶ This change is also **backported** to: 0.8.7

  References: #3085

- **[mysql] [bug]** Added support for reflecting tables where an index includes KEY_BLOCK_SIZE using an equal sign. Pull request courtesy Sean McGivern. ¶ This change is also **backported** to: 0.8.7

  References: pull request bitbucket:15

mssql

- **[mssql] [bug]** Revised the query used to determine the current default schema name to use the `database_principal_id()` function in conjunction with the `sys.database_principals` view so that we can determine the default schema independently of the type of login in progress (e.g., SQL Server, Windows, etc). ¶ References: #3025

firebird

- **[firebird] [bug]** Fixed bug where the combination of "limit" rendering as "SELECT FIRST n ROWS" using a bound parameter (only firebird has both), combined with column-level subqueries which also feature "limit" as well as "positional" bound parameters (e.g. qmark style) would erroneously assign the subquery-level positions before that of the enclosing SELECT, thus returning parameters which are out of order. ¶ References: #3038

misc

- **[feature] [examples]** Added a new example illustrating materialized paths, using the latest relationship features. Example courtesy Jack Zhou. ¶ References: pull request bitbucket:21
- **[bug] [declarative]** The `__mapper_args__` dictionary is copied from a declarative mixin or abstract class when accessed, so that modifications made to this dictionary by declarative itself won't conflict with that of other mappings. The dictionary is modified regarding the `version_id_col` and `polymorphic_on` arguments, replacing the column within with the one that is officially mapped to the local class/table. ¶ This change is also **backported** to: 0.8.7

  References: #3062
- **[bug] [ext]** Fixed bug in mutable extension where *`MutableDict`* did not report change events for the `setdefault()` dictionary operation. ¶ This change is also **backported** to: 0.8.7

  References: #3093, #3051
- **[bug] [ext]** Fixed bug where *`MutableDict.setdefault()`* didn't return the existing or new value (this bug was not released in any 0.8 version). Pull request courtesy Thomas Hervé. ¶ This change is also **backported** to: 0.8.7

  References: #3093, #3051, pull request bitbucket:24
- **[bug] [testsuite]** In public test suite, shanged to use of `String(40)` from less-supported `Text` in `StringTest.test_literal_backslashes`. Pullreq courtesy Jan. ¶ References: pull request github:95
- **[bug] [tests] [py3k]** Corrected for some deprecation warnings involving the `imp` module and Python 3.3 or greater, when running tests. Pull request courtesy Matt Chisholm. ¶ References: #2830, pull request bitbucket:2830

0.9.4

Released: March 28, 2014

general

- **[general] [feature]** Support has been added for pytest to run tests. This runner is currently being supported in addition to nose, and will likely be preferred to nose going forward. The nose plugin system used by SQLAlchemy has been split out so that it works under pytest as well. There are no plans to drop support for nose at the moment and we hope that the test suite itself can continue to remain as agnostic of testing platform as possible. See the file README.unittests.rst for updated information on running tests with pytest.

  The test plugin system has also been enhanced to support running tests against multiple database URLs at once, by specifying the `--db` and/or `--dburi` flags multiple times. This does not run the entire test suite for each database, but instead allows test cases that are specific to certain backends make use of that backend as the test is run. When using pytest as the test runner, the system will also run specific test suites multiple times, once for each database, particularly those tests within the "dialect suite". The plan is that the enhanced system will also be used by Alembic, and allow Alembic to run migration operation tests against multiple backends in one run, including third-party backends not included within Alembic itself. Third party dialects and extensions are also encouraged to standardize on SQLAlchemy's test suite as a basis; see the file README.dialects.rst for background on building out from SQLAlchemy's test platform. ¶
- **[general] [bug]** Adjusted `setup.py` file to support the possible future removal of the `setuptools.Feature` extension from setuptools. If this keyword isn't present, the setup will still succeed with setuptools rather than falling back to distutils. C extension building can be disabled now also

by setting the DISABLE_SQLALCHEMY_CEXT environment variable. This variable works whether or not setuptools is even available. ¶ This change is also **backported** to: 0.8.6

References: #2986

- **[general] [bug]** Fixed some test/feature failures occurring in Python 3.4, in particular the logic used to wrap "column default" callables wouldn't work properly for Python built-ins. ¶ References: #2979

orm

- **[orm] [feature]** Added new parameter `orm.mapper.confirm_deleted_rows`. Defaults to True, indicates that a series of DELETE statements should confirm that the cursor rowcount matches the number of primary keys that should have matched; this behavior had been taken off in most cases (except when version_id is used) to support the unusual edge case of self-referential ON DELETE CASCADE; to accommodate this, the message is now just a warning, not an exception, and the flag can be used to indicate a mapping that expects self-refererntial cascaded deletes of this nature. See also #2403 for background on the original change. ¶ References: #3007
- **[orm] [feature]** A warning is emitted if the `MapperEvents.before_configured()` or `MapperEvents.after_configured()` events are applied to a specific mapper or mapped class, as the events are only invoked for the `Mapper` target at the general level. ¶
- **[orm] [feature]** Added a new keyword argument `once=True` to `event.listen()` and `event.listens_for()`. This is a convenience feature which will wrap the given listener such that it is only invoked once. ¶
- **[orm] [feature]** Added a new option to `relationship.innerjoin` which is to specify the string `"nested"`. When set to `"nested"` as opposed to `True`, the "chaining" of joins will parenthesize the inner join on the right side of an existing outer join, instead of chaining as a string of outer joins. This possibly should have been the default behavior when 0.9 was released, as we introduced the feature of right-nested joins in the ORM, however we are keeping it as a non-default for now to avoid further surprises.

  See also:

  Right-nested inner joins available in joined eager loads

  ¶ References: #2976
- **[orm] [bug]** Fixed ORM bug where changing the primary key of an object, then marking it for DELETE would fail to target the correct row for DELETE. ¶ This change is also **backported** to: 0.8.6

  References: #3006
- **[orm] [bug]** Fixed regression from 0.8.3 as a result of #2818 where `Query.exists()` wouldn't work on a query that only had a `Query.select_from()` entry but no other entities. ¶ This change is also **backported** to: 0.8.6

  References: #2995
- **[orm] [bug]** Improved an error message which would occur if a query() were made against a non-selectable, such as a `literal_column()`, and then an attempt was made to use `Query.join()` such that the "left" side would be determined as `None` and then fail. This condition is now detected explicitly. ¶ This change is also **backported** to: 0.8.6
- **[orm] [bug]** Removed stale names from `sqlalchemy.orm.interfaces.__all__` and refreshed with current names, so that an `import *` from this module again works. ¶ This change is also **backported** to: 0.8.6

  References: #2975
- **[orm] [bug]** Fixed a very old behavior where the lazy load emitted for a one-to-many could inappropriately pull in the parent table, and also return results inconsistent based on what's in the parent table, when the

primaryjoin includes some kind of discriminator against the parent table, such as `and_(parent.id == child.parent_id, parent.deleted == False)`. While this primaryjoin doesn't make that much sense for a one-to-many, it is slightly more common when applied to the many-to-one side, and the one-to-many comes as a result of a backref. Loading rows from `child` in this case would keep `parent.deleted == False` as is within the query, thereby yanking it into the FROM clause and doing a cartesian product. The new behavior will now substitute the value of the local "parent.deleted" for that parameter as is appropriate. Though typically, a real-world app probably wants to use a different primaryjoin for the o2m side in any case. ¶ References: #2948

- **[orm] [bug]** Improved the check for "how to join from A to B" such that when a table has multiple, composite foreign keys targeting a parent table, the *relationship.foreign_keys* argument will be properly interpreted in order to resolve the ambiguity; previously this condition would raise that there were multiple FK paths when in fact the foreign_keys argument should be establishing which one is expected. ¶ References: #2965
- **[orm] [bug]** Added support for the not-quite-yet-documented `insert=True` flag for *event.listen()* to work with mapper / instance events. ¶
- **[orm] [bug] [engine]** Fixed bug where events set to listen at the class level (e.g. on the *Mapper* or *ClassManager* level, as opposed to on an individual mapped class, and also on *Connection*) that also made use of internal argument conversion (which is most within those categories) would fail to be removable. ¶ References: #2973
- **[orm] [bug]** Fixed regression from 0.8 where using an option like *orm.lazyload()* with the "wildcard" expression, e.g. `"*"`, would raise an assertion error in the case where the query didn't contain any actual entities. This assertion is meant for other cases and was catching this one inadvertently. ¶
- **[orm] [bug] [sqlite]** More fixes to SQLite "join rewriting"; the fix from #2967 implemented right before the release of 0.9.3 affected the case where a UNION contained nested joins in it. "Join rewriting" is a feature with a wide range of possibilities and is the first intricate "SQL rewriting" feature we've introduced in years, so we're sort of going through a lot of iterations with it (not unlike eager loading back in the 0.2/0.3 series, polymorphic loading in 0.4/0.5). We should be there soon so thanks for bearing with us :). ¶ References: #2969

engine

- **[engine] [feature]** Added some new event mechanics for dialect-level events; the initial implementation allows an event handler to redefine the specific mechanics by which an arbitrary dialect invokes execute() or executemany() on a DBAPI cursor. The new events, at this point semi-public and experimental, are in support of some upcoming transaction-related extensions. ¶
- **[engine] [feature]** An event listener can now be associated with a *Engine*, after one or more *Connection* objects have been created (such as by an orm *Session* or via explicit connect) and the listener will pick up events from those connections. Previously, performance concerns pushed the event transfer from *Engine* to *Connection* at init-time only, but we've inlined a bunch of conditional checks to make this possible without any additional function calls. ¶ References: #2978
- **[engine] [bug]** A major improvement made to the mechanics by which the *Engine* recycles the connection pool when a "disconnect" condition is detected; instead of discarding the pool and explicitly closing out connections, the pool is retained and a "generational" timestamp is updated to reflect the current time, thereby causing all existing connections to be recycled when they are next checked out. This greatly simplifies the recycle process, removes the need for "waking up" connect attempts waiting on the old pool and eliminates the race condition that many immediately-discarded "pool" objects could be created during the recycle operation. ¶ References: #2985
- **[engine] [bug]** The *ConnectionEvents.after_cursor_execute()* event is now emitted for the "_cursor_execute()" method of *Connection*; this is the "quick" executor that is used for things like when a sequence is executed ahead of an INSERT statement, as well as for dialect startup checks like unicode returns, charset, etc. the *ConnectionEvents.before_cursor_execute()* event was already invoked here.

The "executemany" flag is now always set to False here, as this event always corresponds to a single execution. Previously the flag could be True if we were acting on behalf of an executemany INSERT statement. ¶

sql

- **[sql] [feature]** Added support for literal rendering of boolean values, e.g. "true" / "false" or "1" / "0". ¶
- **[sql] [feature]** Added a new feature `schema.conv()`, the purpose of which is to mark a constraint name as already having had a naming convention applied. This token will be used by Alembic migrations as of Alembic 0.6.4 in order to render constraints in migration scripts with names marked as already having been subject to a naming convention. ¶
- **[sql] [feature]** The new dialect-level keyword argument system for schema-level constructs has been enhanced in order to assist with existing schemes that rely upon addition of ad-hoc keyword arguments to constructs.

  E.g., a construct such as `Index` will again accept ad-hoc keyword arguments within the `Index.kwargs` collection, after construction:

```
idx = Index('a', 'b')
idx.kwargs['mysql_someargument'] = True
```

  To suit the use case of allowing custom arguments at construction time, the `DialectKWArgs.argument_for()` method now allows this registration:

```
Index.argument_for('mysql', 'someargument', False)

idx = Index('a', 'b', mysql_someargument=True)
```

  See also:

  `DialectKWArgs.argument_for()`

  ¶ References: #2866, #2962

- **[sql] [bug]** Fixed bug in `tuple_()` construct where the "type" of essentially the first SQL expression would be applied as the "comparison type" to a compared tuple value; this has the effect in some cases of an inappropriate "type coersion" occurring, such as when a tuple that has a mix of String and Binary values improperly coerces target values to Binary even though that's not what they are on the left side. `tuple_()` now expects heterogeneous types within its list of values. ¶ This change is also **backported** to: 0.8.6

  References: #2977

- **[sql] [bug]** Fixed an 0.9 regression where a `Table` that failed to reflect correctly wouldn't be removed from the parent `MetaData`, even though in an invalid state. Pullreq courtesy Roman Podoliaka. ¶ References: #2988, pull request github:78
- **[sql] [bug]** `MetaData.naming_convention` feature will now also apply to `CheckConstraint` objects that are associated directly with a `Column` instead of just on the `Table`. ¶
- **[sql] [bug]** Fixed bug in new `MetaData.naming_convention` feature where the name of a check constraint making use of the *"%(constraint_name)s"* token would get doubled up for the constraint generated by a boolean or enum type, and overall duplicate events would cause the *"%(constraint_name)s"* token to keep compounding itself. ¶ References: #2991
- **[sql] [bug]** Adjusted the logic which applies names to the .c collection when a no-name `BindParameter` is received, e.g. via `sql.literal()` or similar; the "key" of the bind param is used as the key within .c. rather than the rendered name. Since these binds have "anonymous" names in any case, this allows individual bound parameters to have their own name within a selectable if they are otherwise unlabeled. ¶ References: #2974

- **[sql] [bug]** Some changes to how the `FromClause.c` collection behaves when presented with duplicate columns. The behavior of emitting a warning and replacing the old column with the same name still remains to some degree; the replacement in particular is to maintain backwards compatibility. However, the replaced column still remains associated with the `c` collection now in a collection `._all_columns`, which is used by constructs such as aliases and unions, to deal with the set of columns in `c` more towards what is actually in the list of columns rather than the unique set of key names. This helps with situations where SELECT statements with same-named columns are used in unions and such, so that the union can match the columns up positionally and also there's some chance of `FromClause.corresponding_column()` still being usable here (it can now return a column that is only in selectable.c._all_columns and not otherwise named). The new collection is underscored as we still need to decide where this list might end up. Theoretically it would become the result of iter(selectable.c), however this would mean that the length of the iteration would no longer match the length of keys(), and that behavior needs to be checked out. ¶ References: #2974
- **[sql] [bug]** Fixed issue in new `TextClause.columns()` method where the ordering of columns given positionally would not be preserved. This could have potential impact in positional situations such as applying the resulting `TextAsFrom` object to a union. ¶

postgresql

- **[postgresql] [feature]** Enabled "sane multi-row count" checking for the psycopg2 DBAPI, as this seems to be supported as of psycopg2 2.0.9. ¶ This change is also **backported** to: 0.8.6
- **[postgresql] [bug]** Fixed regression caused by release 0.8.5 / 0.9.3's compatibility enhancements where index reflection on Postgresql versions specific to only the 8.1, 8.2 series again broke, surrounding the ever problematic int2vector type. While int2vector supports array operations as of 8.1, apparently it only supports CAST to a varchar as of 8.3. ¶ This change is also **backported** to: 0.8.6

  References: #3000

mysql

- **[mysql] [bug]** Tweaked the settings for mysql-connector-python; in Py2K, the "supports unicode statements" flag is now False, so that SQLAlchemy will encode the *SQL string* (note: *not* the parameters) to bytes before sending to the database. This seems to allow all unicode-related tests to pass for mysql-connector, including those that use non-ascii table/column names, as well as some tests for the TEXT type using unicode under cursor.executemany(). ¶

oracle

- **[oracle] [feature]** Added a new engine option `coerce_to_unicode=True` to the cx_Oracle dialect, which restores the cx_Oracle outputtypehandler approach to Python unicode conversion under Python 2, which was removed in 0.9.2 as a result of #2911. Some use cases would prefer that unicode coersion is unconditional for all string values, despite performance concerns. Pull request courtesy Christoph Zwerschke. ¶ References: #2911, pull request github:74
- **[oracle] [bug]** Added new datatype `oracle.DATE`, which is a subclass of `DateTime`. As Oracle has no "datetime" type per se, it instead has only `DATE`, it is appropriate here that the `DATE` type as present in the Oracle dialect be an instance of `DateTime`. This issue doesn't change anything as far as the behavior of the type, as data conversion is handled by the DBAPI in any case, however the improved subclass layout will help the use cases of inspecting types for cross-database compatibility. Also removed uppercase `DATETIME` from the Oracle dialect as this type isn't functional in that context. ¶ References: #2987

misc

- **[bug] [ext]** Fixed bug in mutable extension as well as `attributes.flag_modified()` where the change event would not be propagated if the attribute had been reassigned to itself. ¶ This change is also **backported** to: 0.8.6

  References: #2997

- **[bug] [automap] [ext]** Added support to automap for the case where a relationship should not be created between two classes that are in a joined inheritance relationship, for those foreign keys that link the subclass back to the superclass. ¶ References: #3004

- **[bug] [tests]** Fixed a few errant `u''` strings that would prevent tests from passing in Py3.2. Patch courtesy Arfrever Frehtes Taifersar Arahesis. ¶ References: #2980

- **[bug] [pool]** Fixed small issue in `SingletonThreadPool` where the current connection to be returned might get inadvertently cleaned out during the "cleanup" process. Patch courtesy jd23. ¶

- **[bug] [ext] [py3k]** Fixed bug in association proxy where assigning an empty slice (e.g. `x[:] = [...]`) would fail on Py3k. ¶

- **[bug] [ext]** Fixed a regression in association proxy caused by #2810 which caused a user-provided "getter" to no longer receive values of `None` when fetching scalar values from a target that is non-present. The check for None introduced by this change is now moved into the default getter, so a user-provided getter will also again receive values of None. ¶ References: #2810

- **[bug] [examples]** Fixed bug in the versioned_history example where column-level INSERT defaults would prevent history values of NULL from being written. ¶

0.9.3

Released: February 19, 2014

orm

- **[orm] [feature]** Added new `MapperEvents.before_configured()` event which allows an event at the start of `configure_mappers()`, as well as `__declare_first__()` hook within declarative to complement `__declare_last__()`. ¶

- **[orm] [bug]** Fixed bug where `Query.get()` would fail to consistently raise the `InvalidRequestError` that invokes when called on a query with existing criterion, when the given identity is already present in the identity map. ¶ This change is also **backported** to: 0.8.5

  References: #2951

- **[orm] [bug] [sqlite]** Fixed bug in SQLite "join rewriting" where usage of an exists() construct would fail to be rewritten properly, such as when the exists is mapped to a column_property in an intricate nested-join scenario. Also fixed a somewhat related issue where join rewriting would fail on the columns clause of the SELECT statement if the targets were aliased tables, as opposed to individual aliased columns. ¶ References: #2967

- **[orm] [bug]** Fixed an 0.9 regression where ORM instance or mapper events applied to a base class such as a declarative base with the propagate=True flag would fail to apply to existing mapped classes which also used inheritance due to an assertion. Addtionally, repaired an attribute error which could occur during removal of such an event, depending on how it was first assigned. ¶ References: #2949

- **[orm] [bug]** Improved the initialization logic of composite attributes such that calling `MyClass.attribute` will not require that the configure mappers step has occurred, e.g. it will just work without throwing any error. ¶ References: #2935

- **[orm] [bug]** More issues with [ticket:2932] first resolved in 0.9.2 where using a column key of the form `<tablename>_<columnname>` matching that of an aliased column in the text would still not match at the ORM level, which is ultimately due to a core column-matching issue. Additional rules have been added so that the column `_label` is taken into account when working with a `TextAsFrom` construct or with literal columns. ¶ References: #2932

orm declarative

- **[bug] [orm] [declarative]** Fixed bug where `AbstractConcreteBase` would fail to be fully usable within declarative relationship configuration, as its string classname would not be available in the registry of classnames at mapper configuration time. The class now explicitly adds itself to the class regsitry, and additionally both `AbstractConcreteBase` as well as `ConcreteBase` set themselves up *before* mappers are configured within the `configure_mappers()` setup, using the new `MapperEvents.before_configured()` event. ¶ References: #2950

engine

- **[engine] [bug] [pool]** Fixed a critical regression caused by #2880 where the newly concurrent ability to return connections from the pool means that the "first_connect" event is now no longer synchronized either, thus leading to dialect mis-configurations under even minimal concurrency situations. ¶ This change is also **backported** to: 0.8.5

  References: #2964, #2880

sql

- **[sql] [bug]** Fixed bug where calling `Insert.values()` with an empty list or tuple would raise an IndexError. It now produces an empty insert construct as would be the case with an empty dictionary. ¶ This change is also **backported** to: 0.8.5

  References: #2944

- **[sql] [bug]** Fixed bug where `in_()` would go into an endless loop if erroneously passed a column expression whose comparator included the `__getitem__()` method, such as a column that uses the `postgresql.ARRAY` type. ¶ This change is also **backported** to: 0.8.5

  References: #2957

- **[sql] [bug]** Fixed regression in new "naming convention" feature where conventions would fail if the referred table in a foreign key contained a schema name. Pull request courtesy Thomas Farvour. ¶ References: pull request github:67

- **[sql] [bug]** Fixed bug where so-called "literal render" of `bindparam()` constructs would fail if the bind were constructed with a callable, rather than a direct value. This prevented ORM expressions from being rendered with the "literal_binds" compiler flag. ¶

postgresql

- **[postgresql] [feature]** Added the `TypeEngine.python_type` convenience accessor onto the `postgresql.ARRAY` type. Pull request courtesy Alexey Terentev. ¶ References: pull request github:64

- **[postgresql] [bug]** Added an additional message to psycopg2 disconnect detection, "could not send data to server", which complements the existing "could not receive data from server" and has been observed by users. ¶ This change is also **backported** to: 0.8.5

References: #2936

- **[postgresql] [bug]**

  Support has been improved for Postgresql reflection behavior on very old (pre 8.1) versions of Postgresql, and potentially other PG engines such as Redshift (assuming Redshift reports the version as < 8.1). The query for "indexes" as well as "primary keys" relies upon inspecting a so-called "int2vector" datatype, which refuses to coerce to an array prior to 8.1 causing failures regarding the "ANY()" operator used in the query. Extensive googling has located the very hacky, but recommended-by-PG-core-developer query to use when PG version < 8.1 is in use, so index and primary key constraint reflection now work on these versions.

  ¶ This change is also **backported** to: 0.8.5

- **[postgresql] [bug]** Revised this very old issue where the Postgresql "get primary key" reflection query were updated to take into account primary key constraints that were renamed; the newer query fails on very old versions of Postgresql such as version 7, so the old query is restored in those cases when server_version_info < (8, 0) is detected. ¶ This change is also **backported** to: 0.8.5

  References: #2291

- **[postgresql] [bug]** Added server version detection to the newly added dialect startup query for "show standard_conforming_strings"; as this variable was added as of PG 8.2, we skip the query for PG versions who report a version string earlier than that. ¶ References: #2946

mysql

- **[mysql] [feature]** Added new MySQL-specific `mysql.DATETIME` which includes fractional seconds support; also added fractional seconds support to `mysql.TIMESTAMP`. DBAPI support is limited, though fractional seconds are known to be supported by MySQL Connector/Python. Patch courtesy Geert JM Vanderkelen. ¶ This change is also **backported** to: 0.8.5

  References: #2941

- **[mysql] [bug]** Added support for the `PARTITION BY` and `PARTITIONS` MySQL table keywords, specified as `mysql_partition_by='value'` and `mysql_partitions='value'` to `Table`. Pull request courtesy Marcus McCurdy. ¶ This change is also **backported** to: 0.8.5

  References: #2966, pull request bitbucket:12

- **[mysql] [bug]** Fixed bug which prevented MySQLdb-based dialects (e.g. pymysql) from working in Py3K, where a check for "connection charset" would fail due to Py3K's more strict value comparison rules. The call in question wasn't taking the database version into account in any case as the server version was still None at that point, so the method overall has been simplified to rely upon connection.character_set_name(). ¶ This change is also **backported** to: 0.8.5

  References: #2933

- **[mysql] [bug] [cymysql]** Fixed bug in cymysql dialect where a version string such as `'33a-MariaDB'` would fail to parse properly. Pull request courtesy Matt Schmidt. ¶ References: #2934, pull request github:69

sqlite

- **[sqlite] [bug]** The SQLite dialect will now skip unsupported arguments when reflecting types; such as if it encounters a string like `INTEGER(5)`, the `INTEGER` type will be instantiated without the "5" being included, based on detecting a `TypeError` on the first attempt. ¶

- **[sqlite] [bug]** Support has been added to SQLite type reflection to fully support the "type affinity" contract specified at http://www.sqlite.org/datatype3.html. In this scheme, keywords like `INT`, `CHAR`, `BLOB` or `REAL` located in the type name generically associate the type with one of five affinities. Pull request courtesy Erich Blume.

  See also:

  Type Reflection

  ¶ References: pull request github:65

misc

- **[feature] [examples]** Added optional "changed" column to the versioned rows example, as well as support for when the versioned `Table` has an explicit `schema` argument. Pull request courtesy jplaverdure. ¶ References: pull request github:41
- **[bug] [ext]** Fixed bug where the `AutomapBase` class of the new automap extension would fail if classes were pre-arranged in single or potentially joined inheritance patterns. The repaired joined inheritance issue could also potentially apply when using `DeferredReflection` as well. ¶

0.9.2

Released: February 2, 2014

orm

- **[orm] [feature]** Added a new parameter `Operators.op.is_comparison`. This flag allows a custom op from `Operators.op()` to be considered as a "comparison" operator, thus usable for custom `relationship.primaryjoin` conditions.

  See also:

  Using custom operators in join conditions

  ¶
- **[orm] [feature]** Support is improved for supplying a `join()` construct as the target of `relationship.secondary` for the purposes of creating very complex `relationship()` join conditions. The change includes adjustments to query joining, joined eager loading to not render a SELECT subquery, changes to lazy loading such that the "secondary" target is properly included in the SELECT, and changes to declarative to better support specification of a join() object with classes as targets.

  The new use case is somewhat experimental, but a new documentation section has been added.

  See also:

  Composite "Secondary" Joins

  ¶
- **[orm] [bug]** Fixed error message when an iterator object is passed to `class_mapper()` or similar, where the error would fail to render on string formatting. Pullreq courtesy Kyle Stark. ¶ This change is also **backported** to: 0.8.5

  References: pull request github:58

- **[orm] [bug]** Fixed bug in new `TextAsFrom` construct where `Column`- oriented row lookups were not matching up to the ad-hoc `ColumnClause` objects that `TextAsFrom` generates, thereby making it not usable as a target in `Query.from_statement()`. Also fixed `Query.from_statement()` mechanics to not mistake a `TextAsFrom` for a `Select` construct. This bug is also an 0.9 regression as the `Text.columns()` method is called to accommodate the `text.typemap` argument. ¶ References: #2932
- **[orm] [bug]** Added a new directive used within the scope of an attribute "set" operation to disable autoflush, in the case that the attribute needs to lazy-load the "old" value, as in when replacing one-to-one values or some kinds of many-to-one. A flush at this point otherwise occurs at the point that the attribute is None and can cause NULL violations. ¶ References: #2921
- **[orm] [bug]** Fixed an 0.9 regression where the automatic aliasing applied by `Query` and in other situations where selects or joins were aliased (such as joined table inheritance) could fail if a user-defined `Column` subclass were used in the expression. In this case, the subclass would fail to propagate ORM-specific "annotations" along needed by the adaptation. The "expression annotations" system has been corrected to account for this case. ¶ References: #2918
- **[orm] [bug]** Fixed a bug involving the new flattened JOIN structures which are used with `joinedload()` (thereby causing a regression in joined eager loading) as well as `aliased()` in conjunction with the `flat=True` flag and joined-table inheritance; basically multiple joins across a "parent JOIN sub" entity using different paths to get to a target class wouldn't form the correct ON conditions. An adjustment / simplification made in the mechanics of figuring out the "left side" of the join in the case of an aliased, joined-inh class repairs the issue. ¶ References: #2908

engine

- **[engine] [feature] [pool]** Added a new pool event `PoolEvents.invalidate()`. Called when a DBAPI connection is to be marked as "invaldated" and discarded from the pool. ¶

sql

- **[sql] [feature]** Added `MetaData.reflect.**dialect_kwargs` to support dialect-level reflection options for all `Table` objects reflected. ¶
- **[sql] [feature]** Added a new feature which allows automated naming conventions to be applied to `Constraint` and `Index` objects. Based on a recipe in the wiki, the new feature uses schema-events to set up names as various schema objects are associated with each other. The events then expose a configuration system through a new argument `MetaData.naming_convention`. This system allows production of both simple and custom naming schemes for constraints and indexes on a per-`MetaData` basis.

  See also:

  Configuring Constraint Naming Conventions

  ¶ References: #2923
- **[sql] [feature]** Options can now be specified on a `PrimaryKeyConstraint` object independently of the specification of columns in the table with the `primary_key=True` flag; use a `PrimaryKeyConstraint` object with no columns in it to achieve this result.

  Previously, an explicit `PrimaryKeyConstraint` would have the effect of those columns marked as `primary_key=True` being ignored; since this is no longer the case, the `PrimaryKeyConstraint` will now assert that either one style or the other is used to specify the columns, or if both are present, that the column lists match exactly. If an inconsistent set of columns in the `PrimaryKeyConstraint` and within the `Table` marked as `primary_key=True` are present, a warning is emitted, and the list of columns is taken only from the `PrimaryKeyConstraint` alone as was the case in previous releases.

See also:

PrimaryKeyConstraint

¶ References: #2910

- **[sql] [feature]** The system by which schema constructs and certain SQL constructs accept dialect-specific keyword arguments has been enhanced. This system includes commonly the *Table* and *Index* constructs, which accept a wide variety of dialect-specific arguments such as `mysql_engine` and `postgresql_where`, as well as the constructs *PrimaryKeyConstraint*, *UniqueConstraint*, *Update*, *Insert* and *Delete*, and also newly added kwarg capability to *ForeignKeyConstraint* and *ForeignKey*. The change is that participating dialects can now specify acceptable argument lists for these constructs, allowing an argument error to be raised if an invalid keyword is specified for a particular dialect. If the dialect portion of the keyword is unrecognized, a warning is emitted only; while the system will actually make use of setuptools entrypoints in order to locate non-local dialects, the use case where certain dialect-specific arguments are used in an environment where that third-party dialect is uninstalled remains supported. Dialects also have to explicitly opt-in to this system, so that external dialects which aren't making use of this system will remain unaffected. ¶ References: #2866

- **[sql] [bug]** The behavior of *Table.tometadata()* has been adjusted such that the schema target of a *ForeignKey* will not be changed unless that schema matches that of the parent table. That is, if a table "schema_a.user" has a foreign key to "schema_b.order.id", the "schema_b" target will be maintained whether or not the "schema" argument is passed to *Table.tometadata()*. However if a table "schema_a.user" refers to "schema_a.order.id", the presence of "schema_a" will be updated on both the parent and referred tables. This is a behavioral change hence isn't likely to be backported to 0.8; it is assumed that the previous behavior is pretty buggy however and that it's unlikely anyone was relying upon it.

  Additionally, a new parameter has been added *Table.tometadata.referred_schema_fn*. This refers to a callable function which will be used to determine the new referred schema for any *ForeignKeyConstraint* encountered in the tometadata operation. This callable can be used to revert to the previous behavior or to customize how referred schemas are treated on a per-constraint basis. ¶ References: #2913

- **[sql] [bug]** Fixed bug whereby binary type would fail in some cases if used with a "test" dialect, such as a DefaultDialect or other dialect with no DBAPI. ¶

- **[sql] [bug] [py3k]** Fixed bug where "literal binds" wouldn't work with a bound parameter that's a binary type. A similar, but different, issue is fixed in 0.8. ¶

- **[sql] [bug]** Fixed regression whereby the "annotation" system used by the ORM was leaking into the names used by standard functions in *sqlalchemy.sql.functions*, such as `func.coalesce()` and `func.max()`. Using these functions in ORM attributes and thus producing annotated versions of them could corrupt the actual function name rendered in the SQL. ¶ References: #2927

- **[sql] [bug]** Fixed 0.9 regression where the new sortable support for *RowProxy* would lead to `TypeError` when compared to non-tuple types as it attempted to apply tuple() to the "other" object unconditionally. The full range of Python comparison operators have now been implemented on *RowProxy*, using an approach that guarantees a comparison system that is equivalent to that of a tuple, and the "other" object is only coerced if it's an instance of RowProxy. ¶ References: #2924, #2848

- **[sql] [bug]** A *UniqueConstraint* created inline with a *Table* that has no columns within it will be skipped. Pullreq courtesy Derek Harland. ¶ References: pull request bitbucket:11

- **[sql] [bug] [orm]** Fixed the multiple-table "UPDATE..FROM" construct, only usable on MySQL, to correctly render the SET clause among multiple columns with the same name across tables. This also changes the name used for the bound parameter in the SET clause to "<tablename>_<colname>" for the non-primary table only; as this parameter is typically specified using the *Column* object directly this should not have an impact on applications. The fix takes effect for both *Table.update()* as well as *Query.update()* in the ORM. ¶ References: #2912

schema

- **[schema] [bug]** Restored `sqlalchemy.schema.SchemaVisitor` to the `.schema` module. Pullreq courtesy Sean Dague. ¶ References: pull request github:57

postgresql

- **[postgresql] [feature]** Added a new dialect-level argument `postgresql_ignore_search_path`; this argument is accepted by both the *`Table`* constructor as well as by the *`MetaData.reflect()`* method. When in use against Postgresql, a foreign-key referenced table which specifies a remote schema name will retain that schema name even if the name is present in the `search_path`; the default behavior since 0.7.3 has been that schemas present in `search_path` would not be copied to reflected *`ForeignKey`* objects. The documentation has been updated to describe in detail the behavior of the `pg_get_constraintdef()` function and how the `postgresql_ignore_search_path` feature essentially determines if we will honor the schema qualification reported by this function or not.

  See also:

  Remote-Schema Table Introspection and Postgresql search_path

  ¶ References: #2922

mysql

- **[mysql] [bug]** Some missing methods added to the cymysql dialect, including _get_server_version_info() and _detect_charset(). Pullreq courtesy Hajime Nakagami. ¶ This change is also **backported** to: 0.8.5

  References: pull request github:61

- **[mysql] [bug] [sql]** Added new test coverage for so-called "down adaptions" of SQL types, where a more specific type is adapted to a more generic one - this use case is needed by some third party tools such as `sqlacodegen`. The specific cases that needed repair within this test suite were that of *`mysql.ENUM`* being downcast into a *`types.Enum`*, and that of SQLite date types being cast into generic date types. The `adapt()` method needed to become more specific here to counteract the removal of a "catch all" `**kwargs` collection on the base *`TypeEngine`* class that was removed in 0.9. ¶ References: #2917

- **[mysql] [bug]** The MySQL CAST compilation now takes into account aspects of a string type such as "charset" and "collation". While MySQL wants all character- based CAST calls to use the CHAR type, we now create a real CHAR object at CAST time and copy over all the parameters it has, so that an expression like `cast(x, mysql.TEXT(charset='utf8'))` will render `CAST(t.col AS CHAR CHARACTER SET utf8)`. ¶

- **[mysql] [bug]** Added new "unicode returns" detection to the MySQL dialect and to the default dialect system overall, such that any dialect can add extra "tests" to the on-first-connect "does this DBAPI return unicode directly?" detection. In this case, we are adding a check specifically against the "utf8" encoding with an explicit "utf8_bin" collation type (after checking that this collation is available) to test for some buggy unicode behavior observed with MySQLdb version 1.2.3. While MySQLdb has resolved this issue as of 1.2.4, the check here should guard against regressions. The change also allows the "unicode" checks to log in the engine logs, which was not previously the case. ¶ References: #2906

- **[mysql] [bug] [engine] [pool]** *`Connection`* now associates a new `RootTransaction` or *`TwoPhaseTransaction`* with its immediate *`_ConnectionFairy`* as a "reset handler" for the span of that transaction, which takes over the task of calling commit() or rollback() for the "reset on return" behavior of *`Pool`* if the transaction was not otherwise completed. This resolves the issue that a picky transaction like that of MySQL two-phase will be properly closed out when the connection is closed without an explicit rollback or commit (e.g. no longer raises "XAER_RMFAIL" in this case - note this only shows up in logging as the

exception is not propagated within pool reset). This issue would arise e.g. when using an orm `Session` with `twophase` set, and then `Session.close()` is called without an explicit rollback or commit. The change also has the effect that you will now see an explicit "ROLLBACK" in the logs when using a `Session` object in non-autocommit mode regardless of how that session was discarded. Thanks to Jeff Dairiki and Laurence Rowe for isolating the issue here. ¶ References: #2907

sqlite

- **[sqlite] [bug]** Fixed bug whereby SQLite compiler failed to propagate compiler arguments such as "literal binds" into a CAST expression. ¶

mssql

- **[mssql] [feature]** Added an option `mssql_clustered` to the `UniqueConstraint` and `PrimaryKeyConstraint` constructs; on SQL Server, this adds the `CLUSTERED` keyword to the constraint construct within DDL. Pullreq courtesy Derek Harland. ¶ References: pull request bitbucket:11

oracle

- **[oracle] [bug]** It's been observed that the usage of a cx_Oracle "outputtypehandler" in Python 2.xx in order to coerce string values to Unicode is inordinately expensive; even though cx_Oracle is written in C, when you pass the Python `unicode` primitive to cursor.var() and associate with an output handler, the library counts every conversion as a Python function call with all the requisite overhead being recorded; this *despite* the fact when running in Python 3, all strings are also unconditionally coerced to unicode but it does *not* incur this overhead, meaning that cx_Oracle is failing to use performant techniques in Py2K. As SQLAlchemy cannot easily select for this style of type handler on a per-column basis, the handler was assembled unconditionally thereby adding the overhead to all string access.

  So this logic has been replaced with SQLAlchemy's own unicode conversion system, which now only takes effect in Py2K for columns that are requested as unicode. When C extensions are used, SQLAlchemy's system appears to be 2-3x faster than cx_Oracle's. Additionally, SQLAlchemy's unicode conversion has been enhanced such that when the "conditional" converter is required (now needed for the Oracle backend), the check for "already unicode" is now performed in C and no longer introduces significant overhead.

  This change has two impacts on the cx_Oracle backend. One is that string values in Py2K which aren't specifically requested with the Unicode type or convert_unicode=True will now come back as `str`, not `unicode` - this behavior is similar to a backend such as MySQL. Additionally, when unicode values are requested with the cx_Oracle backend, if the C extensions are *not* used, there is now an additional overhead of an isinstance() check per column. This tradeoff has been made as it can be worked around and no longer places a performance burden on the likely majority of Oracle result columns that are non-unicode strings. ¶ References: #2911

misc

- **[bug] [examples]** Added a tweak to the "history_meta" example where the check for "history" on a relationship-bound attribute will now no longer emit any SQL if the relationship is unloaded. ¶
- **[bug] [pool]** The argument names for the `PoolEvents.reset()` event have been renamed to `dbapi_connection` and `connection_record` in order to maintain consistency with all the other pool events. It is expected that any existing listeners for this relatively new and seldom-used event are using positional style to receive arguments in any case. ¶

- **[bug] [cextensions] [py3k]** Fixed an issue where the C extensions in Py3K are using the wrong API to specify the top-level module function, which breaks in Python 3.4b2. Py3.4b2 changes PyMODINIT_FUNC to return "void" instead of `PyObject *`, so we now make sure to use "PyMODINIT_FUNC" instead of `PyObject *` directly. Pull request courtesy cgohlke. ¶ References: pull request github:55

0.9.1

Released: January 5, 2014

orm

- **[orm] [feature] [extensions]** A new, **experimental** extension *`sqlalchemy.ext.automap`* is added. This extension expands upon the functionality of Declarative as well as the *`DeferredReflection`* class to produce a base class which automatically generates mapped classes *and relationships* based on table metadata.

  See also:

  Automap Extension

  Automap

  ¶

- **[orm] [bug] [events]** Fixed regression where using a `functools.partial()` with the event system would cause a recursion overflow due to usage of inspect.getargspec() on it in order to detect a legacy calling signature for certain events, and apparently there's no way to do this with a partial object. Instead we skip the legacy check and assume the modern style; the check itself now only occurs for the SessionEvents.after_bulk_update and SessionEvents.after_bulk_delete events. Those two events will require the new signature style if assigned to a "partial" event listener. ¶ References: #2905

- **[orm] [bug]** Fixed bug where using new *`Session.info`* attribute would fail if the `.info` argument were only passed to the *`sessionmaker`* creation call but not to the object itself. Courtesy Robin Schoonover. ¶ References: pull request bitbucket:9

- **[orm] [bug]** Fixed regression where we don't check the given name against the correct string class when setting up a backref based on a name, therefore causing the error "too many values to unpack". This was related to the Py3k conversion. ¶ References: #2901

- **[orm] [bug]** Fixed regression where we apparently still create an implicit alias when saying query(B).join(B.cs), where "C" is a joined inh class; however, this implicit alias was created only considering the immediate left side, and not a longer chain of joins along different joined-inh subclasses of the same base. As long as we're still implicitly aliasing in this case, the behavior is dialed back a bit so that it will alias the right side in a wider variety of cases. ¶ References: #2903

orm declarative

- **[bug] [orm] [declarative]** Fixed an extremely unlikely memory issue where when using *`DeferredReflection`* to define classes pending for reflection, if some subset of those classes were discarded before the *`DeferredReflection.prepare()`* method were called to reflect and map the class, a strong reference to the class would remain held within the declarative internals. This internal collection of "classes to map" now uses weak references against the classes themselves. ¶

- **[bug] [orm] [declarative]** A quasi-regression where apparently in 0.8 you can set a class-level attribute on declarative to simply refer directly to an *`InstrumentedAttribute`* on a superclass or on the class itself, and it acts more or less like a synonym; in 0.9, this fails to set up enough bookkeeping to keep up with the more liberalized backref logic from #2789. Even though this use case was never directly considered, it is now

detected by declarative at the "setattr()" level as well as when setting up a subclass, and the mirrored/renamed attribute is now set up as a `synonym()` instead. ¶ References: #2900

sql

- **[sql] [feature]** Conjunctions like `and_()` and `or_()` can now accept Python generators as a single argument, e.g.:

```
and_(x == y for x, y in tuples)
```

  The logic here looks for a single argument `*args` where the first element is an instance of `types.GeneratorType`. ¶

schema

- **[schema] [feature]** The `Table.extend_existing` and `Table.autoload_replace` parameters are now available on the `MetaData.reflect()` method. ¶

0.9.0

Released: December 30, 2013

orm

- **[orm] [feature]** The `exc.StatementError` or DBAPI-related subclass now can accommodate additional information about the "reason" for the exception; the `Session` now adds some detail to it when the exception occurs within an autoflush. This approach is taken as opposed to combining `FlushError` with a Python 3 style "chained exception" approach so as to maintain compatibility both with Py2K code as well as code that already catches `IntegrityError` or similar. ¶
- **[orm] [feature] [backrefs]** Added new argument `include_backrefs=True` to the `validates()` function; when set to False, a validation event will not be triggered if the event was initated as a backref to an attribute operation from the other side.

  See also:

  include_backrefs=False option for @validates

  ¶ References: #1535
- **[orm] [feature]** A new API for specifying the `FOR UPDATE` clause of a `SELECT` is added with the new `Query.with_for_update()` method, to complement the new `GenerativeSelect.with_for_update()` method. Pull request courtesy Mario Lassnig.

  See also:

  New FOR UPDATE support on select(), Query()

  ¶ References: pull request github:42
- **[orm] [bug]** An adjustment to the `subqueryload()` strategy which ensures that the query runs after the loading process has begun; this is so that the subqueryload takes precedence over other loaders that may be hitting the same attribute due to other eager/noload situations at the wrong time. ¶ This change is also **backported** to: 0.8.5

References: #2887

- **[orm] [bug]** Fixed bug when using joined table inheritance from a table to a select/alias on the base, where the PK columns were also not same named; the persistence system would fail to copy primary key values from the base table to the inherited table upon INSERT. ¶ This change is also **backported** to: 0.8.5

  References: #2885

- **[orm] [bug]** `composite()` will raise an informative error message when the columns/attribute (names) passed don't resolve to a Column or mapped attribute (such as an erroneous tuple); previously raised an unbound local. ¶ This change is also **backported** to: 0.8.5

  References: #2889

- **[orm] [bug]** Fixed a regression introduced by #2818 where the EXISTS query being generated would produce a "columns being replaced" warning for a statement with two same-named columns, as the internal SELECT wouldn't have use_labels set. ¶ This change is also **backported** to: 0.8.4

  References: #2818

- **[orm] [bug] [collections] [py3k]** Added support for the Python 3 method `list.clear()` within the ORM collection instrumentation system; pull request courtesy Eduardo Schettino. ¶ References: pull request github:40

- **[orm] [bug]** Some refinements to the `AliasedClass` construct with regards to descriptors, like hybrids, synonyms, composites, user-defined descriptors, etc. The attribute adaptation which goes on has been made more robust, such that if a descriptor returns another instrumented attribute, rather than a compound SQL expression element, the operation will still proceed. Addtionally, the "adapted" operator will retain its class; previously, a change in class from `InstrumentedAttribute` to `QueryableAttribute` (a superclass) would interact with Python's operator system such that an expression like `aliased(MyClass.x) > MyClass.x` would reverse itself to read `myclass.x < myclass_1.x`. The adapted attribute will also refer to the new `AliasedClass` as its parent which was not always the case before. ¶ References: #2872

- **[orm] [bug]** The `viewonly` flag on `relationship()` will now prevent attribute history from being written on behalf of the target attribute. This has the effect of the object not being written to the Session.dirty list if it is mutated. Previously, the object would be present in Session.dirty, but no change would take place on behalf of the modified attribute during flush. The attribute still emits events such as backref events and user-defined events and will still receive mutations from backrefs.

  See also:

  viewonly=True on relationship() prevents history from taking effect

  ¶ References: #2833

- **[orm] [bug]** Added support for new `Session.info` attribute to `scoped_session`. ¶

- **[orm] [bug]** Fixed bug where usage of new `Bundle` object would cause the `Query.column_descriptions` attribute to fail. ¶

- **[orm] [bug] [sqlite] [sql]** Fixed a regression introduced by the join rewriting feature of #2369 and #2587 where a nested join with one side already an aliased select would fail to translate the ON clause on the outside correctly; in the ORM this could be seen when using a SELECT statement as a "secondary" table. ¶ References: #2858

orm declarative

- **[bug] [orm] [declarative]** Declarative does an extra check to detect if the same `Column` is mapped multiple times under different properties (which typically should be a `synonym()` instead) or if two or more `Column` objects are given the same name, raising a warning if this condition is detected. ¶ References: #2828

- **[bug] [orm] [declarative]** The `DeferredReflection` class has been enhanced to provide automatic reflection support for the "secondary" table referred to by a `relationship()`. "secondary", when specified either as a string table name, or as a `Table` object with only a name and `MetaData` object will also be included in the reflection process when `DeferredReflection.prepare()` is called. ¶ References: #2865
- **[bug] [orm] [declarative]** Fixed bug where in Py2K a unicode literal would not be accepted as the string name of a class or other argument within declarative using `relationship()`. ¶

engine

- **[engine] [feature]** The `engine_from_config()` function has been improved so that we will be able to parse dialect-specific arguments from string configuration dictionaries. Dialect classes can now provide their own list of parameter types and string-conversion routines. The feature is not yet used by the built-in dialects, however. ¶ References: #2875
- **[engine] [bug]** A DBAPI that raises an error on `connect()` which is not a subclass of dbapi.Error (such as `TypeError`, `NotImplementedError`, etc.) will propagate the exception unchanged. Previously, the error handling specific to the `connect()` routine would both inappropriately run the exception through the dialect's `Dialect.is_disconnect()` routine as well as wrap it in a `sqlalchemy.exc.DBAPIError`. It is now propagated unchanged in the same way as occurs within the execute process. ¶ This change is also **backported** to: 0.8.4

  References: #2881
- **[engine] [bug] [pool]** The `QueuePool` has been enhanced to not block new connection attempts when an existing connection attempt is blocking. Previously, the production of new connections was serialized within the block that monitored overflow; the overflow counter is now altered within its own critical section outside of the connection process itself. ¶ This change is also **backported** to: 0.8.4

  References: #2880
- **[engine] [bug] [pool]** Made a slight adjustment to the logic which waits for a pooled connection to be available, such that for a connection pool with no timeout specified, it will every half a second break out of the wait to check for the so-called "abort" flag, which allows the waiter to break out in case the whole connection pool was dumped; normally the waiter should break out due to a notify_all() but it's possible this notify_all() is missed in very slim cases. This is an extension of logic first introduced in 0.8.0, and the issue has only been observed occasionally in stress tests. ¶ This change is also **backported** to: 0.8.4

  References: #2522
- **[engine] [bug]** Fixed bug where SQL statement would be improperly ASCII-encoded when a pre-DBAPI `StatementError` were raised within `Connection.execute()`, causing encoding errors for non-ASCII statements. The stringification now remains within Python unicode thus avoiding encoding errors. ¶ This change is also **backported** to: 0.8.4

  References: #2871
- **[engine] [bug]** The `create_engine()` routine and the related `make_url()` function no longer considers the + sign to be a space within the password field. The parsing has been adjuted to match RFC 1738 exactly, in that both `username` and `password` expect only `:`, `@`, and `/` to be encoded.

  See also:

  The "password" portion of a create_engine() no longer considers the + sign as an encoded space

  ¶ References: #2873
- **[engine] [bug]** The `RowProxy` object is now sortable in Python as a regular tuple is; this is accomplished via ensuring tuple() conversion on both sides within the `__eq__()` method as well as the addition of a `__lt__()` method.

See also:

RowProxy now has tuple-sorting behavior

¶ References: #2848

sql

- **[sql] [feature]** New improvements to the `text()` construct, including more flexible ways to set up bound parameters and return types; in particular, a `text()` can now be turned into a full FROM-object, embeddable in other statements as an alias or CTE using the new method `TextClause.columns()`. The `text()` construct can also render "inline" bound parameters when the construct is compiled in a "literal bound" context.

  See also:

  New text() Capabilities

  ¶ References: #2882, #2877

- **[sql] [feature]** A new API for specifying the `FOR UPDATE` clause of a `SELECT` is added with the new `GenerativeSelect.with_for_update()` method. This method supports a more straightforward system of setting dialect-specific options compared to the `for_update` keyword argument of `select()`, and also includes support for the SQL standard `FOR UPDATE OF` clause. The ORM also includes a new corresponding method `Query.with_for_update()`. Pull request courtesy Mario Lassnig.

  See also:

  New FOR UPDATE support on select(), Query()

  ¶ References: pull request github:42

- **[sql] [feature]** The precision used when coercing a returned floating point value to Python `Decimal` via string is now configurable. The flag `decimal_return_scale` is now supported by all `Numeric` and `Float` types, which will ensure this many digits are taken from the native floating point value when it is converted to string. If not present, the type will make use of the value of `.scale`, if the type supports this setting and it is non-None. Otherwise the original default length of 10 is used.

  See also:

  Floating Point String-Conversion Precision Configurable for Native Floating Point Types

  ¶ References: #2867

- **[sql] [bug]** Fixed issue where a primary key column that has a Sequence on it, yet the column is not the "auto increment" column, either because it has a foreign key constraint or `autoincrement=False` set, would attempt to fire the Sequence on INSERT for backends that don't support sequences, when presented with an INSERT missing the primary key value. This would take place on non-sequence backends like SQLite, MySQL. ¶ This change is also **backported** to: 0.8.5

  References: #2896

- **[sql] [bug]** Fixed bug with `Insert.from_select()` method where the order of the given names would not be taken into account when generating the INSERT statement, thus producing a mismatch versus the column names in the given SELECT statement. Also noted that `Insert.from_select()` implies that Python-side insert defaults cannot be used, since the statement has no VALUES clause. ¶ This change is also **backported** to: 0.8.5

  References: #2895

- **[sql] [bug]** The `cast()` function, when given a plain literal value, will now apply the given type to the given literal value on the bind parameter side according to the type given to the cast, in the same manner as that of the

`type_coerce()` function. However unlike `type_coerce()`, this only takes effect if a non-clauseelement value is passed to `cast()`; an existing typed construct will retain its type. ¶

- **[sql] [bug]** The `ForeignKey` class more aggressively checks the given column argument. If not a string, it checks that the object is at least a `ColumnClause`, or an object that resolves to one, and that the `.table` attribute, if present, refers to a `TableClause` or subclass, and not something like an `Alias`. Otherwise, a `ArgumentError` is raised. ¶ References: #2883

- **[sql] [bug]** The precedence rules for the `ColumnOperators.collate()` operator have been modified, such that the COLLATE operator is now of lower precedence than the comparison operators. This has the effect that a COLLATE applied to a comparison will not render parenthesis around the comparison, which is not parsed by backends such as MSSQL. The change is backwards incompatible for those setups that were working around the issue by applying `Operators.collate()` to an individual element of the comparison expression, rather than the comparison expression as a whole.

  See also:

  The precedence rules for COLLATE have been changed

  ¶ References: #2879

- **[sql] [enhancement]** The exception raised when a `BindParameter` is present in a compiled statement without a value now includes the key name of the bound parameter in the error message. ¶ This change is also **backported** to: 0.8.5

schema

- **[schema] [bug]** Fixed a regression caused by #2812 where the repr() for table and column names would fail if the name contained non-ascii characters. ¶ References: #2868

postgresql

- **[postgresql] [feature]** Support for Postgresql JSON has been added, using the new `JSON` type. Huge thanks to Nathan Rice for implementing and testing this. ¶ References: #2581, pull request github:50

- **[postgresql] [feature]** Added support for Postgresql TSVECTOR via the `postgresql.TSVECTOR` type. Pull request courtesy Noufal Ibrahim. ¶ References: pull request bitbucket:8

- **[postgresql] [bug]** Fixed bug where index reflection would mis-interpret indkey values when using the pypostgresql adapter, which returns these values as lists vs. psycopg2's return type of string. ¶ This change is also **backported** to: 0.8.4

  References: #2855

- **[postgresql] [bug]** Now using psycopg2 UNICODEARRAY extension for handling unicode arrays with psycopg2 + normal "native unicode" mode, in the same way the UNICODE extension is used. ¶

- **[postgresql] [bug]** Fixed bug where values within an ENUM weren't escaped for single quote signs. Note that this is backwards-incompatible for existing workarounds that manually escape the single quotes.

  See also:

  Postgresql CREATE TYPE <x> AS ENUM now applies quoting to values

  ¶ References: #2878

mysql

- **[mysql] [bug]** Improvements to the system by which SQL types generate within `__repr__()`, particularly with regards to the MySQL integer/numeric/ character types which feature a wide variety of keyword arguments. The `__repr__()` is important for use with Alembic autogenerate for when Python code is rendered in a migration script. ¶ References: #2893

mssql

- **[mssql] [bug] [firebird]** The "asdecimal" flag used with the `Float` type will now work with Firebird as well as the mssql+pyodbc dialects; previously the decimal conversion was not occurring. ¶ This change is also **backported** to: 0.8.5
- **[mssql] [bug] [pymssql]** Added "Net-Lib error during Connection reset by peer" message to the list of messages checked for "disconnect" within the pymssql dialect. Courtesy John Anderson. ¶ This change is also **backported** to: 0.8.5

  References: pull request github:51
- **[mssql] [bug]** Fixed bug introduced in 0.8.0 where the `DROP INDEX` statement for an index in MSSQL would render incorrectly if the index were in an alternate schema; the schemaname/tablename would be reversed. The format has been also been revised to match current MSSQL documentation. Courtesy Derek Harland. ¶ This change is also **backported** to: 0.8.4

  References: pull request bitbucket:7

oracle

- **[oracle] [bug]** Added ORA-02396 "maximum idle time" error code to list of "is disconnect" codes with cx_oracle. ¶ This change is also **backported** to: 0.8.4

  References: #2864
- **[oracle] [bug]** Fixed bug where Oracle `VARCHAR` types given with no length (e.g. for a `CAST` or similar) would incorrectly render `None CHAR` or similar. ¶ This change is also **backported** to: 0.8.4

  References: #2870

firebird

- **[firebird] [bug]** The firebird dialect will quote identifiers which begin with an underscore. Courtesy Treeve Jelbert. ¶ This change is also **backported** to: 0.8.5

  References: #2897
- **[firebird] [bug]** Fixed bug in Firebird index reflection where the columns within the index were not sorted correctly; they are now sorted in order of RDB$FIELD_POSITION. ¶ This change is also **backported** to: 0.8.5
- **[firebird] [bug]** Changed the queries used by Firebird to list table and view names to query from the `rdb$relations` view instead of the `rdb$relation_fields` and `rdb$view_relations` views. Variants of both the old and new queries are mentioned on many FAQ and blogs, however the new queries are taken straight from the "Firebird FAQ" which appears to be the most official source of info. ¶ References: #2898

misc

- **[removed]** The "informix" and "informixdb" dialects have been removed; the code is now available as a separate repository on Bitbucket. The IBM-DB project has provided production-level Informix support since the informixdb dialect was first added. ¶
- **[bug] [declarative]** Error message when a string arg sent to `relationship()` which doesn't resolve to a class or mapper has been corrected to work the same way as when a non-string arg is received, which indicates the name of the relationship which had the configurational error. ¶ This change is also **backported** to: 0.8.5

  References: #2888
- **[bug] [ext]** Fixed bug which prevented the `serializer` extension from working correctly with table or column names that contain non-ASCII characters. ¶ This change is also **backported** to: 0.8.4

  References: #2869
- **[bug] [examples]** Fixed bug which prevented history_meta recipe from working with joined inheritance schemes more than one level deep. ¶

0.9.0b1

Released: October 26, 2013

general

- **[general] [feature] [py3k]** The C extensions are ported to Python 3 and will build under any supported CPython 2 or 3 environment. ¶ References: #2161
- **[general] [feature] [py3k]** The codebase is now "in-place" for Python 2 and 3, the need to run 2to3 has been removed. Compatibility is now against Python 2.6 on forward. ¶ References: #2671
- **[general]** A large refactoring of packages has reorganized the import structure of many Core modules as well as some aspects of the ORM modules. In particular `sqlalchemy.sql` has been broken out into several more modules than before so that the very large size of `sqlalchemy.sql.expression` is now pared down. The effort has focused on a large reduction in import cycles. Additionally, the system of API functions in `sqlalchemy.sql.expression` and `sqlalchemy.orm` has been reorganized to eliminate redundancy in documentation between the functions vs. the objects they produce. ¶

orm

- **[orm] [feature]** Added new option to `relationship()` `distinct_target_key`. This enables the subquery eager loader strategy to apply a DISTINCT to the innermost SELECT subquery, to assist in the case where duplicate rows are generated by the innermost query which corresponds to this relationship (there's not yet a general solution to the issue of dupe rows within subquery eager loading, however, when joins outside of the innermost subquery produce dupes). When the flag is set to `True`, the DISTINCT is rendered unconditionally, and when it is set to `None`, DISTINCT is rendered if the innermost relationship targets columns that do not comprise a full primary key. The option defaults to False in 0.8 (e.g. off by default in all cases), None in 0.9 (e.g. automatic by default). Thanks to Alexander Koval for help with this.

  See also:

  Subquery Eager Loading will apply DISTINCT to the innermost SELECT for some queries

  ¶ This change is also **backported** to: 0.8.3

  References: #2836

- **[orm] [feature]** The association proxy now returns `None` when fetching a scalar attribute off of a scalar relationship, where the scalar relationship itself points to `None`, instead of raising an `AttributeError`.

  See also:

  Association Proxy Missing Scalar returns None

  ¶ References: #2810

- **[orm] [feature]** Added new method `AttributeState.load_history()`, works like `AttributeState.history` but also fires loader callables.

  See also:

  attributes.get_history() will query from the DB by default if value not present

  ¶ References: #2787

- **[orm] [feature]** Added a new load option `orm.load_only()`. This allows a series of column names to be specified as loading "only" those attributes, deferring the rest. ¶ References: #1418

- **[orm] [feature]** The system of loader options has been entirely rearchitected to build upon a much more comprehensive base, the `Load` object. This base allows any common loader option like `joinedload()`, `defer()`, etc. to be used in a "chained" style for the purpose of specifying options down a path, such as `joinedload("foo").subqueryload("bar")`. The new system supersedes the usage of dot-separated path names, multiple attributes within options, and the usage of `_all()` options.

  See also:

  New Query Options API; load_only() option

  ¶ References: #1418

- **[orm] [feature]** The `composite()` construct now maintains the return object when used in a column-oriented `Query`, rather than expanding out into individual columns. This makes use of the new `Bundle` feature internally. This behavior is backwards incompatible; to select from a composite column which will expand out, use `MyClass.some_composite.clauses`.

  See also:

  Composite attributes are now returned as their object form when queried on a per-attribute basis

  ¶ References: #2824

- **[orm] [feature]** A new construct `Bundle` is added, which allows for specification of groups of column expressions to a `Query` construct. The group of columns are returned as a single tuple by default. The behavior of `Bundle` can be overridden however to provide any sort of result processing to the returned row. The behavior of `Bundle` is also embedded into composite attributes now when they are used in a column-oriented `Query`.

  See also:

  Column Bundles for ORM queries

  Composite attributes are now returned as their object form when queried on a per-attribute basis

  ¶ References: #2824

- **[orm] [feature]** The `version_id_generator` parameter of `Mapper` can now be specified to rely upon server generated version identifiers, using triggers or other database-provided versioning features, or via an optional programmatic value, by setting `version_id_generator=False`. When using a server-generated version identfier, the ORM will use RETURNING when available to immediately load the new version value, else it will emit a second SELECT. ¶ References: #2793

- **[orm] [feature]** The `eager_defaults` flag of *Mapper* will now allow the newly generated default values to be fetched using an inline RETURNING clause, rather than a second SELECT statement, for backends that support RETURNING. ¶ References: #2793
- **[orm] [feature]** Added a new attribute *Session.info* to *Session*; this is a dictionary where applications can store arbitrary data local to a *Session*. The contents of *Session.info* can be also be initialized using the `info` argument of *Session* or *sessionmaker*. ¶
- **[orm] [feature]** Removal of event listeners is now implemented. The feature is provided via the *event.remove()* function.

  See also:

  Event Removal API

  ¶ References: #2268
- **[orm] [feature]** The mechanism by which attribute events pass along an `AttributeImpl` as an "initiator" token has been changed; the object is now an event-specific object called *attributes.Event*. Additionally, the attribute system no longer halts events based on a matching "initiator" token; this logic has been moved to be specific to ORM backref event handlers, which are the typical source of the re-propagation of an attribute event onto subsequent append/set/remove operations. End user code which emulates the behavior of backrefs must now ensure that recursive event propagation schemes are halted, if the scheme does not use the backref handlers. Using this new system, backref handlers can now perform a "two-hop" operation when an object is appended to a collection, associated with a new many-to-one, de-associated with the previous many-to-one, and then removed from a previous collection. Before this change, the last step of removal from the previous collection would not occur.

  See also:

  Backref handlers can now propagate more than one level deep

  ¶ References: #2789
- **[orm] [feature]** A major change regarding how the ORM constructs joins where the right side is itself a join or left outer join. The ORM is now configured to allow simple nesting of joins of the form `a JOIN (b JOIN c ON b.id=c.id) ON a.id=b.id`, rather than forcing the right side into a `SELECT` subquery. This should allow significant performance improvements on most backends, most particularly MySQL. The one database backend that has for many years held back this change, SQLite, is now addressed by moving the production of the `SELECT` subquery from the ORM to the SQL compiler; so that a right-nested join on SQLite will still ultimately render with a `SELECT`, while all other backends are no longer impacted by this workaround.

  As part of this change, a new argument `flat=True` has been added to the *orm.aliased()*, *Join.alias()*, and *orm.with_polymorphic()* functions, which allows an "alias" of a JOIN to be produced which applies an anonymous alias to each component table within the join, rather than producing a subquery.

  See also:

  *Many JOIN and LEFT OUTER JOIN expressions will no longer be wrapped in (SELECT * FROM ..) AS ANON_1*

  ¶ References: #2587
- **[orm] [bug]** Fixed bug where using an annotation such as *remote()* or *foreign()* on a *Column* before association with a parent *Table* could produce issues related to the parent table not rendering within joins, due to the inherent copy operation performed by an annotation. ¶ This change is also **backported** to: 0.8.3

  References: #2813
- **[orm] [bug]** Fixed bug where *Query.exists()* failed to work correctly without any WHERE criterion. Courtesy Vladimir Magamedov. ¶ This change is also **backported** to: 0.8.3

References: #2818

- **[orm] [bug]** Fixed a potential issue in an ordered sequence implementation used by the ORM to iterate mapper hierarchies; under the Jython interpreter this implementation wasn't ordered, even though cPython and Pypy maintained ordering. ¶ This change is also **backported** to: 0.8.3

  References: #2794

- **[orm] [bug]** Fixed bug in ORM-level event registration where the "raw" or "propagate" flags could potentially be mis-configured in some "unmapped base class" configurations. ¶ This change is also **backported** to: 0.8.3

  References: #2786

- **[orm] [bug]** A performance fix related to the usage of the `defer()` option when loading mapped entities. The function overhead of applying a per-object deferred callable to an instance at load time was significantly higher than that of just loading the data from the row (note that `defer()` is meant to reduce DB/network overhead, not necessarily function call count); the function call overhead is now less than that of loading data from the column in all cases. There is also a reduction in the number of "lazy callable" objects created per load from N (total deferred values in the result) to 1 (total number of deferred cols). ¶ This change is also **backported** to: 0.8.3

  References: #2778

- **[orm] [bug]** Fixed bug whereby attribute history functions would fail when an object we moved from "persistent" to "pending" using the `make_transient()` function, for operations involving collection-based backrefs. ¶ This change is also **backported** to: 0.8.3

  References: #2773

- **[orm] [bug]** A warning is emitted when trying to flush an object of an inherited class where the polymorphic discriminator has been assigned to a value that is invalid for the class. ¶ This change is also **backported** to: 0.8.2

  References: #2750

- **[orm] [bug]** Fixed bug in polymorphic SQL generation where multiple joined-inheritance entities against the same base class joined to each other as well would not track columns on the base table independently of each other if the string of joins were more than two entities long. ¶ This change is also **backported** to: 0.8.2

  References: #2759

- **[orm] [bug]** Fixed bug where sending a composite attribute into `Query.order_by()` would produce a parenthesized expression not accepted by some databases. ¶ This change is also **backported** to: 0.8.2

  References: #2754

- **[orm] [bug]** Fixed the interaction between composite attributes and the `aliased()` function. Previously, composite attributes wouldn't work correctly in comparison operations when aliasing was applied. ¶ This change is also **backported** to: 0.8.2

  References: #2755

- **[orm] [bug] [ext]** Fixed bug where `MutableDict` didn't report a change event when `clear()` was called. ¶ This change is also **backported** to: 0.8.2

  References: #2730

- **[orm] [bug]** Fixed bug where list instrumentation would fail to represent a setslice of `[0:0]` correctly, which in particular could occur when using `insert(0, item)` with the association proxy. Due to some quirk in Python collections, the issue was much more likely with Python 3 rather than 2. ¶ This change is also **backported** to: 0.8.3, 0.7.11

  References: #2807

- **[orm] [bug]** `attributes.get_history()` when used with a scalar column-mapped attribute will now honor the "passive" flag passed to it; as this defaults to `PASSIVE_OFF`, the function will by default query the database if the value is not present. This is a behavioral change vs. 0.8.

  See also:

  attributes.get_history() will query from the DB by default if value not present

  ¶ References: #2787

- **[orm] [bug] [associationproxy]** Added additional criterion to the ==, != comparators, used with scalar values, for comparisons to None to also take into account the association record itself being non-present, in addition to the existing test for the scalar endpoint on the association record being NULL. Previously, comparing `Cls.scalar == None` would return records for which `Cls.associated` were present and `Cls.associated.scalar` is None, but not rows for which `Cls.associated` is non-present. More significantly, the inverse operation `Cls.scalar != None` *would* return `Cls` rows for which `Cls.associated` was non-present.

  The case for `Cls.scalar != 'somevalue'` is also modified to act more like a direct SQL comparison; only rows for which `Cls.associated` is present and `Associated.scalar` is non-NULL and not equal to `'somevalue'` are returned. Previously, this would be a simple `NOT EXISTS`.

  Also added a special use case where you can call `Cls.scalar.has()` with no arguments, when `Cls.scalar` is a column-based value - this returns whether or not `Cls.associated` has any rows present, regardless of whether or not `Cls.associated.scalar` is NULL or not.

  See also:

  Association Proxy SQL Expression Improvements and Fixes

  ¶ References: #2751

- **[orm] [bug]** Fixed an obscure bug where the wrong results would be fetched when joining/joinedloading across a many-to-many relationship to a single-table-inheriting subclass with a specific discriminator value, due to "secondary" rows that would come back. The "secondary" and right-side tables are now inner joined inside of parenthesis for all ORM joins on many-to-many relationships so that the left->right join can accurately filtered. This change was made possible by finally addressing the issue with right-nested joins outlined in #2587.

  See also:

  *Many JOIN and LEFT OUTER JOIN expressions will no longer be wrapped in (SELECT * FROM ..) AS ANON_1*

  ¶ References: #2369

- **[orm] [bug]** The "auto-aliasing" behavior of the `Query.select_from()` method has been turned off. The specific behavior is now available via a new method `Query.select_entity_from()`. The auto-aliasing behavior here was never well documented and is generally not what's desired, as `Query.select_from()` has become more oriented towards controlling how a JOIN is rendered. `Query.select_entity_from()` will also be made available in 0.8 so that applications which rely on the auto-aliasing can shift their applications to use this method.

  See also:

  Query.select_from() no longer applies the clause to corresponding entities

  ¶ References: #2736

orm declarative

- **[feature] [orm] [declarative]** Added a convenience class decorator `as_declarative()`, is a wrapper for `declarative_base()` which allows an existing base class to be applied using a nifty class-decorated ap-

proach. ¶ This change is also **backported** to: 0.8.3

- **[feature] [orm] [declarative]** ORM descriptors such as hybrid properties can now be referenced by name in a string argument used with `order_by`, `primaryjoin`, or similar in *relationship()*, in addition to column-bound attributes. ¶ This change is also **backported** to: 0.8.2

  References: #2761

engine

- **[engine] [feature]** `repr()` for the *URL* of an *Engine* will now conceal the password using asterisks. Courtesy Gunnlaugur Þór Briem. ¶ This change is also **backported** to: 0.8.3

  References: #2821

- **[engine] [feature]** New events added to *ConnectionEvents*:
  - *ConnectionEvents.engine_connect()*
  - *ConnectionEvents.set_connection_execution_options()*
  - *ConnectionEvents.set_engine_execution_options()*

  ¶ References: #2770

- **[engine] [bug] [oracle]** Dialect.initialize() is not called a second time if an *Engine* is recreated, due to a disconnect error. This fixes a particular issue in the Oracle 8 dialect, but in general the dialect.initialize() phase should only be once per dialect. ¶ This change is also **backported** to: 0.8.3

  References: #2776

- **[engine] [bug] [pool]** Fixed bug where *QueuePool* would lose the correct checked out count if an existing pooled connection failed to reconnect after an invalidate or recycle event. ¶ This change is also **backported** to: 0.8.3

  References: #2772

- **[engine] [bug]** Fixed bug where the `reset_on_return` argument to various *Pool* implementations would not be propagated when the pool was regenerated. Courtesy Eevee. ¶ This change is also **backported** to: 0.8.2

  References: pull request github:6

- **[engine] [bug]** The regexp used by the *make_url()* function now parses ipv6 addresses, e.g. surrounded by brackets. ¶ This change is also **backported** to: 0.8.3, 0.7.11

  References: #2851

- **[engine] [bug]** The method signature of *Dialect.reflecttable()*, which in all known cases is provided by *DefaultDialect*, has been tightened to expect `include_columns` and `exclude_columns` arguments without any kw option, reducing ambiguity - previously `exclude_columns` was missing. ¶ References: #2748

sql

- **[sql] [feature]** Added support for "unique constraint" reflection, via the *Inspector.get_unique_constraints()* method. Thanks for Roman Podolyaka for the patch. ¶ This change is also **backported** to: 0.8.4

  References: #1443

- **[sql] [feature]** The *update()*, *insert()*, and *delete()* constructs will now interpret ORM entities as target tables to be operated upon, e.g.:

```
from sqlalchemy import insert, update, delete

ins = insert(SomeMappedClass).values(x=5)

del_ = delete(SomeMappedClass).where(SomeMappedClass.id == 5)

upd = update(SomeMappedClass).where(SomeMappedClass.id == 5).values(name='ed')
```

  ¶ This change is also **backported** to: 0.8.3

- **[sql] [feature] [postgresql] [mysql]** The Postgresql and MySQL dialects now support reflection/inspection of foreign key options, including ON UPDATE, ON DELETE. Postgresql also reflects MATCH, DEFERRABLE, and INITIALLY. Coutesy ijl. ¶ References: #2183

- **[sql] [feature]** A *bindparam()* construct with a "null" type (e.g. no type specified) is now copied when used in a typed expression, and the new copy is assigned the actual type of the compared column. Previously, this logic would occur on the given *bindparam()* in place. Additionally, a similar process now occurs for *bindparam()* constructs passed to *ValuesBase.values()* for an *Insert* or *Update* construct, within the compilation phase of the construct.

  These are both subtle behavioral changes which may impact some usages.

  See also:

  A bindparam() construct with no type gets upgraded via copy when a type is available

  ¶ References: #2850

- **[sql] [feature]** An overhaul of expression handling for special symbols particularly with conjunctions, e.g. `None` *expression.null()* *expression.true()* *expression.false()*, including consistency in rendering NULL in conjunctions, "short-circuiting" of *and_()* and *or_()* expressions which contain boolean constants, and rendering of boolean constants and expressions as compared to "1" or "0" for backends that don't feature `true`/`false` constants.

  See also:

  Improved rendering of Boolean constants, NULL constants, conjunctions

  ¶ References: #2734, #2804, #2823

- **[sql] [feature]** The typing system now handles the task of rendering "literal bind" values, e.g. values that are normally bound parameters but due to context must be rendered as strings, typically within DDL constructs such as CHECK constraints and indexes (note that "literal bind" values become used by DDL as of #2742). A new method *TypeEngine.literal_processor()* serves as the base, and *TypeDecorator.process_literal_param()* is added to allow wrapping of a native literal rendering method.

  See also:

  The typing system now handles the task of rendering "literal bind" values

  ¶ References: #2838

- **[sql] [feature]** The *Table.tometadata()* method now produces copies of all *SchemaItem.info* dictionaries from all *SchemaItem* objects within the structure including columns, constraints, foreign keys, etc. As these dictionaries are copies, they are independent of the original dictionary. Previously, only the `.info` dictionary of *Column* was transferred within this operation, and it was only linked in place, not copied. ¶ References: #2716

- **[sql] [feature]** The `default` argument of *Column* now accepts a class or object method as an argument, in addition to a standalone function; will properly detect if the "context" argument is accepted or not. ¶

- **[sql] [feature]** Added new method to the *insert()* construct *Insert.from_select()*. Given a list of columns and a selectable, renders `INSERT INTO (table) (columns) SELECT ...`. While this feature is highlighted as part of 0.9 it is also backported to 0.8.3.

  See also:

  INSERT from SELECT

  ¶ References: #722

- **[sql] [feature]** Provided a new attribute for *TypeDecorator* called *TypeDecorator.coerce_to_is_types*, to make it easier to control how comparisons using `==` or `!=` to `None` and boolean types goes about producing an `IS` expression, or a plain equality expression with a bound parameter. ¶ References: #2734, #2744

- **[sql] [feature]** A *label()* construct will now render as its name alone in an `ORDER BY` clause, if that label is also referred to in the columns clause of the select, instead of rewriting the full expression. This gives the database a better chance to optimize the evaluation of the same expression in two different contexts.

  See also:

  Label constructs can now render as their name alone in an ORDER BY

  ¶ References: #1068

- **[sql] [bug]** Fixed bug where *type_coerce()* would not interpret ORM elements with a `__clause_element__()` method properly. ¶ This change is also **backported** to: 0.8.3

  References: #2849

- **[sql] [bug]** The *Enum* and *Boolean* types now bypass any custom (e.g. TypeDecorator) type in use when producing the CHECK constraint for the "non native" type. This so that the custom type isn't involved in the expression within the CHECK, since this expression is against the "impl" value and not the "decorated" value. ¶ This change is also **backported** to: 0.8.3

  References: #2842

- **[sql] [bug]** The `.unique` flag on *Index* could be produced as `None` if it was generated from a *Column* that didn't specify `unique` (where it defaults to `None`). The flag will now always be `True` or `False`. ¶ This change is also **backported** to: 0.8.3

  References: #2825

- **[sql] [bug]** Fixed bug in default compiler plus those of postgresql, mysql, and mssql to ensure that any literal SQL expression values are rendered directly as literals, instead of as bound parameters, within a CREATE INDEX statement. This also changes the rendering scheme for other DDL such as constraints. ¶ This change is also **backported** to: 0.8.3

  References: #2742

- **[sql] [bug]** A *select()* that is made to refer to itself in its FROM clause, typically via in-place mutation, will raise an informative error message rather than causing a recursion overflow. ¶ This change is also **backported** to: 0.8.3

  References: #2815

- **[sql] [bug]** Fixed bug where using the `column_reflect` event to change the `.key` of the incoming *Column* would prevent primary key constraints, indexes, and foreign key constraints from being correctly reflected. ¶ This change is also **backported** to: 0.8.3

  References: #2811

- **[sql] [bug]** The `ColumnOperators.notin_()` operator added in 0.8 now properly produces the negation of the expression "IN" returns when used against an empty collection. ¶ This change is also **backported** to: 0.8.3

- **[sql] [bug] [postgresql]** Fixed bug where the expression system relied upon the `str()` form of a some expressions when referring to the `.c` collection on a `select()` construct, but the `str()` form isn't available since the element relies on dialect-specific compilation constructs, notably the `__getitem__()` operator as used with a Postgresql `ARRAY` element. The fix also adds a new exception class `UnsupportedCompilationError` which is raised in those cases where a compiler is asked to compile something it doesn't know how to. ¶ This change is also **backported** to: 0.8.3

  References: #2780

- **[sql] [bug]** Multiple fixes to the correlation behavior of `Select` constructs, first introduced in 0.8.0:

  - To satisfy the use case where FROM entries should be correlated outwards to a SELECT that encloses another, which then encloses this one, correlation now works across multiple levels when explicit correlation is established via `Select.correlate()`, provided that the target select is somewhere along the chain contained by a WHERE/ORDER BY/columns clause, not just nested FROM clauses. This makes `Select.correlate()` act more compatibly to that of 0.7 again while still maintaining the new "smart" correlation.

  - When explicit correlation is not used, the usual "implicit" correlation limits its behavior to just the immediate enclosing SELECT, to maximize compatibility with 0.7 applications, and also prevents correlation across nested FROMs in this case, maintaining compatibility with 0.8.0/0.8.1.

  - The `Select.correlate_except()` method was not preventing the given FROM clauses from correlation in all cases, and also would cause FROM clauses to be incorrectly omitted entirely (more like what 0.7 would do), this has been fixed.

  - Calling *select.correlate_except(None)* will enter all FROM clauses into correlation as would be expected.

  ¶ This change is also **backported** to: 0.8.2

  References: #2668, #2746

- **[sql] [bug]** Fixed bug whereby joining a select() of a table "A" with multiple foreign key paths to a table "B", to that table "B", would fail to produce the "ambiguous join condition" error that would be reported if you join table "A" directly to "B"; it would instead produce a join condition with multiple criteria. ¶ This change is also **backported** to: 0.8.2

  References: #2738

- **[sql] [bug] [reflection]** Fixed bug whereby using `MetaData.reflect()` across a remote schema as well as a local schema could produce wrong results in the case where both schemas had a table of the same name. ¶ This change is also **backported** to: 0.8.2

  References: #2728

- **[sql] [bug]** Removed the "not implemented" `__iter__()` call from the base `ColumnOperators` class, while this was introduced in 0.8.0 to prevent an endless, memory-growing loop when one also implements a `__getitem__()` method on a custom operator and then calls erroneously `list()` on that object, it had the effect of causing column elements to report that they were in fact iterable types which then throw an error when you try to iterate. There's no real way to have both sides here so we stick with Python best practices. Careful with implementing `__getitem__()` on your custom operators! ¶ This change is also **backported** to: 0.8.2

  References: #2726

- **[sql] [bug]** Fixed regression dating back to 0.7.9 whereby the name of a CTE might not be properly quoted if it was referred to in multiple FROM clauses. ¶ This change is also **backported** to: 0.8.3, 0.7.11

  References: #2801

- **[sql] [bug] [cte]** Fixed bug in common table expression system where if the CTE were used only as an `alias()` construct, it would not render using the WITH keyword. ¶ This change is also **backported** to: 0.8.3, 0.7.11

  References: #2783

- **[sql] [bug]** Fixed bug in `CheckConstraint` DDL where the "quote" flag from a `Column` object would not be propagated. ¶ This change is also **backported** to: 0.8.3, 0.7.11

  References: #2784

- **[sql] [bug]** The "name" attribute is set on `Index` before the "attach" events are called, so that attachment events can be used to dynamically generate a name for the index based on the parent table and/or columns. ¶ References: #2835

- **[sql] [bug]** The erroneous kw arg "schema" has been removed from the `ForeignKey` object. this was an accidental commit that did nothing; a warning is raised in 0.8.3 when this kw arg is used. ¶ References: #2831

- **[sql] [bug]** A rework to the way that "quoted" identifiers are handled, in that instead of relying upon various `quote=True` flags being passed around, these flags are converted into rich string objects with quoting information included at the point at which they are passed to common schema constructs like `Table`, `Column`, etc. This solves the issue of various methods that don't correctly honor the "quote" flag such as `Engine.has_table()` and related methods. The `quoted_name` object is a string subclass that can also be used explicitly if needed; the object will hold onto the quoting preferences passed and will also bypass the "name normalization" performed by dialects that standardize on uppercase symbols, such as Oracle, Firebird and DB2. The upshot is that the "uppercase" backends can now work with force-quoted names, such as lowercase-quoted names and new reserved words.

  See also:

  Schema identifiers now carry along their own quoting information

  ¶ References: #2812

- **[sql] [bug]** The resolution of `ForeignKey` objects to their target `Column` has been reworked to be as immediate as possible, based on the moment that the target `Column` is associated with the same `MetaData` as this `ForeignKey`, rather than waiting for the first time a join is constructed, or similar. This along with other improvements allows earlier detection of some foreign key configuration issues. Also included here is a rework of the type-propagation system, so that it should be reliable now to set the type as `None` on any `Column` that refers to another via `ForeignKey` - the type will be copied from the target column as soon as that other column is associated, and now works for composite foreign keys as well.

  See also:

  Columns can reliably get their type from a column referred to via ForeignKey

  ¶ References: #1765

postgresql

- **[postgresql] [feature]** Support for Postgresql 9.2 range types has been added. Currently, no type translation is provided, so works directly with strings or psycopg2 2.5 range extension types at the moment. Patch courtesy Chris Withers. ¶ This change is also **backported** to: 0.8.2

- **[postgresql] [feature]** Added support for "AUTOCOMMIT" isolation when using the psycopg2 DBAPI. The keyword is available via the `isolation_level` execution option. Patch courtesy Roman Podolyaka. ¶ This change is also **backported** to: 0.8.2

  References: #2072

- **[postgresql] [feature]** Added support for rendering `SMALLSERIAL` when a *SmallInteger* type is used on a primary key autoincrement column, based on server version detection of Postgresql version 9.2 or greater. ¶ References: #2840
- **[postgresql] [bug]** Removed a 128-character truncation from the reflection of the server default for a column; this code was original from PG system views which truncated the string for readability. ¶ This change is also **backported** to: 0.8.3

  References: #2844
- **[postgresql] [bug]** Parenthesis will be applied to a compound SQL expression as rendered in the column list of a CREATE INDEX statement. ¶ This change is also **backported** to: 0.8.3

  References: #2742
- **[postgresql] [bug]** Fixed bug where Postgresql version strings that had a prefix preceding the words "Postgresql" or "EnterpriseDB" would not parse. Courtesy Scott Schaefer. ¶ This change is also **backported** to: 0.8.3

  References: #2819
- **[postgresql] [bug]** The behavior of *extract()* has been simplified on the Postgresql dialect to no longer inject a hardcoded `::timestamp` or similar cast into the given expression, as this interfered with types such as timezone-aware datetimes, but also does not appear to be at all necessary with modern versions of psycopg2. ¶ This change is also **backported** to: 0.8.2

  References: #2740
- **[postgresql] [bug]** Fixed bug in HSTORE type where keys/values that contained backslashed quotes would not be escaped correctly when using the "non native" (i.e. non-psycopg2) means of translating HSTORE data. Patch courtesy Ryan Kelly. ¶ This change is also **backported** to: 0.8.2

  References: #2766
- **[postgresql] [bug]** Fixed bug where the order of columns in a multi-column Postgresql index would be reflected in the wrong order. Courtesy Roman Podolyaka. ¶ This change is also **backported** to: 0.8.2

  References: #2767

mysql

- **[mysql] [feature]** The `mysql_length` parameter used with *Index* can now be passed as a dictionary of column names/lengths, for use with composite indexes. Big thanks to Roman Podolyaka for the patch. ¶ This change is also **backported** to: 0.8.2

  References: #2704
- **[mysql] [feature]** The MySQL *mysql.SET* type now features the same auto-quoting behavior as that of *mysql.ENUM*. Quotes are not required when setting up the value, but quotes that are present will be auto-detected along with a warning. This also helps with Alembic where the SET type doesn't render with quotes. ¶ References: #2817
- **[mysql] [bug]** The change in #2721, which is that the `deferrable` keyword of *ForeignKeyConstraint* is silently ignored on the MySQL backend, will be reverted as of 0.9; this keyword will now render again, raising errors on MySQL as it is not understood - the same behavior will also apply to the `initially` keyword. In 0.8, the keywords will remain ignored but a warning is emitted. Additionally, the `match` keyword now raises a *CompileError* on 0.9 and emits a warning on 0.8; this keyword is not only silently ignored by MySQL but also breaks the ON UPDATE/ON DELETE options.

To use a `ForeignKeyConstraint` that does not render or renders differently on MySQL, use a custom compilation option. An example of this usage has been added to the documentation, see *MySQL Foreign Keys*. ¶ This change is also **backported** to: 0.8.3

References: #2721, #2839

- **[mysql] [bug]** MySQL-connector dialect now allows options in the create_engine query string to override those defaults set up in the connect, including "buffered" and "raise_on_warnings". ¶ This change is also **backported** to: 0.8.3

  References: #2515

- **[mysql] [bug]** Fixed bug when using multi-table UPDATE where a supplemental table is a SELECT with its own bound parameters, where the positioning of the bound parameters would be reversed versus the statement itself when using MySQL's special syntax. ¶ This change is also **backported** to: 0.8.2

  References: #2768

- **[mysql] [bug]** Added another conditional to the `mysql+gaerdbms` dialect to detect so-called "development" mode, where we should use the `rdbms_mysqldb` DBAPI. Patch courtesy Brett Slatkin. ¶ This change is also **backported** to: 0.8.2

  References: #2715

- **[mysql] [bug]** The `deferrable` keyword argument on `ForeignKey` and `ForeignKeyConstraint` will not render the `DEFERRABLE` keyword on the MySQL dialect. For a long time we left this in place because a non-deferrable foreign key would act very differently than a deferrable one, but some environments just disable FKs on MySQL, so we'll be less opinionated here. ¶ This change is also **backported** to: 0.8.2

  References: #2721

- **[mysql] [bug]** Updates to MySQL reserved words for versions 5.5, 5.6, courtesy Hanno Schlichting. ¶ This change is also **backported** to: 0.8.3, 0.7.11

  References: #2791

- **[mysql] [bug]** Fix and test parsing of MySQL foreign key options within reflection; this complements the work in #2183 where we begin to support reflection of foreign key options such as ON UPDATE/ON DELETE cascade. ¶ References: #2839

- **[mysql] [bug]** Improved support for the cymysql driver, supporting version 0.6.5, courtesy Hajime Nakagami. ¶

sqlite

- **[sqlite] [bug]** The newly added SQLite DATETIME arguments storage_format and regexp apparently were not fully implemented correctly; while the arguments were accepted, in practice they would have no effect; this has been fixed. ¶ This change is also **backported** to: 0.8.3

  References: #2781

- **[sqlite] [bug]** Added `sqlalchemy.types.BIGINT` to the list of type names that can be reflected by the SQLite dialect; courtesy Russell Stuart. ¶ This change is also **backported** to: 0.8.2

  References: #2764

mssql

- **[mssql] [bug]** When querying the information schema on SQL Server 2000, removed a CAST call that was added in 0.8.1 to help with driver issues, which apparently is not compatible on 2000. The CAST remains in

place for SQL Server 2005 and greater. ¶ This change is also **backported** to: 0.8.2

References: #2747

- **[mssql] [bug] [pyodbc]** Fixes to MSSQL with Python 3 + pyodbc, including that statements are passed correctly. ¶ References: #2355

oracle

- **[oracle] [feature] [py3k]** The Oracle unit tests with cx_oracle now pass fully under Python 3. ¶
- **[oracle] [bug]** Fixed bug where Oracle table reflection using synonyms would fail if the synonym and the table were in different remote schemas. Patch to fix courtesy Kyle Derr. ¶ This change is also **backported** to: 0.8.3

  References: #2853

firebird

- **[firebird] [feature]** Added new flag `retaining=True` to the kinterbasdb and fdb dialects. This controls the value of the `retaining` flag sent to the `commit()` and `rollback()` methods of the DBAPI connection. Due to historical concerns, this flag defaults to `True` in 0.8.2, however in 0.9.0b1 this flag defaults to `False`. ¶ This change is also **backported** to: 0.8.2

  References: #2763
- **[firebird] [feature]** The `fdb` dialect is now the default dialect when specified without a dialect qualifier, i.e. `firebird://`, per the Firebird project publishing `fdb` as their official Python driver. ¶ References: #2504
- **[firebird] [bug]** Type lookup when reflecting the Firebird types LONG and INT64 has been fixed so that LONG is treated as INTEGER, INT64 treated as BIGINT, unless the type has a "precision" in which case it's treated as NUMERIC. Patch courtesy Russell Stuart. ¶ This change is also **backported** to: 0.8.2

  References: #2757

misc

- **[feature]** Added a new flag `system=True` to *`Column`*, which marks the column as a "system" column which is automatically made present by the database (such as Postgresql `oid` or `xmin`). The column will be omitted from the `CREATE TABLE` statement but will otherwise be available for querying. In addition, the *`CreateColumn`* construct can be appled to a custom compilation rule which allows skipping of columns, by producing a rule that returns `None`. ¶ This change is also **backported** to: 0.8.3
- **[feature] [examples]** Improved the examples in `examples/generic_associations`, including that `discriminator_on_association.py` makes use of single table inheritance do the work with the "discriminator". Also added a true "generic foreign key" example, which works similarly to other popular frameworks in that it uses an open-ended integer to point to any other table, foregoing traditional referential integrity. While we don't recommend this pattern, information wants to be free. ¶ This change is also **backported** to: 0.8.3
- **[feature] [core]** Added a new variant to *`UpdateBase.returning()`* called *`ValuesBase.return_defaults()`*; this allows arbitrary columns to be added to the RETURNING clause of the statement without interfering with the compilers usual "implicit returning" feature, which is used to efficiently fetch newly generated primary key values. For supporting backends, a dictionary of all fetched values is present at *`ResultProxy.returned_defaults`*. ¶ References: #2793
- **[feature] [pool]** Added pool logging for "rollback-on-return" and the less used "commit-on-return". This is enabled with the rest of pool "debug" logging. ¶ References: #2752

- **[bug] [examples]** Added "autoincrement=False" to the history table created in the versioning example, as this table shouldn't have autoinc on it in any case, courtesy Patrick Schmid. ¶ This change is also **backported** to: 0.8.3
- **[bug] [ext]** Fixed bug whereby if a composite type were set up with a function instead of a class, the mutable extension would trip up when it tried to check that column for being a `MutableComposite` (which it isn't). Courtesy asldevi. ¶ This change is also **backported** to: 0.8.2
- **[bug] [examples]** Fixed an issue with the "versioning" recipe whereby a many-to-one reference could produce a meaningless version for the target, even though it was not changed, when backrefs were present. Patch courtesy Matt Chisholm. ¶ This change is also **backported** to: 0.8.2
- **[requirements]** The Python mock library is now required in order to run the unit test suite. While part of the standard library as of Python 3.3, previous Python installations will need to install this in order to run unit tests or to use the `sqlalchemy.testing` package for external dialects. ¶ This change is also **backported** to: 0.8.2

6.2.4 0.8 Changelog

0.8.7

Released: July 22, 2014

orm

- **[orm] [bug]** Fixed bug in subquery eager loading where a long chain of eager loads across a polymorphic-subclass boundary in conjunction with polymorphic loading would fail to locate the subclass-link in the chain, erroring out with a missing property name on an `AliasedClass`. ¶ References: #3055
- **[orm] [bug]** Fixed ORM bug where the `class_mapper()` function would mask AttributeErrors or KeyErrors that should raise during mapper configuration due to user errors. The catch for attribute/keyerror has been made more specific to not include the configuration step. ¶ References: #3047

sql

- **[sql] [bug]** Fixed bug in `Enum` and other `SchemaType` subclasses where direct association of the type with a `MetaData` would lead to a hang when events (like create events) were emitted on the `MetaData`. ¶ References: #3124
- **[sql] [bug]** Fixed a bug within the custom operator plus `TypeEngine.with_variant()` system, whereby using a `TypeDecorator` in conjunction with variant would fail with an MRO error when a comparison operator was used. ¶ References: #3102
- **[sql] [bug]** Fixed bug in INSERT..FROM SELECT construct where selecting from a UNION would wrap the union in an anonymous (e.g. unlabled) subquery. ¶ References: #3044
- **[sql] [bug]** Fixed bug where `Table.update()` and `Table.delete()` would produce an empty WHERE clause when an empty `and_()` or `or_()` or other blank expression were applied. This is now consistent with that of `select()`. ¶ References: #3045

postgresql

- **[postgresql] [bug]** Added the `hashable=False` flag to the PG `HSTORE` type, which is needed to allow the ORM to skip over trying to "hash" an ORM-mapped HSTORE column when requesting it in a mixed

column/entity list. Patch courtesy Gunnlaugur Þór Briem. ¶ References: #3053

- **[postgresql] [bug]** Added a new "disconnect" message "connection has been closed unexpectedly". This appears to be related to newer versions of SSL. Pull request courtesy Antti Haapala. ¶ References: pull request bitbucket:13

mysql

- **[mysql] [bug]** MySQL error 2014 "commands out of sync" appears to be raised as a ProgrammingError, not OperationalError, in modern MySQL-Python versions; all MySQL error codes that are tested for "is disconnect" are now checked within OperationalError and ProgrammingError regardless. ¶ References: #3101
- **[mysql] [bug]** Fixed bug where column names added to `mysql_length` parameter on an index needed to have the same quoting for quoted names in order to be recognized. The fix makes the quotes optional but also provides the old behavior for backwards compatibility with those using the workaround. ¶ References: #3085
- **[mysql] [bug]** Added support for reflecting tables where an index includes KEY_BLOCK_SIZE using an equal sign. Pull request courtesy Sean McGivern. ¶ References: pull request bitbucket:15

mssql

- **[mssql] [bug]** Added statement encoding to the "SET IDENTITY_INSERT" statements which operate when an explicit INSERT is being interjected into an IDENTITY column, to support non-ascii table identifiers on drivers such as pyodbc + unix + py2k that don't support unicode statements. ¶
- **[mssql] [bug]** In the SQL Server pyodbc dialect, repaired the implementation for the `description_encoding` dialect parameter, which when not explicitly set was preventing cursor.description from being parsed correctly in the case of result sets that contained names in alternate encodings. This parameter shouldn't be needed going forward. ¶ References: #3091

misc

- **[bug] [declarative]** The `__mapper_args__` dictionary is copied from a declarative mixin or abstract class when accessed, so that modifications made to this dictionary by declarative itself won't conflict with that of other mappings. The dictionary is modified regarding the `version_id_col` and `polymorphic_on` arguments, replacing the column within with the one that is officially mapped to the local class/table. ¶ References: #3062
- **[bug] [ext]** Fixed bug in mutable extension where *`MutableDict`* did not report change events for the `setdefault()` dictionary operation. ¶ References: #3093, #3051
- **[bug] [ext]** Fixed bug where *`MutableDict.setdefault()`* didn't return the existing or new value (this bug was not released in any 0.8 version). Pull request courtesy Thomas Hervé. ¶ References: #3093, #3051, pull request bitbucket:24

0.8.6

Released: March 28, 2014

general

- **[general] [bug]** Adjusted `setup.py` file to support the possible future removal of the `setuptools.Feature` extension from setuptools. If this keyword isn't present, the setup will still succeed with setuptools rather than falling back to distutils. C extension building can be disabled now also

by setting the DISABLE_SQLALCHEMY_CEXT environment variable. This variable works whether or not setuptools is even available. ¶ References: #2986

orm

- **[orm] [bug]** Fixed ORM bug where changing the primary key of an object, then marking it for DELETE would fail to target the correct row for DELETE. ¶ References: #3006
- **[orm] [bug]** Fixed regression from 0.8.3 as a result of #2818 where `Query.exists()` wouldn't work on a query that only had a `Query.select_from()` entry but no other entities. ¶ References: #2995
- **[orm] [bug]** Improved an error message which would occur if a query() were made against a non-selectable, such as a `literal_column()`, and then an attempt was made to use `Query.join()` such that the "left" side would be determined as `None` and then fail. This condition is now detected explicitly. ¶
- **[orm] [bug]** Removed stale names from `sqlalchemy.orm.interfaces.__all__` and refreshed with current names, so that an `import *` from this module again works. ¶ References: #2975

sql

- **[sql] [bug]** Fixed bug in `tuple_()` construct where the "type" of essentially the first SQL expression would be applied as the "comparison type" to a compared tuple value; this has the effect in some cases of an inappropriate "type coersion" occurring, such as when a tuple that has a mix of String and Binary values improperly coerces target values to Binary even though that's not what they are on the left side. `tuple_()` now expects heterogeneous types within its list of values. ¶ References: #2977

postgresql

- **[postgresql] [feature]** Enabled "sane multi-row count" checking for the psycopg2 DBAPI, as this seems to be supported as of psycopg2 2.0.9. ¶
- **[postgresql] [bug]** Fixed regression caused by release 0.8.5 / 0.9.3's compatibility enhancements where index reflection on Postgresql versions specific to only the 8.1, 8.2 series again broke, surrounding the ever problematic int2vector type. While int2vector supports array operations as of 8.1, apparently it only supports CAST to a varchar as of 8.3. ¶ References: #3000

misc

- **[bug] [ext]** Fixed bug in mutable extension as well as `attributes.flag_modified()` where the change event would not be propagated if the attribute had been reassigned to itself. ¶ References: #2997

0.8.5

Released: February 19, 2014

orm

- **[orm] [bug]** Fixed bug where `Query.get()` would fail to consistently raise the `InvalidRequestError` that invokes when called on a query with existing criterion, when the given identity is already present in the identity map. ¶ References: #2951

- **[orm] [bug]** Fixed error message when an iterator object is passed to `class_mapper()` or similar, where the error would fail to render on string formatting. Pullreq courtesy Kyle Stark. ¶ References: pull request github:58
- **[orm] [bug]** An adjustment to the `subqueryload()` strategy which ensures that the query runs after the loading process has begun; this is so that the subqueryload takes precedence over other loaders that may be hitting the same attribute due to other eager/noload situations at the wrong time. ¶ References: #2887
- **[orm] [bug]** Fixed bug when using joined table inheritance from a table to a select/alias on the base, where the PK columns were also not same named; the persistence system would fail to copy primary key values from the base table to the inherited table upon INSERT. ¶ References: #2885
- **[orm] [bug]** `composite()` will raise an informative error message when the columns/attribute (names) passed don't resolve to a Column or mapped attribute (such as an erroneous tuple); previously raised an unbound local. ¶ References: #2889

engine

- **[engine] [bug] [pool]** Fixed a critical regression caused by #2880 where the newly concurrent ability to return connections from the pool means that the "first_connect" event is now no longer synchronized either, thus leading to dialect mis-configurations under even minimal concurrency situations. ¶ References: #2964, #2880

sql

- **[sql] [bug]** Fixed bug where calling `Insert.values()` with an empty list or tuple would raise an IndexError. It now produces an empty insert construct as would be the case with an empty dictionary. ¶ References: #2944
- **[sql] [bug]** Fixed bug where `in_()` would go into an endless loop if erroneously passed a column expression whose comparator included the `__getitem__()` method, such as a column that uses the `postgresql.ARRAY` type. ¶ References: #2957
- **[sql] [bug]** Fixed issue where a primary key column that has a Sequence on it, yet the column is not the "auto increment" column, either because it has a foreign key constraint or `autoincrement=False` set, would attempt to fire the Sequence on INSERT for backends that don't support sequences, when presented with an INSERT missing the primary key value. This would take place on non-sequence backends like SQLite, MySQL. ¶ References: #2896
- **[sql] [bug]** Fixed bug with `Insert.from_select()` method where the order of the given names would not be taken into account when generating the INSERT statement, thus producing a mismatch versus the column names in the given SELECT statement. Also noted that `Insert.from_select()` implies that Python-side insert defaults cannot be used, since the statement has no VALUES clause. ¶ References: #2895
- **[sql] [enhancement]** The exception raised when a `BindParameter` is present in a compiled statement without a value now includes the key name of the bound parameter in the error message. ¶

postgresql

- **[postgresql] [bug]** Added an additional message to psycopg2 disconnect detection, "could not send data to server", which complements the existing "could not receive data from server" and has been observed by users. ¶ References: #2936
- **[postgresql] [bug]**

  Support has been improved for Postgresql reflection behavior on very old (pre 8.1) versions of Postgresql, and potentially other PG engines such as Redshift (assuming Redshift reports the version as < 8.1). The query for "indexes" as well as "primary keys" relies upon inspecting a so-called

"int2vector" datatype, which refuses to coerce to an array prior to 8.1 causing failures regarding the "ANY()" operator used in the query. Extensive googling has located the very hacky, but recommended-by-PG-core-developer query to use when PG version < 8.1 is in use, so index and primary key constraint reflection now work on these versions.

¶

- **[postgresql] [bug]** Revised this very old issue where the Postgresql "get primary key" reflection query were updated to take into account primary key constraints that were renamed; the newer query fails on very old versions of Postgresql such as version 7, so the old query is restored in those cases when server_version_info < (8, 0) is detected. ¶ References: #2291

mysql

- **[mysql] [feature]** Added new MySQL-specific `mysql.DATETIME` which includes fractional seconds support; also added fractional seconds support to `mysql.TIMESTAMP`. DBAPI support is limited, though fractional seconds are known to be supported by MySQL Connector/Python. Patch courtesy Geert JM Vanderkelen. ¶ References: #2941
- **[mysql] [bug]** Added support for the `PARTITION BY` and `PARTITIONS` MySQL table keywords, specified as `mysql_partition_by='value'` and `mysql_partitions='value'` to `Table`. Pull request courtesy Marcus McCurdy. ¶ References: #2966, pull request bitbucket:12
- **[mysql] [bug]** Fixed bug which prevented MySQLdb-based dialects (e.g. pymysql) from working in Py3K, where a check for "connection charset" would fail due to Py3K's more strict value comparison rules. The call in question wasn't taking the database version into account in any case as the server version was still None at that point, so the method overall has been simplified to rely upon connection.character_set_name(). ¶ References: #2933
- **[mysql] [bug]** Some missing methods added to the cymysql dialect, including _get_server_version_info() and _detect_charset(). Pullreq courtesy Hajime Nakagami. ¶ References: pull request github:61

sqlite

- **[sqlite] [bug]** Restored a change that was missed in the backport of unique constraint reflection to 0.8, where `UniqueConstraint` with SQLite would fail if reserved keywords were included in the names of columns. Pull request courtesy Roman Podolyaka. ¶ References: pull request github:72

mssql

- **[mssql] [bug] [firebird]** The "asdecimal" flag used with the `Float` type will now work with Firebird as well as the mssql+pyodbc dialects; previously the decimal conversion was not occurring. ¶
- **[mssql] [bug] [pymssql]** Added "Net-Lib error during Connection reset by peer" message to the list of messages checked for "disconnect" within the pymssql dialect. Courtesy John Anderson. ¶ References: pull request github:51

firebird

- **[firebird] [bug]** The firebird dialect will quote identifiers which begin with an underscore. Courtesy Treeve Jelbert. ¶ References: #2897
- **[firebird] [bug]** Fixed bug in Firebird index reflection where the columns within the index were not sorted correctly; they are now sorted in order of RDB$FIELD_POSITION. ¶

misc

- **[bug] [py3k]** Fixed Py3K bug where a missing import would cause "literal binary" mode to fail to import "util.binary_type" when rendering a bound parameter. 0.9 handles this differently. Pull request courtesy Andreas Zeidler. ¶ References: pull request github:63
- **[bug] [declarative]** Error message when a string arg sent to `relationship()` which doesn't resolve to a class or mapper has been corrected to work the same way as when a non-string arg is received, which indicates the name of the relationship which had the configurational error. ¶ References: #2888

0.8.4

Released: December 8, 2013

orm

- **[orm] [bug]** Fixed a regression introduced by #2818 where the EXISTS query being generated would produce a "columns being replaced" warning for a statement with two same-named columns, as the internal SELECT wouldn't have use_labels set. ¶ References: #2818

engine

- **[engine] [bug]** A DBAPI that raises an error on `connect()` which is not a subclass of dbapi.Error (such as `TypeError`, `NotImplementedError`, etc.) will propagate the exception unchanged. Previously, the error handling specific to the `connect()` routine would both inappropriately run the exception through the dialect's `Dialect.is_disconnect()` routine as well as wrap it in a `sqlalchemy.exc.DBAPIError`. It is now propagated unchanged in the same way as occurs within the execute process. ¶ References: #2881
- **[engine] [bug] [pool]** The `QueuePool` has been enhanced to not block new connection attempts when an existing connection attempt is blocking. Previously, the production of new connections was serialized within the block that monitored overflow; the overflow counter is now altered within its own critical section outside of the connection process itself. ¶ References: #2880
- **[engine] [bug] [pool]** Made a slight adjustment to the logic which waits for a pooled connection to be available, such that for a connection pool with no timeout specified, it will every half a second break out of the wait to check for the so-called "abort" flag, which allows the waiter to break out in case the whole connection pool was dumped; normally the waiter should break out due to a notify_all() but it's possible this notify_all() is missed in very slim cases. This is an extension of logic first introduced in 0.8.0, and the issue has only been observed occasionally in stress tests. ¶ References: #2522
- **[engine] [bug]** Fixed bug where SQL statement would be improperly ASCII-encoded when a pre-DBAPI `StatementError` were raised within `Connection.execute()`, causing encoding errors for non-ASCII statements. The stringification now remains within Python unicode thus avoiding encoding errors. ¶ References: #2871

sql

- **[sql] [feature]** Added support for "unique constraint" reflection, via the `Inspector.get_unique_constraints()` method. Thanks for Roman Podolyaka for the patch. ¶ References: #1443

postgresql

- **[postgresql] [bug]** Fixed bug where index reflection would mis-interpret indkey values when using the pypostgresql adapter, which returns these values as lists vs. psycopg2's return type of string. ¶ References: #2855

mssql

- **[mssql] [bug]** Fixed bug introduced in 0.8.0 where the `DROP INDEX` statement for an index in MSSQL would render incorrectly if the index were in an alternate schema; the schemaname/tablename would be reversed. The format has been also been revised to match current MSSQL documentation. Courtesy Derek Harland. ¶ References: pull request bitbucket:7

oracle

- **[oracle] [bug]** Added ORA-02396 "maximum idle time" error code to list of "is disconnect" codes with cx_oracle. ¶ References: #2864
- **[oracle] [bug]** Fixed bug where Oracle `VARCHAR` types given with no length (e.g. for a `CAST` or similar) would incorrectly render `None CHAR` or similar. ¶ References: #2870

misc

- **[bug] [ext]** Fixed bug which prevented the `serializer` extension from working correctly with table or column names that contain non-ASCII characters. ¶ References: #2869

0.8.3

Released: October 26, 2013

orm

- **[orm] [feature]** Added new option to *`relationship()`* `distinct_target_key`. This enables the subquery eager loader strategy to apply a DISTINCT to the innermost SELECT subquery, to assist in the case where duplicate rows are generated by the innermost query which corresponds to this relationship (there's not yet a general solution to the issue of dupe rows within subquery eager loading, however, when joins outside of the innermost subquery produce dupes). When the flag is set to `True`, the DISTINCT is rendered unconditionally, and when it is set to `None`, DISTINCT is rendered if the innermost relationship targets columns that do not comprise a full primary key. The option defaults to False in 0.8 (e.g. off by default in all cases), None in 0.9 (e.g. automatic by default). Thanks to Alexander Koval for help with this.

  See also:

  Subquery Eager Loading will apply DISTINCT to the innermost SELECT for some queries

  ¶ References: #2836

- **[orm] [bug]** Fixed bug where list instrumentation would fail to represent a setslice of `[0:0]` correctly, which in particular could occur when using `insert(0, item)` with the association proxy. Due to some quirk in Python collections, the issue was much more likely with Python 3 rather than 2. ¶ This change is also **backported** to: 0.7.11

  References: #2807

- **[orm] [bug]** Fixed bug where using an annotation such as `remote()` or `foreign()` on a `Column` before association with a parent `Table` could produce issues related to the parent table not rendering within joins, due to the inherent copy operation performed by an annotation. ¶ References: #2813
- **[orm] [bug]** Fixed bug where `Query.exists()` failed to work correctly without any WHERE criterion. Courtesy Vladimir Magamedov. ¶ References: #2818
- **[orm] [bug]** Backported a change from 0.9 whereby the iteration of a hierarchy of mappers used in polymorphic inheritance loads is sorted, which allows the SELECT statements generated for polymorphic queries to have deterministic rendering, which in turn helps with caching schemes that cache on the SQL string itself. ¶ References: #2779
- **[orm] [bug]** Fixed a potential issue in an ordered sequence implementation used by the ORM to iterate mapper hierarchies; under the Jython interpreter this implementation wasn't ordered, even though cPython and Pypy maintained ordering. ¶ References: #2794
- **[orm] [bug]** Fixed bug in ORM-level event registration where the "raw" or "propagate" flags could potentially be mis-configured in some "unmapped base class" configurations. ¶ References: #2786
- **[orm] [bug]** A performance fix related to the usage of the `defer()` option when loading mapped entities. The function overhead of applying a per-object deferred callable to an instance at load time was significantly higher than that of just loading the data from the row (note that `defer()` is meant to reduce DB/network overhead, not necessarily function call count); the function call overhead is now less than that of loading data from the column in all cases. There is also a reduction in the number of "lazy callable" objects created per load from N (total deferred values in the result) to 1 (total number of deferred cols). ¶ References: #2778
- **[orm] [bug]** Fixed bug whereby attribute history functions would fail when an object we moved from "persistent" to "pending" using the `make_transient()` function, for operations involving collection-based backrefs. ¶ References: #2773

orm declarative

- **[feature] [orm] [declarative]** Added a convenience class decorator `as_declarative()`, is a wrapper for `declarative_base()` which allows an existing base class to be applied using a nifty class-decorated approach. ¶

engine

- **[engine] [feature]** `repr()` for the `URL` of an `Engine` will now conceal the password using asterisks. Courtesy Gunnlaugur Þór Briem. ¶ References: #2821
- **[engine] [bug]** The regexp used by the `make_url()` function now parses ipv6 addresses, e.g. surrounded by brackets. ¶ This change is also **backported** to: 0.7.11

  References: #2851
- **[engine] [bug] [oracle]** Dialect.initialize() is not called a second time if an `Engine` is recreated, due to a disconnect error. This fixes a particular issue in the Oracle 8 dialect, but in general the dialect.initialize() phase should only be once per dialect. ¶ References: #2776
- **[engine] [bug] [pool]** Fixed bug where `QueuePool` would lose the correct checked out count if an existing pooled connection failed to reconnect after an invalidate or recycle event. ¶ References: #2772

sql

- **[sql] [feature]** Added new method to the *insert()* construct *Insert.from_select()*. Given a list of columns and a selectable, renders `INSERT INTO (table) (columns) SELECT ...` ¶ References: #722
- **[sql] [feature]** The *update()*, *insert()*, and *delete()* constructs will now interpret ORM entities as target tables to be operated upon, e.g.:

```
from sqlalchemy import insert, update, delete

ins = insert(SomeMappedClass).values(x=5)

del_ = delete(SomeMappedClass).where(SomeMappedClass.id == 5)

upd = update(SomeMappedClass).where(SomeMappedClass.id == 5).values(name='ed')
```

  ¶
- **[sql] [bug]** Fixed regression dating back to 0.7.9 whereby the name of a CTE might not be properly quoted if it was referred to in multiple FROM clauses. ¶ This change is also **backported** to: 0.7.11

  References: #2801
- **[sql] [bug] [cte]** Fixed bug in common table expression system where if the CTE were used only as an `alias()` construct, it would not render using the WITH keyword. ¶ This change is also **backported** to: 0.7.11

  References: #2783
- **[sql] [bug]** Fixed bug in *CheckConstraint* DDL where the "quote" flag from a *Column* object would not be propagated. ¶ This change is also **backported** to: 0.7.11

  References: #2784
- **[sql] [bug]** Fixed bug where *type_coerce()* would not interpret ORM elements with a `__clause_element__()` method properly. ¶ References: #2849
- **[sql] [bug]** The *Enum* and *Boolean* types now bypass any custom (e.g. TypeDecorator) type in use when producing the CHECK constraint for the "non native" type. This so that the custom type isn't involved in the expression within the CHECK, since this expression is against the "impl" value and not the "decorated" value. ¶ References: #2842
- **[sql] [bug]** The `.unique` flag on *Index* could be produced as `None` if it was generated from a *Column* that didn't specify `unique` (where it defaults to `None`). The flag will now always be `True` or `False`. ¶ References: #2825
- **[sql] [bug]** Fixed bug in default compiler plus those of postgresql, mysql, and mssql to ensure that any literal SQL expression values are rendered directly as literals, instead of as bound parameters, within a CREATE INDEX statement. This also changes the rendering scheme for other DDL such as constraints. ¶ References: #2742
- **[sql] [bug]** A *select()* that is made to refer to itself in its FROM clause, typically via in-place mutation, will raise an informative error message rather than causing a recursion overflow. ¶ References: #2815
- **[sql] [bug]** Non-working "schema" argument on *ForeignKey* is deprecated; raises a warning. Removed in 0.9. ¶ References: #2831
- **[sql] [bug]** Fixed bug where using the `column_reflect` event to change the `.key` of the incoming *Column* would prevent primary key constraints, indexes, and foreign key constraints from being correctly reflected. ¶ References: #2811

- **[sql] [bug]** The `ColumnOperators.notin_()` operator added in 0.8 now properly produces the negation of the expression "IN" returns when used against an empty collection. ¶
- **[sql] [bug] [postgresql]** Fixed bug where the expression system relied upon the `str()` form of a some expressions when referring to the `.c` collection on a `select()` construct, but the `str()` form isn't available since the element relies on dialect-specific compilation constructs, notably the `__getitem__()` operator as used with a Postgresql `ARRAY` element. The fix also adds a new exception class `UnsupportedCompilationError` which is raised in those cases where a compiler is asked to compile something it doesn't know how to. ¶ References: #2780

postgresql

- **[postgresql] [bug]** Removed a 128-character truncation from the reflection of the server default for a column; this code was original from PG system views which truncated the string for readability. ¶ References: #2844
- **[postgresql] [bug]** Parenthesis will be applied to a compound SQL expression as rendered in the column list of a CREATE INDEX statement. ¶ References: #2742
- **[postgresql] [bug]** Fixed bug where Postgresql version strings that had a prefix preceding the words "Postgresql" or "EnterpriseDB" would not parse. Courtesy Scott Schaefer. ¶ References: #2819

mysql

- **[mysql] [bug]** Updates to MySQL reserved words for versions 5.5, 5.6, courtesy Hanno Schlichting. ¶ This change is also **backported** to: 0.7.11

  References: #2791
- **[mysql] [bug]** The change in #2721, which is that the `deferrable` keyword of `ForeignKeyConstraint` is silently ignored on the MySQL backend, will be reverted as of 0.9; this keyword will now render again, raising errors on MySQL as it is not understood - the same behavior will also apply to the `initially` keyword. In 0.8, the keywords will remain ignored but a warning is emitted. Additionally, the `match` keyword now raises a `CompileError` on 0.9 and emits a warning on 0.8; this keyword is not only silently ignored by MySQL but also breaks the ON UPDATE/ON DELETE options.

  To use a `ForeignKeyConstraint` that does not render or renders differently on MySQL, use a custom compilation option. An example of this usage has been added to the documentation, see *MySQL Foreign Keys*. ¶ References: #2721, #2839
- **[mysql] [bug]** MySQL-connector dialect now allows options in the create_engine query string to override those defaults set up in the connect, including "buffered" and "raise_on_warnings". ¶ References: #2515

sqlite

- **[sqlite] [bug]** The newly added SQLite DATETIME arguments storage_format and regexp apparently were not fully implemented correctly; while the arguments were accepted, in practice they would have no effect; this has been fixed. ¶ References: #2781

oracle

- **[oracle] [bug]** Fixed bug where Oracle table reflection using synonyms would fail if the synonym and the table were in different remote schemas. Patch to fix courtesy Kyle Derr. ¶ References: #2853

misc

- **[feature]** Added a new flag `system=True` to *Column*, which marks the column as a "system" column which is automatically made present by the database (such as Postgresql `oid` or `xmin`). The column will be omitted from the `CREATE TABLE` statement but will otherwise be available for querying. In addition, the *CreateColumn* construct can be appled to a custom compilation rule which allows skipping of columns, by producing a rule that returns `None`. ¶
- **[feature] [examples]** Improved the examples in `examples/generic_associations`, including that `discriminator_on_association.py` makes use of single table inheritance do the work with the "discriminator". Also added a true "generic foreign key" example, which works similarly to other popular frameworks in that it uses an open-ended integer to point to any other table, foregoing traditional referential integrity. While we don't recommend this pattern, information wants to be free. ¶
- **[bug] [examples]** Added "autoincrement=False" to the history table created in the versioning example, as this table shouldn't have autoinc on it in any case, courtesy Patrick Schmid. ¶

0.8.2

Released: July 3, 2013

orm

- **[orm] [feature]** Added a new method *Query.select_entity_from()* which will in 0.9 replace part of the functionality of *Query.select_from()*. In 0.8, the two methods perform the same function, so that code can be migrated to use the *Query.select_entity_from()* method as appropriate. See the 0.9 migration guide for details. ¶ References: #2736
- **[orm] [bug]** A warning is emitted when trying to flush an object of an inherited class where the polymorphic discriminator has been assigned to a value that is invalid for the class. ¶ References: #2750
- **[orm] [bug]** Fixed bug in polymorphic SQL generation where multiple joined-inheritance entities against the same base class joined to each other as well would not track columns on the base table independently of each other if the string of joins were more than two entities long. ¶ References: #2759
- **[orm] [bug]** Fixed bug where sending a composite attribute into *Query.order_by()* would produce a parenthesized expression not accepted by some databases. ¶ References: #2754
- **[orm] [bug]** Fixed the interaction between composite attributes and the *aliased()* function. Previously, composite attributes wouldn't work correctly in comparison operations when aliasing was applied. ¶ References: #2755
- **[orm] [bug] [ext]** Fixed bug where *MutableDict* didn't report a change event when `clear()` was called. ¶ References: #2730
- **[orm] [bug]** Fixed a regression caused by #2682 whereby the evaluation invoked by *Query.update()* and *Query.delete()* would hit upon unsupported `True` and `False` symbols which now appear due to the usage of `IS`. ¶ References: #2737
- **[orm] [bug]** Fixed a regression from 0.7 caused by this ticket, which made the check for recursion overflow in self-referential eager joining too loose, missing a particular circumstance where a subclass had lazy="joined" or "subquery" configured and the load was a "with_polymorphic" against the base. ¶ References: #2481
- **[orm] [bug]** Fixed a regression from 0.7 where the contextmanager feature of *Session.begin_nested()* would fail to correctly roll back the transaction when a flush error occurred, instead raising its own exception while leaving the session still pending a rollback. ¶ References: #2718

orm declarative

- **[feature] [orm] [declarative]** ORM descriptors such as hybrid properties can now be referenced by name in a string argument used with `order_by`, `primaryjoin`, or similar in *relationship()*, in addition to column-bound attributes. ¶ References: #2761

engine

- **[engine] [bug]** Fixed bug where the `reset_on_return` argument to various *Pool* implementations would not be propagated when the pool was regenerated. Courtesy Eevee. ¶ References: pull request github:6
- **[engine] [bug] [sybase]** Fixed a bug where the routine to detect the correct kwargs being sent to *create_engine()* would fail in some cases, such as with the Sybase dialect. ¶ References: #2732

sql

- **[sql] [feature]** Provided a new attribute for *TypeDecorator* called *TypeDecorator.coerce_to_is_types*, to make it easier to control how comparisons using `==` or `!=` to `None` and boolean types goes about producing an `IS` expression, or a plain equality expression with a bound parameter. ¶ References: #2734, #2744
- **[sql] [bug]** Multiple fixes to the correlation behavior of *Select* constructs, first introduced in 0.8.0:
  - To satisfy the use case where FROM entries should be correlated outwards to a SELECT that encloses another, which then encloses this one, correlation now works across multiple levels when explicit correlation is established via *Select.correlate()*, provided that the target select is somewhere along the chain contained by a WHERE/ORDER BY/columns clause, not just nested FROM clauses. This makes *Select.correlate()* act more compatibly to that of 0.7 again while still maintaining the new "smart" correlation.
  - When explicit correlation is not used, the usual "implicit" correlation limits its behavior to just the immediate enclosing SELECT, to maximize compatibility with 0.7 applications, and also prevents correlation across nested FROMs in this case, maintaining compatibility with 0.8.0/0.8.1.
  - The *Select.correlate_except()* method was not preventing the given FROM clauses from correlation in all cases, and also would cause FROM clauses to be incorrectly omitted entirely (more like what 0.7 would do), this has been fixed.
  - Calling *select.correlate_except(None)* will enter all FROM clauses into correlation as would be expected.

  ¶ References: #2668, #2746
- **[sql] [bug]** Fixed bug whereby joining a select() of a table "A" with multiple foreign key paths to a table "B", to that table "B", would fail to produce the "ambiguous join condition" error that would be reported if you join table "A" directly to "B"; it would instead produce a join condition with multiple criteria. ¶ References: #2738
- **[sql] [bug] [reflection]** Fixed bug whereby using *MetaData.reflect()* across a remote schema as well as a local schema could produce wrong results in the case where both schemas had a table of the same name. ¶ References: #2728
- **[sql] [bug]** Removed the "not implemented" `__iter__()` call from the base *ColumnOperators* class, while this was introduced in 0.8.0 to prevent an endless, memory-growing loop when one also implements a `__getitem__()` method on a custom operator and then calls erroneously `list()` on that object, it had the effect of causing column elements to report that they were in fact iterable types which then throw an error when you try to iterate. There's no real way to have both sides here so we stick with Python best practices. Careful with implementing `__getitem__()` on your custom operators! ¶ References: #2726

- **[sql] [bug] [mssql]** Regression from this ticket caused the unsupported keyword "true" to render, added logic to convert this to 1/0 for SQL server. ¶ References: #2682

postgresql

- **[postgresql] [feature]** Support for Postgresql 9.2 range types has been added. Currently, no type translation is provided, so works directly with strings or psycopg2 2.5 range extension types at the moment. Patch courtesy Chris Withers. ¶
- **[postgresql] [feature]** Added support for "AUTOCOMMIT" isolation when using the psycopg2 DBAPI. The keyword is available via the `isolation_level` execution option. Patch courtesy Roman Podolyaka. ¶ References: #2072
- **[postgresql] [bug]** The behavior of *`extract()`* has been simplified on the Postgresql dialect to no longer inject a hardcoded `::timestamp` or similar cast into the given expression, as this interfered with types such as timezone-aware datetimes, but also does not appear to be at all necessary with modern versions of psycopg2. ¶ References: #2740
- **[postgresql] [bug]** Fixed bug in HSTORE type where keys/values that contained backslashed quotes would not be escaped correctly when using the "non native" (i.e. non-psycopg2) means of translating HSTORE data. Patch courtesy Ryan Kelly. ¶ References: #2766
- **[postgresql] [bug]** Fixed bug where the order of columns in a multi-column Postgresql index would be reflected in the wrong order. Courtesy Roman Podolyaka. ¶ References: #2767
- **[postgresql] [bug]** Fixed the HSTORE type to correctly encode/decode for unicode. This is always on, as the hstore is a textual type, and matches the behavior of psycopg2 when using Python 3. Courtesy Dmitry Mugtasimov. ¶ References: #2735, pull request github:2

mysql

- **[mysql] [feature]** The `mysql_length` parameter used with *`Index`* can now be passed as a dictionary of column names/lengths, for use with composite indexes. Big thanks to Roman Podolyaka for the patch. ¶ References: #2704
- **[mysql] [bug]** Fixed bug when using multi-table UPDATE where a supplemental table is a SELECT with its own bound parameters, where the positioning of the bound parameters would be reversed versus the statement itself when using MySQL's special syntax. ¶ References: #2768
- **[mysql] [bug]** Added another conditional to the `mysql+gaerdbms` dialect to detect so-called "development" mode, where we should use the `rdbms_mysqldb` DBAPI. Patch courtesy Brett Slatkin. ¶ References: #2715
- **[mysql] [bug]** The `deferrable` keyword argument on *`ForeignKey`* and *`ForeignKeyConstraint`* will not render the `DEFERRABLE` keyword on the MySQL dialect. For a long time we left this in place because a non-deferrable foreign key would act very differently than a deferrable one, but some environments just disable FKs on MySQL, so we'll be less opinionated here. ¶ References: #2721
- **[mysql] [bug]** Updated mysqlconnector dialect to check for disconnect based on the apparent string message sent in the exception; tested against mysqlconnector 1.0.9. ¶

sqlite

- **[sqlite] [bug]** Added *`sqlalchemy.types.BIGINT`* to the list of type names that can be reflected by the SQLite dialect; courtesy Russell Stuart. ¶ References: #2764

mssql

- **[mssql] [bug]** When querying the information schema on SQL Server 2000, removed a CAST call that was added in 0.8.1 to help with driver issues, which apparently is not compatible on 2000. The CAST remains in place for SQL Server 2005 and greater. ¶ References: #2747

firebird

- **[firebird] [feature]** Added new flag `retaining=True` to the kinterbasdb and fdb dialects. This controls the value of the `retaining` flag sent to the `commit()` and `rollback()` methods of the DBAPI connection. Due to historical concerns, this flag defaults to `True` in 0.8.2, however in 0.9.0b1 this flag defaults to `False`. ¶ References: #2763
- **[firebird] [bug]** Type lookup when reflecting the Firebird types LONG and INT64 has been fixed so that LONG is treated as INTEGER, INT64 treated as BIGINT, unless the type has a "precision" in which case it's treated as NUMERIC. Patch courtesy Russell Stuart. ¶ References: #2757

misc

- **[bug] [ext]** Fixed bug whereby if a composite type were set up with a function instead of a class, the mutable extension would trip up when it tried to check that column for being a *`MutableComposite`* (which it isn't). Courtesy asldevi. ¶
- **[bug] [examples]** Fixed an issue with the "versioning" recipe whereby a many-to-one reference could produce a meaningless version for the target, even though it was not changed, when backrefs were present. Patch courtesy Matt Chisholm. ¶
- **[bug] [examples]** Fixed a small bug in the dogpile example where the generation of SQL cache keys wasn't applying deduping labels to the statement the same way *`Query`* normally does. ¶
- **[requirements]** The Python mock library is now required in order to run the unit test suite. While part of the standard library as of Python 3.3, previous Python installations will need to install this in order to run unit tests or to use the `sqlalchemy.testing` package for external dialects. ¶

0.8.1

Released: April 27, 2013

orm

- **[orm] [feature]** Added a convenience method to Query that turns a query into an EXISTS subquery of the form `EXISTS (SELECT 1 FROM ... WHERE ...)`. ¶ References: #2673
- **[orm] [bug]** Fixed bug when a query of the form: `query(SubClass).options(subqueryload(Baseclass.attrna` where `SubClass` is a joined inh of `BaseClass`, would fail to apply the `JOIN` inside the subquery on the attribute load, producing a cartesian product. The populated results still tended to be correct as additional rows are just ignored, so this issue may be present as a performance degradation in applications that are otherwise working correctly. ¶ This change is also **backported** to: 0.7.11

  References: #2699
- **[orm] [bug]** Fixed bug in unit of work whereby a joined-inheritance subclass could insert the row for the "sub" table before the parent table, if the two tables had no ForeignKey constraints set up between them. ¶ This change is also **backported** to: 0.7.11

References: #2689

- **[orm] [bug]** Fixes to the `sqlalchemy.ext.serializer` extension, including that the "id" passed from the pickler is turned into a string to prevent against bytes being parsed on Py3K, as well as that `relationship()` and `orm.join()` constructs are now properly serialized. ¶ References: #2698
- **[orm] [bug]** A significant improvement to the inner workings of query.join(), such that the decisionmaking involved on how to join has been dramatically simplified. New test cases now pass such as multiple joins extending from the middle of an already complex series of joins involving inheritance and such. Joining from deeply nested subquery structures is still complicated and not without caveats, but with these improvements the edge cases are hopefully pushed even farther out to the edges. ¶ References: #2714
- **[orm] [bug]** Added a conditional to the unpickling process for ORM mapped objects, such that if the reference to the object were lost when the object was pickled, we don't erroneously try to set up _sa_instance_state - fixes a NoneType error. ¶
- **[orm] [bug]** Fixed bug where many-to-many relationship with uselist=False would fail to delete the association row and raise an error if the scalar attribute were set to None. This was a regression introduced by the changes for #2229. ¶ References: #2710
- **[orm] [bug]** Improved the behavior of instance management regarding the creation of strong references within the Session; an object will no longer have an internal reference cycle created if it's in the transient state or moves into the detached state - the strong ref is created only when the object is attached to a Session and is removed when the object is detached. This makes it somewhat safer for an object to have a *__del__()* method, even though this is not recommended, as relationships with backrefs produce cycles too. A warning has been added when a class with a *__del__()* method is mapped. ¶ References: #2708
- **[orm] [bug]** Fixed bug whereby ORM would run the wrong kind of query when refreshing an inheritance-mapped class where the superclass was mapped to a non-Table object, like a custom join() or a select(), running a query that assumed a hierarchy that's mapped to individual Table-per-class. ¶ References: #2697
- **[orm] [bug]** Fixed *__repr__()* on mapper property constructs to work before the object is initialized, so that Sphinx builds with recent Sphinx versions can read them. ¶

orm declarative

- **[bug] [orm] [declarative]** Fixed indirect regression regarding *`has_inherited_table()`*, where since it considers the current class' `__table__`, was sensitive to when it was called. This is 0.7's behavior also, but in 0.7 things tended to "work out" within events like `__mapper_args__()`. *`has_inherited_table()`* now only considers superclasses, so should return the same answer regarding the current class no matter when it's called (obviously assuming the state of the superclass). ¶ References: #2656

sql

- **[sql] [feature]** Loosened the check on dialect-specific argument names passed to Table(); since we want to support external dialects and also want to support args without a certain dialect being installed, it only checks the format of the arg now, rather than looking for that dialect in sqlalchemy.dialects. ¶
- **[sql] [bug] [mysql]** Fully implemented the IS and IS NOT operators with regards to the True/False constants. An expression like `col.is_(True)` will now render `col IS true` on the target platform, rather than converting the True/ False constant to an integer bound parameter. This allows the `is_()` operator to work on MySQL when given True/False constants. ¶ References: #2682
- **[sql] [bug]** A major fix to the way in which a select() object produces labeled columns when apply_labels() is used; this mode produces a SELECT where each column is labeled as in <tablename>_<columnname>, to remove column name collisions for a multiple table select. The fix is that if two labels collide when combined with the table name, i.e. "foo.bar_id" and "foo_bar.id", anonymous aliasing will be applied to one of the dupes.

This allows the ORM to handle both columns independently; previously, 0.7 would in some cases silently emit a second SELECT for the column that was "duped", and in 0.8 an ambiguous column error would be emitted. The "keys" applied to the .c. collection of the select() will also be deduped, so that the "column being replaced" warning will no longer emit for any select() that specifies use_labels, though the dupe key will be given an anonymous label which isn't generally user-friendly. ¶ References: #2702

- **[sql] [bug]** Fixed bug where disconnect detect on error would raise an attribute error if the error were being raised after the Connection object had already been closed. ¶ References: #2691
- **[sql] [bug]** Reworked internal exception raises that emit a rollback() before re-raising, so that the stack trace is preserved from sys.exc_info() before entering the rollback. This so that the traceback is preserved when using coroutine frameworks which may have switched contexts before the rollback function returns. ¶ References: #2703
- **[sql] [bug] [postgresql]** The _Binary base type now converts values through the bytes() callable when run on Python 3; in particular psycopg2 2.5 with Python 3.3 seems to now be returning the "memoryview" type, so this is converted to bytes before return. ¶
- **[sql] [bug]** Improvements to Connection auto-invalidation handling. If a non-disconnect error occurs, but leads to a delayed disconnect error within error handling (happens with MySQL), the disconnect condition is detected. The Connection can now also be closed when in an invalid state, meaning it will raise "closed" on next usage, and additionally the "close with result" feature will work even if the autorollback in an error handling routine fails and regardless of whether the condition is a disconnect or not. ¶ References: #2695
- **[sql] [bug]** Fixed bug whereby a DBAPI that can return "0" for cursor.lastrowid would not function correctly in conjunction with `ResultProxy.inserted_primary_key`. ¶

postgresql

- **[postgresql] [bug]** Opened up the checking for "disconnect" with psycopg2/libpq to check for all the various "disconnect" messages within the full exception hierarchy. Specifically the "closed the connection unexpectedly" message has now been seen in at least three different exception types. Courtesy Eli Collins. ¶ References: #2712
- **[postgresql] [bug]** The operators for the Postgresql ARRAY type supports input types of sets, generators, etc. even when a dimension is not specified, by turning the given iterable into a collection unconditionally. ¶ References: #2681
- **[postgresql] [bug]** Added missing HSTORE type to postgresql type names so that the type can be reflected. ¶ References: #2680

mysql

- **[mysql] [bug]** Fixes to support the latest cymysql DBAPI, courtesy Hajime Nakagami. ¶
- **[mysql] [bug]** Improvements to the operation of the pymysql dialect on Python 3, including some important decode/bytes steps. Issues remain with BLOB types due to driver issues. Courtesy Ben Trofatter. ¶ References: #2663
- **[mysql] [bug]** Updated a regexp to correctly extract error code on google app engine v1.7.5 and newer. Courtesy Dan Ring. ¶

mssql

- **[mssql] [bug]** Part of a longer series of fixes needed for pyodbc+ mssql, a CAST to NVARCHAR(max) has been added to the bound parameter for the table name and schema name in all information schema queries

to avoid the issue of comparing NVARCHAR to NTEXT, which seems to be rejected by the ODBC driver in some cases, such as FreeTDS (0.91 only?) plus unicode bound parameters being passed. The issue seems to be specific to the SQL Server information schema tables and the workaround is harmless for those cases where the problem doesn't exist in the first place. ¶ References: #2355

- **[mssql] [bug]** Added support for additional "disconnect" messages to the pymssql dialect. Courtesy John Anderson. ¶
- **[mssql] [bug]** Fixed Py3K bug regarding "binary" types and pymssql. Courtesy Marc Abramowitz. ¶ References: #2683

misc

- **[bug] [examples]** Fixed a long-standing bug in the caching example, where the limit/offset parameter values wouldn't be taken into account when computing the cache key. The _key_from_query() function has been simplified to work directly from the final compiled statement in order to get at both the full statement as well as the fully processed parameter list. ¶

0.8.0

Released: March 9, 2013

Note: There are some new behavioral changes as of 0.8.0 not present in 0.8.0b2. They are present in the migration document as follows:

- *The consideration of a "pending" object as an "orphan" has been made more aggressive*
- *create_all() and drop_all() will now honor an empty list as such*
- *Correlation is now always context-specific*

orm

- **[orm] [feature]** A meaningful `QueryableAttribute.info` attribute is added, which proxies down to the `.info` attribute on either the `schema.Column` object if directly present, or the `MapperProperty` otherwise. The full behavior is documented and ensured by tests to remain stable. ¶ References: #2675
- **[orm] [feature]** Can set/change the "cascade" attribute on a `relationship()` construct after it's been constructed already. This is not a pattern for normal use but we like to change the setting for demonstration purposes in tutorials. ¶
- **[orm] [feature]** Added new helper function `was_deleted()`, returns True if the given object was the subject of a `Session.delete()` operation. ¶ References: #2658
- **[orm] [feature]** Extended the Runtime Inspection API system so that all Python descriptors associated with the ORM or its extensions can be retrieved. This fulfills the common request of being able to inspect all `QueryableAttribute` descriptors in addition to extension types such as `hybrid_property` and `AssociationProxy`. See `Mapper.all_orm_descriptors`. ¶
- **[orm] [removed]** The undocumented (and hopefully unused) system of producing custom collections using an `__instrumentation__` datastructure associated with the collection has been removed, as this was a complex and untested feature which was also essentially redundant versus the decorator approach. Other internal simplifcations to the orm.collections module have been made as well. ¶

- **[orm] [bug]** Improved checking for an existing backref name conflict during mapper configuration; will now test for name conflicts on superclasses and subclasses, in addition to the current mapper, as these conflicts break things just as much. This is new for 0.8, but see below for a warning that will also be triggered in 0.7.11. ¶ References: #2674
- **[orm] [bug]** Improved the error message emitted when a "backref loop" is detected, that is when an attribute event triggers a bidirectional assignment between two other attributes with no end. This condition can occur not just when an object of the wrong type is assigned, but also when an attribute is mis-configured to backref into an existing backref pair. Also in 0.7.11. ¶ References: #2674
- **[orm] [bug]** A warning is emitted when a MapperProperty is assigned to a mapper that replaces an existing property, if the properties in question aren't plain column-based properties. Replacement of relationship properties is rarely (ever?) what is intended and usually refers to a mapper mis-configuration. Also in 0.7.11. ¶ References: #2674
- **[orm] [bug]** A clear error message is emitted if an event handler attempts to emit SQL on a Session within the after_commit() handler, where there is not a viable transaction in progress. ¶ References: #2662
- **[orm] [bug]** Detection of a primary key change within the process of cascading a natural primary key update will succeed even if the key is composite and only some of the attributes have changed. ¶ References: #2665
- **[orm] [bug]** An object that's deleted from a session will be de-associated with that session fully after the transaction is committed, that is the `object_session()` function will return None. ¶ References: #2658
- **[orm] [bug]** Fixed bug whereby `Query.yield_per()` would set the execution options incorrectly, thereby breaking subsequent usage of the `Query.execution_options()` method. Courtesy Ryan Kelly. ¶ References: #2661
- **[orm] [bug]** Fixed the consideration of the `between()` operator so that it works correctly with the new relationship local/remote system. ¶ References: #1768
- **[orm] [bug]** the consideration of a pending object as an "orphan" has been modified to more closely match the behavior as that of persistent objects, which is that the object is expunged from the `Session` as soon as it is de-associated from any of its orphan-enabled parents. Previously, the pending object would be expunged only if de-associated from all of its orphan-enabled parents. The new flag `legacy_is_orphan` is added to `orm.mapper()` which re-establishes the legacy behavior.

  See the change note and example case at *The consideration of a "pending" object as an "orphan" has been made more aggressive* for a detailed discussion of this change. ¶ References: #2655
- **[orm] [bug]** Fixed the (most likely never used) "@collection.link" collection method, which fires off each time the collection is associated or de-associated with a mapped object - the decorator was not tested or functional. The decorator method is now named `collection.linker()` though the name "link" remains for backwards compatibility. Courtesy Luca Wehrstedt. ¶ References: #2653
- **[orm] [bug]** Made some fixes to the system of producing custom instrumented collections, mainly that the usage of the @collection decorators will now honor the __mro__ of the given class, applying the logic of the sub-most classes' version of a particular collection method. Previously, it wasn't predictable when subclassing an existing instrumented class such as `MappedCollection` whether or not custom methods would resolve correctly. ¶ References: #2654
- **[orm] [bug]** Fixed potential memory leak which could occur if an arbitrary number of `sessionmaker` objects were created. The anonymous subclass created by the sessionmaker, when dereferenced, would not be garbage collected due to remaining class-level references from the event package. This issue also applies to any custom system that made use of ad-hoc subclasses in conjunction with an event dispatcher. Also in 0.7.10. ¶ References: #2650
- **[orm] [bug]** `Query.merge_result()` can now load rows from an outer join where an entity may be `None` without throwing an error. Also in 0.7.10. ¶ References: #2640

- **[orm] [bug]** Fixes to the "dynamic" loader on *relationship()*, includes that backrefs will work properly even when autoflush is disabled, history events are more accurate in scenarios where multiple add/remove of the same object occurs. ¶ References: #2637

sql

- **[sql] [feature]** Added a new argument to *Enum* and its base *SchemaType* `inherit_schema`. When set to `True`, the type will set its `schema` attribute of that of the *Table* to which it is associated. This also occurs during a *Table.tometadata()* operation; the *SchemaType* is now copied in all cases when *Table.tometadata()* happens, and if `inherit_schema=True`, the type will take on the new schema name passed to the method. The `schema` is important when used with the Postgresql backend, as the type results in a `CREATE TYPE` statement. ¶ References: #2657
- **[sql] [feature]** *Index* now supports arbitrary SQL expressions and/or functions, in addition to straight columns. Common modifiers include using `somecolumn.desc()` for a descending index and `func.lower(somecolumn)` for a case-insensitive index, depending on the capabilities of the target backend. ¶ References: #695
- **[sql] [bug]** The behavior of SELECT correlation has been improved such that the *Select.correlate()* and *Select.correlate_except()* methods, as well as their ORM analogues, will still retain "auto-correlation" behavior in that the FROM clause is modified only if the output would be legal SQL; that is, the FROM clause is left intact if the correlated SELECT is not used in the context of an enclosing SELECT inside of the WHERE, columns, or HAVING clause. The two methods now only specify conditions to the default "auto correlation", rather than absolute FROM lists. ¶ References: #2668
- **[sql] [bug]** Fixed a bug regarding column annotations which in particular could impact some usages of the new *orm.remote()* and `orm.local()` annotation functions, where annotations could be lost when the column were used in a subsequent expression. ¶ References: #1768, #2660
- **[sql] [bug]** The *ColumnOperators.in_()* operator will now coerce values of `None` to *null()*. ¶ References: #2496
- **[sql] [bug]** Fixed bug where *Table.tometadata()* would fail if a *Column* had both a foreign key as well as an alternate ".key" name for the column. Also in 0.7.10. ¶ References: #2643
- **[sql] [bug]** insert().returning() raises an informative CompileError if attempted to compile on a dialect that doesn't support RETURNING. ¶ References: #2629
- **[sql] [bug]** Tweaked the "REQUIRED" symbol used by the compiler to identify INSERT/UPDATE bound parameters that need to be passed, so that it's more easily identifiable when writing custom bind-handling code. ¶ References: #2648

schema

- **[schema] [bug]** *MetaData.create_all()* and *MetaData.drop_all()* will now accommodate an empty list as an instruction to not create/drop any items, rather than ignoring the collection. ¶ References: #2664

postgresql

- **[postgresql] [feature]** Added support for Postgresql's traditional SUBSTRING function syntax, renders as "SUBSTRING(x FROM y FOR z)" when regular `func.substring()` is used. Courtesy Gunnlaugur Þór Briem. ¶ This change is also **backported** to: 0.7.11

  References: #2676

- **[postgresql] [feature]** Added `postgresql.ARRAY.Comparator.any()` and `postgresql.ARRAY.Comparator.all()` methods, as well as standalone expression constructs. Big thanks to Audrius Kažukauskas for the terrific work here. ¶
- **[postgresql] [bug]** Fixed bug in *array()* construct whereby using it inside of an *expression.insert()* construct would produce an error regarding a parameter issue in the `self_group()` method. ¶

mysql

- **[mysql] [feature]** New dialect for CyMySQL added, courtesy Hajime Nakagami. ¶
- **[mysql] [feature]** GAE dialect now accepts username/password arguments in the URL, courtesy Owen Nelson. ¶
- **[mysql] [bug] [gae]** Added a conditional import to the `gaerdbms` dialect which attempts to import rdbms_apiproxy vs. rdbms_googleapi to work on both dev and production platforms. Also now honors the `instance` attribute. Courtesy Sean Lynch. Also in 0.7.10. ¶ References: #2649
- **[mysql] [bug]** GAE dialect won't fail on None match if the error code can't be extracted from the exception throw; courtesy Owen Nelson. ¶

mssql

- **[mssql] [feature]** Added `mssql_include` and `mssql_clustered` options to *Index*, renders the `INCLUDE` and `CLUSTERED` keywords, respectively. Courtesy Derek Harland. ¶
- **[mssql] [feature]** DDL for IDENTITY columns is now supported on non-primary key columns, by establishing a *Sequence* construct on any integer column. Courtesy Derek Harland. ¶ References: #2644
- **[mssql] [bug]** Added a py3K conditional around unnecessary .decode() call in mssql information schema, fixes reflection in Py3K. Also in 0.7.10. ¶ References: #2638
- **[mssql] [bug]** Fixed a regression whereby the "collation" parameter of the character types CHAR, NCHAR, etc. stopped working, as "collation" is now supported by the base string types. The TEXT, NCHAR, CHAR, VARCHAR types within the MSSQL dialect are now synonyms for the base types. ¶

oracle

- **[oracle] [bug]** The cx_oracle dialect will no longer run the bind parameter names through `encode()`, as this is not valid on Python 3, and prevented statements from functioning correctly on Python 3. We now encode only if `supports_unicode_binds` is False, which is not the case for cx_oracle when at least version 5 of cx_oracle is used. ¶

misc

- **[bug] [tests]** Fixed an import of "logging" in test_execute which was not working on some linux platforms. Also in 0.7.11. ¶ References: #2669
- **[bug] [examples]** Fixed a regression in the examples/dogpile_caching example which was due to the change in #2614. ¶

0.8.0b2

Released: December 14, 2012

orm

- **[orm] [feature]** Added `KeyedTuple._asdict()` and `KeyedTuple._fields` to the `KeyedTuple` class to provide some degree of compatibility with the Python standard library `collections.namedtuple()`. ¶ References: #2601
- **[orm] [feature]** Allow synonyms to be used when defining primary and secondary joins for relationships. ¶
- **[orm] [feature] [extensions]** The `sqlalchemy.ext.mutable` extension now includes the example `MutableDict` class as part of the extension. ¶
- **[orm] [bug]** The `Query.select_from()` method can now be used with a `aliased()` construct without it interfering with the entities being selected. Basically, a statement like this:

```
ua = aliased(User)
session.query(User.name).select_from(ua).join(User, User.name > ua.name)
```

Will maintain the columns clause of the SELECT as coming from the unaliased "user", as specified; the select_from only takes place in the FROM clause:

```
SELECT users.name AS users_name FROM users AS users_1
JOIN users ON users.name < users_1.name
```

Note that this behavior is in contrast to the original, older use case for `Query.select_from()`, which is that of restating the mapped entity in terms of a different selectable:

```
session.query(User.name).\
  select_from(user_table.select().where(user_table.c.id > 5))
```

Which produces:

```
SELECT anon_1.name AS anon_1_name FROM (SELECT users.id AS id,
users.name AS name FROM users WHERE users.id > :id_1) AS anon_1
```

It was the "aliasing" behavior of the latter use case that was getting in the way of the former use case. The method now specifically considers a SQL expression like `expression.select()` or `expression.alias()` separately from a mapped entity like a `aliased()` construct. ¶ References: #2635

- **[orm] [bug]** The `MutableComposite` type did not allow for the `MutableBase.coerce()` method to be used, even though the code seemed to indicate this intent, so this now works and a brief example is added. As a side-effect, the mechanics of this event handler have been changed so that new `MutableComposite` types no longer add per-type global event handlers. Also in 0.7.10. ¶ References: #2624
- **[orm] [bug]** A second overhaul of aliasing/internal pathing mechanics now allows two subclasses to have different relationships of the same name, supported with subquery or joined eager loading on both simultaneously when a full polymorphic load is used. ¶ References: #2614
- **[orm] [bug]** Fixed bug whereby a multi-hop subqueryload within a particular with_polymorphic load would produce a KeyError. Takes advantage of the same internal pathing overhaul as #2614. ¶ References: #2617
- **[orm] [bug]** Fixed regression where query.update() would produce an error if an object matched by the "fetch" synchronization strategy wasn't locally present. Courtesy Scott Torborg. ¶ References: #2602

engine

- **[engine] [feature]** The `Connection.connect()` and `Connection.contextual_connect()` methods now return a "branched" version so that the `Connection.close()` method can be called on the returned connection without affecting the original. Allows symmetry when using `Engine` and `Connection` objects as context managers:

```
with conn.connect() as c: # leaves the Connection open
  c.execute("...")

with engine.connect() as c:  # closes the Connection
  c.execute("...")
```

  ¶

- **[engine] [bug]** Fixed `MetaData.reflect()` to correctly use the given `Connection`, if given, without opening a second connection from that connection's `Engine`. ¶ This change is also **backported** to: 0.7.10

  References: #2604

- **[engine]** The "reflect=True" argument to `MetaData` is deprecated. Please use the `MetaData.reflect()` method. ¶

sql

- **[sql] [feature]** The `Insert` construct now supports multi-valued inserts, that is, an INSERT that renders like "INSERT INTO table VALUES (...), (...), ...". Supported by Postgresql, SQLite, and MySQL. Big thanks to Idan Kamara for doing the legwork on this one.

  See also:

  Multiple-VALUES support for Insert

  ¶ References: #2623

- **[sql] [bug]** Fixed bug where using server_onupdate=<FetchedValuelDefaultClause> without passing the "for_update=True" flag would apply the default object to the server_default, blowing away whatever was there. The explicit for_update=True argument shouldn't be needed with this usage (especially since the documentation shows an example without it being used) so it is now arranged internally using a copy of the given default object, if the flag isn't set to what corresponds to that argument. ¶ This change is also **backported** to: 0.7.10

  References: #2631

- **[sql] [bug]** Fixed a regression caused by #2410 whereby a `CheckConstraint` would apply itself back to the original table during a `Table.tometadata()` operation, as it would parse the SQL expression for a parent table. The operation now copies the given expression to correspond to the new table. ¶ References: #2633

- **[sql] [bug]** Fixed bug whereby using a label_length on dialect that was smaller than the size of actual column identifiers would fail to render the columns correctly in a SELECT statement. ¶ References: #2610

- **[sql] [bug]** The `DECIMAL` type now honors the "precision" and "scale" arguments when rendering DDL. ¶ References: #2618

- **[sql] [bug]** Made an adjustment to the "boolean", (i.e. `__nonzero__`) evaluation of binary expressions, i.e. `x1 == x2`, such that the "auto-grouping" applied by `BinaryExpression` in some cases won't get in the way of this comparison. Previously, an expression like:

```
expr1 = mycolumn > 2
bool(expr1 == expr1)
```

Would evaluate as `False`, even though this is an identity comparison, because `mycolumn > 2` would be "grouped" before being placed into the *BinaryExpression*, thus changing its identity. *BinaryExpression* now keeps track of the "original" objects passed in. Additionally the `__nonzero__` method now only returns if the operator is == or `!=` - all others raise `TypeError`. ¶ References: #2621

- **[sql] [bug]** Fixed a gotcha where inadvertently calling list() on a *ColumnElement* would go into an endless loop, if *ColumnOperators.__getitem__()* were implemented. A new NotImplementedError is emitted via `__iter__()`. ¶
- **[sql] [bug]** Fixed bug in type_coerce() whereby typing information could be lost if the statement were used as a subquery inside of another statement, as well as other similar situations. Among other things, would cause typing information to be lost when the Oracle/mssql dialects would apply limit/offset wrappings. ¶ References: #2603
- **[sql] [bug]** Fixed bug whereby the ".key" of a Column wasn't being used when producing a "proxy" of the column against a selectable. This probably didn't occur in 0.7 since 0.7 doesn't respect the ".key" in a wider range of scenarios. ¶ References: #2597

postgresql

- **[postgresql] [feature]** *HSTORE* is now available in the Postgresql dialect. Will also use psycopg2's extensions if available. Courtesy Audrius Kažukauskas. ¶ References: #2606

sqlite

- **[sqlite] [bug]** More adjustment to this SQLite related issue which was released in 0.7.9, to intercept legacy SQLite quoting characters when reflecting foreign keys. In addition to intercepting double quotes, other quoting characters such as brackets, backticks, and single quotes are now also intercepted. ¶ This change is also **backported** to: 0.7.10

  References: #2568

mssql

- **[mssql] [feature]** Support for reflection of the "name" of primary key constraints added, courtesy Dave Moore. ¶ References: #2600
- **[mssql] [bug]** Fixed bug whereby using "key" with Column in conjunction with "schema" for the owning Table would fail to locate result rows due to the MSSQL dialect's "schema rendering" logic's failure to take .key into account. ¶ This change is also **backported** to: 0.7.10

oracle

- **[oracle] [bug]** Fixed table reflection for Oracle when accessing a synonym that refers to a DBLINK remote database; while the syntax has been present in the Oracle dialect for some time, up until now it has never been tested. The syntax has been tested against a sample database linking to itself, however there's still some uncertainty as to what should be used for the "owner" when querying the remote database for table information. Currently, the value of "username" from user_db_links is used to match the "owner". ¶ References: #2619
- **[oracle] [bug]** The Oracle LONG type, while an unbounded text type, does not appear to use the cx_Oracle.LOB type when result rows are returned, so the dialect has been repaired to exclude LONG from having cx_Oracle.LOB filtering applied. Also in 0.7.10. ¶ References: #2620

- **[oracle] [bug]** Repaired the usage of `.prepare()` in conjunction with cx_Oracle so that a return value of `False` will result in no call to `connection.commit()`, hence avoiding "no transaction" errors. Two-phase transactions have now been shown to work in a rudimental fashion with SQLAlchemy and cx_oracle, however are subject to caveats observed with the driver; check the documentation for details. Also in 0.7.10. ¶ References: #2611

firebird

- **[firebird] [bug]** Added missing import for "fdb" to the experimental "firebird+fdb" dialect. ¶ References: #2622

misc

- **[feature] [sybase]** Reflection support has been added to the Sybase dialect. Big thanks to Ben Trofatter for all the work developing and testing this. ¶ References: #1753
- **[feature] [pool]** The `Pool` will now log all connection.close() operations equally, including closes which occur for invalidated connections, detached connections, and connections beyond the pool capacity. ¶
- **[feature] [pool]** The `Pool` now consults the `Dialect` for functionality regarding how the connection should be "auto rolled back", as well as closed. This grants more control of transaction scope to the dialect, so that we will be better able to implement transactional workarounds like those potentially needed for pysqlite and cx_oracle. ¶ References: #2611
- **[feature] [pool]** Added new `PoolEvents.reset()` hook to capture the event before a connection is auto-rolled back, upon return to the pool. Together with `ConnectionEvents.rollback()` this allows all rollback events to be intercepted. ¶
- **[informix]** Some cruft regarding informix transaction handling has been removed, including a feature that would skip calling commit()/rollback() as well as some hardcoded isolation level assumptions on begin().. The status of this dialect is not well understood as we don't have any users working with it, nor any access to an Informix database. If someone with access to Informix wants to help test this dialect, please let us know. ¶

0.8.0b1

Released: October 30, 2012

general

- **[general] [removed]** The "sqlalchemy.exceptions" synonym for "sqlalchemy.exc" is removed fully. ¶ References: #2433
- **[general]** SQLAlchemy 0.8 now targets Python 2.5 and above. Python 2.4 is no longer supported. ¶

orm

- **[orm] [feature]** Major rewrite of relationship() internals now allow join conditions which include columns pointing to themselves within composite foreign keys. A new API for very specialized primaryjoin conditions is added, allowing conditions based on SQL functions, CAST, etc. to be handled by placing the annotation functions remote() and foreign() inline within the expression when necessary. Previous recipes using the semi-private _local_remote_pairs approach can be upgraded to this new approach.

  See also:

Rewritten relationship() mechanics

¶ References: #1401

- **[orm] [feature]** New standalone function with_polymorphic() provides the functionality of query.with_polymorphic() in a standalone form. It can be applied to any entity within a query, including as the target of a join in place of the "of_type()" modifier. ¶ References: #2333
- **[orm] [feature]** The of_type() construct on attributes now accepts aliased() class constructs as well as with_polymorphic constructs, and works with query.join(), any(), has(), and also eager loaders subqueryload(), joinedload(), contains_eager() ¶ References: #1106, #2438
- **[orm] [feature]** Improvements to event listening for mapped classes allows that unmapped classes can be specified for instance- and mapper-events. The established events will be automatically set up on subclasses of that class when the propagate=True flag is passed, and the events will be set up for that class itself if and when it is ultimately mapped. ¶ References: #2585
- **[orm] [feature]** The "deferred declarative reflection" system has been moved into the declarative extension itself, using the new DeferredReflection class. This class is now tested with both single and joined table inheritance use cases. ¶ References: #2485
- **[orm] [feature]** Added new core function "inspect()", which serves as a generic gateway to introspection into mappers, objects, others. The Mapper and InstanceState objects have been enhanced with a public API that allows inspection of mapped attributes, including filters for column-bound or relationship-bound properties, inspection of current object state, history of attributes, etc. ¶ References: #2208
- **[orm] [feature]** Calling rollback() within a session.begin_nested() will now only expire those objects that had net changes within the scope of that transaction, that is objects which were dirty or were modified on a flush. This allows the typical use case for begin_nested(), that of altering a small subset of objects, to leave in place the data from the larger enclosing set of objects that weren't modified in that sub-transaction. ¶ References: #2452
- **[orm] [feature]** Added utility feature Session.enable_relationship_loading(), supersedes relationship.load_on_pending. Both features should be avoided, however. ¶ References: #2372
- **[orm] [feature]** Added support for .info dictionary argument to column_property(), relationship(), composite(). All MapperProperty classes have an auto-creating .info dict available overall. ¶
- **[orm] [feature]** Adding/removing None from a mapped collection now generates attribute events. Previously, a None append would be ignored in some cases. Related to. ¶ References: #2229
- **[orm] [feature]** The presence of None in a mapped collection now raises an error during flush. Previously, None values in collections would be silently ignored. ¶ References: #2229
- **[orm] [feature]** The Query.update() method is now more lenient as to the table being updated. Plain Table objects are better supported now, and additional a joined-inheritance subclass may be used with update(); the subclass table will be the target of the update, and if the parent table is referenced in the WHERE clause, the compiler will call upon UPDATE..FROM syntax as allowed by the dialect to satisfy the WHERE clause. MySQL's multi-table update feature is also supported if columns are specified by object in the "values" dicitionary. PG's DELETE..USING is also not available in Core yet. ¶
- **[orm] [feature]** New session events after_transaction_create and after_transaction_end allows tracking of new SessionTransaction objects. If the object is inspected, can be used to determine when a session first becomes active and when it deactivates. ¶
- **[orm] [feature]** The Query can now load entity/scalar-mixed "tuple" rows that contain types which aren't hashable, by setting the flag "hashable=False" on the corresponding TypeEngine object in use. Custom types that return unhashable types (typically lists) can set this flag to False. ¶ References: #2592
- **[orm] [feature]** Query now "auto correlates" by default in the same way as select() does. Previously, a Query used as a subquery in another would require the correlate() method be called explicitly in order to correlate a table on the inside to the outside. As always, correlate(None) disables correlation. ¶ References: #2179

- **[orm] [feature]** The after_attach event is now emitted after the object is established in Session.new or Session.identity_map upon Session.add(), Session.merge(), etc., so that the object is represented in these collections when the event is called. Added before_attach event to accommodate use cases that need autoflush w pre-attached object. ¶ References: #2464
- **[orm] [feature]** The Session will produce warnings when unsupported methods are used inside the "execute" portion of the flush. These are the familiar methods add(), delete(), etc. as well as collection and related-object manipulations, as called within mapper-level flush events like after_insert(), after_update(), etc. It's been prominently documented for a long time that SQLAlchemy cannot guarantee results when the Session is manipulated within the execution of the flush plan, however users are still doing it, so now there's a warning. Maybe someday the Session will be enhanced to support these operations inside of the flush, but for now, results can't be guaranteed. ¶
- **[orm] [feature]** ORM entities can be passed to the core select() construct as well as to the select_from(), correlate(), and correlate_except() methods of select(), where they will be unwrapped into selectables. ¶ References: #2245
- **[orm] [feature]** Some support for auto-rendering of a relationship join condition based on the mapped attribute, with usage of core SQL constructs. E.g. select([SomeClass]).where(SomeClass.somerelationship) would render SELECT from "someclass" and use the primaryjoin of "somerelationship" as the WHERE clause. This changes the previous meaning of "SomeClass.somerelationship" when used in a core SQL context; previously, it would "resolve" to the parent selectable, which wasn't generally useful. Also works with query.filter(). Related to. ¶ References: #2245
- **[orm] [feature]** The registry of classes in declarative_base() is now a WeakValueDictionary. So subclasses of "Base" that are dereferenced will be garbage collected, *if they are not referred to by any other mappers/superclass mappers*. See the next note for this ticket. ¶ References: #2526
- **[orm] [feature]** Conflicts between columns on single-inheritance declarative subclasses, with or without using a mixin, can be resolved using a new @declared_attr usage described in the documentation. ¶ References: #2472
- **[orm] [feature]** declared_attr can now be used on non-mixin classes, even though this is generally only useful for single-inheritance subclass column conflict resolution. ¶ References: #2472
- **[orm] [feature]** declared_attr can now be used with attributes that are not Column or MapperProperty; including any user-defined value as well as association proxy objects. ¶ References: #2517
- **[orm] [feature]** *Very limited* support for inheriting mappers to be GC'ed when the class itself is deferenced. The mapper must not have its own table (i.e. single table inh only) without polymorphic attributes in place. This allows for the use case of creating a temporary subclass of a declarative mapped class, with no table or mapping directives of its own, to be garbage collected when dereferenced by a unit test. ¶ References: #2526
- **[orm] [feature]** Declarative now maintains a registry of classes by string name as well as by full module-qualified name. Multiple classes with the same name can now be looked up based on a module-qualified string within relationship(). Simple class name lookups where more than one class shares the same name now raises an informative error message. ¶ References: #2338
- **[orm] [feature]** Can now provide class-bound attributes that override columns which are of any non-ORM type, not just descriptors. ¶ References: #2535
- **[orm] [feature]** Added with_labels and reduce_columns keyword arguments to Query.subquery(), to provide two alternate strategies for producing queries with uniquely- named columns. . ¶ References: #1729
- **[orm] [feature]** A warning is emitted when a reference to an instrumented collection is no longer associated with the parent class due to expiration/attribute refresh/collection replacement, but an append or remove operation is received on the now-detached collection. ¶ References: #2476
- **[orm] [removed]** The legacy "mutable" system of the ORM, including the MutableType class as well as the mutable=True flag on PickleType and postgresql.ARRAY has been removed. In-place mutations are detected by

the ORM using the sqlalchemy.ext.mutable extension, introduced in 0.7. The removal of MutableType and associated constructs removes a great deal of complexity from SQLAlchemy's internals. The approach performed poorly as it would incur a scan of the full contents of the Session when in use. ¶ References: #2442

- **[orm] [removed]** Deprecated identifiers removed:
  - allow_null_pks mapper() argument (use allow_partial_pks)
  - _get_col_to_prop() mapper method (use get_property_by_column())
  - dont_load argument to Session.merge() (use load=True)
  - sqlalchemy.orm.shard module (use sqlalchemy.ext.horizontal_shard)

  ¶
- **[orm] [bug]** ORM will perform extra effort to determine that an FK dependency between two tables is not significant during flush if the tables are related via joined inheritance and the FK dependency is not part of the inherit_condition, saves the user a use_alter directive. ¶ References: #2527
- **[orm] [bug]** The instrumentation events class_instrument(), class_uninstrument(), and attribute_instrument() will now fire off only for descendant classes of the class assigned to listen(). Previously, an event listener would be assigned to listen for all classes in all cases regardless of the "target" argument passed. ¶ References: #2590
- **[orm] [bug]** with_polymorphic() produces JOINs in the correct order and with correct inheriting tables in the case of sending multi-level subclasses in an arbitrary order or with intermediary classes missing. ¶ References: #1900
- **[orm] [bug]** Improvements to joined/subquery eager loading dealing with chains of subclass entities sharing a common base, with no specific "join depth" provided. Will chain out to each subclass mapper individually before detecting a "cycle", rather than considering the base class to be the source of the "cycle". ¶ References: #2481
- **[orm] [bug]** The "passive" flag on Session.is_modified() no longer has any effect. is_modified() in all cases looks only at local in-memory modified flags and will not emit any SQL or invoke loader callables/initializers. ¶ References: #2320
- **[orm] [bug]** The warning emitted when using delete-orphan cascade with one-to-many or many-to-many without single-parent=True is now an error. The ORM would fail to function subsequent to this warning in any case. ¶ References: #2405
- **[orm] [bug]** Lazy loads emitted within flush events such as before_flush(), before_update(), etc. will now function as they would within non-event code, regarding consideration of the PK/FK values used in the lazy-emitted query. Previously, special flags would be established that would cause lazy loads to load related items based on the "previous" value of the parent PK/FK values specifically when called upon within a flush; the signal to load in this way is now localized to where the unit of work actually needs to load that way. Note that the UOW does sometimes load these collections before the before_update() event is called, so the usage of "passive_updates" or not can affect whether or not a collection will represent the "old" or "new" data, when accessed within a flush event, based on when the lazy load was emitted. The change is backwards incompatible in the exceedingly small chance that user event code depended on the old behavior. ¶ References: #2350
- **[orm] [bug]** Continuing regarding extra state post-flush due to event listeners; any states that are marked as "dirty" from an attribute perspective, usually via column-attribute set events within after_insert(), after_update(), etc., will get the "history" flag reset in all cases, instead of only those instances that were part of the flush. This has the effect that this "dirty" state doesn't carry over after the flush and won't result in UPDATE statements. A warning is emitted to this effect; the set_committed_state() method can be used to assign attributes on objects without producing history events. ¶ References: #2582, #2566
- **[orm] [bug]** Fixed a disconnect that slowly evolved between a @declared_attr Column and a directly-defined Column on a mixin. In both cases, the Column will be applied to the declared class' table, but not to that of a

joined inheritance subclass. Previously, the directly-defined Column would be placed on both the base and the sub table, which isn't typically what's desired. ¶ References: #2565

- **[orm] [bug]** Declarative can now propagate a column declared on a single-table inheritance subclass up to the parent class' table, when the parent class is itself mapped to a join() or select() statement, directly or via joined inheritance, and not just a Table. ¶ References: #2549
- **[orm] [bug]** An error is emitted when uselist=False is combined with a "dynamic" loader. This is a warning in 0.7.9. ¶
- **[orm] [moved]** The InstrumentationManager interface and the entire related system of alternate class implementation is now moved out to sqlalchemy.ext.instrumentation. This is a seldom used system that adds significant complexity and overhead to the mechanics of class instrumentation. The new architecture allows it to remain unused until InstrumentationManager is actually imported, at which point it is bootstrapped into the core. ¶

engine

- **[engine] [feature]** Connection event listeners can now be associated with individual Connection objects, not just Engine objects. ¶ References: #2511
- **[engine] [feature]** The before_cursor_execute event fires off for so-called "_cursor_execute" events, which are usually special-case executions of primary-key bound sequences and default-generation SQL phrases that invoke separately when RETURNING is not used with INSERT. ¶ References: #2459
- **[engine] [feature]** The libraries used by the test suite have been moved around a bit so that they are part of the SQLAlchemy install again. In addition, a new suite of tests is present in the new sqlalchemy.testing.suite package. This is an under-development system that hopes to provide a universal testing suite for external dialects. Dialects which are maintained outside of SQLAlchemy can use the new test fixture as the framework for their own tests, and will get for free a "compliance" suite of dialect-focused tests, including an improved "requirements" system where specific capabilities and features can be enabled or disabled for testing. ¶
- **[engine] [feature]** Added a new system for registration of new dialects in-process without using an entrypoint. See the docs for "Registering New Dialects". ¶ References: #2462
- **[engine] [feature]** The "required" flag is set to True by default, if not passed explicitly, on bindparam() if the "value" or "callable" parameters are not passed. This will cause statement execution to check for the parameter being present in the final collection of bound parameters, rather than implicitly assigning None. ¶ References: #2556
- **[engine] [feature]** Various API tweaks to the "dialect" API to better support highly specialized systems such as the Akiban database, including more hooks to allow an execution context to access type processors. ¶
- **[engine] [feature]** Inspector.get_primary_keys() is deprecated; use Inspector.get_pk_constraint(). Courtesy Diana Clarke. ¶ References: #2422
- **[engine] [feature]** New C extension module "utils" has been added for additional function speedups as we have time to implement. ¶
- **[engine] [bug]** The Inspector.get_table_names() order_by="foreign_key" feature now sorts tables by dependee first, to be consistent with util.sort_tables and metadata.sorted_tables. ¶
- **[engine] [bug]** Fixed bug whereby if a database restart affected multiple connections, each connection would individually invoke a new disposal of the pool, even though only one disposal is needed. ¶ References: #2522
- **[engine] [bug]** The names of the columns on the .c. attribute of a select().apply_labels() is now based on <tablename>_<colkey> instead of <tablename>_<colname>, for those columns that have a distinctly named .key. ¶ References: #2397

- **[engine] [bug]** The autoload_replace flag on Table, when False, will cause any reflected foreign key constraints which refer to already-declared columns to be skipped, assuming that the in-Python declared column will take over the task of specifying in-Python ForeignKey or ForeignKeyConstraint declarations. ¶
- **[engine] [bug]** The ResultProxy methods inserted_primary_key, last_updated_params(), last_inserted_params(), postfetch_cols(), prefetch_cols() all assert that the given statement is a compiled construct, and is an insert() or update() statement as is appropriate, else raise InvalidRequestError. ¶ References: #2498
- **[engine]** ResultProxy.last_inserted_ids is removed, replaced by inserted_primary_key. ¶

sql

- **[sql] [feature]** Added a new method `Engine.execution_options()` to `Engine`. This method works similarly to `Connection.execution_options()` in that it creates a copy of the parent object which will refer to the new set of options. The method can be used to build sharding schemes where each engine shares the same underlying pool of connections. The method has been tested against the horizontal shard recipe in the ORM as well.

  See also:

  `Engine.execution_options()`

  ¶
- **[sql] [feature]** Major rework of operator system in Core, to allow redefinition of existing operators as well as addition of new operators at the type level. New types can be created from existing ones which add or redefine operations that are exported out to column expressions, in a similar manner to how the ORM has allowed comparator_factory. The new architecture moves this capability into the Core so that it is consistently usable in all cases, propagating cleanly using existing type propagation behavior. ¶ References: #2547
- **[sql] [feature]** To complement, types can now provide "bind expressions" and "column expressions" which allow compile-time injection of SQL expressions into statements on a per-column or per-bind level. This is to suit the use case of a type which needs to augment bind- and result- behavior at the SQL level, as opposed to in the Python level. Allows for schemes like transparent encryption/ decryption, usage of Postgis functions, etc. ¶ References: #1534, #2547
- **[sql] [feature]** The Core oeprator system now includes the *getitem* operator, i.e. the bracket operator in Python. This is used at first to provide index and slice behavior to the Postgresql ARRAY type, and also provides a hook for end-user definition of custom __getitem__ schemes which can be applied at the type level as well as within ORM-level custom operator schemes. *lshift* (<<) and *rshift* (>>) are also supported as optional operators.

  Note that this change has the effect that descriptor-based __getitem__ schemes used by the ORM in conjunction with synonym() or other "descriptor-wrapped" schemes will need to start using a custom comparator in order to maintain this behavior. ¶
- **[sql] [feature]** Revised the rules used to determine the operator precedence for the user-defined operator, i.e. that granted using the `op()` method. Previously, the smallest precedence was applied in all cases, now the default precedence is zero, lower than all operators except "comma" (such as, used in the argument list of a `func` call) and "AS", and is also customizable via the "precedence" argument on the `op()` method. ¶ References: #2537
- **[sql] [feature]** Added "collation" parameter to all String types. When present, renders as COLLATE <collation>. This to support the COLLATE keyword now supported by several databases including MySQL, SQLite, and Postgresql. ¶ References: #2276
- **[sql] [feature]** Custom unary operators can now be used by combining operators.custom_op() with UnaryExpression(). ¶

- **[sql] [feature]** Enhanced GenericFunction and func.* to allow for user-defined GenericFunction subclasses to be available via the func.* namespace automatically by classname, optionally using a package name, as well as with the ability to have the rendered name different from the identified name in func.*. ¶
- **[sql] [feature]** The cast() and extract() constructs will now be produced via the func.* accessor as well, as users naturally try to access these names from func.* they might as well do what's expected, even though the returned object is not a FunctionElement. ¶ References: #2562
- **[sql] [feature]** The Inspector object can now be acquired using the new inspect() service, part of ¶ References: #2208
- **[sql] [feature]** The column_reflect event now accepts the Inspector object as the first argument, preceding "table". Code which uses the 0.7 version of this very new event will need modification to add the "inspector" object as the first argument. ¶ References: #2418
- **[sql] [feature]** The behavior of column targeting in result sets is now case sensitive by default. SQLAlchemy for many years would run a case-insensitive conversion on these values, probably to alleviate early case sensitivity issues with dialects like Oracle and Firebird. These issues have been more cleanly solved in more modern versions so the performance hit of calling lower() on identifiers is removed. The case insensitive comparisons can be re-enabled by setting "case_insensitive=False" on create_engine(). ¶ References: #2423
- **[sql] [feature]** The "unconsumed column names" warning emitted when keys are present in insert.values() or update.values() that aren't in the target table is now an exception. ¶ References: #2415
- **[sql] [feature]** Added "MATCH" clause to ForeignKey, ForeignKeyConstraint, courtesy Ryan Kelly. ¶ References: #2502
- **[sql] [feature]** Added support for DELETE and UPDATE from an alias of a table, which would assumedly be related to itself elsewhere in the query, courtesy Ryan Kelly. ¶ References: #2507
- **[sql] [feature]** select() features a correlate_except() method, auto correlates all selectables except those passed. ¶
- **[sql] [feature]** The prefix_with() method is now available on each of select(), insert(), update(), delete(), all with the same API, accepting multiple prefix calls, as well as a "dialect name" so that the prefix can be limited to one kind of dialect. ¶ References: #2431
- **[sql] [feature]** Added reduce_columns() method to select() construct, replaces columns inline using the util.reduce_columns utility function to remove equivalent columns. reduce_columns() also adds "with_only_synonyms" to limit the reduction just to those columns which have the same name. The deprecated fold_equivalents() feature is removed. ¶ References: #1729
- **[sql] [feature]** Reworked the startswith(), endswith(), contains() operators to do a better job with negation (NOT LIKE), and also to assemble them at compilation time so that their rendered SQL can be altered, such as in the case for Firebird STARTING WITH ¶ References: #2470
- **[sql] [feature]** Added a hook to the system of rendering CREATE TABLE that provides access to the render for each Column individually, by constructing a @compiles function against the new schema.CreateColumn construct. ¶ References: #2463
- **[sql] [feature]** "scalar" selects now have a WHERE method to help with generative building. Also slight adjustment regarding how SS "correlates" columns; the new methodology no longer applies meaning to the underlying Table column being selected. This improves some fairly esoteric situations, and the logic that was there didn't seem to have any purpose. ¶
- **[sql] [feature]** An explicit error is raised when a ForeignKeyConstraint() that was constructed to refer to multiple remote tables is first used. ¶ References: #2455
- **[sql] [feature]** Added `ColumnOperators.notin_()`, `ColumnOperators.notlike()`, `ColumnOperators.notilike()` to `ColumnOperators`. ¶ References: #2580

- **[sql] [changed]** Most classes in expression.sql are no longer preceded with an underscore, i.e. Label, SelectBase, Generative, CompareMixin. _BindParamClause is also renamed to BindParameter. The old underscore names for these classes will remain available as synonyms for the foreseeable future. ¶
- **[sql] [removed]** The long-deprecated and non-functional `assert_unicode` flag on *create_engine()* as well as *String* is removed. ¶
- **[sql] [bug]** Fixed bug where keyword arguments passed to `Compiler.process()` wouldn't get propagated to the column expressions present in the columns clause of a SELECT statement. In particular this would come up when used by custom compilation schemes that relied upon special flags. ¶ References: #2593
- **[sql] [bug] [orm]** The auto-correlation feature of *select()*, and by proxy that of *Query*, will not take effect for a SELECT statement that is being rendered directly in the FROM list of the enclosing SELECT. Correlation in SQL only applies to column expressions such as those in the WHERE, ORDER BY, columns clause. ¶ References: #2595
- **[sql] [bug]** A tweak to column precedence which moves the "concat" and "match" operators to be the same as that of "is", "like", and others; this helps with parenthesization rendering when used in conjunction with "IS". ¶ References: #2564
- **[sql] [bug]** Applying a column expression to a select statement using a label with or without other modifying constructs will no longer "target" that expression to the underlying Column; this affects ORM operations that rely upon Column targeting in order to retrieve results. That is, a query like query(User.id, User.id.label('foo')) will now track the value of each "User.id" expression separately instead of munging them together. It is not expected that any users will be impacted by this; however, a usage that uses select() in conjunction with query.from_statement() and attempts to load fully composed ORM entities may not function as expected if the select() named Column objects with arbitrary .label() names, as these will no longer target to the Column objects mapped by that entity. ¶ References: #2591
- **[sql] [bug]** Fixes to the interpretation of the Column "default" parameter as a callable to not pass ExecutionContext into a keyword argument parameter. ¶ References: #2520
- **[sql] [bug]** All of UniqueConstraint, ForeignKeyConstraint, CheckConstraint, and PrimaryKeyConstraint will attach themselves to their parent table automatically when they refer to a Table-bound Column object directly (i.e. not just string column name), and refer to one and only one Table. Prior to 0.8 this behavior occurred for UniqueConstraint and PrimaryKeyConstraint, but not ForeignKeyConstraint or CheckConstraint. ¶ References: #2410
- **[sql] [bug]** TypeDecorator now includes a generic repr() that works in terms of the "impl" type by default. This is a behavioral change for those TypeDecorator classes that specify a custom __init__ method; those types will need to re-define __repr__() if they need __repr__() to provide a faithful constructor representation. ¶ References: #2594
- **[sql] [bug]** column.label(None) now produces an anonymous label, instead of returning the column object itself, consistent with the behavior of label(column, None). ¶ References: #2168
- **[sql] [change]** The Text() type renders the length given to it, if a length was specified. ¶

postgresql

- **[postgresql] [feature]** postgresql.ARRAY features an optional "dimension" argument, will assign a specific number of dimensions to the array which will render in DDL as ARRAY[][]..., also improves performance of bind/result processing. ¶ References: #2441
- **[postgresql] [feature]** postgresql.ARRAY now supports indexing and slicing. The Python [] operator is available on all SQL expressions that are of type ARRAY; integer or simple slices can be passed. The slices can also be used on the assignment side in the SET clause of an UPDATE statement by passing them into Update.values(); see the docs for examples. ¶

- **[postgresql] [feature]** Added new "array literal" construct postgresql.array(). Basically a "tuple" that renders as ARRAY[1,2,3]. ¶
- **[postgresql] [feature]** Added support for the Postgresql ONLY keyword, which can appear corresponding to a table in a SELECT, UPDATE, or DELETE statement. The phrase is established using with_hint(). Courtesy Ryan Kelly ¶ References: #2506
- **[postgresql] [feature]** The "ischema_names" dictionary of the Postgresql dialect is "unofficially" customizable. Meaning, new types such as PostGIS types can be added into this dictionary, and the PG type reflection code should be able to handle simple types with variable numbers of arguments. The functionality here is "unofficial" for three reasons:

  1. this is not an "official" API. Ideally an "official" API would allow custom type-handling callables at the dialect or global level in a generic way.
  2. This is only implemented for the PG dialect, in particular because PG has broad support for custom types vs. other database backends. A real API would be implemented at the default dialect level.
  3. The reflection code here is only tested against simple types and probably has issues with more compositional types.

  patch courtesy Éric Lemoine. ¶

mysql

- **[mysql] [feature]** Added TIME type to mysql dialect, accepts "fst" argument which is the new "fractional seconds" specifier for recent MySQL versions. The datatype will interpret a microseconds portion received from the driver, however note that at this time most/all MySQL DBAPIs do not support returning this value. ¶ References: #2534
- **[mysql] [bug]** Dialect no longer emits expensive server collations query, as well as server casing, on first connect. These functions are still available as semi-private. ¶ References: #2404

sqlite

- **[sqlite] [feature]** the SQLite date and time types have been overhauled to support a more open ended format for input and output, using name based format strings and regexps. A new argument "microseconds" also provides the option to omit the "microseconds" portion of timestamps. Thanks to Nathan Wright for the work and tests on this. ¶ References: #2363
- **[sqlite]** Added `types.NCHAR`, `types.NVARCHAR` to the SQLite dialect's list of recognized type names for reflection. SQLite returns the name given to a type as the name returned. ¶ References: rc3addcc9ffad

mssql

- **[mssql] [feature]** SQL Server dialect can be given database-qualified schema names, i.e. "schema='mydatabase.dbo'"; reflection operations will detect this, split the schema among the "." to get the owner separately, and emit a "USE mydatabase" statement before reflecting targets within the "dbo" owner; the existing database returned from DB_NAME() is then restored. ¶
- **[mssql] [feature]** updated support for the mxodbc driver; mxodbc 3.2.1 is recommended for full compatibility. ¶
- **[mssql] [bug]** removed legacy behavior whereby a column comparison to a scalar SELECT via == would coerce to an IN with the SQL server dialect. This is implicit behavior which fails in other scenarios so is removed. Code which relies on this needs to be modified to use column.in_(select) explicitly. ¶ References: #2277

oracle

- **[oracle] [feature]** The types of columns excluded from the setinputsizes() set can be customized by sending a list of string DBAPI type names to exclude, using the exclude_setinputsizes dialect parameter. This list was previously fixed. The list also now defaults to STRING, UNICODE, removing CLOB, NCLOB from the list. ¶ References: #2561
- **[oracle] [bug]** Quoting information is now passed along from a Column with quote=True when generating a same-named bound parameter to the bindparam() object, as is the case in generated INSERT and UPDATE statements, so that unknown reserved names can be fully supported. ¶ References: #2437
- **[oracle] [bug]** The CreateIndex construct in Oracle will now schema-qualify the name of the index to be that of the parent table. Previously this name was omitted which apparently creates the index in the default schema, rather than that of the table. ¶

firebird

- **[firebird] [feature]** The "startswith()" operator renders as "STARTING WITH", "~startswith()" renders as "NOT STARTING WITH", using FB's more efficient operator. ¶ References: #2470
- **[firebird] [feature]** An experimental dialect for the fdb driver is added, but is untested as I cannot get the fdb package to build. ¶ References: #2504
- **[firebird] [bug]** CompileError is raised when VARCHAR with no length is attempted to be emitted, same way as MySQL. ¶ References: #2505
- **[firebird] [bug]** Firebird now uses strict "ansi bind rules" so that bound parameters don't render in the columns clause of a statement - they render literally instead. ¶
- **[firebird] [bug]** Support for passing datetime as date when using the DateTime type with Firebird; other dialects support this. ¶

misc

- **[feature] [access]** the MS Access dialect has been moved to its own project on Bitbucket, taking advantage of the new SQLAlchemy dialect compliance suite. The dialect is still in very rough shape and probably not ready for general use yet, however it does have *extremely* rudimental functionality now. https://bitbucket.org/zzzeek/sqlalchemy-access ¶
- **[moved] [maxdb]** The MaxDB dialect, which hasn't been functional for several years, is moved out to a pending bitbucket project, https://bitbucket.org/zzzeek/sqlalchemy-maxdb. ¶
- **[examples]** The Beaker caching example has been converted to use dogpile.cache. This is a new caching library written by the same creator of Beaker's caching internals, and represents a vastly improved, simplified, and modernized system of caching.

  See also:

  Dogpile Caching

  ¶ References: #2589

6.2.5 0.7 Changelog

0.7.11

no release date

orm

- **[orm] [bug]** Fixed bug where list instrumentation would fail to represent a setslice of `[0:0]` correctly, which in particular could occur when using `insert(0, item)` with the association proxy. Due to some quirk in Python collections, the issue was much more likely with Python 3 rather than 2. ¶ References: #2807
- **[orm] [bug]** Fixed bug when a query of the form: `query(SubClass).options(subqueryload(Baseclass.attrna` where `SubClass` is a joined inh of `BaseClass`, would fail to apply the `JOIN` inside the subquery on the attribute load, producing a cartesian product. The populated results still tended to be correct as additional rows are just ignored, so this issue may be present as a performance degradation in applications that are otherwise working correctly. ¶ References: #2699
- **[orm] [bug]** Fixed bug in unit of work whereby a joined-inheritance subclass could insert the row for the "sub" table before the parent table, if the two tables had no ForeignKey constraints set up between them. ¶ References: #2689
- **[orm] [bug]** Improved the error message emitted when a "backref loop" is detected, that is when an attribute event triggers a bidirectional assignment between two other attributes with no end. This condition can occur not just when an object of the wrong type is assigned, but also when an attribute is mis-configured to backref into an existing backref pair. ¶ References: #2674
- **[orm] [bug]** A warning is emitted when a MapperProperty is assigned to a mapper that replaces an existing property, if the properties in question aren't plain column-based properties. Replacement of relationship properties is rarely (ever?) what is intended and usually refers to a mapper mis-configuration. This will also warn if a backref configures itself on top of an existing one in an inheritance relationship (which is an error in 0.8). ¶ References: #2674

engine

- **[engine] [bug]** The regexp used by the `make_url()` function now parses ipv6 addresses, e.g. surrounded by brackets. ¶ References: #2851

sql

- **[sql] [bug]** Fixed regression dating back to 0.7.9 whereby the name of a CTE might not be properly quoted if it was referred to in multiple FROM clauses. ¶ References: #2801
- **[sql] [bug] [cte]** Fixed bug in common table expression system where if the CTE were used only as an `alias()` construct, it would not render using the WITH keyword. ¶ References: #2783
- **[sql] [bug]** Fixed bug in `CheckConstraint` DDL where the "quote" flag from a `Column` object would not be propagated. ¶ References: #2784

postgresql

- **[postgresql] [feature]** Added support for Postgresql's traditional SUBSTRING function syntax, renders as "SUBSTRING(x FROM y FOR z)" when regular `func.substring()` is used. Courtesy Gunnlaugur Þór

Briem. ¶ References: #2676

mysql

- **[mysql] [bug]** Updates to MySQL reserved words for versions 5.5, 5.6, courtesy Hanno Schlichting. ¶ References: #2791

misc

- **[bug] [tests]** Fixed an import of "logging" in test_execute which was not working on some linux platforms. ¶ References: #2669, pull request 41

0.7.10

Released: Thu Feb 7 2013

orm

- **[orm] [bug]** Fixed potential memory leak which could occur if an arbitrary number of `sessionmaker` objects were created. The anonymous subclass created by the sessionmaker, when dereferenced, would not be garbage collected due to remaining class-level references from the event package. This issue also applies to any custom system that made use of ad-hoc subclasses in conjunction with an event dispatcher. ¶ References: #2650
- **[orm] [bug]** `Query.merge_result()` can now load rows from an outer join where an entity may be `None` without throwing an error. ¶ References: #2640
- **[orm] [bug]** The `MutableComposite` type did not allow for the `MutableBase.coerce()` method to be used, even though the code seemed to indicate this intent, so this now works and a brief example is added. As a side-effect, the mechanics of this event handler have been changed so that new `MutableComposite` types no longer add per-type global event handlers. Also in 0.8.0b2. ¶ References: #2624
- **[orm] [bug]** Fixed Session accounting bug whereby replacing a deleted object in the identity map with another object of the same primary key would raise a "conflicting state" error on rollback(), if the replaced primary key were established either via non-unitofwork-established INSERT statement or by primary key switch of another instance. ¶ References: #2583

engine

- **[engine] [bug]** Fixed `MetaData.reflect()` to correctly use the given `Connection`, if given, without opening a second connection from that connection's `Engine`. ¶ References: #2604

sql

- **[sql] [bug]** Backported adjustment to `__repr__` for `TypeDecorator` to 0.7, allows `PickleType` to produce a clean `repr()` to help with Alembic. ¶ References: #2594, #2584
- **[sql] [bug]** Fixed bug where `Table.tometadata()` would fail if a `Column` had both a foreign key as well as an alternate ".key" name for the column. ¶ References: #2643

- **[sql] [bug]** Fixed bug where using server_onupdate=<FetchedValuelDefaultClause> without passing the "for_update=True" flag would apply the default object to the server_default, blowing away whatever was there. The explicit for_update=True argument shouldn't be needed with this usage (especially since the documentation shows an example without it being used) so it is now arranged internally using a copy of the given default object, if the flag isn't set to what corresponds to that argument. ¶ References: #2631
- **[sql] [gae] [mysql]** Added a conditional import to the `gaerdbms` dialect which attempts to import rdbms_apiproxy vs. rdbms_googleapi to work on both dev and production platforms. Also now honors the `instance` attribute. Courtesy Sean Lynch. Also backported enhancements to allow username/password as well as fixing error code interpretation from 0.8. ¶ References: #2649

mysql

- **[mysql] [feature]** Added "raise_on_warnings" flag to OurSQL dialect. ¶ References: #2523
- **[mysql] [feature]** Added "read_timeout" flag to MySQLdb dialect. ¶ References: #2554

sqlite

- **[sqlite] [bug]** More adjustment to this SQLite related issue which was released in 0.7.9, to intercept legacy SQLite quoting characters when reflecting foreign keys. In addition to intercepting double quotes, other quoting characters such as brackets, backticks, and single quotes are now also intercepted. ¶ References: #2568

mssql

- **[mssql] [bug]** Fixed bug whereby using "key" with Column in conjunction with "schema" for the owning Table would fail to locate result rows due to the MSSQL dialect's "schema rendering" logic's failure to take .key into account. ¶
- **[mssql] [bug]** Added a Py3K conditional around unnecessary .decode() call in mssql information schema, fixes reflection in Py3k. ¶ References: #2638

oracle

- **[oracle] [bug]** The Oracle LONG type, while an unbounded text type, does not appear to use the cx_Oracle.LOB type when result rows are returned, so the dialect has been repaired to exclude LONG from having cx_Oracle.LOB filtering applied. ¶ References: #2620
- **[oracle] [bug]** Repaired the usage of `.prepare()` in conjunction with cx_Oracle so that a return value of `False` will result in no call to `connection.commit()`, hence avoiding "no transaction" errors. Two-phase transactions have now been shown to work in a rudimental fashion with SQLAlchemy and cx_oracle, however are subject to caveats observed with the driver; check the documentation for details. ¶ References: #2611
- **[oracle] [bug]** changed the list of cx_oracle types that are excluded from the setinputsizes() step to only include STRING and UNICODE; CLOB and NCLOB are removed. This is to work around cx_oracle behavior which is broken for the executemany() call. In 0.8, this same change is applied however it is also configurable via the exclude_setinputsizes argument. ¶ References: #2561

0.7.9

Released: Mon Oct 01 2012

orm

- **[orm] [bug]** Fixed bug mostly local to new AbstractConcreteBase helper where the "type" attribute from the superclass would not be overridden on the subclass to produce the "reserved for base" error message, instead placing a do-nothing attribute there. This was inconsistent vs. using ConcreteBase as well as all the behavior of classical concrete mappings, where the "type" column from the polymorphic base would be explicitly disabled on subclasses, unless overridden explicitly. ¶
- **[orm] [bug]** A warning is emitted when lazy='dynamic' is combined with uselist=False. This is an exception raise in 0.8. ¶
- **[orm] [bug]** Fixed bug whereby user error in related-object assignment could cause recursion overflow if the assignment triggered a backref of the same name as a bi-directional attribute on the incorrect class to the same target. An informative error is raised now. ¶
- **[orm] [bug]** Fixed bug where incorrect type information would be passed when the ORM would bind the "version" column, when using the "version" feature. Tests courtesy Daniel Miller. ¶ References: #2539
- **[orm] [bug]** Extra logic has been added to the "flush" that occurs within Session.commit(), such that the extra state added by an after_flush() or after_flush_postexec() hook is also flushed in a subsequent flush, before the "commit" completes. Subsequent calls to flush() will continue until the after_flush hooks stop adding new state. An "overflow" counter of 100 is also in place, in the event of a broken after_flush() hook adding new content each time. ¶ References: #2566

engine

- **[engine] [feature]** Dramatic improvement in memory usage of the event system; instance-level collections are no longer created for a particular type of event until instance-level listeners are established for that event. ¶ References: #2516
- **[engine] [bug]** Fixed bug whereby a disconnect detect + dispose that occurs when the QueuePool has threads waiting for connections would leave those threads waiting for the duration of the timeout on the old pool (or indefinitely if timeout was disabled). The fix now notifies those waiters with a special exception case and has them move onto the new pool. ¶ References: #2522
- **[engine] [bug]** Added gaerdbms import to mysql/__init__.py, the absence of which was preventing the new GAE dialect from being loaded. ¶ References: #2529
- **[engine] [bug]** Fixed cextension bug whereby the "ambiguous column error" would fail to function properly if the given index were a Column object and not a string. Note there are still some column-targeting issues here which are fixed in 0.8. ¶ References: #2553
- **[engine] [bug]** Fixed the repr() of Enum to include the "name" and "native_enum" flags. Helps Alembic autogenerate. ¶

sql

- **[sql] [bug]** Fixed the DropIndex construct to support an Index associated with a Table in a remote schema. ¶ References: #2571
- **[sql] [bug]** Fixed bug in over() construct whereby passing an empty list for either partition_by or order_by, as opposed to None, would fail to generate correctly. Courtesy Gunnlaugur Þór Briem. ¶ References: #2574
- **[sql] [bug]** Fixed CTE bug whereby positional bound parameters present in the CTEs themselves would corrupt the overall ordering of bound parameters. This primarily affected SQL Server as the platform with positional binds + CTE support. ¶ References: #2521

- **[sql] [bug]** Fixed more un-intuitivenesses in CTEs which prevented referring to a CTE in a union of itself without it being aliased. CTEs now render uniquely on name, rendering the outermost CTE of a given name only - all other references are rendered just as the name. This even includes other CTE/SELECTs that refer to different versions of the same CTE object, such as a SELECT or a UNION ALL of that SELECT. We are somewhat loosening the usual link between object identity and lexical identity in this case. A true name conflict between two unrelated CTEs now raises an error. ¶
- **[sql] [bug]** quoting is applied to the column names inside the WITH RECURSIVE clause of a common table expression according to the quoting rules for the originating Column. ¶ References: #2512
- **[sql] [bug]** Fixed regression introduced in 0.7.6 whereby the FROM list of a SELECT statement could be incorrect in certain "clone+replace" scenarios. ¶ References: #2518
- **[sql] [bug]** Fixed bug whereby usage of a UNION or similar inside of an embedded subquery would interfere with result-column targeting, in the case that a result-column had the same ultimate name as a name inside the embedded UNION. ¶ References: #2552
- **[sql] [bug]** Fixed a regression since 0.6 regarding result-row targeting. It should be possible to use a select() statement with string based columns in it, that is select(['id', 'name']).select_from('mytable'), and have this statement be targetable by Column objects with those names; this is the mechanism by which query(MyClass).from_statement(some_statement) works. At some point the specific case of using select(['id']), which is equivalent to select([literal_column('id')]), stopped working here, so this has been re-instated and of course tested. ¶ References: #2558
- **[sql] [bug]** Added missing operators is_(), isnot() to the ColumnOperators base, so that these long-available operators are present as methods like all the other operators. ¶ References: #2544

postgresql

- **[postgresql] [bug]** Columns in reflected primary key constraint are now returned in the order in which the constraint itself defines them, rather than how the table orders them. Courtesy Gunnlaugur Þór Briem.. ¶ References: #2531
- **[postgresql] [bug]** Added 'terminating connection' to the list of messages we use to detect a disconnect with PG, which appears to be present in some versions when the server is restarted. ¶ References: #2570

mysql

- **[mysql] [bug]** Updated mysqlconnector interface to use updated "client flag" and "charset" APIs, courtesy David McNelis. ¶

sqlite

- **[sqlite] [feature]** Added support for the localtimestamp() SQL function implemented in SQLite, courtesy Richard Mitchell. ¶
- **[sqlite] [bug]** Adjusted a very old bugfix which attempted to work around a SQLite issue that itself was "fixed" as of sqlite 3.6.14, regarding quotes surrounding a table name when using the "foreign_key_list" pragma. The fix has been adjusted to not interfere with quotes that are *actually in the name* of a column or table, to as much a degree as possible; sqlite still doesn't return the correct result for foreign_key_list() if the target table actually has quotes surrounding its name, as *part* of its name (i.e. """mytable"""). ¶ References: #2568
- **[sqlite] [bug]** Adjusted column default reflection code to convert non-string values to string, to accommodate old SQLite versions that don't deliver default info as a string. ¶ References: #2265

mssql

- **[mssql] [bug]** Fixed compiler bug whereby using a correlated subquery within an ORDER BY would fail to render correctly if the stament also used LIMIT/OFFSET, due to mis-rendering within the ROW_NUMBER() OVER clause. Fix courtesy sayap ¶ References: #2538
- **[mssql] [bug]** Fixed compiler bug whereby a given select() would be modified if it had an "offset" attribute, causing the construct to not compile correctly a second time. ¶ References: #2545
- **[mssql] [bug]** Fixed bug where reflection of primary key constraint would double up columns if the same constraint/table existed in multiple schemas. ¶

0.7.8

Released: Sat Jun 16 2012

orm

- **[orm] [feature]** The 'objects' argument to flush() is no longer deprecated, as some valid use cases have been identified. ¶
- **[orm] [bug]** Fixed bug whereby subqueryload() from a polymorphic mapping to a target would incur a new invocation of the query for each distinct class encountered in the polymorphic result. ¶ References: #2480
- **[orm] [bug]** Fixed bug in declarative whereby the precedence of columns in a joined-table, composite column (typically for id) would fail to be correct if the columns contained names distinct from their attribute names. This would cause things like primaryjoin conditions made against the entity attributes to be incorrect. Related to as this was supposed to be part of that, this is. ¶ References: #2491, #1892
- **[orm] [bug]** Fixed identity_key() function which was not accepting a scalar argument for the identity. . ¶ References: #2508
- **[orm] [bug]** Fixed bug whereby populate_existing option would not propagate to subquery eager loaders. . ¶ References: #2497

engine

- **[engine] [bug]** Fixed memory leak in C version of result proxy whereby DBAPIs which don't deliver pure Python tuples for result rows would fail to decrement refcounts correctly. The most prominently affected DBAPI is pyodbc. ¶ References: #2489
- **[engine] [bug]** Fixed bug affecting Py3K whereby string positional parameters passed to engine/connection execute() would fail to be interpreted correctly, due to __iter__ being present on Py3K string.. ¶ References: #2503

sql

- **[sql] [bug]** added BIGINT to types.__all__, BIGINT, BINARY, VARBINARY to sqlalchemy module namespace, plus test to ensure this breakage doesn't occur again. ¶ References: #2499
- **[sql] [bug]** Repaired common table expression rendering to function correctly when the SELECT statement contains UNION or other compound expressions, courtesy btbuilder. ¶ References: #2490
- **[sql] [bug]** Fixed bug whereby append_column() wouldn't function correctly on a cloned select() construct, courtesy Gunnlaugur Þór Briem. ¶ References: #2482

postgresql

- **[postgresql] [bug]** removed unnecessary table clause when reflecting enums,. Courtesy Gunnlaugur Þór Briem. ¶ References: #2510

mysql

- **[mysql] [feature]** Added a new dialect for Google App Engine. Courtesy Richie Foreman. ¶ References: #2484

oracle

- **[oracle] [bug]** Added ROWID to oracle.*. ¶ References: #2483

0.7.7

Released: Sat May 05 2012

orm

- **[orm] [feature]** Added prefix_with() method to Query, calls upon select().prefix_with() to allow placement of MySQL SELECT directives in statements. Courtesy Diana Clarke ¶ References: #2443
- **[orm] [feature]** Added new flag to @validates include_removes. When True, collection remove and attribute del events will also be sent to the validation function, which accepts an additional argument "is_remove" when this flag is used. ¶
- **[orm] [bug]** Fixed issue in unit of work whereby setting a non-None self-referential many-to-one relationship to None would fail to persist the change if the former value was not already loaded.. ¶ References: #2477
- **[orm] [bug]** Fixed bug in 0.7.6 introduced by whereby column_mapped_collection used against columns that were mapped as joins or other indirect selectables would fail to function. ¶ References: #2409
- **[orm] [bug]** Fixed bug whereby polymorphic_on column that's not otherwise mapped on the class would be incorrectly included in a merge() operation, raising an error. ¶ References: #2449
- **[orm] [bug]** Fixed bug in expression annotation mechanics which could lead to incorrect rendering of SELECT statements with aliases and joins, particularly when using column_property(). ¶ References: #2453
- **[orm] [bug]** Fixed bug which would prevent OrderingList from being pickleable. Courtesy Jeff Dairiki ¶ References: #2454
- **[orm] [bug]** Fixed bug in relationship comparisons whereby calling unimplemented methods like SomeClass.somerelationship.like() would produce a recursion overflow, instead of NotImplementedError. ¶

sql

- **[sql] [feature]** Added new connection event dbapi_error(). Is called for all DBAPI-level errors passing the original DBAPI exception before SQLAlchemy modifies the state of the cursor. ¶
- **[sql] [bug]** Removed warning when Index is created with no columns; while this might not be what the user intended, it is a valid use case as an Index could be a placeholder for just an index of a certain name. ¶

- **[sql] [bug]** If conn.begin() fails when calling "with engine.begin()", the newly acquired Connection is closed explicitly before propagating the exception onward normally. ¶
- **[sql] [bug]** Add BINARY, VARBINARY to types.__all__. ¶ References: #2474

postgresql

- **[postgresql] [feature]** Added new for_update/with_lockmode() options for Postgresql: for_update="read"/ with_lockmode("read"), for_update="read_nowait"/ with_lockmode("read_nowait"). These emit "FOR SHARE" and "FOR SHARE NOWAIT", respectively. Courtesy Diana Clarke ¶ References: #2445
- **[postgresql] [bug]** removed unnecessary table clause when reflecting domains. ¶ References: #2473

mysql

- **[mysql] [bug]** Fixed bug whereby column name inside of "KEY" clause for autoincrement composite column with InnoDB would double quote a name that's a reserved word. Courtesy Jeff Dairiki. ¶ References: #2460
- **[mysql] [bug]** Fixed bug whereby get_view_names() for "information_schema" schema would fail to retrieve views marked as "SYSTEM VIEW". courtesy Matthew Turland. ¶
- **[mysql] [bug]** Fixed bug whereby if cast() is used on a SQL expression whose type is not supported by cast() and therefore CAST isn't rendered by the dialect, the order of evaluation could change if the casted expression required that it be grouped; grouping is now applied to those expressions. ¶ References: #2467

sqlite

- **[sqlite] [feature]** Added SQLite execution option "sqlite_raw_colnames=True", will bypass attempts to remove "." from column names returned by SQLite cursor.description. ¶ References: #2475
- **[sqlite] [bug]** When the primary key column of a Table is replaced, such as via extend_existing, the "auto increment" column used by insert() constructs is reset. Previously it would remain referring to the previous primary key column. ¶ References: #2525

mssql

- **[mssql] [feature]** Added interim create_engine flag supports_unicode_binds to PyODBC dialect, to force whether or not the dialect passes Python unicode literals to PyODBC or not. ¶
- **[mssql] [bug]** Repaired the use_scope_identity create_engine() flag when using the pyodbc dialect. Previously this flag would be ignored if set to False. When set to False, you'll get "SELECT @@identity" after each INSERT to get at the last inserted ID, for those tables which have "implicit_returning" set to False. ¶
- **[mssql] [bug]** UPDATE..FROM syntax with SQL Server requires that the updated table be present in the FROM clause when an alias of that table is also present in the FROM clause. The updated table is now always present in the FROM, when FROM is present in the first place. Courtesy sayap. ¶ References: #2468

0.7.6

Released: Wed Mar 14 2012

orm

- **[orm] [feature]** Added "no_autoflush" context manager to Session, used with with: will temporarily disable autoflush. ¶
- **[orm] [feature]** Added cte() method to Query, invokes common table expression support from the Core (see below). ¶ References: #1859
- **[orm] [feature]** Added the ability to query for Table-bound column names when using query(sometable).filter_by(colname=value). ¶ References: #2400
- **[orm] [bug]** Fixed event registration bug which would primarily show up as events not being registered with sessionmaker() instances created after the event was associated with the Session class. ¶ References: #2424
- **[orm] [bug]** Fixed bug whereby a primaryjoin condition with a "literal" in it would raise an error on compile with certain kinds of deeply nested expressions which also needed to render the same bound parameter name more than once. ¶ References: #2425
- **[orm] [bug]** Removed the check for number of rows affected when doing a multi-delete against mapped objects. If an ON DELETE CASCADE exists between two rows, we can't get an accurate rowcount from the DBAPI; this particular count is not supported on most DBAPIs in any case, MySQLdb is the notable case where it is. ¶ References: #2403
- **[orm] [bug]** Fixed bug whereby objects using attribute_mapped_collection or column_mapped_collection could not be pickled. ¶ References: #2409
- **[orm] [bug]** Fixed bug whereby MappedCollection would not get the appropriate collection instrumentation if it were only used in a custom subclass that used @collection.internally_instrumented. ¶ References: #2406
- **[orm] [bug]** Fixed bug whereby SQL adaption mechanics would fail in a very nested scenario involving joined-inheritance, joinedload(), limit(), and a derived function in the columns clause. ¶ References: #2419
- **[orm] [bug]** Fixed the repr() for CascadeOptions to include refresh-expire. Also reworked CascadeOptions to be a <frozenset>. ¶ References: #2417
- **[orm] [bug]** Improved the "declarative reflection" example to support single-table inheritance, multiple calls to prepare(), tables that are present in alternate schemas, establishing only a subset of classes as reflected. ¶
- **[orm] [bug]** Scaled back the test applied within flush() to check for UPDATE against partially NULL PK within one table to only actually happen if there's really an UPDATE to occur. ¶ References: #2390
- **[orm] [bug]** Fixed bug whereby if a method name conflicted with a column name, a TypeError would be raised when the mapper tried to inspect the __get__() method on the method object. ¶ References: #2352

engine

- **[engine] [feature]** Added "no_parameters=True" execution option for connections. If no parameters are present, will pass the statement as cursor.execute(statement), thereby invoking the DBAPIs behavior when no parameter collection is present; for psycopg2 and mysql-python, this means not interpreting % signs in the string. This only occurs with this option, and not just if the param list is blank, as otherwise this would produce inconsistent behavior of SQL expressions that normally escape percent signs (and while compiling, can't know ahead of time if parameters will be present in some cases). ¶ References: #2407
- **[engine] [feature]** Added pool_reset_on_return argument to create_engine, allows control over "connection return" behavior. Also added new arguments 'rollback', 'commit', None to pool.reset_on_return to allow more control over connection return activity. ¶ References: #2378
- **[engine] [feature]** Added some decent context managers to Engine, Connection:

```
with engine.begin() as conn:
    <work with conn in a transaction>
```

and:

```
with engine.connect() as conn:
    <work with conn>
```

Both close out the connection when done, commit or rollback transaction with errors on engine.begin(). ¶

- **[engine] [bug]** Added execution_options() call to MockConnection (i.e., that used with strategy="mock") which acts as a pass through for arguments. ¶

sql

- **[sql] [feature]** Added support for SQL standard common table expressions (CTE), allowing SELECT objects as the CTE source (DML not yet supported). This is invoked via the cte() method on any select() construct. ¶ References: #1859
- **[sql] [bug]** Fixed memory leak in core which would occur when C extensions were used with particular types of result fetches, in particular when orm query.count() were called. ¶ References: #2427
- **[sql] [bug]** Fixed issue whereby attribute-based column access on a row would raise AttributeError with non-C version, NoSuchColumnError with C version. Now raises AttributeError in both cases. ¶ References: #2398
- **[sql] [bug]** Added support for using the .key of a Column as a string identifier in a result set row. The .key is currently listed as an "alternate" name for a column, and is superseded by the name of a column which has that key value as its regular name. For the next major release of SQLAlchemy we may reverse this precedence so that .key takes precedence, but this is not decided on yet. ¶ References: #2392
- **[sql] [bug]** A warning is emitted when a not-present column is stated in the values() clause of an insert() or update() construct. Will move to an exception in 0.8. ¶ References: #2413
- **[sql] [bug]** A significant change to how labeling is applied to columns in SELECT statements allows "truncated" labels, that is label names that are generated in Python which exceed the maximum identifier length (note this is configurable via label_length on create_engine()), to be properly referenced when rendered inside of a subquery, as well as to be present in a result set row using their original in-Python names. ¶ References: #2396
- **[sql] [bug]** Fixed bug in new "autoload_replace" flag which would fail to preserve the primary key constraint of the reflected table. ¶ References: #2402
- **[sql] [bug]** Index will raise when arguments passed cannot be interpreted as columns or expressions. Will warn when Index is created with no columns at all. ¶ References: #2380

mysql

- **[mysql] [feature]** Added support for MySQL index and primary key constraint types (i.e. USING) via new mysql_using parameter to Index and PrimaryKeyConstraint, courtesy Diana Clarke. ¶ References: #2386
- **[mysql] [feature]** Added support for the "isolation_level" parameter to all MySQL dialects. Thanks to mu_mind for the patch here. ¶ References: #2394

sqlite

- **[sqlite] [bug]** Fixed bug in C extensions whereby string format would not be applied to a Numeric value returned as integer; this affected primarily SQLite which does not maintain numeric scale settings. ¶ References: #2432

mssql

- **[mssql] [feature]** Added support for MSSQL INSERT, UPDATE, and DELETE table hints, using new with_hint() method on UpdateBase. ¶ References: #2430

oracle

- **[oracle] [feature]** Added a new create_engine() flag coerce_to_decimal=False, disables the precision numeric handling which can add lots of overhead by converting all numeric values to Decimal. ¶ References: #2399
- **[oracle] [bug]** Added missing compilation support for LONG ¶ References: #2401
- **[oracle] [bug]** Added 'LEVEL' to the list of reserved words for Oracle. ¶ References: #2435

misc

- **[bug] [examples]** Altered _params_from_query() function in Beaker example to pull bindparams from the fully compiled statement, as a quick means to get everything including subqueries in the columns clause, etc. ¶

0.7.5

Released: Sat Jan 28 2012

orm

- **[orm] [feature]** Added "class_registry" argument to declarative_base(). Allows two or more declarative bases to share the same registry of class names. ¶
- **[orm] [feature]** query.filter() accepts multiple criteria which will join via AND, i.e. query.filter(x==y, z>q, ...) ¶
- **[orm] [feature]** Added new capability to relationship loader options to allow "default" loader strategies. Pass '*' to any of joinedload(), lazyload(), subqueryload(), or noload() and that becomes the loader strategy used for all relationships, except for those explicitly stated in the Query. Thanks to up-and-coming contributor Kent Bower for an exhaustive and well written test suite ! ¶ References: #2351
- **[orm] [feature]** New declarative reflection example added, illustrates how best to mix table reflection with declarative as well as uses some new features from. ¶ References: #2356
- **[orm] [bug]** Fixed issue where modified session state established after a failed flush would be committed as part of the subsequent transaction that begins automatically after manual call to rollback(). The state of the session is checked within rollback(), and if new state is present, a warning is emitted and restore_snapshot() is called a second time, discarding those changes. ¶ References: #2389
- **[orm] [bug]** Fixed regression from 0.7.4 whereby using an already instrumented column from a superclass as "polymorphic_on" failed to resolve the underlying Column. ¶ References: #2345

- **[orm] [bug]** Raise an exception if xyzload_all() is used inappropriately with two non-connected relationships. ¶ References: #2370
- **[orm] [bug]** Fixed bug whereby event.listen(SomeClass) forced an entirely unnecessary compile of the mapper, making events very hard to set up at module import time (nobody noticed this ??) ¶ References: #2367
- **[orm] [bug]** Fixed bug whereby hybrid_property didn't work as a kw arg in any(), has(). ¶
- **[orm] [bug]** ensure pickleability of all ORM exceptions for multiprocessing compatibility. ¶ References: #2371
- **[orm] [bug]** implemented standard "can't set attribute" / "can't delete attribute" AttributeError when setattr/delattr used on a hybrid that doesn't define fset or fdel. ¶ References: #2353
- **[orm] [bug]** Fixed bug where unpickled object didn't have enough of its state set up to work correctly within the unpickle() event established by the mutable object extension, if the object needed ORM attribute access within __eq__() or similar. ¶ References: #2362
- **[orm] [bug]** Fixed bug where "merge" cascade could mis-interpret an unloaded attribute, if the load_on_pending flag were used with relationship(). Thanks to Kent Bower for tests. ¶ References: #2374
- **[orm]** Fixed regression from 0.6 whereby if "load_on_pending" relationship() flag were used where a non-"get()" lazy clause needed to be emitted on a pending object, it would fail to load. ¶

engine

- **[engine] [bug]** Added __reduce__ to StatementError, DBAPIError, column errors so that exceptions are pickleable, as when using multiprocessing. However, not all DBAPIs support this yet, such as psycopg2. ¶ References: #2371
- **[engine] [bug]** Improved error messages when a non-string or invalid string is passed to any of the date/time processors used by SQLite, including C and Python versions. ¶ References: #2382
- **[engine] [bug]** Fixed bug whereby a table-bound Column object named "<a>_<b>" which matched a column labeled as "<tablename>_<colname>" could match inappropriately when targeting in a result set row. ¶ References: #2377
- **[engine] [bug]** Fixed bug in "mock" strategy whereby correct DDL visit method wasn't called, resulting in "CREATE/DROP SEQUENCE" statements being duplicated ¶ References: #2384

sql

- **[sql] [feature]** New reflection feature "autoload_replace"; when set to False on Table, the Table can be autoloaded without existing columns being replaced. Allows more flexible chains of Table construction/reflection to be constructed, including that it helps with combining Declarative with table reflection. See the new example on the wiki. ¶ References: #2356
- **[sql] [feature]** Added "false()" and "true()" expression constructs to sqlalchemy.sql namespace, though not part of __all__ as of yet. ¶
- **[sql] [feature]** Dialect-specific compilers now raise CompileError for all type/statement compilation issues, instead of InvalidRequestError or ArgumentError. The DDL for CREATE TABLE will re-raise CompileError to include table/column information for the problematic column. ¶ References: #2361
- **[sql] [bug]** Improved the API for add_column() such that if the same column is added to its own table, an error is not raised and the constraints don't get doubled up. Also helps with some reflection/declarative patterns. ¶ References: #2356

- **[sql] [bug]** Fixed issue where the "required" exception would not be raised for bindparam() with required=True, if the statement were given no parameters at all. ¶ References: #2381

mysql

- **[mysql] [bug]** fixed regexp that filters out warnings for non-reflected "PARTITION" directives, thanks to George Reilly ¶ References: #2376

sqlite

- **[sqlite] [bug]** the "name" of an FK constraint in SQLite is reflected as "None", not "0" or other integer value. SQLite does not appear to support constraint naming in any case. ¶ References: #2364
- **[sqlite] [bug]** sql.false() and sql.true() compile to 0 and 1, respectively in sqlite ¶ References: #2368
- **[sqlite] [bug]** removed an erroneous "raise" in the SQLite dialect when getting table names and view names, where logic is in place to fall back to an older version of SQLite that doesn't have the "sqlite_temp_master" table. ¶

mssql

- **[mssql] [bug]** Adjusted the regexp used in the mssql.TIME type to ensure only six digits are received for the "microseconds" portion of the value, which is expected by Python's datetime.time(). Note that support for sending microseconds doesn't seem to be possible yet with pyodbc at least. ¶ References: #2340
- **[mssql] [bug]** Dropped the "30 char" limit on pymssql, based on reports that it's doing things better these days. pymssql hasn't been well tested and as the DBAPI is in flux it's still not clear what the status is on this driver and how SQLAlchemy's implementation should adapt. ¶ References: #2347

oracle

- **[oracle] [bug]** Added ORA-03135 to the never ending list of oracle "connection lost" errors ¶ References: #2388

misc

- **[feature] [examples]** Simplified the versioning example a bit to use a declarative mixin as well as an event listener, instead of a metaclass + SessionExtension. ¶ References: #2313
- **[bug] [core]** Changed LRUCache, used by the mapper to cache INSERT/UPDATE/DELETE statements, to use an incrementing counter instead of a timestamp to track entries, for greater reliability versus using time.time(), which can cause test failures on some platforms. ¶ References: #2379
- **[bug] [core]** Added a boolean check for the "finalize" function within the pool connection proxy's weakref callback before calling it, so that a warning isn't emitted that this function is None when the application is exiting and gc has removed the function from the module before the weakref callback was invoked. ¶ References: #2383
- **[bug] [py3k]** Fixed inappropriate usage of util.py3k flag and renamed it to util.py3k_warning, since this flag is intended to detect the -3 flag series of import restrictions only. ¶ References: #2348
- **[bug] [examples]** Fixed large_collection.py to close the session before dropping tables. ¶ References: #2346

0.7.4

Released: Fri Dec 09 2011

orm

- **[orm] [feature]** polymorphic_on now accepts many new kinds of values:
  - standalone expressions that aren't otherwise mapped
  - column_property() objects
  - string names of any column_property() or attribute name of a mapped Column

  The docs include an example using the case() construct, which is likely to be a common constructed used here. and part of

  Standalone expressions in polymorphic_on propagate to single-table inheritance subclasses so that they are used in the WHERE /JOIN clause to limit rows to that subclass as is the usual behavior. ¶ References: #2345, #2238
- **[orm] [feature]** IdentitySet supports the - operator as the same as difference(), handy when dealing with Session.dirty etc. ¶ References: #2301
- **[orm] [feature]** Added new value for Column autoincrement called "ignore_fk", can be used to force autoincrement on a column that's still part of a ForeignKeyConstraint. New example in the relationship docs illustrates its use. ¶
- **[orm] [bug]** Fixed backref behavior when "popping" the value off of a many-to-one in response to a removal from a stale one-to-many - the operation is skipped, since the many-to-one has since been updated. ¶ References: #2315
- **[orm] [bug]** After some years of not doing this, added more granularity to the "is X a parent of Y" functionality, which is used when determining if the FK on "Y" needs to be "nulled out" as well as if "Y" should be deleted with delete-orphan cascade. The test now takes into account the Python identity of the parent as well its identity key, to see if the last known parent of Y is definitely X. If a decision can't be made, a StaleDataError is raised. The conditions where this error is raised are fairly rare, requiring that the previous parent was garbage collected, and previously could very well inappropriately update/delete a record that's since moved onto a new parent, though there may be some cases where "silent success" occurred previously that will now raise in the face of ambiguity. Expiring "Y" resets the "parent" tracker, meaning X.remove(Y) could then end up deleting Y even if X is stale, but this is the same behavior as before; it's advised to expire X also in that case. ¶ References: #2264
- **[orm] [bug]** fixed inappropriate evaluation of user-mapped object in a boolean context within query.get(). Also in 0.6.9. ¶ References: #2310
- **[orm] [bug]** Added missing comma to PASSIVE_RETURN_NEVER_SET symbol ¶ References: #2304
- **[orm] [bug]** Cls.column.collate("some collation") now works. Also in 0.6.9 ¶ References: #1776
- **[orm] [bug]** the value of a composite attribute is now expired after an insert or update operation, instead of regenerated in place. This ensures that a column value which is expired within a flush will be loaded first, before the composite is regenerated using that value. ¶ References: #2309
- **[orm] [bug]** The fix in also emits the "refresh" event when the composite value is loaded on access, even if all column values were already present, as is appropriate. This fixes the "mutable" extension which relies upon the "load" event to ensure the _parents dictionary is up to date, fixes. Thanks to Scott Torborg for the test case here. ¶ References: #2309, #2308
- **[orm] [bug]** Fixed bug whereby a subclass of a subclass using concrete inheritance in conjunction with the new ConcreteBase or AbstractConcreteBase would fail to apply the subclasses deeper than one level to the "polymorphic loader" of each base ¶ References: #2312

- **[orm] [bug]** Fixed bug whereby a subclass of a subclass using the new AbstractConcreteBase would fail to acquire the correct "base_mapper" attribute when the "base" mapper was generated, thereby causing failures later on. ¶ References: #2312
- **[orm] [bug]** Fixed bug whereby column_property() created against ORM-level column could be treated as a distinct entity when producing certain kinds of joined-inh joins. ¶ References: #2316
- **[orm] [bug]** Fixed the error formatting raised when a tuple is inadvertently passed to session.query(). Also in 0.6.9. ¶ References: #2297
- **[orm] [bug]** Calls to query.join() to a single-table inheritance subclass are now tracked, and are used to eliminate the additional WHERE.. IN criterion normally tacked on with single table inheritance, since the join should accommodate it. This allows OUTER JOIN to a single table subclass to produce the correct results, and overall will produce fewer WHERE criterion when dealing with single table inheritance joins. ¶ References: #2328
- **[orm] [bug]** __table_args__ can now be passed as an empty tuple as well as an empty dict.. Thanks to Fayaz Yusuf Khan for the patch. ¶ References: #2339
- **[orm] [bug]** Updated warning message when setting delete-orphan without delete to no longer refer to 0.6, as we never got around to upgrading this to an exception. Ideally this might be better as an exception but it's not critical either way. ¶ References: #2325
- **[orm] [bug]** Fixed bug in get_history() when referring to a composite attribute that has no value; added coverage for get_history() regarding composites which is otherwise just a userland function. ¶

engine

- **[engine] [bug]** Fixed bug whereby transaction.rollback() would throw an error on an invalidated connection if the transaction were a two-phase or savepoint transaction. For plain transactions, rollback() is a no-op if the connection is invalidated, so while it wasn't 100% clear if it should be a no-op, at least now the interface is consistent. ¶ References: #2317

sql

- **[sql] [feature]** The update() construct can now accommodate multiple tables in the WHERE clause, which will render an "UPDATE..FROM" construct, recognized by Postgresql and MSSQL. When compiled on MySQL, will instead generate "UPDATE t1, t2, ..". MySQL additionally can render against multiple tables in the SET clause, if Column objects are used as keys in the "values" parameter or generative method. ¶ References: #2166, #1944
- **[sql] [feature]** Added accessor to types called "python_type", returns the rudimentary Python type object for a particular TypeEngine instance, if known, else raises NotImplementedError. ¶ References: #77
- **[sql] [bug]** related to, made some adjustments to the change from regarding the "from" list on a select(). The _froms collection is no longer memoized, as this simplifies various use cases and removes the need for a "warning" if a column is attached to a table after it was already used in an expression - the select() construct will now always produce the correct expression. There's probably no real-world performance hit here; select() objects are almost always made ad-hoc, and systems that wish to optimize the re-use of a select() would be using the "compiled_cache" feature. A hit which would occur when calling select.bind has been reduced, but the vast majority of users shouldn't be using "bound metadata" anyway :). ¶ References: #2316, #2261
- **[sql] [bug]** further tweak to the fix from, so that generative methods work a bit better off of cloned (this is almost a non-use case though). In particular this allows with_only_columns() to behave more consistently. Added additional documentation to with_only_columns() to clarify expected behavior, which changed as a result of. ¶ References: #2261, #2319

schema

- **[schema] [feature]** Added new support for remote "schemas": ¶
- **[schema] [feature]** The "extend_existing" flag on Table now allows for the reflection process to take effect for a Table object that's already been defined; when autoload=True and extend_existing=True are both set, the full set of columns will be reflected from the Table which will then *overwrite* those columns already present, rather than no activity occurring. Columns that are present directly in the autoload run will be used as always, however. ¶ References: #1410
- **[schema] [bug]** Fixed bug whereby TypeDecorator would return a stale value for _type_affinity, when using a TypeDecorator that "switches" types, like the CHAR/UUID type. ¶
- **[schema] [bug]** Fixed bug whereby "order_by='foreign_key"' option to Inspector.get_table_names wasn't implementing the sort properly, replaced with the existing sort algorithm ¶
- **[schema] [bug]** the "name" of a column-level CHECK constraint, if present, is now rendered in the CREATE TABLE statement using "CONSTRAINT <name> CHECK <expression>". ¶ References: #2305
- **[schema]** MetaData() accepts "schema" and "quote_schema" arguments, which will be applied to the same-named arguments of a Table or Sequence which leaves these at their default of `None`. ¶
- **[schema]** Sequence accepts "quote_schema" argument ¶
- **[schema]** tometadata() for Table will use the "schema" of the incoming MetaData for the new Table if the schema argument is explicitly "None" ¶
- **[schema]** Added CreateSchema and DropSchema DDL constructs - these accept just the string name of a schema and a "quote" flag. ¶
- **[schema]** When using default "schema" with MetaData, ForeignKey will also assume the "default" schema when locating remote table. This allows the "schema" argument on MetaData to be applied to any set of Table objects that otherwise don't have a "schema". ¶
- **[schema]** a "has_schema" method has been implemented on dialect, but only works on Postgresql so far. Courtesy Manlio Perillo. ¶ References: #1679

postgresql

- **[postgresql] [feature]** Added create_type constructor argument to pg.ENUM. When False, no CREATE/DROP or checking for the type will be performed as part of a table create/drop event; only the create()/drop)() methods called directly will do this. Helps with Alembic "offline" scripts. ¶
- **[postgresql] [bug]** Postgresql dialect memoizes that an ENUM of a particular name was processed during a create/drop sequence. This allows a create/drop sequence to work without any calls to "checkfirst", and also means with "checkfirst" turned on it only needs to check for the ENUM once. ¶ References: #2311

mysql

- **[mysql] [bug]** Unicode adjustments allow latest pymysql (post 0.4) to pass 100% on Python 2. ¶

mssql

- **[mssql] [feature]** lifted the restriction on SAVEPOINT for SQL Server. All tests pass using it, it's not known if there are deeper issues however. ¶ References: #822

- **[mssql] [bug]** repaired the with_hint() feature which wasn't implemented correctly on MSSQL - usually used for the "WITH (NOLOCK)" hint (which you shouldn't be using anyway ! use snapshot isolation instead :)) ¶ References: #2336
- **[mssql] [bug]** use new pyodbc version detection for _need_decimal_fix option. ¶ References: #2318
- **[mssql] [bug]** don't cast "table name" as NVARCHAR on SQL Server 2000. Still mostly in the dark what incantations are needed to make PyODBC work fully with FreeTDS 0.91 here, however. ¶ References: #2343
- **[mssql] [bug]** Decode incoming values when retrieving list of index names and the names of columns within those indexes. ¶ References: #2269

misc

- **[feature] [ext]** Added an example to the hybrid docs of a "transformer" - a hybrid that returns a query-transforming callable in combination with a custom comparator. Uses a new method on Query called with_transformation(). The use case here is fairly experimental, but only adds one line of code to Query. ¶
- **[bug] [pyodbc]** pyodbc-based dialects now parse the pyodbc accurately as far as observed pyodbc strings, including such gems as "py3-3.0.1-beta4" ¶ References: #2318
- **[bug] [ext]** the @compiles decorator raises an informative error message when no "default" compilation handler is present, rather than KeyError. ¶
- **[bug] [examples]** Fixed bug in history_meta.py example where the "unique" flag was not removed from a single-table-inheritance subclass which generates columns to put up onto the base. ¶

0.7.3

Released: Sun Oct 16 2011

general

- **[general]** Adjusted the "importlater" mechanism, which is used internally to resolve import cycles, such that the usage of __import__ is completed when the import of sqlalchemy or sqlalchemy.orm is done, thereby avoiding any usage of __import__ after the application starts new threads, fixes. Also in 0.6.9. ¶ References: #2279

orm

- **[orm]** Improved query.join() such that the "left" side can more flexibly be a non-ORM selectable, such as a subquery. A selectable placed in select_from() will now be used as the left side, favored over implicit usage of a mapped entity. If the join still fails based on lack of foreign keys, the error message includes this detail. Thanks to brianrhude on IRC for the test case. ¶ References: #2298
- **[orm]** Added after_soft_rollback() Session event. This event fires unconditionally whenever rollback() is called, regardless of if an actual DBAPI level rollback occurred. This event is specifically designed to allow operations with the Session to proceed after a rollback when the Session.is_active is True. ¶ References: #2241
- **[orm]** added "adapt_on_names" boolean flag to orm.aliased() construct. Allows an aliased() construct to link the ORM entity to a selectable that contains aggregates or other derived forms of a particular attribute, provided the name is the same as that of the entity mapped column. ¶
- **[orm]** Added new flag expire_on_flush=False to column_property(), marks those properties that would otherwise be considered to be "readonly", i.e. derived from SQL expressions, to retain their value after a flush has occurred, including if the parent object itself was involved in an update. ¶

- **[orm]** Enhanced the instrumentation in the ORM to support Py3K's new argument style of "required kw arguments", i.e. fn(a, b, *, c, d), fn(a, b, *args, c, d). Argument signatures of mapped object's __init__ method will be preserved, including required kw rules. ¶ References: #2237
- **[orm]** Fixed bug in unit of work whereby detection of "cycles" among classes in highly interlinked patterns would not produce a deterministic result; thereby sometimes missing some nodes that should be considered cycles and causing further issues down the road. Note this bug is in 0.6 also; not backported at the moment. ¶ References: #2282
- **[orm]** Fixed a variety of synonym()-related regressions from 0.6:
  - making a synonym against a synonym now works.
  - synonyms made against a relationship() can be passed to query.join(), options sent to query.options(), passed by name to query.with_parent().

  ¶
- **[orm]** Fixed bug whereby mapper.order_by attribute would be ignored in the "inner" query within a subquery eager load. . Also in 0.6.9. ¶ References: #2287
- **[orm]** Identity map .discard() uses dict.pop(,None) internally instead of "del" to avoid KeyError/warning during a non-determinate gc teardown ¶ References: #2267
- **[orm]** Fixed regression in new composite rewrite where deferred=True option failed due to missing import ¶ References: #2253
- **[orm]** Reinstated "comparator_factory" argument to composite(), removed when 0.7 was released. ¶ References: #2248
- **[orm]** Fixed bug in query.join() which would occur in a complex multiple-overlapping path scenario, where the same table could be joined to twice. Thanks *much* to Dave Vitek for the excellent fix here. ¶ References: #2247
- **[orm]** Query will convert an OFFSET of zero when slicing into None, so that needless OFFSET clauses are not invoked. ¶
- **[orm]** Repaired edge case where mapper would fail to fully update internal state when a relationship on a new mapper would establish a backref on the first mapper. ¶
- **[orm]** Fixed bug whereby if __eq__() was redefined, a relationship many-to-one lazyload would hit the __eq__() and fail. Does not apply to 0.6.9. ¶ References: #2260
- **[orm]** Calling class_mapper() and passing in an object that is not a "type" (i.e. a class that could potentially be mapped) now raises an informative ArgumentError, rather than UnmappedClassError. ¶ References: #2196
- **[orm]** New event hook, MapperEvents.after_configured(). Called after a configure() step has completed and mappers were in fact affected. Theoretically this event is called once per application, unless new mappings are constructed after existing ones have been used already. ¶
- **[orm]** When an open Session is garbage collected, the objects within it which remain are considered detached again when they are add()-ed to a new Session. This is accomplished by an extra check that the previous "session_key" doesn't actually exist among the pool of Sessions. ¶ References: #2281
- **[orm]** New declarative features:
  - __declare_last__() method, establishes an event listener for the class method that will be called when mappers are completed with the final "configure" step.
  - __abstract__ flag. The class will not be mapped at all when this flag is present on the class.
  - New helper classes ConcreteBase, AbstractConcreteBase. Allow concrete mappings using declarative which automatically set up the "polymorphic_union" when the "configure" mapper step is invoked.

  - The mapper itself has semi-private methods that allow the "with_polymorphic" selectable to be assigned to the mapper after it has already been configured.

  ¶ References: #2239

- **[orm]** Declarative will warn when a subclass' base uses @declared_attr for a regular column - this attribute does not propagate to subclasses. ¶ References: #2283
- **[orm]** The integer "id" used to link a mapped instance with its owning Session is now generated by a sequence generation function rather than id(Session), to eliminate the possibility of recycled id() values causing an incorrect result, no need to check that object actually in the session. ¶ References: #2280
- **[orm]** Behavioral improvement: empty conjunctions such as and_() and or_() will be flattened in the context of an enclosing conjunction, i.e. and_(x, or_()) will produce 'X' and not 'X AND ()'.. ¶ References: #2257
- **[orm]** Fixed bug regarding calculation of "from" list for a select() element. The "from" calc is now delayed, so that if the construct uses a Column object that is not yet attached to a Table, but is later associated with a Table, it generates SQL using the table as a FROM. This change impacted fairly deeply the mechanics of how the FROM list as well as the "correlates" collection is calculated, as some "clause adaption" schemes (these are used very heavily in the ORM) were relying upon the fact that the "froms" collection would typically be cached before the adaption completed. The rework allows it such that the "froms" collection can be cleared and re-generated at any time. ¶ References: #2261
- **[orm]** Fixed bug whereby with_only_columns() method of Select would fail if a selectable were passed.. Also in 0.6.9. ¶ References: #2270

engine

- **[engine]** The recreate() method in all pool classes uses self.__class__ to get at the type of pool to produce, in the case of subclassing. Note there's no usual need to subclass pools. ¶ References: #2254
- **[engine]** Improvement to multi-param statement logging, long lists of bound parameter sets will be compressed with an informative indicator of the compression taking place. Exception messages use the same improved formatting. ¶ References: #2243
- **[engine]** Added optional "sa_pool_key" argument to pool.manage(dbapi).connect() so that serialization of args is not necessary. ¶
- **[engine]** The entry point resolution supported by create_engine() now supports resolution of individual DBAPI drivers on top of a built-in or entry point-resolved dialect, using the standard '+' notation - it's converted to a '.' before being resolved as an entry point. ¶ References: #2286
- **[engine]** Added an exception catch + warning for the "return unicode detection" step within connect, allows databases that crash on NVARCHAR to continue initializing, assuming no NVARCHAR type implemented. ¶ References: #2299

schema

- **[schema]** Modified Column.copy() to use _constructor(), which defaults to self.__class__, in order to create the new object. This allows easier support of subclassing Column. ¶ References: #2284
- **[schema]** Added a slightly nicer __repr__() to SchemaItem classes. Note the repr here can't fully support the "repr is the constructor" idea since schema items can be very deeply nested/cyclical, have late initialization of some things, etc. ¶ References: #2223

postgresql

- **[postgresql]** Added "postgresql_using" argument to Index(), produces USING clause to specify index implementation for PG. . Thanks to Ryan P. Kelly for the patch. ¶ References: #2290
- **[postgresql]** Added client_encoding parameter to create_engine() when the postgresql+psycopg2 dialect is used; calls the psycopg2 set_client_encoding() method with the value upon connect. ¶ References: #1839
- **[postgresql]** Fixed bug related to whereby the same modified index behavior in PG 9 affected primary key reflection on a renamed column.. Also in 0.6.9. ¶ References: #2291, #2141
- **[postgresql]** Reflection functions for Table, Sequence no longer case insensitive. Names can be differ only in case and will be correctly distinguished. ¶ References: #2256
- **[postgresql]** Use an atomic counter as the "random number" source for server side cursor names; conflicts have been reported in rare cases. ¶
- **[postgresql]** Narrowed the assumption made when reflecting a foreign-key referenced table with schema in the current search path; an explicit schema will be applied to the referenced table only if it actually matches that of the referencing table, which also has an explicit schema. Previously it was assumed that "current" schema was synonymous with the full search_path. ¶ References: #2249

mysql

- **[mysql]** a CREATE TABLE will put the COLLATE option after CHARSET, which appears to be part of MySQL's arbitrary rules regarding if it will actually work or not. Also in 0.6.9. ¶ References: #2225
- **[mysql]** Added mysql_length parameter to Index construct, specifies "length" for indexes. ¶ References: #2293

sqlite

- **[sqlite]** Ensured that the same ValueError is raised for illegal date/time/datetime string parsed from the database regardless of whether C extensions are in use or not. ¶

mssql

- **[mssql]** Changes to attempt support of FreeTDS 0.91 with Pyodbc. This includes that string binds are sent as Python unicode objects when FreeTDS 0.91 is detected, and a CAST(? AS NVARCHAR) is used when we detect for a table. However, I'd continue to characterize Pyodbc + FreeTDS 0.91 behavior as pretty crappy, there are still many queries such as used in reflection which cause a core dump on Linux, and it is not really usable at all on OSX, MemoryErrors abound and just plain broken unicode support. ¶ References: #2273
- **[mssql]** The behavior of =/!= when comparing a scalar select to a value will no longer produce IN/NOT IN as of 0.8; this behavior is a little too heavy handed (use in_() if you want to emit IN) and now emits a deprecation warning. To get the 0.8 behavior immediately and remove the warning, a compiler recipe is given at http://www.sqlalchemy.org/docs/07/dialects/mssql.html#scalar-select-comparisons to override the behavior of visit_binary(). ¶ References: #2277
- **[mssql]** "0" is accepted as an argument for limit() which will produce "TOP 0". ¶ References: #2222

oracle

- **[oracle]** Fixed ReturningResultProxy for zxjdbc dialect.. Regression from 0.6. ¶ References: #2272

- **[oracle]** The String type now generates VARCHAR2 on Oracle which is recommended as the default VARCHAR. Added an explicit VARCHAR2 and NVARCHAR2 to the Oracle dialect as well. Using NVARCHAR still generates "NVARCHAR2" - there is no "NVARCHAR" on Oracle - this remains a slight breakage of the "uppercase types always give exactly that" policy. VARCHAR still generates "VARCHAR", keeping with the policy. If Oracle were to ever define "VARCHAR" as something different as they claim (IMHO this will never happen), the type would be available. ¶ References: #2252

misc

- **[types]** Extra keyword arguments to the base Float type beyond "precision" and "asdecimal" are ignored; added a deprecation warning here and additional docs, related to ¶ References: #2258
- **[ext]** SQLSoup will not be included in version 0.8 of SQLAlchemy; while useful, we would like to keep SQLAlchemy itself focused on one ORM usage paradigm. SQLSoup will hopefully soon be superseded by a third party project. ¶ References: #2262
- **[ext]** Added local_attr, remote_attr, attr accessors to AssociationProxy, providing quick access to the proxied attributes at the class level. ¶ References: #2236
- **[ext]** Changed the update() method on association proxy dictionary to use a duck typing approach, i.e. checks for "keys", to discern between update({}) and update((a, b)). Previously, passing a dictionary that had tuples as keys would be misinterpreted as a sequence. ¶ References: #2275
- **[examples]** Adjusted dictlike-polymorphic.py example to apply the CAST such that it works on PG, other databases. Also in 0.6.9. ¶ References: #2266

0.7.2

Released: Sun Jul 31 2011

orm

- **[orm]** Feature enhancement: joined and subquery loading will now traverse already-present related objects and collections in search of unpopulated attributes throughout the scope of the eager load being defined, so that the eager loading that is specified via mappings or query options unconditionally takes place for the full depth, populating whatever is not already populated. Previously, this traversal would stop if a related object or collection were already present leading to inconsistent behavior (though would save on loads/cycles for an already-loaded graph). For a subqueryload, this means that the additional SELECT statements emitted by subqueryload will invoke unconditionally, no matter how much of the existing graph is already present (hence the controversy). The previous behavior of "stopping" is still in effect when a query is the result of an attribute-initiated lazyload, as otherwise an "N+1" style of collection iteration can become needlessly expensive when the same related object is encountered repeatedly. There's also an as-yet-not-public generative Query method _with_invoke_all_eagers() which selects old/new behavior ¶ References: #2213
- **[orm]** A rework of "replacement traversal" within the ORM as it alters selectables to be against aliases of things (i.e. clause adaption) includes a fix for multiply-nested any()/has() constructs against a joined table structure. ¶ References: #2195
- **[orm]** Fixed bug where query.join() + aliased=True from a joined-inh structure to itself on relationship() with join condition on the child table would convert the lead entity into the joined one inappropriately. Also in 0.6.9. ¶ References: #2234
- **[orm]** Fixed regression from 0.6 where Session.add() against an object which contained None in a collection would raise an internal exception. Reverted this to 0.6's behavior which is to accept the None but obviously

nothing is persisted. Ideally, collections with None present or on append() should at least emit a warning, which is being considered for 0.8. ¶ References: #2205

- **[orm]** Load of a deferred() attribute on an object where row can't be located raises ObjectDeletedError instead of failing later on; improved the message in ObjectDeletedError to include other conditions besides a simple "delete". ¶ References: #2191
- **[orm]** Fixed regression from 0.6 where a get history operation on some relationship() based attributes would fail when a lazyload would emit; this could trigger within a flush() under certain conditions. Thanks to the user who submitted the great test for this. ¶ References: #2224
- **[orm]** Fixed bug apparent only in Python 3 whereby sorting of persistent + pending objects during flush would produce an illegal comparison, if the persistent object primary key is not a single integer. Also in 0.6.9 ¶ References: #2228
- **[orm]** Fixed bug whereby the source clause used by query.join() would be inconsistent if against a column expression that combined multiple entities together. Also in 0.6.9 ¶ References: #2197
- **[orm]** Fixed bug whereby if a mapped class redefined __hash__() or __eq__() to something non-standard, which is a supported use case as SQLA should never consult these, the methods would be consulted if the class was part of a "composite" (i.e. non-single-entity) result set. Also in 0.6.9. ¶ References: #2215
- **[orm]** Added public attribute ".validators" to Mapper, an immutable dictionary view of all attributes that have been decorated with the @validates decorator. courtesy Stefano Fontanelli ¶ References: #2240
- **[orm]** Fixed subtle bug that caused SQL to blow up if: column_property() against subquery + joinedload + LIMIT + order by the column property() occurred. . Also in 0.6.9 ¶ References: #2188
- **[orm]** The join condition produced by with_parent as well as when using a "dynamic" relationship against a parent will generate unique bindparams, rather than incorrectly repeating the same bindparam. . Also in 0.6.9. ¶ References: #2207
- **[orm]** Added the same "columns-only" check to mapper.polymorphic_on as used when receiving user arguments to relationship.order_by, foreign_keys, remote_side, etc. ¶
- **[orm]** Fixed bug whereby comparison of column expression to a Query() would not call as_scalar() on the underlying SELECT statement to produce a scalar subquery, in the way that occurs if you called it on Query().subquery(). ¶ References: #2190
- **[orm]** Fixed declarative bug where a class inheriting from a superclass of the same name would fail due to an unnecessary lookup of the name in the _decl_class_registry. ¶ References: #2194
- **[orm]** Repaired the "no statement condition" assertion in Query which would attempt to raise if a generative method were called after from_statement() were called.. Also in 0.6.9. ¶ References: #2199

engine

- **[engine]** Context manager provided by Connection.begin() will issue rollback() if the commit() fails, not just if an exception occurs. ¶
- **[engine]** Use urllib.parse_qsl() in Python 2.6 and above, no deprecation warning about cgi.parse_qsl() ¶ References: #1682
- **[engine]** Added mixin class sqlalchemy.ext.DontWrapMixin. User-defined exceptions of this type are never wrapped in StatementException when they occur in the context of a statement execution. ¶
- **[engine]** StatementException wrapping will display the original exception class in the message. ¶
- **[engine]** Failures on connect which raise dbapi.Error will forward the error to dialect.is_disconnect() and set the "connection_invalidated" flag if the dialect knows this to be a potentially "retryable" condition. Only Oracle ORA-01033 implemented for now. ¶ References: #2201

sql

- **[sql]** Fixed two subtle bugs involving column correspondence in a selectable, one with the same labeled subquery repeated, the other when the label has been "grouped" and loses itself. Affects. ¶ References: #2188

schema

- **[schema]** New feature: with_variant() method on all types. Produces an instance of Variant(), a special TypeDecorator which will select the usage of a different type based on the dialect in use. ¶ References: #2187
- **[schema]** Added an informative error message when ForeignKeyConstraint refers to a column name in the parent that is not found. Also in 0.6.9. ¶
- **[schema]** Fixed bug whereby adaptation of old append_ddl_listener() function was passing unexpected **kw through to the Table event. Table gets no kws, the MetaData event in 0.6 would get "tables=somecollection", this behavior is preserved. ¶ References: #2206
- **[schema]** Fixed bug where "autoincrement" detection on Table would fail if the type had no "affinity" value, in particular this would occur when using the UUID example on the site that uses TypeEngine as the "impl". ¶
- **[schema]** Added an improved repr() to TypeEngine objects that will only display constructor args which are positional or kwargs that deviate from the default. ¶ References: #2209

postgresql

- **[postgresql]** Added new "postgresql_ops" argument to Index, allows specification of PostgreSQL operator classes for indexed columns. Courtesy Filip Zyzniewski. ¶ References: #2198

mysql

- **[mysql]** Fixed OurSQL dialect to use ansi-neutral quote symbol "" for XA commands instead of "". . Also in 0.6.9. ¶ References: #2186

sqlite

- **[sqlite]** SQLite dialect no longer strips quotes off of reflected default value, allowing a round trip CREATE TABLE to work. This is consistent with other dialects that also maintain the exact form of the default. ¶ References: #2189

mssql

- **[mssql]** Adjusted the pyodbc dialect such that bound values are passed as bytes and not unicode if the "Easysoft" unix drivers are detected. This is the same behavior as occurs with FreeTDS. Easysoft appears to segfault if Python unicodes are passed under certain circumstances. ¶

oracle

- **[oracle]** Added ORA-00028 to disconnect codes, use cx_oracle _Error.code to get at the code,. Also in 0.6.9. ¶ References: #2200

- **[oracle]** Added ORA-01033 to disconnect codes, which can be caught during a connection event. ¶ References: #2201
- **[oracle]** repaired the oracle.RAW type which did not generate the correct DDL. Also in 0.6.9. ¶ References: #2220
- **[oracle]** added CURRENT to reserved word list. Also in 0.6.9. ¶ References: #2212
- **[oracle]** Fixed bug in the mutable extension whereby if the same type were used twice in one mapping, the attributes beyond the first would not get instrumented. ¶
- **[oracle]** Fixed bug in the mutable extension whereby if None or a non-corresponding type were set, an error would be raised. None is now accepted which assigns None to all attributes, illegal values raise ValueError. ¶

misc

- **[examples]** Repaired the examples/versioning test runner to not rely upon SQLAlchemy test libs, nosetests must be run from within examples/versioning to get around setup.cfg breaking it. ¶
- **[examples]** Tweak to examples/versioning to pick the correct foreign key in a multi-level inheritance situation. ¶
- **[examples]** Fixed the attribute shard example to check for bind param callable correctly in 0.7 style. ¶

0.7.1

Released: Sun Jun 05 2011

general

- **[general]** Added a workaround for Python bug 7511 where failure of C extension build does not raise an appropriate exception on Windows 64 bit + VC express ¶ References: #2184

orm

- **[orm]** "delete-orphan" cascade is now allowed on self-referential relationships - this since SQLA 0.7 no longer enforces "parent with no child" at the ORM level; this check is left up to foreign key nullability. Related to ¶ References: #1912
- **[orm]** Repaired new "mutable" extension to propagate events to subclasses correctly; don't create multiple event listeners for subclasses either. ¶ References: #2180
- **[orm]** Modify the text of the message which occurs when the "identity" key isn't detected on flush, to include the common cause that the Column isn't set up to detect auto-increment correctly;. Also in 0.6.8. ¶ References: #2170
- **[orm]** Fixed bug where transaction-level "deleted" collection wouldn't be cleared of expunged states, raising an error if they later became transient. Also in 0.6.8. ¶ References: #2182

engine

- **[engine]** Deprecate schema/SQL-oriented methods on Connection/Engine that were never well known and are redundant: reflecttable(), create(), drop(), text(), engine.func ¶

- **[engine]** Adjusted the __contains__() method of a RowProxy result row such that no exception throw is generated internally; NoSuchColumnError() also will generate its message regardless of whether or not the column construct can be coerced to a string.. Also in 0.6.8. ¶ References: #2178

sql

- **[sql]** Fixed bug whereby metadata.reflect(bind) would close a Connection passed as a bind argument. Regression from 0.6. ¶
- **[sql]** Streamlined the process by which a Select determines what's in its '.c' collection. Behaves identically, except that a raw ClauseList() passed to select([]) (which is not a documented case anyway) will now be expanded into its individual column elements instead of being ignored. ¶

postgresql

- **[postgresql]** Some unit test fixes regarding numeric arrays, MATCH operator. A potential floating-point inaccuracy issue was fixed, and certain tests of the MATCH operator only execute within an EN-oriented locale for now. . Also in 0.6.8. ¶ References: #2175

mysql

- **[mysql]** Unit tests pass 100% on MySQL installed on windows. ¶
- **[mysql]** Removed the "adjust casing" step that would fail when reflecting a table on MySQL on windows with a mixed case name. After some experimenting with a windows MySQL server, it's been determined that this step wasn't really helping the situation much; MySQL does not return FK names with proper casing on non-windows platforms either, and removing the step at least allows the reflection to act more like it does on other OSes. A warning here has been considered but its difficult to determine under what conditions such a warning can be raised, so punted on that for now - added some docs instead. ¶ References: #2181
- **[mysql]** supports_sane_rowcount will be set to False if using MySQLdb and the DBAPI doesn't provide the constants.CLIENT module. ¶

sqlite

- **[sqlite]** Accept None from cursor.fetchone() when "PRAGMA read_uncommitted" is called to determine current isolation mode at connect time and default to SERIALIZABLE; this to support SQLite versions pre-3.3.0 that did not have this feature. ¶ References: #2173

0.7.0

Released: Fri May 20 2011

orm

- **[orm]** Fixed regression introduced in 0.7b4 (!) whereby query.options(someoption("nonexistent name")) would fail to raise an error. Also added additional error catching for cases where the option would try to build off a column-based element, further fixed up some of the error messages tailored in ¶ References: #2069
- **[orm]** query.count() emits "count(*)" instead of "count(1)". ¶ References: #2162

- **[orm]** Fine tuning of Query clause adaptation when from_self(), union(), or other "select from myself" operation, such that plain SQL expression elements added to filter(), order_by() etc. which are present in the nested "from myself" query *will* be adapted in the same way an ORM expression element will, since these elements are otherwise not easily accessible. ¶ References: #2155
- **[orm]** Fixed bug where determination of "self referential" relationship would fail with no workaround for joined-inh subclass related to itself, or joined-inh subclass related to a subclass of that with no cols in the sub-sub class in the join condition. Also in 0.6.8. ¶ References: #2149
- **[orm]** mapper() will ignore non-configured foreign keys to unrelated tables when determining inherit condition between parent and child class, but will raise as usual for unresolved columns and table names regarding the inherited table. This is an enhanced generalization of behavior that was already applied to declarative previously. 0.6.8 has a more conservative version of this which doesn't fundamentally alter how join conditions are determined. ¶ References: #2153
- **[orm]** It is an error to call query.get() when the given entity is not a single, full class entity or mapper (i.e. a column). This is a deprecation warning in 0.6.8. ¶ References: #2144
- **[orm]** Fixed a potential KeyError which under some circumstances could occur with the identity map, part of ¶ References: #2148
- **[orm]** added Query.with_session() method, switches Query to use a different session. ¶
- **[orm]** horizontal shard query should use execution options per connection as per ¶ References: #2131
- **[orm]** a non_primary mapper will inherit the _identity_class of the primary mapper. This so that a non_primary established against a class that's normally in an inheritance mapping will produce results that are identity-map compatible with that of the primary mapper (also in 0.6.8) ¶ References: #2151
- **[orm]** Fixed the error message emitted for "can't execute syncrule for destination column 'q'; mapper 'X' does not map this column" to reference the correct mapper. . Also in 0.6.8. ¶ References: #2163
- **[orm]** polymorphic_union() gets a "cast_nulls" option, disables the usage of CAST when it renders the labeled NULL columns. ¶ References: #1502
- **[orm]** polymorphic_union() renders the columns in their original table order, as according to the first table/selectable in the list of polymorphic unions in which they appear. (which is itself an unordered mapping unless you pass an OrderedDict). ¶
- **[orm]** Fixed bug whereby mapper mapped to an anonymous alias would fail if logging were used, due to unescaped % sign in the alias name. Also in 0.6.8. ¶ References: #2171

sql

- **[sql]** Fixed bug whereby nesting a label of a select() with another label in it would produce incorrect exported columns. Among other things this would break an ORM column_property() mapping against another column_property(). . Also in 0.6.8 ¶ References: #2167
- **[sql]** Changed the handling in determination of join conditions such that foreign key errors are only considered between the two given tables. That is, t1.join(t2) will report FK errors that involve 't1' or 't2', but anything involving 't3' will be skipped. This affects join(), as well as ORM relationship and inherit condition logic. ¶
- **[sql]** Some improvements to error handling inside of the execute procedure to ensure auto-close connections are really closed when very unusual DBAPI errors occur. ¶
- **[sql]** metadata.reflect() and reflection.Inspector() had some reliance on GC to close connections which were internally procured, fixed this. ¶
- **[sql]** Added explicit check for when Column .name is assigned as blank string ¶ References: #2140

- **[sql]** Fixed bug whereby if FetchedValue was passed to column server_onupdate, it would not have its parent "column" assigned, added test coverage for all column default assignment patterns. also in 0.6.8 ¶ References: #2147

postgresql

- **[postgresql]** Fixed the psycopg2_version parsing in the psycopg2 dialect. ¶
- **[postgresql]** Fixed bug affecting PG 9 whereby index reflection would fail if against a column whose name had changed. . Also in 0.6.8. ¶ References: #2141

mssql

- **[mssql]** Fixed bug in MSSQL dialect whereby the aliasing applied to a schema-qualified table would leak into enclosing select statements. Also in 0.6.8. ¶ References: #2169

misc

- This section documents those changes from 0.7b4 to 0.7.0. For an overview of what's new in SQLAlchemy 0.7, see http://www.sqlalchemy.org/trac/wiki/07Migration ¶
- **[documentation]** Removed the usage of the "collections.MutableMapping" abc from the ext.mutable docs as it was being used incorrectly and makes the example more difficult to understand in any case. ¶ References: #2152
- **[examples]** removed the ancient "polymorphic association" examples and replaced with an updated set of examples that use declarative mixins, "generic_associations". Each presents an alternative table layout. ¶
- **[ext]** Fixed bugs in sqlalchemy.ext.mutable extension where *None* was not appropriately handled, replacement events were not appropriately handled. ¶ References: #2143

0.7.0b4

Released: Sun Apr 17 2011

general

- **[general]** Changes to the format of CHANGES, this file. The format changes have been applied to the 0.7 releases. ¶
- **[general]** The "-declarative" changes will now be listed directly under the "-orm" section, as these are closely related. ¶
- **[general]** The 0.5 series changes have been moved to the file CHANGES_PRE_06 which replaces CHANGES_PRE_05. ¶
- **[general]** The changelog for 0.6.7 and subsequent within the 0.6 series is now listed only in the CHANGES file within the 0.6 branch. In the 0.7 CHANGES file (i.e. this file), all the 0.6 changes are listed inline within the 0.7 section in which they were also applied (since all 0.6 changes are in 0.7 as well). Changes that apply to an 0.6 version here are noted as are if any differences in implementation/behavior are present. ¶

orm

- **[orm]** Some fixes to "evaluate" and "fetch" evaluation when query.update(), query.delete() are called. The retrieval of records is done after autoflush in all cases, and before update/delete is emitted, guarding against unflushed data present as well as expired objects failing during the evaluation. ¶ References: #2122
- **[orm]** Reworded the exception raised when a flush is attempted of a subclass that is not polymorphic against the supertype. ¶ References: #2063
- **[orm]** Still more wording adjustments when a query option can't find the target entity. Explain that the path must be from one of the root entities. ¶
- **[orm]** Some fixes to the state handling regarding backrefs, typically when autoflush=False, where the back-referenced collection wouldn't properly handle add/removes with no net change. Thanks to Richard Murri for the test case + patch. (also in 0.6.7). ¶ References: #2123
- **[orm]** Added checks inside the UOW to detect the unusual condition of being asked to UPDATE or DELETE on a primary key value that contains NULL in it. ¶ References: #2127
- **[orm]** Some refinements to attribute history. More changes are pending possibly in 0.8, but for now history has been modified such that scalar history doesn't have a "side effect" of populating None for a non-present value. This allows a slightly better ability to distinguish between a None set and no actual change, affects as well. ¶ References: #2127
- **[orm]** a "having" clause would be copied from the inside to the outside query if from_self() were used; in particular this would break an 0.7 style count() query. (also in 0.6.7) ¶ References: #2130
- **[orm]** the Query.execution_options() method now passes those options to the Connection rather than the SELECT statement, so that all available options including isolation level and compiled cache may be used. ¶ References: #2131

engine

- **[engine]** The C extension is now enabled by default on CPython 2.x with a fallback to pure python if it fails to compile. ¶ References: #2129

sql

- **[sql]** The "compiled_cache" execution option now raises an error when passed to a SELECT statement rather than a Connection. Previously it was being ignored entirely. We may look into having this option work on a per-statement level at some point. ¶ References: #2131
- **[sql]** Restored the "catchall" constructor on the base TypeEngine class, with a deprecation warning. This so that code which does something like Integer(11) still succeeds. ¶
- **[sql]** Fixed regression whereby MetaData() coming back from unpickling did not keep track of new things it keeps track of now, i.e. collection of Sequence objects, list of schema names. ¶ References: #2104
- **[sql]** The limit/offset keywords to select() as well as the value passed to select.limit()/offset() will be coerced to integer. (also in 0.6.7) ¶ References: #2116
- **[sql]** fixed bug where "from" clause gathering from an over() clause would be an itertools.chain() and not a list, causing "can only concatenate list" TypeError when combined with other clauses. ¶
- **[sql]** Fixed incorrect usage of "," in over() clause being placed between the "partition" and "order by" clauses. ¶ References: #2134

- **[sql]** Before/after attach events for PrimaryKeyConstraint now function, tests added for before/after events on all constraint types. ¶ References: #2105
- **[sql]** Added explicit true()/false() constructs to expression lib - coercion rules will intercept "False"/"True" into these constructs. In 0.6, the constructs were typically converted straight to string, which was no longer accepted in 0.7. ¶ References: #2117

schema

- **[schema]** The 'useexisting' flag on Table has been superseded by a new pair of flags 'keep_existing' and 'extend_existing'. 'extend_existing' is equivalent to 'useexisting' - the existing Table is returned, and additional constructor elements are added. With 'keep_existing', the existing Table is returned, but additional constructor elements are not added - these elements are only applied when the Table is newly created. ¶ References: #2109

postgresql

- **[postgresql]** Psycopg2 for Python 3 is now supported. ¶
- **[postgresql]** Fixed support for precision numerics when using pg8000. ¶ References: #2132

sqlite

- **[sqlite]** Fixed bug where reflection of foreign key created as "REFERENCES <tablename>" without col name would fail. (also in 0.6.7) ¶ References: #2115

oracle

- **[oracle]** Using column names that would require quotes for the column itself or for a name-generated bind parameter, such as names with special characters, underscores, non-ascii characters, now properly translate bind parameter keys when talking to cx_oracle. (Also in 0.6.7) ¶ References: #2100
- **[oracle]** Oracle dialect adds use_binds_for_limits=False create_engine() flag, will render the LIMIT/OFFSET values inline instead of as binds, reported to modify the execution plan used by Oracle. (Also in 0.6.7) ¶ References: #2116

misc

- **[types]** REAL has been added to the core types. Supported by Postgresql, SQL Server, MySQL, SQLite. Note that the SQL Server and MySQL versions, which add extra arguments, are also still available from those dialects. ¶ References: #2081
- **[types]** Added @event.listens_for() decorator, given target + event name, applies the decorated function as a listener. ¶ References: #2106
- **[pool]** AssertionPool now stores the traceback indicating where the currently checked out connection was acquired; this traceback is reported within the assertion raised upon a second concurrent checkout; courtesy Gunnlaugur Briem ¶ References: #2103
- **[pool]** The "pool.manage" feature doesn't use pickle anymore to hash the arguments for each pool. ¶
- **[documentation]** Documented SQLite DATE/TIME/DATETIME types. (also in 0.6.7) ¶ References: #2029
- **[documentation]** Fixed mutable extension docs to show the correct type-association methods. ¶ References: #2118

0.7.0b3

Released: Sun Mar 20 2011

general

- **[general]** Lots of fixes to unit tests when run under Pypy (courtesy Alex Gaynor). ¶

orm

- **[orm]** Changed the underlying approach to query.count(). query.count() is now in all cases exactly:

  query. from_self(func.count(literal_column('1'))). scalar()

  That is, "select count(1) from (<full query>)". This produces a subquery in all cases, but vastly simplifies all the guessing count() tried to do previously, which would still fail in many scenarios particularly when joined table inheritance and other joins were involved. If the subquery produced for an otherwise very simple count is really an issue, use query(func.count()) as an optimization. ¶ References: #2093
- **[orm]** some changes to the identity map regarding rare weakref callbacks during iterations. The mutex has been removed as it apparently can cause a reentrant (i.e. in one thread) deadlock, perhaps when gc collects objects at the point of iteration in order to gain more memory. It is hoped that "dictionary changed during iteration" will be exceedingly rare as iteration methods internally acquire the full list of objects in a single values() call. Note 0.6.7 has a more conservative fix here which still keeps the mutex in place. ¶ References: #2087
- **[orm]** A tweak to the unit of work causes it to order the flush along relationship() dependencies even if the given objects don't have any inter-attribute references in memory, which was the behavior in 0.5 and earlier, so a flush of Parent/Child with only foreign key/primary key set will succeed. This while still maintaining 0.6 and above's not generating a ton of useless internal dependency structures within the flush that don't correspond to state actually within the current flush. ¶ References: #2082
- **[orm]** Improvements to the error messages emitted when querying against column-only entities in conjunction with (typically incorrectly) using loader options, where the parent entity is not fully present. ¶ References: #2069
- **[orm]** Fixed bug in query.options() whereby a path applied to a lazyload using string keys could overlap a same named attribute on the wrong entity. Note 0.6.7 has a more conservative fix to this. ¶ References: #2098

engine

- **[engine]** Fixed AssertionPool regression bug. ¶ References: #2097
- **[engine]** Changed exception raised to ArgumentError when an invalid dialect is specified. ¶ References: #2060

sql

- **[sql]** Added a fully descriptive error message for the case where Column is subclassed and _make_proxy() fails to make a copy due to TypeError on the constructor. The method _constructor should be implemented in this case. ¶
- **[sql]** Added new event "column_reflect" for Table objects. Receives the info dictionary about a Column before the object is generated within reflection, and allows modification to the dictionary for control over most aspects of the resulting Column including key, name, type, info dictionary. ¶ References: #2095

- **[sql]** To help with the "column_reflect" event being used with specific Table objects instead of all instances of Table, listeners can be added to a Table object inline with its construction using a new argument "listeners", a list of tuples of the form (<eventname>, <fn>), which are applied to the Table before the reflection process begins. ¶
- **[sql]** Added new generic function "next_value()", accepts a Sequence object as its argument and renders the appropriate "next value" generation string on the target platform, if supported. Also provides ".next_value()" method on Sequence itself. ¶ References: #2085
- **[sql]** func.next_value() or other SQL expression can be embedded directly into an insert() construct, and if implicit or explicit "returning" is used in conjunction with a primary key column, the newly generated value will be present in result.inserted_primary_key. ¶ References: #2084
- **[sql]** Added accessors to ResultProxy "returns_rows", "is_insert" (also in 0.6.7) ¶ References: #2089

postgresql

- **[postgresql]** Added RESERVED_WORDS for postgresql dialect. (also in 0.6.7) ¶ References: #2092
- **[postgresql]** Fixed the BIT type to allow a "length" parameter, "varying" parameter. Reflection also fixed. (also in 0.6.7) ¶ References: #2073

mssql

- **[mssql]** Rewrote the query used to get the definition of a view, typically when using the Inspector interface, to use sys.sql_modules instead of the information schema, thereby allowing views definitions longer than 4000 characters to be fully returned. (also in 0.6.7) ¶ References: #2071

firebird

- **[firebird]** The "implicit_returning" flag on create_engine() is honored if set to False. (also in 0.6.7) ¶ References: #2083

misc

- **[declarative]** Arguments in __mapper_args__ that aren't "hashable" aren't mistaken for always-hashable, possibly-column arguments. (also in 0.6.7) ¶ References: #2091
- **[informix]** Added RESERVED_WORDS informix dialect. (also in 0.6.7) ¶ References: #2092
- **[ext]** The horizontal_shard ShardedSession class accepts the common Session argument "query_cls" as a constructor argument, to enable further subclassing of ShardedQuery. (also in 0.6.7) ¶ References: #2090
- **[examples]** Updated the association, association proxy examples to use declarative, added a new example dict_of_sets_with_default.py, a "pushing the envelope" example of association proxy. ¶
- **[examples]** The Beaker caching example allows a "query_cls" argument to the query_callable() function. (also in 0.6.7) ¶ References: #2090

0.7.0b2

Released: Sat Feb 19 2011

orm

- **[orm]** Fixed bug whereby Session.merge() would call the load() event with one too few arguments. ¶ References: #2053
- **[orm]** Added logic which prevents the generation of events from a MapperExtension or SessionExtension from generating do-nothing events for all the methods not overridden. ¶ References: #2052

sql

- **[sql]** Renamed the EngineEvents event class to ConnectionEvents. As these classes are never accessed directly by end-user code, this strictly is a documentation change for end users. Also simplified how events get linked to engines and connections internally. ¶ References: #2059
- **[sql]** The Sequence() construct, when passed a MetaData() object via its 'metadata' argument, will be included in CREATE/DROP statements within metadata.create_all() and metadata.drop_all(), including "checkfirst" logic. ¶ References: #2055
- **[sql]** The Column.references() method now returns True if it has a foreign key referencing the given column exactly, not just its parent table. ¶ References: #2064

postgresql

- **[postgresql]** Fixed regression from 0.6 where SMALLINT and BIGINT types would both generate SERIAL on an integer PK column, instead of SMALLINT and BIGSERIAL ¶ References: #2065

misc

- **[declarative]** Fixed regression whereby composite() with Column objects placed inline would fail to initialize. The Column objects can now be inline with the composite() or external and pulled in via name or object ref. ¶ References: #2058
- **[declarative]** Fix error message referencing old @classproperty name to reference @declared_attr (also in 0.6.7) ¶ References: #2061
- **[declarative]** the dictionary at the end of the __table_args__ tuple is now optional. ¶ References: #1468
- **[ext]** Association proxy now has correct behavior for any(), has(), and contains() when proxying a many-to-one scalar attribute to a one-to-many collection (i.e. the reverse of the 'typical' association proxy use case) ¶ References: #2054
- **[examples]** Beaker example now takes into account 'limit' and 'offset', bind params within embedded FROM clauses (like when you use union() or from_self()) when generating a cache key. ¶

0.7.0b1

Released: Sat Feb 12 2011

general

- **[general]** New event system, supersedes all extensions, listeners, etc. ¶ References: #1902
- **[general]** Logging enhancements ¶ References: #1926

- **[general]** Setup no longer installs a Nose plugin ¶ References: #1949
- **[general]** The "sqlalchemy.exceptions" alias in sys.modules has been removed. Base SQLA exceptions are available via "from sqlalchemy import exc". The "exceptions" alias for "exc" remains in "sqlalchemy" for now, it's just not patched into sys.modules. ¶

orm

- **[orm]** More succinct form of query.join(target, onclause) ¶ References: #1923
- **[orm]** Hybrid Attributes, implements/supersedes synonym() ¶ References: #1903
- **[orm]** Rewrite of composites ¶ References: #2008
- **[orm]** Mutation Event Extension, supersedes "mutable=True"

  See also:

  Mutation event extension, supersedes "mutable=True"

  ¶
- **[orm]** PickleType and ARRAY mutability turned off by default ¶ References: #1980
- **[orm]** Simplified polymorphic_on assignment ¶ References: #1895
- **[orm]** Flushing of Orphans that have no parent is allowed ¶ References: #1912
- **[orm]** Adjusted flush accounting step to occur before the commit in the case of autocommit=True. This allows autocommit=True to work appropriately with expire_on_commit=True, and also allows post-flush session hooks to operate in the same transactional context as when autocommit=False. ¶ References: #2041
- **[orm]** Warnings generated when collection members, scalar referents not part of the flush ¶ References: #1973
- **[orm]** Non-*Table*-derived constructs can be mapped ¶ References: #1876
- **[orm]** Tuple label names in Query Improved ¶ References: #1942
- **[orm]** Mapped column attributes reference the most specific column first ¶ References: #1892
- **[orm]** Mapping to joins with two or more same-named columns requires explicit declaration ¶ References: #1896
- **[orm]** Mapper requires that polymorphic_on column be present in the mapped selectable ¶ References: #1875
- **[orm]** compile_mappers() renamed configure_mappers(), simplified configuration internals ¶ References: #1966
- **[orm]** the aliased() function, if passed a SQL FromClause element (i.e. not a mapped class), will return element.alias() instead of raising an error on AliasedClass. ¶ References: #2018
- **[orm]** Session.merge() will check the version id of the incoming state against that of the database, assuming the mapping uses version ids and incoming state has a version_id assigned, and raise StaleDataError if they don't match. ¶ References: #2027
- **[orm]** Session.connection(), Session.execute() accept 'bind', to allow execute/connection operations to participate in the open transaction of an engine explicitly. ¶ References: #1996
- **[orm]** Query.join(), Query.outerjoin(), eagerload(), eagerload_all(), others no longer allow lists of attributes as arguments (i.e. option([x, y, z]) form, deprecated since 0.5) ¶
- **[orm]** ScopedSession.mapper is removed (deprecated since 0.5). ¶
- **[orm]** Horizontal shard query places 'shard_id' in context.attributes where it's accessible by the "load()" event. ¶ References: #2031

- **[orm]** A single contains_eager() call across multiple entities will indicate all collections along that path should load, instead of requiring distinct contains_eager() calls for each endpoint (which was never correctly documented). ¶ References: #2032
- **[orm]** The "name" field used in orm.aliased() now renders in the resulting SQL statement. ¶
- **[orm]** Session weak_instance_dict=False is deprecated. ¶ References: #1473
- **[orm]** An exception is raised in the unusual case that an append or similar event on a collection occurs after the parent object has been dereferenced, which prevents the parent from being marked as "dirty" in the session. Was a warning in 0.6.6. ¶ References: #2046
- **[orm]** Query.distinct() now accepts column expressions as *args, interpreted by the Postgresql dialect as DISTINCT ON (<expr>). ¶ References: #1069
- **[orm]** Additional tuning to "many-to-one" relationship loads during a flush(). A change in version 0.6.6 ([ticket:2002]) required that more "unnecessary" m2o loads during a flush could occur. Extra loading modes have been added so that the SQL emitted in this specific use case is trimmed back, while still retrieving the information the flush needs in order to not miss anything. ¶ References: #2049
- **[orm]** the value of "passive" as passed to attributes.get_history() should be one of the constants defined in the attributes package. Sending True or False is deprecated. ¶
- **[orm]** Added a *name* argument to *Query.subquery()*, to allow a fixed name to be assigned to the alias object. (also in 0.6.7) ¶ References: #2030
- **[orm]** A warning is emitted when a joined-table inheriting mapper has no primary keys on the locally mapped table (but has pks on the superclass table). (also in 0.6.7) ¶ References: #2019
- **[orm]** Fixed bug where "middle" class in a polymorphic hierarchy would have no 'polymorphic_on' column if it didn't also specify a 'polymorphic_identity', leading to strange errors upon refresh, wrong class loaded when querying from that target. Also emits the correct WHERE criterion when using single table inheritance. (also in 0.6.7) ¶ References: #2038
- **[orm]** Fixed bug where a column with a SQL or server side default that was excluded from a mapping with include_properties or exclude_properties would result in UnmappedColumnError. (also in 0.6.7) ¶ References: #1995
- **[orm]** A warning is emitted in the unusual case that an append or similar event on a collection occurs after the parent object has been dereferenced, which prevents the parent from being marked as "dirty" in the session. This will be an exception in 0.7. (also in 0.6.7) ¶ References: #2046

sql

- **[sql]** Added over() function, method to FunctionElement classes, produces the _Over() construct which in turn generates "window functions", i.e. "<window function> OVER (PARTITION BY <partition by>, ORDER BY <order by>)". ¶ References: #1844
- **[sql]** LIMIT/OFFSET clauses now use bind parameters ¶ References: #805
- **[sql]** select.distinct() now accepts column expressions as *args, interpreted by the Postgresql dialect as DISTINCT ON (<expr>). Note this was already available via passing a list to the *distinct* keyword argument to select(). ¶ References: #1069
- **[sql]** select.prefix_with() accepts multiple expressions (i.e. *expr), 'prefix' keyword argument to select() accepts a list or tuple. ¶
- **[sql]** Passing a string to the *distinct* keyword argument of *select()* for the purpose of emitting special MySQL keywords (DISTINCTROW etc.) is deprecated - use *prefix_with()* for this. ¶
- **[sql]** TypeDecorator works with primary key columns ¶ References: #2006, #2005

- **[sql]** DDL() constructs now escape percent signs ¶ References: #1897
- **[sql]** Table.c / MetaData.tables refined a bit, don't allow direct mutation ¶ References: #1917, #1893
- **[sql]** Callables passed to *bindparam()* don't get evaluated ¶ References: #1950
- **[sql]** types.type_map is now private, types._type_map ¶ References: #1870
- **[sql]** Non-public Pool methods underscored ¶ References: #1982
- **[sql]** Added NULLS FIRST and NULLS LAST support. It's implemented as an extension to the asc() and desc() operators, called nullsfirst() and nullslast(). ¶ References: #723
- **[sql]** The Index() construct can be created inline with a Table definition, using strings as column names, as an alternative to the creation of the index outside of the Table. ¶
- **[sql]** execution_options() on Connection accepts "isolation_level" argument, sets transaction isolation level for that connection only until returned to the connection pool, for those backends which support it (SQLite, Postgresql) ¶ References: #2001
- **[sql]** A TypeDecorator of Integer can be used with a primary key column, and the "autoincrement" feature of various dialects as well as the "sqlite_autoincrement" flag will honor the underlying database type as being Integer-based. ¶ References: #2005
- **[sql]** Established consistency when server_default is present on an Integer PK column. SQLA doesn't pre-fetch these, nor do they come back in cursor.lastrowid (DBAPI). Ensured all backends consistently return None in result.inserted_primary_key for these. Regarding reflection for this case, reflection of an int PK col with a server_default sets the "autoincrement" flag to False, except in the case of a PG SERIAL col where we detected a sequence default. ¶ References: #2020, #2021
- **[sql]** Result-row processors are applied to pre-executed SQL defaults, as well as cursor.lastrowid, when determining the contents of result.inserted_primary_key. ¶ References: #2006
- **[sql]** Bind parameters present in the "columns clause" of a select are now auto-labeled like other "anonymous" clauses, which among other things allows their "type" to be meaningful when the row is fetched, as in result row processors. ¶
- **[sql]** TypeDecorator is present in the "sqlalchemy" import space. ¶
- **[sql]** Non-DBAPI errors which occur in the scope of an *execute()* call are now wrapped in sqlalchemy.exc.StatementError, and the text of the SQL statement and repr() of params is included. This makes it easier to identify statement executions which fail before the DBAPI becomes involved. ¶ References: #2015
- **[sql]** The concept of associating a ".bind" directly with a ClauseElement has been explicitly moved to Executable, i.e. the mixin that describes ClauseElements which represent engine-executable constructs. This change is an improvement to internal organization and is unlikely to affect any real-world usage. ¶ References: #2048
- **[sql]** Column.copy(), as used in table.tometadata(), copies the 'doc' attribute. (also in 0.6.7) ¶ References: #2028
- **[sql]** Added some defs to the resultproxy.c extension so that the extension compiles and runs on Python 2.4. (also in 0.6.7) ¶ References: #2023
- **[sql]** The compiler extension now supports overriding the default compilation of expression._BindParamClause including that the auto-generated binds within the VALUES/SET clause of an insert()/update() statement will also use the new compilation rules. (also in 0.6.7) ¶ References: #2042
- **[sql]** SQLite dialect now uses *NullPool* for file-based databases ¶ References: #1921
- **[sql]** The path given as the location of a sqlite database is now normalized via os.path.abspath(), so that directory changes within the process don't affect the ultimate location of a relative file path. ¶ References: #2036

postgresql

- **[postgresql]** When explicit sequence execution derives the name of the auto-generated sequence of a SERIAL column, which currently only occurs if implicit_returning=False, now accommodates if the table + column name is greater than 63 characters using the same logic Postgresql uses. (also in 0.6.7) ¶ References: #1083
- **[postgresql]** Added an additional libpq message to the list of "disconnect" exceptions, "could not receive data from server" (also in 0.6.7) ¶ References: #2044

mysql

- **[mysql]** New DBAPI support for pymysql, a pure Python port of MySQL-python. ¶ References: #1991
- **[mysql]** oursql dialect accepts the same "ssl" arguments in create_engine() as that of MySQLdb. (also in 0.6.7) ¶ References: #2047

mssql

- **[mssql]** the String/Unicode types, and their counterparts VARCHAR/ NVARCHAR, emit "max" as the length when no length is specified, so that the default length, normally '1' as per SQL server documentation, is instead 'unbounded'. This also occurs for the VARBINARY type..

  This behavior makes these types more closely compatible with Postgresql's VARCHAR type which is similarly unbounded when no length is specified. ¶ References: #1833

firebird

- **[firebird]** Some adjustments so that Interbase is supported as well. FB/Interbase version idents are parsed into a structure such as (8, 1, 1, 'interbase') or (2, 1, 588, 'firebird') so they can be distinguished. ¶ References: #1885

misc

- Detailed descriptions of each change below are described at: http://www.sqlalchemy.org/trac/wiki/07Migration ¶
- **[declarative]** Added an explicit check for the case that the name 'metadata' is used for a column attribute on a declarative class. (also in 0.6.7) ¶ References: #2050

6.2.6 0.6 Changelog

0.6.9

Released: Sat May 05 2012

general

- **[general]** Adjusted the "importlater" mechanism, which is used internally to resolve import cycles, such that the usage of __import__ is completed when the import of sqlalchemy or sqlalchemy.orm is done, thereby avoiding any usage of __import__ after the application starts new threads, fixes. ¶ References: #2279

orm

- **[orm] [bug]** fixed inappropriate evaluation of user-mapped object in a boolean context within query.get(). ¶ References: #2310
- **[orm] [bug]** Fixed the error formatting raised when a tuple is inadvertently passed to session.query(). ¶ References: #2297
- **[orm]** Fixed bug whereby the source clause used by query.join() would be inconsistent if against a column expression that combined multiple entities together. ¶ References: #2197
- **[orm]** Fixed bug apparent only in Python 3 whereby sorting of persistent + pending objects during flush would produce an illegal comparison, if the persistent object primary key is not a single integer. ¶ References: #2228
- **[orm]** Fixed bug where query.join() + aliased=True from a joined-inh structure to itself on relationship() with join condition on the child table would convert the lead entity into the joined one inappropriately. ¶ References: #2234
- **[orm]** Fixed bug whereby mapper.order_by attribute would be ignored in the "inner" query within a subquery eager load. . ¶ References: #2287
- **[orm]** Fixed bug whereby if a mapped class redefined __hash__() or __eq__() to something non-standard, which is a supported use case as SQLA should never consult these, the methods would be consulted if the class was part of a "composite" (i.e. non-single-entity) result set. ¶ References: #2215
- **[orm]** Fixed subtle bug that caused SQL to blow up if: column_property() against subquery + joinedload + LIMIT + order by the column property() occurred. . ¶ References: #2188
- **[orm]** The join condition produced by with_parent as well as when using a "dynamic" relationship against a parent will generate unique bindparams, rather than incorrectly repeating the same bindparam. . ¶ References: #2207
- **[orm]** Repaired the "no statement condition" assertion in Query which would attempt to raise if a generative method were called after from_statement() were called.. ¶ References: #2199
- **[orm]** Cls.column.collate("some collation") now works. ¶ References: #1776

engine

- **[engine]** Backported the fix for introduced in 0.7.4, which ensures that the connection is in a valid state before attempting to call rollback()/prepare()/release() on savepoint and two-phase transactions. ¶ References: #2317

sql

- **[sql]** Fixed two subtle bugs involving column correspondence in a selectable, one with the same labeled subquery repeated, the other when the label has been "grouped" and loses itself. Affects. ¶ References: #2188
- **[sql]** Fixed bug whereby "warn on unicode" flag would get set for the String type when used with certain dialects. This bug is not in 0.7. ¶
- **[sql]** Fixed bug whereby with_only_columns() method of Select would fail if a selectable were passed.. However, the FROM behavior is still incorrect here, so you need 0.7 in any case for this use case to be usable. ¶ References: #2270

schema

- **[schema]** Added an informative error message when ForeignKeyConstraint refers to a column name in the parent that is not found. ¶

postgresql

- **[postgresql]** Fixed bug related to whereby the same modified index behavior in PG 9 affected primary key reflection on a renamed column.. ¶ References: #2291, #2141

mysql

- **[mysql]** Fixed OurSQL dialect to use ansi-neutral quote symbol "'" for XA commands instead of "'". . ¶ References: #2186
- **[mysql]** a CREATE TABLE will put the COLLATE option after CHARSET, which appears to be part of MySQL's arbitrary rules regarding if it will actually work or not. ¶ References: #2225

mssql

- **[mssql] [bug]** Decode incoming values when retrieving list of index names and the names of columns within those indexes. ¶ References: #2269

oracle

- **[oracle]** Added ORA-00028 to disconnect codes, use cx_oracle _Error.code to get at the code,. ¶ References: #2200
- **[oracle]** repaired the oracle.RAW type which did not generate the correct DDL. ¶ References: #2220
- **[oracle]** added CURRENT to reserved word list. ¶ References: #2212

misc

- **[examples]** Adjusted dictlike-polymorphic.py example to apply the CAST such that it works on PG, other databases. ¶ References: #2266

0.6.8

Released: Sun Jun 05 2011

orm

- **[orm]** Calling query.get() against a column-based entity is invalid, this condition now raises a deprecation warning. ¶ References: #2144
- **[orm]** a non_primary mapper will inherit the _identity_class of the primary mapper. This so that a non_primary established against a class that's normally in an inheritance mapping will produce results that are identity-map compatible with that of the primary mapper ¶ References: #2151

- **[orm]** Backported 0.7's identity map implementation, which does not use a mutex around removal. This as some users were still getting deadlocks despite the adjustments in 0.6.7; the 0.7 approach that doesn't use a mutex does not appear to produce "dictionary changed size" issues, the original rationale for the mutex. ¶ References: #2148
- **[orm]** Fixed the error message emitted for "can't execute syncrule for destination column 'q'; mapper 'X' does not map this column" to reference the correct mapper. . ¶ References: #2163
- **[orm]** Fixed bug where determination of "self referential" relationship would fail with no workaround for joined-inh subclass related to itself, or joined-inh subclass related to a subclass of that with no cols in the sub-sub class in the join condition. ¶ References: #2149
- **[orm]** mapper() will ignore non-configured foreign keys to unrelated tables when determining inherit condition between parent and child class. This is equivalent to behavior already applied to declarative. Note that 0.7 has a more comprehensive solution to this, altering how join() itself determines an FK error. ¶ References: #2153
- **[orm]** Fixed bug whereby mapper mapped to an anonymous alias would fail if logging were used, due to unescaped % sign in the alias name. ¶ References: #2171
- **[orm]** Modify the text of the message which occurs when the "identity" key isn't detected on flush, to include the common cause that the Column isn't set up to detect auto-increment correctly;. ¶ References: #2170
- **[orm]** Fixed bug where transaction-level "deleted" collection wouldn't be cleared of expunged states, raising an error if they later became transient. ¶ References: #2182

engine

- **[engine]** Adjusted the __contains__() method of a RowProxy result row such that no exception throw is generated internally; NoSuchColumnError() also will generate its message regardless of whether or not the column construct can be coerced to a string.. ¶ References: #2178

sql

- **[sql]** Fixed bug whereby if FetchedValue was passed to column server_onupdate, it would not have its parent "column" assigned, added test coverage for all column default assignment patterns. ¶ References: #2147
- **[sql]** Fixed bug whereby nesting a label of a select() with another label in it would produce incorrect exported columns. Among other things this would break an ORM column_property() mapping against another column_property(). . ¶ References: #2167

postgresql

- **[postgresql]** Fixed bug affecting PG 9 whereby index reflection would fail if against a column whose name had changed. . ¶ References: #2141
- **[postgresql]** Some unit test fixes regarding numeric arrays, MATCH operator. A potential floating-point inaccuracy issue was fixed, and certain tests of the MATCH operator only execute within an EN-oriented locale for now. . ¶ References: #2175

mssql

- **[mssql]** Fixed bug in MSSQL dialect whereby the aliasing applied to a schema-qualified table would leak into enclosing select statements. ¶ References: #2169

- **[mssql]** Fixed bug whereby DATETIME2 type would fail on the "adapt" step when used in result sets or bound parameters. This issue is not in 0.7. ¶ References: #2159

0.6.7

Released: Wed Apr 13 2011

orm

- **[orm]** Tightened the iterate vs. remove mutex around the identity map iteration, attempting to reduce the chance of an (extremely rare) reentrant gc operation causing a deadlock. Might remove the mutex in 0.7. ¶ References: #2087
- **[orm]** Added a *name* argument to *Query.subquery()*, to allow a fixed name to be assigned to the alias object. ¶ References: #2030
- **[orm]** A warning is emitted when a joined-table inheriting mapper has no primary keys on the locally mapped table (but has pks on the superclass table). ¶ References: #2019
- **[orm]** Fixed bug where "middle" class in a polymorphic hierarchy would have no 'polymorphic_on' column if it didn't also specify a 'polymorphic_identity', leading to strange errors upon refresh, wrong class loaded when querying from that target. Also emits the correct WHERE criterion when using single table inheritance. ¶ References: #2038
- **[orm]** Fixed bug where a column with a SQL or server side default that was excluded from a mapping with include_properties or exclude_properties would result in UnmappedColumnError. ¶ References: #1995
- **[orm]** A warning is emitted in the unusual case that an append or similar event on a collection occurs after the parent object has been dereferenced, which prevents the parent from being marked as "dirty" in the session. This will be an exception in 0.7. ¶ References: #2046
- **[orm]** Fixed bug in query.options() whereby a path applied to a lazyload using string keys could overlap a same named attribute on the wrong entity. Note 0.7 has an updated version of this fix. ¶ References: #2098
- **[orm]** Reworded the exception raised when a flush is attempted of a subclass that is not polymorphic against the supertype. ¶ References: #2063
- **[orm]** Some fixes to the state handling regarding backrefs, typically when autoflush=False, where the back-referenced collection wouldn't properly handle add/removes with no net change. Thanks to Richard Murri for the test case + patch. ¶ References: #2123
- **[orm]** a "having" clause would be copied from the inside to the outside query if from_self() were used.. ¶ References: #2130

engine

- **[engine]** Fixed bug in QueuePool, SingletonThreadPool whereby connections that were discarded via overflow or periodic cleanup() were not explicitly closed, leaving garbage collection to the task instead. This generally only affects non-reference-counting backends like Jython and Pypy. Thanks to Jaimy Azle for spotting this. ¶ References: #2102

sql

- **[sql]** Column.copy(), as used in table.tometadata(), copies the 'doc' attribute. ¶ References: #2028

- **[sql]** Added some defs to the resultproxy.c extension so that the extension compiles and runs on Python 2.4. ¶ References: #2023
- **[sql]** The compiler extension now supports overriding the default compilation of expression._BindParamClause including that the auto-generated binds within the VALUES/SET clause of an insert()/update() statement will also use the new compilation rules. ¶ References: #2042
- **[sql]** Added accessors to ResultProxy "returns_rows", "is_insert" ¶ References: #2089
- **[sql]** The limit/offset keywords to select() as well as the value passed to select.limit()/offset() will be coerced to integer. ¶ References: #2116

postgresql

- **[postgresql]** When explicit sequence execution derives the name of the auto-generated sequence of a SERIAL column, which currently only occurs if implicit_returning=False, now accommodates if the table + column name is greater than 63 characters using the same logic Postgresql uses. ¶ References: #1083
- **[postgresql]** Added an additional libpq message to the list of "disconnect" exceptions, "could not receive data from server" ¶ References: #2044
- **[postgresql]** Added RESERVED_WORDS for postgresql dialect. ¶ References: #2092
- **[postgresql]** Fixed the BIT type to allow a "length" parameter, "varying" parameter. Reflection also fixed. ¶ References: #2073

mysql

- **[mysql]** oursql dialect accepts the same "ssl" arguments in create_engine() as that of MySQLdb. ¶ References: #2047

sqlite

- **[sqlite]** Fixed bug where reflection of foreign key created as "REFERENCES <tablename>" without col name would fail. ¶ References: #2115

mssql

- **[mssql]** Rewrote the query used to get the definition of a view, typically when using the Inspector interface, to use sys.sql_modules instead of the information schema, thereby allowing views definitions longer than 4000 characters to be fully returned. ¶ References: #2071

oracle

- **[oracle]** Using column names that would require quotes for the column itself or for a name-generated bind parameter, such as names with special characters, underscores, non-ascii characters, now properly translate bind parameter keys when talking to cx_oracle. ¶ References: #2100
- **[oracle]** Oracle dialect adds use_binds_for_limits=False create_engine() flag, will render the LIMIT/OFFSET values inline instead of as binds, reported to modify the execution plan used by Oracle. ¶ References: #2116

firebird

- **[firebird]** The "implicit_returning" flag on create_engine() is honored if set to False. ¶ References: #2083

misc

- **[informix]** Added RESERVED_WORDS informix dialect. ¶ References: #2092
- **[ext]** The horizontal_shard ShardedSession class accepts the common Session argument "query_cls" as a constructor argument, to enable further subclassing of ShardedQuery. ¶ References: #2090
- **[declarative]** Added an explicit check for the case that the name 'metadata' is used for a column attribute on a declarative class. ¶ References: #2050
- **[declarative]** Fix error message referencing old @classproperty name to reference @declared_attr ¶ References: #2061
- **[declarative]** Arguments in __mapper_args__ that aren't "hashable" aren't mistaken for always-hashable, possibly-column arguments. ¶ References: #2091
- **[documentation]** Documented SQLite DATE/TIME/DATETIME types. ¶ References: #2029
- **[examples]** The Beaker caching example allows a "query_cls" argument to the query_callable() function. ¶ References: #2090

0.6.6

Released: Sat Jan 08 2011

orm

- **[orm]** Fixed bug whereby a non-"mutable" attribute modified event which occurred on an object that was clean except for preceding mutable attribute changes would fail to strongly reference itself in the identity map. This would cause the object to be garbage collected, losing track of any changes that weren't previously saved in the "mutable changes" dictionary. ¶
- **[orm]** Fixed bug whereby "passive_deletes='all"' wasn't passing the correct symbols to lazy loaders during flush, thereby causing an unwarranted load. ¶ References: #2013
- **[orm]** Fixed bug which prevented composite mapped attributes from being used on a mapped select statement.. Note the workings of composite are slated to change significantly in 0.7. ¶ References: #1997
- **[orm]** active_history flag also added to composite(). The flag has no effect in 0.6, but is instead a placeholder flag for forwards compatibility, as it applies in 0.7 for composites. ¶ References: #1976
- **[orm]** Fixed uow bug whereby expired objects passed to Session.delete() would not have unloaded references or collections taken into account when deleting objects, despite passive_deletes remaining at its default of False. ¶ References: #2002
- **[orm]** A warning is emitted when version_id_col is specified on an inheriting mapper when the inherited mapper already has one, if those column expressions are not the same. ¶ References: #1987
- **[orm]** "innerjoin" flag doesn't take effect along the chain of joinedload() joins if a previous join in that chain is an outer join, thus allowing primary rows without a referenced child row to be correctly returned in results. ¶ References: #1954
- **[orm]** Fixed bug regarding "subqueryload" strategy whereby strategy would fail if the entity was an aliased() construct. ¶ References: #1964

- **[orm]** Fixed bug regarding "subqueryload" strategy whereby the join would fail if using a multi-level load of the form from A->joined-subclass->C ¶ References: #2014
- **[orm]** Fixed indexing of Query objects by -1. It was erroneously transformed to the empty slice -1:0 that resulted in IndexError. ¶ References: #1968
- **[orm]** The mapper argument "primary_key" can be passed as a single column as well as a list or tuple. The documentation examples that illustrated it as a scalar value have been changed to lists. ¶ References: #1971
- **[orm]** Added active_history flag to relationship() and column_property(), forces attribute events to always load the "old" value, so that it's available to attributes.get_history(). ¶ References: #1961
- **[orm]** Query.get() will raise if the number of params in a composite key is too large, as well as too small. ¶ References: #1977
- **[orm]** Backport of "optimized get" fix from 0.7, improves the generation of joined-inheritance "load expired row" behavior. ¶ References: #1992
- **[orm]** A little more verbiage to the "primaryjoin" error, in an unusual condition that the join condition "works" for viewonly but doesn't work for non-viewonly, and foreign_keys wasn't used - adds "foreign_keys" to the suggestion. Also add "foreign_keys" to the suggestion for the generic "direction" error. ¶

engine

- **[engine]** The "unicode warning" against non-unicode bind data is now raised only when the Unicode type is used explicitly; not when convert_unicode=True is used on the engine or String type. ¶
- **[engine]** Fixed memory leak in C version of Decimal result processor. ¶ References: #1978
- **[engine]** Implemented sequence check capability for the C version of RowProxy, as well as 2.7 style "collections.Sequence" registration for RowProxy. ¶ References: #1871
- **[engine]** Threadlocal engine methods rollback(), commit(), prepare() won't raise if no transaction is in progress; this was a regression introduced in 0.6. ¶ References: #1998
- **[engine]** Threadlocal engine returns itself upon begin(), begin_nested(); engine then implements contextmanager methods to allow the "with" statement. ¶ References: #2004

sql

- **[sql]** Fixed operator precedence rules for multiple chains of a single non-associative operator. I.e. "x - (y - z)" will compile as "x - (y - z)" and not "x - y - z". Also works with labels, i.e. "x - (y - z).label('foo')" ¶ References: #1984
- **[sql]** The 'info' attribute of Column is copied during Column.copy(), i.e. as occurs when using columns in declarative mixins. ¶ References: #1967
- **[sql]** Added a bind processor for booleans which coerces to int, for DBAPIs such as pymssql that naively call str() on values. ¶
- **[sql]** CheckConstraint will copy its 'initially', 'deferrable', and '_create_rule' attributes within a copy()/tometadata() ¶ References: #2000

postgresql

- **[postgresql]** Single element tuple expressions inside an IN clause parenthesize correctly, also from ¶ References: #1984

- **[postgresql]** Ensured every numeric, float, int code, scalar + array, are recognized by psycopg2 and pg8000's "numeric" base type. ¶ References: #1955
- **[postgresql]** Added as_uuid=True flag to the UUID type, will receive and return values as Python UUID() objects rather than strings. Currently, the UUID type is only known to work with psycopg2. ¶ References: #1956
- **[postgresql]** Fixed bug whereby KeyError would occur with non-ENUM supported PG versions after a pool dispose+recreate would occur. ¶ References: #1989

mysql

- **[mysql]** Fixed error handling for Jython + zxjdbc, such that has_table() property works again. Regression from 0.6.3 (we don't have a Jython buildbot, sorry) ¶ References: #1960

sqlite

- **[sqlite]** The REFERENCES clause in a CREATE TABLE that includes a remote schema to another table with the same schema name now renders the remote name without the schema clause, as required by SQLite. ¶ References: #1851
- **[sqlite]** On the same theme, the REFERENCES clause in a CREATE TABLE that includes a remote schema to a *different* schema than that of the parent table doesn't render at all, as cross-schema references do not appear to be supported. ¶

mssql

- **[mssql]** The rewrite of index reflection in was unfortunately not tested correctly, and returned incorrect results. This regression is now fixed. ¶ References: #1770

oracle

- **[oracle]** The cx_oracle "decimal detection" logic, which takes place for result set columns with ambiguous numeric characteristics, now uses the decimal point character determined by the locale/ NLS_LANG setting, using an on-first-connect detection of this character. cx_oracle 5.0.3 or greater is also required when using a non-period-decimal-point NLS_LANG setting.. ¶ References: #1953

firebird

- **[firebird]** Firebird numeric type now checks for Decimal explicitly, lets float() pass right through, thereby allowing special values such as float('inf'). ¶ References: #2012

misc

- **[declarative]** An error is raised if __table_args__ is not in tuple or dict format, and is not None. ¶ References: #1972
- **[sqlsoup]** Added "map_to()" method to SqlSoup, which is a "master" method which accepts explicit arguments for each aspect of the selectable and mapping, including a base class per mapping. ¶ References: #1975

- **[sqlsoup]** Mapped selectables used with the map(), with_labels(), join() methods no longer put the given argument into the internal "cache" dictionary. Particularly since the join() and select() objects are created in the method itself this was pretty much a pure memory leaking behavior. ¶
- **[examples]** The versioning example now supports detection of changes in an associated relationship(). ¶

0.6.5

Released: Sun Oct 24 2010

orm

- **[orm]** Added a new "lazyload" option "immediateload". Issues the usual "lazy" load operation automatically as the object is populated. The use case here is when loading objects to be placed in an offline cache, or otherwise used after the session isn't available, and straight 'select' loading, not 'joined' or 'subquery', is desired. ¶ References: #1914
- **[orm]** New Query methods: query.label(name), query.as_scalar(), return the query's statement as a scalar subquery with /without label; query.with_entities(*ent), replaces the SELECT list of the query with new entities. Roughly equivalent to a generative form of query.values() which accepts mapped entities as well as column expressions. ¶ References: #1920
- **[orm]** Fixed recursion bug which could occur when moving an object from one reference to another, with backrefs involved, where the initiating parent was a subclass (with its own mapper) of the previous parent. ¶
- **[orm]** Fixed a regression in 0.6.4 which occurred if you passed an empty list to "include_properties" on mapper() ¶ References: #1918
- **[orm]** Fixed labeling bug in Query whereby the NamedTuple would mis-apply labels if any of the column expressions were un-labeled. ¶
- **[orm]** Patched a case where query.join() would adapt the right side to the right side of the left's join inappropriately ¶ References: #1925
- **[orm]** Query.select_from() has been beefed up to help ensure that a subsequent call to query.join() will use the select_from() entity, assuming it's a mapped entity and not a plain selectable, as the default "left" side, not the first entity in the Query object's list of entities. ¶
- **[orm]** The exception raised by Session when it is used subsequent to a subtransaction rollback (which is what happens when a flush fails in autocommit=False mode) has now been reworded (this is the "inactive due to a rollback in a subtransaction" message). In particular, if the rollback was due to an exception during flush(), the message states this is the case, and reiterates the string form of the original exception that occurred during flush. If the session is closed due to explicit usage of subtransactions (not very common), the message just states this is the case. ¶
- **[orm]** The exception raised by Mapper when repeated requests to its initialization are made after initialization already failed no longer assumes the "hasattr" case, since there's other scenarios in which this message gets emitted, and the message also does not compound onto itself multiple times - you get the same message for each attempt at usage. The misnomer "compiles" is being traded out for "initialize". ¶
- **[orm]** Fixed bug in query.update() where 'evaluate' or 'fetch' expiration would fail if the column expression key was a class attribute with a different keyname as the actual column name. ¶ References: #1935
- **[orm]** Added an assertion during flush which ensures that no NULL-holding identity keys were generated on "newly persistent" objects. This can occur when user defined code inadvertently triggers flushes on not-fully-loaded objects. ¶

- **[orm]** lazy loads for relationship attributes now use the current state, not the "committed" state, of foreign and primary key attributes when issuing SQL, if a flush is not in process. Previously, only the database-committed state would be used. In particular, this would cause a many-to-one get()-on-lazyload operation to fail, as autoflush is not triggered on these loads when the attributes are determined and the "committed" state may not be available. ¶ References: #1910
- **[orm]** A new flag on relationship(), load_on_pending, allows the lazy loader to fire off on pending objects without a flush taking place, as well as a transient object that's been manually "attached" to the session. Note that this flag blocks attribute events from taking place when an object is loaded, so backrefs aren't available until after a flush. The flag is only intended for very specific use cases. ¶
- **[orm]** Another new flag on relationship(), cascade_backrefs, disables the "save-update" cascade when the event was initiated on the "reverse" side of a bidirectional relationship. This is a cleaner behavior so that many-to-ones can be set on a transient object without it getting sucked into the child object's session, while still allowing the forward collection to cascade. We *might* default this to False in 0.7. ¶
- **[orm]** Slight improvement to the behavior of "passive_updates=False" when placed only on the many-to-one side of a relationship; documentation has been clarified that passive_updates=False should really be on the one-to-many side. ¶
- **[orm]** Placing passive_deletes=True on a many-to-one emits a warning, since you probably intended to put it on the one-to-many side. ¶
- **[orm]** Fixed bug that would prevent "subqueryload" from working correctly with single table inheritance for a relationship from a subclass - the "where type in (x, y, z)" only gets placed on the inside, instead of repeatedly. ¶
- **[orm]** When using from_self() with single table inheritance, the "where type in (x, y, z)" is placed on the outside of the query only, instead of repeatedly. May make some more adjustments to this. ¶
- **[orm]** scoped_session emits a warning when configure() is called if a Session is already present (checks only the current thread) ¶ References: #1924
- **[orm]** reworked the internals of mapper.cascade_iterator() to cut down method calls by about 9% in some circumstances. ¶ References: #1932

engine

- **[engine]** Fixed a regression in 0.6.4 whereby the change that allowed cursor errors to be raised consistently broke the result.lastrowid accessor. Test coverage has been added for result.lastrowid. Note that lastrowid is only supported by Pysqlite and some MySQL drivers, so isn't super-useful in the general case. ¶
- **[engine]** the logging message emitted by the engine when a connection is first used is now "BEGIN (implicit)" to emphasize that DBAPI has no explicit begin(). ¶
- **[engine]** added "views=True" option to metadata.reflect(), will add the list of available views to those being reflected. ¶ References: #1936
- **[engine]** engine_from_config() now accepts 'debug' for 'echo', 'echo_pool', 'force' for 'convert_unicode', boolean values for 'use_native_unicode'. ¶ References: #1899

sql

- **[sql]** Fixed bug in TypeDecorator whereby the dialect-specific type was getting pulled in to generate the DDL for a given type, which didn't always return the correct result. ¶
- **[sql]** TypeDecorator can now have a fully constructed type specified as its "impl", in addition to a type class. ¶

- **[sql]** TypeDecorator will now place itself as the resulting type for a binary expression where the type coercion rules would normally return its impl type - previously, a copy of the impl type would be returned which would have the TypeDecorator embedded into it as the "dialect" impl, this was probably an unintentional way of achieving the desired effect. ¶
- **[sql]** TypeDecorator.load_dialect_impl() returns "self.impl" by default, i.e. not the dialect implementation type of "self.impl". This to support compilation correctly. Behavior can be user-overridden in exactly the same way as before to the same effect. ¶
- **[sql]** Added type_coerce(expr, type_) expression element. Treats the given expression as the given type when evaluating expressions and processing result rows, but does not affect the generation of SQL, other than an anonymous label. ¶
- **[sql]** Table.tometadata() now copies Index objects associated with the Table as well. ¶
- **[sql]** Table.tometadata() issues a warning if the given Table is already present in the target MetaData - the existing Table object is returned. ¶
- **[sql]** An informative error message is raised if a Column which has not yet been assigned a name, i.e. as in declarative, is used in a context where it is exported to the columns collection of an enclosing select() construct, or if any construct involving that column is compiled before its name is assigned. ¶
- **[sql]** as_scalar(), label() can be called on a selectable which contains a Column that is not yet named. ¶ References: #1862
- **[sql]** Fixed recursion overflow which could occur when operating with two expressions both of type "NullType", but not the singleton NULLTYPE instance. ¶ References: #1907

postgresql

- **[postgresql]** Added "as_tuple" flag to ARRAY type, returns results as tuples instead of lists to allow hashing. ¶
- **[postgresql]** Fixed bug which prevented "domain" built from a custom type such as "enum" from being reflected. ¶ References: #1933

mysql

- **[mysql]** Fixed bug involving reflection of CURRENT_TIMESTAMP default used with ON UPDATE clause, thanks to Taavi Burns ¶ References: #1940

mssql

- **[mssql]** Fixed reflection bug which did not properly handle reflection of unknown types. ¶ References: #1946
- **[mssql]** Fixed bug where aliasing of tables with "schema" would fail to compile properly. ¶ References: #1943
- **[mssql]** Rewrote the reflection of indexes to use sys. catalogs, so that column names of any configuration (spaces, embedded commas, etc.) can be reflected. Note that reflection of indexes requires SQL Server 2005 or greater. ¶ References: #1770
- **[mssql]** mssql+pymssql dialect now honors the "port" portion of the URL instead of discarding it. ¶ References: #1952

oracle

- **[oracle]** The implicit_retunring argument to create_engine() is now honored regardless of detected version of Oracle. Previously, the flag would be forced to False if server version info was < 10. ¶ References: #1878

misc

- **[declarative]** @classproperty (soon/now @declared_attr) takes effect for __mapper_args__, __table_args__, __tablename__ on a base class that is not a mixin, as well as mixins. ¶ References: #1922
- **[declarative]** @classproperty 's official name/location for usage with declarative is sqlalchemy.ext.declarative.declared_attr. Same thing, but moving there since it is more of a "marker" that's specific to declararative, not just an attribute technique. ¶ References: #1915
- **[declarative]** Fixed bug whereby columns on a mixin wouldn't propagate correctly to a single-table, or joined-table, inheritance scheme where the attribute name is different than that of the column.,. ¶ References: #1931, #1930
- **[declarative]** A mixin can now specify a column that overrides a column of the same name associated with a superclass. Thanks to Oystein Haaland. ¶
- **[informix]** *Major* cleanup / modernization of the Informix dialect for 0.6, courtesy Florian Apolloner. ¶ References: #1906
- **[tests]** the NoseSQLAlchemyPlugin has been moved to a new package "sqlalchemy_nose" which installs along with "sqlalchemy". This so that the "nosetests" script works as always but also allows the –with-coverage option to turn on coverage before SQLAlchemy modules are imported, allowing coverage to work correctly. ¶
- **[misc]** CircularDependencyError now has .cycles and .edges members, which are the set of elements involved in one or more cycles, and the set of edges as 2-tuples. ¶ References: #1890

0.6.4

Released: Tue Sep 07 2010

orm

- **[orm]** The name ConcurrentModificationError has been changed to StaleDataError, and descriptive error messages have been revised to reflect exactly what the issue is. Both names will remain available for the forseeable future for schemes that may be specifying ConcurrentModificationError in an "except:" clause. ¶
- **[orm]** Added a mutex to the identity map which mutexes remove operations against iteration methods, which now pre-buffer before returning an iterable. This because asyncrhonous gc can remove items via the gc thread at any time. ¶ References: #1891
- **[orm]** The Session class is now present in sqlalchemy.orm.*. We're moving away from the usage of create_session(), which has non-standard defaults, for those situations where a one-step Session constructor is desired. Most users should stick with sessionmaker() for general use, however. ¶
- **[orm]** query.with_parent() now accepts transient objects and will use the non-persistent values of their pk/fk attributes in order to formulate the criterion. Docs are also clarified as to the purpose of with_parent(). ¶
- **[orm]** The include_properties and exclude_properties arguments to mapper() now accept Column objects as members in addition to strings. This so that same-named Column objects, such as those within a join(), can be disambiguated. ¶

- **[orm]** A warning is now emitted if a mapper is created against a join or other single selectable that includes multiple columns with the same name in its .c. collection, and those columns aren't explicitly named as part of the same or separate attributes (or excluded). In 0.7 this warning will be an exception. Note that this warning is not emitted when the combination occurs as a result of inheritance, so that attributes still allow being overridden naturally.. In 0.7 this will be improved further. ¶ References: #1896
- **[orm]** The primary_key argument to mapper() can now specify a series of columns that are only a subset of the calculated "primary key" columns of the mapped selectable, without an error being raised. This helps for situations where a selectable's effective primary key is simpler than the number of columns in the selectable that are actually marked as "primary_key", such as a join against two tables on their primary key columns. ¶ References: #1896
- **[orm]** An object that's been deleted now gets a flag 'deleted', which prohibits the object from being re-add()ed to the session, as previously the object would live in the identity map silently until its attributes were accessed. The make_transient() function now resets this flag along with the "key" flag. ¶
- **[orm]** make_transient() can be safely called on an already transient instance. ¶
- **[orm]** a warning is emitted in mapper() if the polymorphic_on column is not present either in direct or derived form in the mapped selectable or in the with_polymorphic selectable, instead of silently ignoring it. Look for this to become an exception in 0.7. ¶
- **[orm]** Another pass through the series of error messages emitted when relationship() is configured with ambiguous arguments. The "foreign_keys" setting is no longer mentioned, as it is almost never needed and it is preferable users set up correct ForeignKey metadata, which is now the recommendation. If 'foreign_keys' is used and is incorrect, the message suggests the attribute is probably unnecessary. Docs for the attribute are beefed up. This because all confused relationship() users on the ML appear to be attempting to use foreign_keys due to the message, which only confuses them further since Table metadata is much clearer. ¶
- **[orm]** If the "secondary" table has no ForeignKey metadata and no foreign_keys is set, even though the user is passing screwed up information, it is assumed that primary/secondaryjoin expressions should consider only and all cols in "secondary" to be foreign. It's not possible with "secondary" for the foreign keys to be elsewhere in any case. A warning is now emitted instead of an error, and the mapping succeeds. ¶ References: #1877
- **[orm]** Moving an o2m object from one collection to another, or vice versa changing the referenced object by an m2o, where the foreign key is also a member of the primary key, will now be more carefully checked during flush if the change in value of the foreign key on the "many" side is the result of a change in the primary key of the "one" side, or if the "one" is just a different object. In one case, a cascade-capable DB would have cascaded the value already and we need to look at the "new" PK value to do an UPDATE, in the other we need to continue looking at the "old". We now look at the "old", assuming passive_updates=True, unless we know it was a PK switch that triggered the change. ¶ References: #1856
- **[orm]** The value of version_id_col can be changed manually, and this will result in an UPDATE of the row. Versioned UPDATEs and DELETEs now use the "committed" value of the version_id_col in the WHERE clause and not the pending changed value. The version generator is also bypassed if manual changes are present on the attribute. ¶ References: #1857
- **[orm]** Repaired the usage of merge() when used with concrete inheriting mappers. Such mappers frequently have so-called "concrete" attributes, which are subclass attributes that "disable" propagation from the parent - these needed to allow a merge() operation to pass through without effect. ¶
- **[orm]** Specifying a non-column based argument for column_mapped_collection, including string, text() etc., will raise an error message that specifically asks for a column element, no longer misleads with incorrect information about text() or literal(). ¶ References: #1863
- **[orm]** Similarly, for relationship(), foreign_keys, remote_side, order_by - all column-based expressions are enforced - lists of strings are explicitly disallowed since this is a very common error ¶
- **[orm]** Dynamic attributes don't support collection population - added an assertion for when

set_committed_value() is called, as well as when joinedload() or subqueryload() options are applied to a dynamic attribute, instead of failure / silent failure. ¶ References: #1864

- **[orm]** Fixed bug whereby generating a Query derived from one which had the same column repeated with different label names, typically in some UNION situations, would fail to propagate the inner columns completely to the outer query. ¶ References: #1852
- **[orm]** object_session() raises the proper UnmappedInstanceError when presented with an unmapped instance. ¶ References: #1881
- **[orm]** Applied further memoizations to calculated Mapper properties, with significant (~90%) runtime mapper.py call count reduction in heavily polymorphic mapping configurations. ¶
- **[orm]** mapper _get_col_to_prop private method used by the versioning example is deprecated; now use mapper.get_property_by_column() which will remain the public method for this. ¶
- **[orm]** the versioning example works correctly now if versioning on a col that was formerly NULL. ¶

engine

- **[engine]** Calling fetchone() or similar on a result that has already been exhausted, has been closed, or is not a result-returning result now raises ResourceClosedError, a subclass of InvalidRequestError, in all cases, regardless of backend. Previously, some DBAPIs would raise ProgrammingError (i.e. pysqlite), others would return None leading to downstream breakages (i.e. MySQL-python). ¶
- **[engine]** Fixed bug in Connection whereby if a "disconnect" event occurred in the "initialize" phase of the first connection pool connect, an AttributeError would be raised when the Connection would attempt to invalidate the DBAPI connection. ¶ References: #1894
- **[engine]** Connection, ResultProxy, as well as Session use ResourceClosedError for all "this connection/transaction/result is closed" types of errors. ¶
- **[engine]** Connection.invalidate() can be called more than once and subsequent calls do nothing. ¶

sql

- **[sql]** Calling execute() on an alias() construct is pending deprecation for 0.7, as it is not itself an "executable" construct. It currently "proxies" its inner element and is conditionally "executable" but this is not the kind of ambiguity we like these days. ¶
- **[sql]** The execute() and scalar() methods of ClauseElement are now moved appropriately to the Executable subclass. ClauseElement.execute()/ scalar() are still present and are pending deprecation in 0.7, but note these would always raise an error anyway if you were not an Executable (unless you were an alias(), see previous note). ¶
- **[sql]** Added basic math expression coercion for Numeric->Integer, so that resulting type is Numeric regardless of the direction of the expression. ¶
- **[sql]** Changed the scheme used to generate truncated "auto" index names when using the "index=True" flag on Column. The truncation only takes place with the auto-generated name, not one that is user-defined (an error would be raised instead), and the truncation scheme itself is now based on a fragment of an md5 hash of the identifier name, so that multiple indexes on columns with similar names still have unique names. ¶ References: #1855
- **[sql]** The generated index name also is based on a "max index name length" attribute which is separate from the "max identifier length" - this to appease MySQL who has a max length of 64 for index names, separate from their overall max length of 255. ¶ References: #1412

- **[sql]** the text() construct, if placed in a column oriented situation, will at least return NULLTYPE for its type instead of None, allowing it to be used a little more freely for ad-hoc column expressions than before. literal_column() is still the better choice, however. ¶
- **[sql]** Added full description of parent table/column, target table/column in error message raised when ForeignKey can't resolve target. ¶
- **[sql]** Fixed bug whereby replacing composite foreign key columns in a reflected table would cause an attempt to remove the reflected constraint from the table a second time, raising a KeyError. ¶ References: #1865
- **[sql]** the _Label construct, i.e. the one that is produced whenever you say somecol.label(), now counts itself in its "proxy_set" unioned with that of its contained column's proxy set, instead of directly returning that of the contained column. This allows column correspondence operations which depend on the identity of the _Labels themselves to return the correct result ¶
- **[sql]** fixes ORM bug. ¶ References: #1852

postgresql

- **[postgresql]** Fixed the psycopg2 dialect to use its set_isolation_level() method instead of relying upon the base "SET SESSION ISOLATION" command, as psycopg2 resets the isolation level on each new transaction otherwise. ¶

mssql

- **[mssql]** Fixed "default schema" query to work with pymssql backend. ¶

oracle

- **[oracle]** Added ROWID type to the Oracle dialect, for those cases where an explicit CAST might be needed. ¶ References: #1879
- **[oracle]** Oracle reflection of indexes has been tuned so that indexes which include some or all primary key columns, but not the same set of columns as that of the primary key, are reflected. Indexes which contain the identical columns as that of the primary key are skipped within reflection, as the index in that case is assumed to be the auto-generated primary key index. Previously, any index with PK columns present would be skipped. Thanks to Kent Bower for the patch. ¶ References: #1867
- **[oracle]** Oracle now reflects the names of primary key constraints - also thanks to Kent Bower. ¶ References: #1868

firebird

- **[firebird]** Fixed bug whereby a column default would fail to reflect if the "default" keyword were lower case. ¶

misc

- **[declarative]** if @classproperty is used with a regular class-bound mapper property attribute, it will be called to get the actual attribute value during initialization. Currently, there's no advantage to using @classproperty on a column or relationship attribute of a declarative class that isn't a mixin - evaluation is at the same time as if @classproperty weren't used. But here we at least allow it to function as expected. ¶

- **[declarative]** Fixed bug where "Can't add additional column" message would display the wrong name. ¶
- **[informix]** Applied patches from to get basic Informix functionality up again. We rely upon end-user testing to ensure that Informix is working to some degree. ¶ References: #1904
- **[documentation]** The docs have been reorganized such that the "API Reference" section is gone - all the docstrings from there which were public API are moved into the context of the main doc section that talks about it. Main docs divided into "SQLAlchemy Core" and "SQLAlchemy ORM" sections, mapper/relationship docs have been broken out. Lots of sections rewritten and/or reorganized. ¶
- **[examples]** The beaker_caching example has been reorganized such that the Session, cache manager, declarative_base are part of environment, and custom cache code is portable and now within "caching_query.py". This allows the example to be easier to "drop in" to existing projects. ¶
- **[examples]** the history_meta versioning recipe sets "unique=False" when copying columns, so that the versioning table handles multiple rows with repeating values. ¶ References: #1887

0.6.3

Released: Thu Jul 15 2010

orm

- **[orm]** Removed errant many-to-many load in unitofwork which triggered unnecessarily on expired/unloaded collections. This load now takes place only if passive_updates is False and the parent primary key has changed, or if passive_deletes is False and a delete of the parent has occurred. ¶ References: #1845
- **[orm]** Column-entities (i.e. query(Foo.id)) copy their state more fully when queries are derived from themselves + a selectable (i.e. from_self(), union(), etc.), so that join() and such have the correct state to work from. ¶ References: #1853
- **[orm]** Fixed bug where Query.join() would fail if querying a non-ORM column then joining without an on clause when a FROM clause is already present, now raises a checked exception the same way it does when the clause is not present. ¶ References: #1853
- **[orm]** Improved the check for an "unmapped class", including the case where the superclass is mapped but the subclass is not. Any attempts to access cls._sa_class_manager.mapper now raise UnmappedClassError(). ¶ References: #1142
- **[orm]** Added "column_descriptions" accessor to Query, returns a list of dictionaries containing naming/typing information about the entities the Query will return. Can be helpful for building GUIs on top of ORM queries. ¶

mysql

- **[mysql]** The _extract_error_code() method now works correctly with each MySQL dialect (MySQL-python, OurSQL, MySQL-Connector-Python, PyODBC). Previously, the reconnect logic would fail for OperationalError conditions, however since MySQLdb and OurSQL have their own reconnect feature, there was no symptom for these drivers here unless one watched the logs. ¶ References: #1848

oracle

- **[oracle]** More tweaks to cx_oracle Decimal handling. "Ambiguous" numerics with no decimal place are coerced to int at the connection handler level. The advantage here is that ints come back as ints without SQLA type objects being involved and without needless conversion to Decimal first.

Unfortunately, some exotic subquery cases can even see different types between individual result rows, so the Numeric handler, when instructed to return Decimal, can't take full advantage of "native decimal" mode and must run isinstance() on every value to check if its Decimal already. Reopen of ¶ References: #1840

0.6.2

Released: Tue Jul 06 2010

orm

- **[orm]** Query.join() will check for a call of the form query.join(target, clause_expression), i.e. missing the tuple, and raise an informative error message that this is the wrong calling form. ¶
- **[orm]** Fixed bug regarding flushes on self-referential bi-directional many-to-many relationships, where two objects made to mutually reference each other in one flush would fail to insert a row for both sides. Regression from 0.5. ¶ References: #1824
- **[orm]** the post_update feature of relationship() has been reworked architecturally to integrate more closely with the new 0.6 unit of work. The motivation for the change is so that multiple "post update" calls, each affecting different foreign key columns of the same row, are executed in a single UPDATE statement, rather than one UPDATE statement per column per row. Multiple row updates are also batched into executemany()s as possible, while maintaining consistent row ordering. ¶
- **[orm]** Query.statement, Query.subquery(), etc. now transfer the values of bind parameters, i.e. those specified by query.params(), into the resulting SQL expression. Previously the values would not be transferred and bind parameters would come out as None. ¶
- **[orm]** Subquery-eager-loading now works with Query objects which include params(), as well as get() Queries. ¶
- **[orm]** Can now call make_transient() on an instance that is referenced by parent objects via many-to-one, without the parent's foreign key value getting temporarily set to None - this was a function of the "detect primary key switch" flush handler. It now ignores objects that are no longer in the "persistent" state, and the parent's foreign key identifier is left unaffected. ¶
- **[orm]** query.order_by() now accepts False, which cancels any existing order_by() state on the Query, allowing subsequent generative methods to be called which do not support ORDER BY. This is not the same as the already existing feature of passing None, which suppresses any existing order_by() settings, including those configured on the mapper. False will make it as though order_by() was never called, while None is an active setting. ¶
- **[orm]** An instance which is moved to "transient", has an incomplete or missing set of primary key attributes, and contains expired attributes, will raise an InvalidRequestError if an expired attribute is accessed, instead of getting a recursion overflow. ¶
- **[orm]** The make_transient() function is now in the generated documentation. ¶
- **[orm]** make_transient() removes all "loader" callables from the state being made transient, removing any "expired" state - all unloaded attributes reset back to undefined, None/empty on access. ¶

sql

- **[sql]** The warning emitted by the Unicode and String types with convert_unicode=True no longer embeds the actual value passed. This so that the Python warning registry does not continue to grow in size, the warning is emitted once as per the warning filter settings, and large string values don't pollute the output. ¶ References: #1822

- **[sql]** Fixed bug that would prevent overridden clause compilation from working for "annotated" expression elements, which are often generated by the ORM. ¶
- **[sql]** The argument to "ESCAPE" of a LIKE operator or similar is passed through render_literal_value(), which may implement escaping of backslashes. ¶ References: #1400
- **[sql]** Fixed bug in Enum type which blew away native_enum flag when used with TypeDecorators or other adaption scenarios. ¶
- **[sql]** Inspector hits bind.connect() when invoked to ensure initialize has been called. the internal name ".conn" is changed to ".bind", since that's what it is. ¶
- **[sql]** Modified the internals of "column annotation" such that a custom Column subclass can safely override _constructor to return Column, for the purposes of making "configurational" column classes that aren't involved in proxying, etc. ¶
- **[sql]** Column.copy() takes along the "unique" attribute among others, fixes regarding declarative mixins ¶ References: #1829

postgresql

- **[postgresql]** render_literal_value() is overridden which escapes backslashes, currently applies to the ESCAPE clause of LIKE and similar expressions. Ultimately this will have to detect the value of "standard_conforming_strings" for full behavior. ¶ References: #1400
- **[postgresql]** Won't generate "CREATE TYPE" / "DROP TYPE" if using types.Enum on a PG version prior to 8.3 - the supports_native_enum flag is fully honored. ¶ References: #1836

mysql

- **[mysql]** MySQL dialect doesn't emit CAST() for MySQL version detected < 4.0.2. This allows the unicode check on connect to proceed. ¶ References: #1826
- **[mysql]** MySQL dialect now detects NO_BACKSLASH_ESCAPES sql mode, in addition to ANSI_QUOTES. ¶
- **[mysql]** render_literal_value() is overridden which escapes backslashes, currently applies to the ESCAPE clause of LIKE and similar expressions. This behavior is derived from detecting the value of NO_BACKSLASH_ESCAPES. ¶ References: #1400

mssql

- **[mssql]** If server_version_info is outside the usual range of (8,), (9,), (10,), a warning is emitted which suggests checking that the FreeTDS version configuration is using 7.0 or 8.0, not 4.2. ¶ References: #1825

oracle

- **[oracle]** Fixed ora-8 compatibility flags such that they don't cache a stale value from before the first database connection actually occurs. ¶ References: #1819
- **[oracle]** Oracle's "native decimal" metadata begins to return ambiguous typing information about numerics when columns are embedded in subqueries as well as when ROWNUM is consulted with subqueries, as we do for limit/offset. We've added these ambiguous conditions to the cx_oracle "convert to Decimal()" handler, so that we receive numerics as Decimal in more cases instead of as floats. These are then converted, if requested, into Integer or Float, or otherwise kept as the lossless Decimal. ¶ References: #1840

firebird

- **[firebird]** Fixed incorrect signature in do_execute(), error introduced in 0.6.1. ¶ References: #1823
- **[firebird]** Firebird dialect adds CHAR, VARCHAR types which accept a "charset" flag, to support Firebird "CHARACTER SET" clause. ¶ References: #1813

misc

- **[declarative]** Added support for @classproperty to provide any kind of schema/mapping construct from a declarative mixin, including columns with foreign keys, relationships, column_property, deferred. This solves all such issues on declarative mixins. An error is raised if any MapperProperty subclass is specified on a mixin without using @classproperty. ¶ References: #1805, #1796, #1751
- **[declarative]** a mixin class can now define a column that matches one which is present on a __table__ defined on a subclass. It cannot, however, define one that is not present in the __table__, and the error message here now works. ¶ References: #1821
- **[extension] [compiler]** The 'default' compiler is automatically copied over when overriding the compilation of a built in clause construct, so no KeyError is raised if the user-defined compiler is specific to certain backends and compilation for a different backend is invoked. ¶ References: #1838
- **[documentation]** Added documentation for the Inspector. ¶ References: #1820
- **[documentation]** Fixed @memoized_property and @memoized_instancemethod decorators so that Sphinx documentation picks up these attributes and methods, such as ResultProxy.inserted_primary_key. ¶ References: #1830

0.6.1

Released: Mon May 31 2010

orm

- **[orm]** Fixed regression introduced in 0.6.0 involving improper history accounting on mutable attributes. ¶ References: #1782
- **[orm]** Fixed regression introduced in 0.6.0 unit of work refactor that broke updates for bi-directional relationship() with post_update=True. ¶ References: #1807
- **[orm]** session.merge() will not expire attributes on the returned instance if that instance is "pending". ¶ References: #1789
- **[orm]** fixed __setstate__ method of CollectionAdapter to not fail during deserialize where parent InstanceState not yet unserialized. ¶ References: #1802
- **[orm]** Added internal warning in case an instance without a full PK happened to be expired and then was asked to refresh. ¶ References: #1797
- **[orm]** Added more aggressive caching to the mapper's usage of UPDATE, INSERT, and DELETE expressions. Assuming the statement has no per-object SQL expressions attached, the expression objects are cached by the mapper after the first create, and their compiled form is stored persistently in a cache dictionary for the duration of the related Engine. The cache is an LRUCache for the rare case that a mapper receives an extremely high number of different column patterns as UPDATEs. ¶

sql

- **[sql]** expr.in_() now accepts a text() construct as the argument. Grouping parenthesis are added automatically, i.e. usage is like *col.in_(text("select id from table"))*. ¶ References: #1793
- **[sql]** Columns of _Binary type (i.e. LargeBinary, BLOB, etc.) will coerce a "basestring" on the right side into a _Binary as well so that required DBAPI processing takes place. ¶
- **[sql]** Added table.add_is_dependent_on(othertable), allows manual placement of dependency rules between two Table objects for use within create_all(), drop_all(), sorted_tables. ¶ References: #1801
- **[sql]** Fixed bug that prevented implicit RETURNING from functioning properly with composite primary key that contained zeroes. ¶ References: #1778
- **[sql]** Fixed errant space character when generating ADD CONSTRAINT for a named UNIQUE constraint. ¶
- **[sql]** Fixed "table" argument on constructor of ForeginKeyConstraint ¶ References: #1571
- **[sql]** Fixed bug in connection pool cursor wrapper whereby if a cursor threw an exception on close(), the logging of the message would fail. ¶ References: #1786
- **[sql]** the _make_proxy() method of ColumnClause and Column now use self.__class__ to determine the class of object to be returned instead of hardcoding to ColumnClause/Column, making it slightly easier to produce specific subclasses of these which work in alias/subquery situations. ¶
- **[sql]** func.XXX() doesn't inadvertently resolve to non-Function classes (e.g. fixes func.text()). ¶ References: #1798

mysql

- **[mysql]** func.sysdate() emits "SYSDATE()", i.e. with the ending parenthesis, on MySQL. ¶ References: #1794

sqlite

- **[sqlite]** Fixed concatenation of constraints when "PRIMARY KEY" constraint gets moved to column level due to SQLite AUTOINCREMENT keyword being rendered. ¶ References: #1812

oracle

- **[oracle]** Added a check for cx_oracle versions lower than version 5, in which case the incompatible "output type handler" won't be used. This will impact decimal accuracy and some unicode handling issues. ¶ References: #1775
- **[oracle]** Fixed use_ansi=False mode, which was producing broken WHERE clauses in pretty much all cases. ¶ References: #1790
- **[oracle]** Re-established support for Oracle 8 with cx_oracle, including that use_ansi is set to False automatically, NVARCHAR2 and NCLOB are not rendered for Unicode, "native unicode" check doesn't fail, cx_oracle "native unicode" mode is disabled, VARCHAR() is emitted with bytes count instead of char count. ¶ References: #1808
- **[oracle]** oracle_xe 5 doesn't accept a Python unicode object in its connect string in normal Python 2.x mode - so we coerce to str() directly. non-ascii characters aren't supported in connect strings here since we don't know what encoding we could use. ¶ References: #1670
- **[oracle]** FOR UPDATE is emitted in the syntactically correct position when limit/offset is used, i.e. the ROWNUM subquery. However, Oracle can't really handle FOR UPDATE with ORDER BY or with subqueries, so its still not very usable, but at least SQLA gets the SQL past the Oracle parser. ¶ References: #1815

firebird

- **[firebird]** Added a label to the query used within has_table() and has_sequence() to work with older versions of Firebird that don't provide labels for result columns. ¶ References: #1521
- **[firebird]** Added integer coercion to the "type_conv" attribute when passed via query string, so that it is properly interpreted by Kinterbasdb. ¶ References: #1779
- **[firebird]** Added 'connection shutdown' to the list of exception strings which indicate a dropped connection. ¶ References: #1646

misc

- **[engines]** Fixed building the C extensions on Python 2.4. ¶ References: #1781
- **[engines]** Pool classes will reuse the same "pool_logging_name" setting after a dispose() occurs. ¶
- **[engines]** Engine gains an "execution_options" argument and update_execution_options() method, which will apply to all connections generated by this engine. ¶
- **[sqlsoup]** the SqlSoup constructor accepts a *base* argument which specifies the base class to use for mapped classes, the default being *object*. ¶ References: #1783

0.6.0

Released: Sun Apr 18 2010

orm

- **[orm]** Unit of work internals have been rewritten. Units of work with large numbers of objects interdependent objects can now be flushed without recursion overflows as there is no longer reliance upon recursive calls. The number of internal structures now stays constant for a particular session state, regardless of how many relationships are present on mappings. The flow of events now corresponds to a linear list of steps, generated by the mappers and relationships based on actual work to be done, filtered through a single topological sort for correct ordering. Flush actions are assembled using far fewer steps and less memory. ¶ References: #1742, #1081
- **[orm]** Along with the UOW rewrite, this also removes an issue introduced in 0.6beta3 regarding topological cycle detection for units of work with long dependency cycles. We now use an algorithm written by Guido (thanks Guido!). ¶
- **[orm]** one-to-many relationships now maintain a list of positive parent-child associations within the flush, preventing previous parents marked as deleted from cascading a delete or NULL foreign key set on those child objects, despite the end-user not removing the child from the old association. ¶ References: #1764
- **[orm]** A collection lazy load will switch off default eagerloading on the reverse many-to-one side, since that loading is by definition unnecessary. ¶ References: #1495
- **[orm]** Session.refresh() now does an equivalent expire() on the given instance first, so that the "refresh-expire" cascade is propagated. Previously, refresh() was not affected in any way by the presence of "refresh-expire" cascade. This is a change in behavior versus that of 0.6beta2, where the "lockmode" flag passed to refresh() would cause a version check to occur. Since the instance is first expired, refresh() always upgrades the object to the most recent version. ¶
- **[orm]** The 'refresh-expire' cascade, when reaching a pending object, will expunge the object if the cascade also includes "delete-orphan", or will simply detach it otherwise. ¶ References: #1754

- **[orm]** id(obj) is no longer used internally within topological.py, as the sorting functions now require hashable objects only. ¶ References: #1756
- **[orm]** The ORM will set the docstring of all generated descriptors to None by default. This can be overridden using 'doc' (or if using Sphinx, attribute docstrings work too). ¶
- **[orm]** Added kw argument 'doc' to all mapper property callables as well as Column(). Will assemble the string 'doc' as the '__doc__' attribute on the descriptor. ¶
- **[orm]** Usage of version_id_col on a backend that supports cursor.rowcount for execute() but not executemany() now works when a delete is issued (already worked for saves, since those don't use executemany()). For a backend that doesn't support cursor.rowcount at all, a warning is emitted the same as with saves. ¶ References: #1761
- **[orm]** The ORM now short-term caches the "compiled" form of insert() and update() constructs when flushing lists of objects of all the same class, thereby avoiding redundant compilation per individual INSERT/UPDATE within an individual flush() call. ¶
- **[orm]** internal getattr(), setattr(), getcommitted() methods on ColumnProperty, CompositeProperty, RelationshipProperty have been underscored (i.e. are private), signature has changed. ¶

sql

- **[sql]** Restored some bind-labeling logic from 0.5 which ensures that tables with column names that overlap another column of the form "<tablename>_<columnname>" won't produce errors if column._label is used as a bind name during an UPDATE. Test coverage which wasn't present in 0.5 has been added. ¶ References: #1755
- **[sql]** somejoin.select(fold_equivalents=True) is no longer deprecated, and will eventually be rolled into a more comprehensive version of the feature for. ¶ References: #1729
- **[sql]** the Numeric type raises an *enormous* warning when expected to convert floats to Decimal from a DBAPI that returns floats. This includes SQLite, Sybase, MS-SQL. ¶ References: #1759
- **[sql]** Fixed an error in expression typing which caused an endless loop for expressions with two NULL types. ¶
- **[sql]** Fixed bug in execution_options() feature whereby the existing Transaction and other state information from the parent connection would not be propagated to the sub-connection. ¶
- **[sql]** Added new 'compiled_cache' execution option. A dictionary where Compiled objects will be cached when the Connection compiles a clause expression into a dialect- and parameter- specific Compiled object. It is the user's responsibility to manage the size of this dictionary, which will have keys corresponding to the dialect, clause element, the column names within the VALUES or SET clause of an INSERT or UPDATE, as well as the "batch" mode for an INSERT or UPDATE statement. ¶
- **[sql]** Added get_pk_constraint() to reflection.Inspector, similar to get_primary_keys() except returns a dict that includes the name of the constraint, for supported backends (PG so far). ¶ References: #1769
- **[sql]** Table.create() and Table.drop() no longer apply metadata- level create/drop events. ¶ References: #1771

postgresql

- **[postgresql]** Postgresql now reflects sequence names associated with SERIAL columns correctly, after the name of the sequence has been changed. Thanks to Kumar McMillan for the patch. ¶ References: #1071
- **[postgresql]** Repaired missing import in psycopg2._PGNumeric type when unknown numeric is received. ¶
- **[postgresql]** psycopg2/pg8000 dialects now aware of REAL[], FLOAT[], DOUBLE_PRECISION[], NUMERIC[] return types without raising an exception. ¶

- **[postgresql]** Postgresql reflects the name of primary key constraints, if one exists. ¶ References: #1769

oracle

- **[oracle]** Now using cx_oracle output converters so that the DBAPI returns natively the kinds of values we prefer: ¶
- **[oracle]** NUMBER values with positive precision + scale convert to cx_oracle.STRING and then to Decimal. This allows perfect precision for the Numeric type when using cx_oracle. ¶ References: #1759
- **[oracle]** STRING/FIXED_CHAR now convert to unicode natively. SQLAlchemy's String types then don't need to apply any kind of conversions. ¶

firebird

- **[firebird]** The functionality of result.rowcount can be disabled on a per-engine basis by setting 'enable_rowcount=False' on create_engine(). Normally, cursor.rowcount is called after any UPDATE or DELETE statement unconditionally, because the cursor is then closed and Firebird requires an open cursor in order to get a rowcount. This call is slightly expensive however so it can be disabled. To re-enable on a per-execution basis, the 'enable_rowcount=True' execution option may be used. ¶

misc

- **[engines]** The C extension now also works with DBAPIs which use custom sequences as row (and not only tuples). ¶ References: #1757
- **[ext]** the compiler extension now allows @compiles decorators on base classes that extend to child classes, @compiles decorators on child classes that aren't broken by a @compiles decorator on the base class. ¶
- **[ext]** Declarative will raise an informative error message if a non-mapped class attribute is referenced in the string-based relationship() arguments. ¶
- **[ext]** Further reworked the "mixin" logic in declarative to additionally allow __mapper_args__ as a @classproperty on a mixin, such as to dynamically assign polymorphic_identity. ¶
- **[examples]** Updated attribute_shard.py example to use a more robust method of searching a Query for binary expressions which compare columns against literal values. ¶

0.6beta3

Released: Sun Mar 28 2010

orm

- **[orm]** Major feature: Added new "subquery" loading capability to relationship(). This is an eager loading option which generates a second SELECT for each collection represented in a query, across all parents at once. The query re-issues the original end-user query wrapped in a subquery, applies joins out to the target collection, and loads all those collections fully in one result, similar to "joined" eager loading but using all inner joins and not re-fetching full parent rows repeatedly (as most DBAPIs seem to do, even if columns are skipped). Subquery loading is available at mapper config level using "lazy='subquery'" and at the query options level using "subqueryload(props..)", "subqueryload_all(props...)". ¶ References: #1675

- **[orm]** To accommodate the fact that there are now two kinds of eager loading available, the new names for eagerload() and eagerload_all() are joinedload() and joinedload_all(). The old names will remain as synonyms for the foreseeable future. ¶
- **[orm]** The "lazy" flag on the relationship() function now accepts a string argument for all kinds of loading: "select", "joined", "subquery", "noload" and "dynamic", where the default is now "select". The old values of True/ False/None still retain their usual meanings and will remain as synonyms for the foreseeable future. ¶
- **[orm]** Added with_hint() method to Query() construct. This calls directly down to select().with_hint() and also accepts entities as well as tables and aliases. See with_hint() in the SQL section below. ¶ References: #921
- **[orm]** Fixed bug in Query whereby calling q.join(prop).from_self(...). join(prop) would fail to render the second join outside the subquery, when joining on the same criterion as was on the inside. ¶
- **[orm]** Fixed bug in Query whereby the usage of aliased() constructs would fail if the underlying table (but not the actual alias) were referenced inside the subquery generated by q.from_self() or q.select_from(). ¶
- **[orm]** Fixed bug which affected all eagerload() and similar options such that "remote" eager loads, i.e. eagerloads off of a lazy load such as query(A).options(eagerload(A.b, B.c)) wouldn't eagerload anything, but using eagerload("b.c") would work fine. ¶
- **[orm]** Query gains an add_columns(*columns) method which is a multi- version of add_column(col). add_column(col) is future deprecated. ¶
- **[orm]** Query.join() will detect if the end result will be "FROM A JOIN A", and will raise an error if so. ¶
- **[orm]** Query.join(Cls.propname, from_joinpoint=True) will check more carefully that "Cls" is compatible with the current joinpoint, and act the same way as Query.join("propname", from_joinpoint=True) in that regard. ¶

sql

- **[sql]** Added with_hint() method to select() construct. Specify a table/alias, hint text, and optional dialect name, and "hints" will be rendered in the appropriate place in the statement. Works for Oracle, Sybase, MySQL. ¶ References: #921
- **[sql]** Fixed bug introduced in 0.6beta2 where column labels would render inside of column expressions already assigned a label. ¶ References: #1747

postgresql

- **[postgresql]** The psycopg2 dialect will log NOTICE messages via the "sqlalchemy.dialects.postgresql" logger name. ¶ References: #877
- **[postgresql]** the TIME and TIMESTAMP types are now available from the postgresql dialect directly, which add the PG-specific argument 'precision' to both. 'precision' and 'timezone' are correctly reflected for both TIME and TIMEZONE types. ¶ References: #997

mysql

- **[mysql]** No longer guessing that TINYINT(1) should be BOOLEAN when reflecting - TINYINT(1) is returned. Use Boolean/ BOOLEAN in table definition to get boolean conversion behavior. ¶ References: #1752

oracle

- **[oracle]** The Oracle dialect will issue VARCHAR type definitions using character counts, i.e. VARCHAR2(50 CHAR), so that the column is sized in terms of characters and not bytes. Column reflection of character types will also use ALL_TAB_COLUMNS.CHAR_LENGTH instead of ALL_TAB_COLUMNS.DATA_LENGTH. Both of these behaviors take effect when the server version is 9 or higher - for version 8, the old behaviors are used. ¶ References: #1744

misc

- **[declarative]** Using a mixin won't break if the mixin implements an unpredictable __getattribute__(), i.e. Zope interfaces. ¶ References: #1746
- **[declarative]** Using @classdecorator and similar on mixins to define __tablename__, __table_args__, etc. now works if the method references attributes on the ultimate subclass. ¶ References: #1749
- **[declarative]** relationships and columns with foreign keys aren't allowed on declarative mixins, sorry. ¶ References: #1751
- **[ext]** The sqlalchemy.orm.shard module now becomes an extension, sqlalchemy.ext.horizontal_shard. The old import works with a deprecation warning. ¶

0.6beta2

Released: Sat Mar 20 2010

orm

- **[orm]** The official name for the relation() function is now relationship(), to eliminate confusion over the relational algebra term. relation() however will remain available in equal capacity for the foreseeable future. ¶ References: #1740
- **[orm]** Added "version_id_generator" argument to Mapper, this is a callable that, given the current value of the "version_id_col", returns the next version number. Can be used for alternate versioning schemes such as uuid, timestamps. ¶ References: #1692
- **[orm]** added "lockmode" kw argument to Session.refresh(), will pass through the string value to Query the same as in with_lockmode(), will also do version check for a version_id_col-enabled mapping. ¶
- **[orm]** Fixed bug whereby calling query(A).join(A.bs).add_entity(B) in a joined inheritance scenario would double-add B as a target and produce an invalid query. ¶ References: #1188
- **[orm]** Fixed bug in session.rollback() which involved not removing formerly "pending" objects from the session before re-integrating "deleted" objects, typically occurred with natural primary keys. If there was a primary key conflict between them, the attach of the deleted would fail internally. The formerly "pending" objects are now expunged first. ¶ References: #1674
- **[orm]** Removed a lot of logging that nobody really cares about, logging that remains will respond to live changes in the log level. No significant overhead is added. ¶ References: #1719
- **[orm]** Fixed bug in session.merge() which prevented dict-like collections from merging. ¶
- **[orm]** session.merge() works with relations that specifically don't include "merge" in their cascade options - the target is ignored completely. ¶

- **[orm]** session.merge() will not expire existing scalar attributes on an existing target if the target has a value for that attribute, even if the incoming merged doesn't have a value for the attribute. This prevents unnecessary loads on existing items. Will still mark the attr as expired if the destination doesn't have the attr, though, which fulfills some contracts of deferred cols. ¶ References: #1681
- **[orm]** The "allow_null_pks" flag is now called "allow_partial_pks", defaults to True, acts like it did in 0.5 again. Except, it also is implemented within merge() such that a SELECT won't be issued for an incoming instance with partially NULL primary key if the flag is False. ¶ References: #1680
- **[orm]** Fixed bug in 0.6-reworked "many-to-one" optimizations such that a many-to-one that is against a non-primary key column on the remote table (i.e. foreign key against a UNIQUE column) will pull the "old" value in from the database during a change, since if it's in the session we will need it for proper history/backref accounting, and we can't pull from the local identity map on a non-primary key column. ¶ References: #1737
- **[orm]** fixed internal error which would occur if calling has() or similar complex expression on a single-table inheritance relation(). ¶ References: #1731
- **[orm]** query.one() no longer applies LIMIT to the query, this to ensure that it fully counts all object identities present in the result, even in the case where joins may conceal multiple identities for two or more rows. As a bonus, one() can now also be called with a query that issued from_statement() to start with since it no longer modifies the query. ¶ References: #1688
- **[orm]** query.get() now returns None if queried for an identifier that is present in the identity map with a different class than the one requested, i.e. when using polymorphic loading. ¶ References: #1727
- **[orm]** A major fix in query.join(), when the "on" clause is an attribute of an aliased() construct, but there is already an existing join made out to a compatible target, query properly joins to the right aliased() construct instead of sticking onto the right side of the existing join. ¶ References: #1706
- **[orm]** Slight improvement to the fix for to not issue needless updates of the primary key column during a so-called "row switch" operation, i.e. add + delete of two objects with the same PK. ¶ References: #1362
- **[orm]** Now uses sqlalchemy.orm.exc.DetachedInstanceError when an attribute load or refresh action fails due to object being detached from any Session. UnboundExecutionError is specific to engines bound to sessions and statements. ¶
- **[orm]** Query called in the context of an expression will render disambiguating labels in all cases. Note that this does not apply to the existing .statement and .subquery() accessor/method, which still honors the .with_labels() setting that defaults to False. ¶
- **[orm]** Query.union() retains disambiguating labels within the returned statement, thus avoiding various SQL composition errors which can result from column name conflicts. ¶ References: #1676
- **[orm]** Fixed bug in attribute history that inadvertently invoked __eq__ on mapped instances. ¶
- **[orm]** Some internal streamlining of object loading grants a small speedup for large results, estimates are around 10-15%. Gave the "state" internals a good solid cleanup with less complexity, datamembers, method calls, blank dictionary creates. ¶
- **[orm]** Documentation clarification for query.delete() ¶ References: #1689
- **[orm]** Fixed cascade bug in many-to-one relation() when attribute was set to None, introduced in r6711 (cascade deleted items into session during add()). ¶
- **[orm]** Calling query.order_by() or query.distinct() before calling query.select_from(), query.with_polymorphic(), or query.from_statement() raises an exception now instead of silently dropping those criterion. ¶ References: #1736
- **[orm]** query.scalar() now raises an exception if more than one row is returned. All other behavior remains the same. ¶ References: #1735

- **[orm]** Fixed bug which caused "row switch" logic, that is an INSERT and DELETE replaced by an UPDATE, to fail when version_id_col was in use. ¶ References: #1692

sql

- **[sql]** join() will now simulate a NATURAL JOIN by default. Meaning, if the left side is a join, it will attempt to join the right side to the rightmost side of the left first, and not raise any exceptions about ambiguous join conditions if successful even if there are further join targets across the rest of the left. ¶ References: #1714
- **[sql]** The most common result processors conversion function were moved to the new "processors" module. Dialect authors are encouraged to use those functions whenever they correspond to their needs instead of implementing custom ones. ¶
- **[sql]** SchemaType and subclasses Boolean, Enum are now serializable, including their ddl listener and other event callables. ¶ References: #1694, #1698
- **[sql]** Some platforms will now interpret certain literal values as non-bind parameters, rendered literally into the SQL statement. This to support strict SQL-92 rules that are enforced by some platforms including MS-SQL and Sybase. In this model, bind parameters aren't allowed in the columns clause of a SELECT, nor are certain ambiguous expressions like "?=?". When this mode is enabled, the base compiler will render the binds as inline literals, but only across strings and numeric values. Other types such as dates will raise an error, unless the dialect subclass defines a literal rendering function for those. The bind parameter must have an embedded literal value already or an error is raised (i.e. won't work with straight bindparam('x')). Dialects can also expand upon the areas where binds are not accepted, such as within argument lists of functions (which don't work on MS-SQL when native SQL binding is used). ¶
- **[sql]** Added "unicode_errors" parameter to String, Unicode, etc. Behaves like the 'errors' keyword argument to the standard library's string.decode() functions. This flag requires that *convert_unicode* is set to *"force"* - otherwise, SQLAlchemy is not guaranteed to handle the task of unicode conversion. Note that this flag adds significant performance overhead to row-fetching operations for backends that already return unicode objects natively (which most DBAPIs do). This flag should only be used as an absolute last resort for reading strings from a column with varied or corrupted encodings, which only applies to databases that accept invalid encodings in the first place (i.e. MySQL. *not* PG, Sqlite, etc.) ¶
- **[sql]** Added math negation operator support, -x. ¶
- **[sql]** FunctionElement subclasses are now directly executable the same way any func.foo() construct is, with automatic SELECT being applied when passed to execute(). ¶
- **[sql]** The "type" and "bind" keyword arguments of a func.foo() construct are now local to "func." constructs and are not part of the FunctionElement base class, allowing a "type" to be handled in a custom constructor or class-level variable. ¶
- **[sql]** Restored the keys() method to ResultProxy. ¶
- **[sql]** The type/expression system now does a more complete job of determining the return type from an expression as well as the adaptation of the Python operator into a SQL operator, based on the full left/right/operator of the given expression. In particular the date/time/interval system created for Postgresql EXTRACT in has now been generalized into the type system. The previous behavior which often occurred of an expression "column + literal" forcing the type of "literal" to be the same as that of "column" will now usually not occur - the type of "literal" is first derived from the Python type of the literal, assuming standard native Python types + date types, before falling back to that of the known type on the other side of the expression. If the "fallback" type is compatible (i.e. CHAR from String), the literal side will use that. TypeDecorator types override this by default to coerce the "literal" side unconditionally, which can be changed by implementing the coerce_compared_value() method. Also part of. ¶ References: #1647, #1683
- **[sql]** Made sqlalchemy.sql.expressions.Executable part of public API, used for any expression construct that can be sent to execute(). FunctionElement now inherits Executable so that it gains execution_options(), which

are also propagated to the select() that's generated within execute(). Executable in turn subclasses _Generative which marks any ClauseElement that supports the @_generative decorator - these may also become "public" for the benefit of the compiler extension at some point. ¶

- **[sql]** A change to the solution for - an end-user defined bind parameter name that directly conflicts with a column-named bind generated directly from the SET or VALUES clause of an update/insert generates a compile error. This reduces call counts and eliminates some cases where undesirable name conflicts could still occur. ¶ References: #1579
- **[sql]** Column() requires a type if it has no foreign keys (this is not new). An error is now raised if a Column() has no type and no foreign keys. ¶ References: #1705
- **[sql]** the "scale" argument of the Numeric() type is honored when coercing a returned floating point value into a string on its way to Decimal - this allows accuracy to function on SQLite, MySQL. ¶ References: #1717
- **[sql]** the copy() method of Column now copies over uninitialized "on table attach" events. Helps with the new declarative "mixin" capability. ¶

mysql

- **[mysql]** Fixed reflection bug whereby when COLLATE was present, nullable flag and server defaults would not be reflected. ¶ References: #1655
- **[mysql]** Fixed reflection of TINYINT(1) "boolean" columns defined with integer flags like UNSIGNED. ¶
- **[mysql]** Further fixes for the mysql-connector dialect. ¶ References: #1668
- **[mysql]** Composite PK table on InnoDB where the "autoincrement" column isn't first will emit an explicit "KEY" phrase within CREATE TABLE thereby avoiding errors. ¶ References: #1496
- **[mysql]** Added reflection/create table support for a wide range of MySQL keywords. ¶ References: #1634
- **[mysql]** Fixed import error which could occur reflecting tables on a Windows host ¶ References: #1580

sqlite

- **[sqlite]** Added "native_datetime=True" flag to create_engine(). This will cause the DATE and TIMESTAMP types to skip all bind parameter and result row processing, under the assumption that PARSE_DECLTYPES has been enabled on the connection. Note that this is not entirely compatible with the "func.current_date()", which will be returned as a string. ¶ References: #1685

mssql

- **[mssql]** Re-established support for the pymssql dialect. ¶
- **[mssql]** Various fixes for implicit returning, reflection, etc. - the MS-SQL dialects aren't quite complete in 0.6 yet (but are close) ¶
- **[mssql]** Added basic support for mxODBC. ¶ References: #1710
- **[mssql]** Removed the text_as_varchar option. ¶

oracle

- **[oracle]** "out" parameters require a type that is supported by cx_oracle. An error will be raised if no cx_oracle type can be found. ¶

- **[oracle]** Oracle 'DATE' now does not perform any result processing, as the DATE type in Oracle stores full date+time objects, that's what you'll get. Note that the generic types.Date type *will* still call value.date() on incoming values, however. When reflecting a table, the reflected type will be 'DATE'. ¶
- **[oracle]** Added preliminary support for Oracle's WITH_UNICODE mode. At the very least this establishes initial support for cx_Oracle with Python 3. When WITH_UNICODE mode is used in Python 2.xx, a large and scary warning is emitted asking that the user seriously consider the usage of this difficult mode of operation. ¶ References: #1670
- **[oracle]** The except_() method now renders as MINUS on Oracle, which is more or less equivalent on that platform. ¶ References: #1712
- **[oracle]** Added support for rendering and reflecting TIMESTAMP WITH TIME ZONE, i.e. TIMESTAMP(timezone=True). ¶ References: #651
- **[oracle]** Oracle INTERVAL type can now be reflected. ¶

misc

- **[py3k]** Improved the installation/test setup regarding Python 3, now that Distribute runs on Py3k. distribute_setup.py is now included. See README.py3k for Python 3 installation/ testing instructions. ¶
- **[engines]** Added an optional C extension to speed up the sql layer by reimplementing RowProxy and the most common result processors. The actual speedups will depend heavily on your DBAPI and the mix of datatypes used in your tables, and can vary from a 30% improvement to more than 200%. It also provides a modest (~15-20%) indirect improvement to ORM speed for large queries. Note that it is *not* built/installed by default. See README for installation instructions. ¶
- **[engines]** the execution sequence pulls all rowcount/last inserted ID info from the cursor before commit() is called on the DBAPI connection in an "autocommit" scenario. This helps mxodbc with rowcount and is probably a good idea overall. ¶
- **[engines]** Opened up logging a bit such that isEnabledFor() is called more often, so that changes to the log level for engine/pool will be reflected on next connect. This adds a small amount of method call overhead. It's negligible and will make life a lot easier for all those situations when logging just happens to be configured after create_engine() is called. ¶ References: #1719
- **[engines]** The assert_unicode flag is deprecated. SQLAlchemy will raise a warning in all cases where it is asked to encode a non-unicode Python string, as well as when a Unicode or UnicodeType type is explicitly passed a bytestring. The String type will do nothing for DBAPIs that already accept Python unicode objects. ¶
- **[engines]** Bind parameters are sent as a tuple instead of a list. Some backend drivers will not accept bind parameters as a list. ¶
- **[engines]** threadlocal engine wasn't properly closing the connection upon close() - fixed that. ¶
- **[engines]** Transaction object doesn't rollback or commit if it isn't "active", allows more accurate nesting of begin/rollback/commit. ¶
- **[engines]** Python unicode objects as binds result in the Unicode type, not string, thus eliminating a certain class of unicode errors on drivers that don't support unicode binds. ¶
- **[engines]** Added "logging_name" argument to create_engine(), Pool() constructor as well as "pool_logging_name" argument to create_engine() which filters down to that of Pool. Issues the given string name within the "name" field of logging messages instead of the default hex identifier string. ¶ References: #1555
- **[engines]** The visit_pool() method of Dialect is removed, and replaced with on_connect(). This method returns a callable which receives the raw DBAPI connection after each one is created. The callable is assembled into

a first_connect/connect pool listener by the connection strategy if non-None. Provides a simpler interface for dialects. ¶

- **[engines]** StaticPool now initializes, disposes and recreates without opening a new connection - the connection is only opened when first requested. dispose() also works on AssertionPool now. ¶ References: #1728
- **[ticket: 1673] [metadata]** Added the ability to strip schema information when using "tometadata" by passing "schema=None" as an argument. If schema is not specified then the table's schema is retained. ¶
- **[declarative]** DeclarativeMeta exclusively uses cls.__dict__ (not dict_) as the source of class information; _as_declarative exclusively uses the dict_ passed to it as the source of class information (which when using DeclarativeMeta is cls.__dict__). This should in theory make it easier for custom metaclasses to modify the state passed into _as_declarative. ¶
- **[declarative]** declarative now accepts mixin classes directly, as a means to provide common functional and column-based elements on all subclasses, as well as a means to propagate a fixed set of __table_args__ or __mapper_args__ to subclasses. For custom combinations of __table_args__/__mapper_args__ from an inherited mixin to local, descriptors can now be used. New details are all up in the Declarative documentation. Thanks to Chris Withers for putting up with my strife on this. ¶ References: #1707
- **[declarative]** the __mapper_args__ dict is copied when propagating to a subclass, and is taken straight off the class __dict__ to avoid any propagation from the parent. mapper inheritance already propagates the things you want from the parent mapper. ¶ References: #1393
- **[declarative]** An exception is raised when a single-table subclass specifies a column that is already present on the base class. ¶ References: #1732
- **[sybase]** Implemented a preliminary working dialect for Sybase, with sub-implementations for Python-Sybase as well as Pyodbc. Handles table creates/drops and basic round trip functionality. Does not yet include reflection or comprehensive support of unicode/special expressions/etc. ¶
- **[examples]** Changed the beaker cache example a bit to have a separate RelationCache option for lazyload caching. This object does a lookup among any number of potential attributes more efficiently by grouping several into a common structure. Both FromCache and RelationCache are simpler individually. ¶
- **[documentation]** Major cleanup work in the docs to link class, function, and method names into the API docs. ¶ References: #1700

0.6beta1

Released: Wed Feb 03 2010

orm

- **[orm]**

  Changes to query.update() and query.delete():

  - the 'expire' option on query.update() has been renamed to 'fetch', thus matching that of query.delete(). 'expire' is deprecated and issues a warning.
  - query.update() and query.delete() both default to 'evaluate' for the synchronize strategy.
  - the 'synchronize' strategy for update() and delete() raises an error on failure. There is no implicit fallback onto "fetch". Failure of evaluation is based on the structure of criteria, so success/failure is deterministic based on code structure.

  ¶
- **[orm]**

Enhancements on many-to-one relations:

- many-to-one relations now fire off a lazyload in fewer cases, including in most cases will not fetch the "old" value when a new one is replaced.
- many-to-one relation to a joined-table subclass now uses get() for a simple load (known as the "use_get" condition), i.e. Related->Sub(Base), without the need to redefine the primaryjoin condition in terms of the base table.
- specifying a foreign key with a declarative column, i.e. ForeignKey(MyRelatedClass.id) doesn't break the "use_get" condition from taking place
- relation(), eagerload(), and eagerload_all() now feature an option called "innerjoin". Specify *True* or *False* to control whether an eager join is constructed as an INNER or OUTER join. Default is *False* as always. The mapper options will override whichever setting is specified on relation(). Should generally be set for many-to-one, not nullable foreign key relations to allow improved join performance.
- the behavior of eagerloading such that the main query is wrapped in a subquery when LIMIT/OFFSET are present now makes an exception for the case when all eager loads are many-to-one joins. In those cases, the eager joins are against the parent table directly along with the limit/offset without the extra overhead of a subquery, since a many-to-one join does not add rows to the result.

¶ References: #1186, #1492, #1544

- **[orm]** Enhancements / Changes on Session.merge(): ¶
- **[orm]** the "dont_load=True" flag on Session.merge() is deprecated and is now "load=False". ¶
- **[orm]** Session.merge() is performance optimized, using half the call counts for "load=False" mode compared to 0.5 and significantly fewer SQL queries in the case of collections for "load=True" mode. ¶
- **[orm]** merge() will not issue a needless merge of attributes if the given instance is the same instance which is already present. ¶
- **[orm]** merge() now also merges the "options" associated with a given state, i.e. those passed through query.options() which follow along with an instance, such as options to eagerly- or lazyily- load various attributes. This is essential for the construction of highly integrated caching schemes. This is a subtle behavioral change vs. 0.5. ¶
- **[orm]** A bug was fixed regarding the serialization of the "loader path" present on an instance's state, which is also necessary when combining the usage of merge() with serialized state and associated options that should be preserved. ¶
- **[orm]** The all new merge() is showcased in a new comprehensive example of how to integrate Beaker with SQLAlchemy. See the notes in the "examples" note below. ¶
- **[orm]** Primary key values can now be changed on a joined-table inheritance object, and ON UPDATE CASCADE will be taken into account when the flush happens. Set the new "passive_updates" flag to False on mapper() when using SQLite or MySQL/MyISAM. ¶ References: #1362
- **[orm]** flush() now detects when a primary key column was updated by an ON UPDATE CASCADE operation from another primary key, and can then locate the row for a subsequent UPDATE on the new PK value. This occurs when a relation() is there to establish the relationship as well as passive_updates=True. ¶ References: #1671
- **[orm]** the "save-update" cascade will now cascade the pending *removed* values from a scalar or collection attribute into the new session during an add() operation. This so that the flush() operation will also delete or modify rows of those disconnected items. ¶

- **[orm]** Using a "dynamic" loader with a "secondary" table now produces a query where the "secondary" table is *not* aliased. This allows the secondary Table object to be used in the "order_by" attribute of the relation(), and also allows it to be used in filter criterion against the dynamic relation. ¶ References: #1531
- **[orm]** relation() with uselist=False will emit a warning when an eager or lazy load locates more than one valid value for the row. This may be due to primaryjoin/secondaryjoin conditions which aren't appropriate for an eager LEFT OUTER JOIN or for other conditions. ¶ References: #1643
- **[orm]** an explicit check occurs when a synonym() is used with map_column=True, when a ColumnProperty (deferred or otherwise) exists separately in the properties dictionary sent to mapper with the same keyname. Instead of silently replacing the existing property (and possible options on that property), an error is raised. ¶ References: #1633
- **[orm]** a "dynamic" loader sets up its query criterion at construction time so that the actual query is returned from non-cloning accessors like "statement". ¶
- **[orm]** the "named tuple" objects returned when iterating a Query() are now pickleable. ¶
- **[orm]** mapping to a select() construct now requires that you make an alias() out of it distinctly. This to eliminate confusion over such issues as ¶ References: #1542
- **[orm]** query.join() has been reworked to provide more consistent behavior and more flexibility (includes) ¶ References: #1537
- **[orm]** query.select_from() accepts multiple clauses to produce multiple comma separated entries within the FROM clause. Useful when selecting from multiple-homed join() clauses. ¶
- **[orm]** query.select_from() also accepts mapped classes, aliased() constructs, and mappers as arguments. In particular this helps when querying from multiple joined-table classes to ensure the full join gets rendered. ¶
- **[orm]** query.get() can be used with a mapping to an outer join where one or more of the primary key values are None. ¶ References: #1135
- **[orm]** query.from_self(), query.union(), others which do a "SELECT * from (SELECT...)" type of nesting will do a better job translating column expressions within the subquery to the columns clause of the outer query. This is potentially backwards incompatible with 0.5, in that this may break queries with literal expressions that do not have labels applied (i.e. literal('foo'), etc.) ¶ References: #1568
- **[orm]** relation primaryjoin and secondaryjoin now check that they are column-expressions, not just clause elements. this prohibits things like FROM expressions being placed there directly. ¶ References: #1622
- **[orm]** *expression.null()* is fully understood the same way None is when comparing an object/collection-referencing attribute within query.filter(), filter_by(), etc. ¶ References: #1415
- **[orm]** added "make_transient()" helper function which transforms a persistent/ detached instance into a transient one (i.e. deletes the instance_key and removes from any session.) ¶ References: #1052
- **[orm]** the allow_null_pks flag on mapper() is deprecated, and the feature is turned "on" by default. This means that a row which has a non-null value for any of its primary key columns will be considered an identity. The need for this scenario typically only occurs when mapping to an outer join. ¶ References: #1339
- **[orm]** the mechanics of "backref" have been fully merged into the finer grained "back_populates" system, and take place entirely within the _generate_backref() method of RelationProperty. This makes the initialization procedure of RelationProperty simpler and allows easier propagation of settings (such as from subclasses of RelationProperty) into the reverse reference. The internal BackRef() is gone and backref() returns a plain tuple that is understood by RelationProperty. ¶
- **[orm]** The version_id_col feature on mapper() will raise a warning when used with dialects that don't support "rowcount" adequately. ¶ References: #1569

- **[orm]** added "execution_options()" to Query, to so options can be passed to the resulting statement. Currently only Select-statements have these options, and the only option used is "stream_results", and the only dialect which knows "stream_results" is psycopg2. ¶
- **[orm]** Query.yield_per() will set the "stream_results" statement option automatically. ¶
- **[orm]**

  Deprecated or removed:

  - 'allow_null_pks' flag on mapper() is deprecated. It does nothing now and the setting is "on" in all cases.
  - 'transactional' flag on sessionmaker() and others is removed. Use 'autocommit=True' to indicate 'transactional=False'.
  - 'polymorphic_fetch' argument on mapper() is removed. Loading can be controlled using the 'with_polymorphic' option.
  - 'select_table' argument on mapper() is removed. Use 'with_polymorphic=("*", <some selectable>)' for this functionality.
  - 'proxy' argument on synonym() is removed. This flag did nothing throughout 0.5, as the "proxy generation" behavior is now automatic.
  - Passing a single list of elements to eagerload(), eagerload_all(), contains_eager(), lazyload(), defer(), and undefer() instead of multiple positional *args is deprecated.
  - Passing a single list of elements to query.order_by(), query.group_by(), query.join(), or query.outerjoin() instead of multiple positional *args is deprecated.
  - query.iterate_instances() is removed. Use query.instances().
  - Query.query_from_parent() is removed. Use the sqlalchemy.orm.with_parent() function to produce a "parent" clause, or alternatively query.with_parent().
  - query._from_self() is removed, use query.from_self() instead.
  - the "comparator" argument to composite() is removed. Use "comparator_factory".
  - RelationProperty._get_join() is removed.
  - the 'echo_uow' flag on Session is removed. Use logging on the "sqlalchemy.orm.unitofwork" name.
  - session.clear() is removed. use session.expunge_all().
  - session.save(), session.update(), session.save_or_update() are removed. Use session.add() and session.add_all().
  - the "objects" flag on session.flush() remains deprecated.
  - the "dont_load=True" flag on session.merge() is deprecated in favor of "load=False".
  - ScopedSession.mapper remains deprecated. See the usage recipe at http://www.sqlalchemy.org/trac/wiki/UsageRecipes/SessionAwareMapper
  - passing an InstanceState (internal SQLAlchemy state object) to attributes.init_collection() or attributes.get_history() is deprecated. These functions are public API and normally expect a regular mapped object instance.
  - the 'engine' parameter to declarative_base() is removed. Use the 'bind' keyword argument.

¶

sql

- **[sql]** the "autocommit" flag on select() and text() as well as select().autocommit() are deprecated - now call .execution_options(autocommit=True) on either of those constructs, also available directly on Connection and orm.Query. ¶
- **[sql]** the autoincrement flag on column now indicates the column which should be linked to cursor.lastrowid, if that method is used. See the API docs for details. ¶
- **[sql]** an executemany() now requires that all bound parameter sets require that all keys are present which are present in the first bound parameter set. The structure and behavior of an insert/update statement is very much determined by the first parameter set, including which defaults are going to fire off, and a minimum of guesswork is performed with all the rest so that performance is not impacted. For this reason defaults would otherwise silently "fail" for missing parameters, so this is now guarded against. ¶ References: #1566
- **[sql]** returning() support is native to insert(), update(), delete(). Implementations of varying levels of functionality exist for Postgresql, Firebird, MSSQL and Oracle. returning() can be called explicitly with column expressions which are then returned in the resultset, usually via fetchone() or first().

  insert() constructs will also use RETURNING implicitly to get newly generated primary key values, if the database version in use supports it (a version number check is performed). This occurs if no end-user returning() was specified. ¶
- **[sql]** union(), intersect(), except() and other "compound" types of statements have more consistent behavior w.r.t. parenthesizing. Each compound element embedded within another will now be grouped with parenthesis - previously, the first compound element in the list would not be grouped, as SQLite doesn't like a statement to start with parenthesis. However, Postgresql in particular has precedence rules regarding INTERSECT, and it is more consistent for parenthesis to be applied equally to all sub-elements. So now, the workaround for SQLite is also what the workaround for PG was previously - when nesting compound elements, the first one usually needs ".alias().select()" called on it to wrap it inside of a subquery. ¶ References: #1665
- **[sql]** insert() and update() constructs can now embed bindparam() objects using names that match the keys of columns. These bind parameters will circumvent the usual route to those keys showing up in the VALUES or SET clause of the generated SQL. ¶ References: #1579
- **[sql]** the Binary type now returns data as a Python string (or a "bytes" type in Python 3), instead of the built- in "buffer" type. This allows symmetric round trips of binary data. ¶ References: #1524
- **[sql]** Added a tuple_() construct, allows sets of expressions to be compared to another set, typically with IN against composite primary keys or similar. Also accepts an IN with multiple columns. The "scalar select can have only one column" error message is removed - will rely upon the database to report problems with col mismatch. ¶
- **[sql]** User-defined "default" and "onupdate" callables which accept a context should now call upon "context.current_parameters" to get at the dictionary of bind parameters currently being processed. This dict is available in the same way regardless of single-execute or executemany-style statement execution. ¶
- **[sql]** multi-part schema names, i.e. with dots such as "dbo.master", are now rendered in select() labels with underscores for dots, i.e. "dbo_master_table_column". This is a "friendly" label that behaves better in result sets. ¶ References: #1428
- **[sql]** removed needless "counter" behavior with select() labelnames that match a column name in the table, i.e. generates "tablename_id" for "id", instead of "tablename_id_1" in an attempt to avoid naming conflicts, when the table has a column actually named "tablename_id" - this is because the labeling logic is always applied to all columns so a naming conflict will never occur. ¶
- **[sql]** calling expr.in_([]), i.e. with an empty list, emits a warning before issuing the usual "expr != expr" clause. The "expr != expr" can be very expensive, and it's preferred that the user not issue in_() if the list is empty, instead simply not querying, or modifying the criterion as appropriate for more complex situations. ¶ References: #1628

- **[sql]** Added "execution_options()" to select()/text(), which set the default options for the Connection. See the note in "engines". ¶
- **[sql]**

  Deprecated or removed:

  - "scalar" flag on select() is removed, use select.as_scalar().
  - "shortname" attribute on bindparam() is removed.
  - postgres_returning, firebird_returning flags on insert(), update(), delete() are deprecated, use the new returning() method.
  - fold_equivalents flag on join is deprecated (will remain until is implemented)

  ¶ References: #1131

schema

- **[schema]** the *__contains__()* method of *MetaData* now accepts strings or *Table* objects as arguments. If given a *Table*, the argument is converted to *table.key* first, i.e. "[schemaname.]<tablename>" ¶ References: #1541
- **[schema]** deprecated MetaData.connect() and ThreadLocalMetaData.connect() have been removed - send the "bind" attribute to bind a metadata. ¶
- **[schema]** deprecated metadata.table_iterator() method removed (use sorted_tables) ¶
- **[schema]** deprecated PassiveDefault - use DefaultClause. ¶
- **[schema]** the "metadata" argument is removed from DefaultGenerator and subclasses, but remains locally present on Sequence, which is a standalone construct in DDL. ¶
- **[schema]** Removed public mutability from Index and Constraint objects:

  - ForeignKeyConstraint.append_element()
  - Index.append_column()
  - UniqueConstraint.append_column()
  - PrimaryKeyConstraint.add()
  - PrimaryKeyConstraint.remove()

  These should be constructed declaratively (i.e. in one construction). ¶
- **[schema]** The "start" and "increment" attributes on Sequence now generate "START WITH" and "INCREMENT BY" by default, on Oracle and Postgresql. Firebird doesn't support these keywords right now. ¶ References: #1545
- **[schema]** UniqueConstraint, Index, PrimaryKeyConstraint all accept lists of column names or column objects as arguments. ¶
- **[schema]**

  Other removed things:

  - Table.key (no idea what this was for)
  - Table.primary_key is not assignable - use table.append_constraint(PrimaryKeyConstraint(...))
  - Column.bind (get via column.table.bind)
  - Column.metadata (get via column.table.metadata)

  - Column.sequence (use column.default)
  - ForeignKey(constraint=some_parent) (is now private _constraint)

¶

- **[schema]** The use_alter flag on ForeignKey is now a shortcut option for operations that can be hand-constructed using the DDL() event system. A side effect of this refactor is that ForeignKeyConstraint objects with use_alter=True will *not* be emitted on SQLite, which does not support ALTER for foreign keys. ¶
- **[schema]** ForeignKey and ForeignKeyConstraint objects now correctly copy() all their public keyword arguments. ¶ References: #1605

postgresql

- **[postgresql]** New dialects: pg8000, zxjdbc, and pypostgresql on py3k. ¶
- **[postgresql]** The "postgres" dialect is now named "postgresql" ! Connection strings look like:

  postgresql://scott:tiger@localhost/test postgresql+pg8000://scott:tiger@localhost/test

  The "postgres" name remains for backwards compatibility in the following ways:

  - There is a "postgres.py" dummy dialect which allows old URLs to work, i.e. postgres://scott:tiger@localhost/test
  - The "postgres" name can be imported from the old "databases" module, i.e. "from sqlalchemy.databases import postgres" as well as "dialects", "from sqlalchemy.dialects.postgres import base as pg", will send a deprecation warning.
  - Special expression arguments are now named "postgresql_returning" and "postgresql_where", but the older "postgres_returning" and "postgres_where" names still work with a deprecation warning.

  ¶

- **[postgresql]** "postgresql_where" now accepts SQL expressions which can also include literals, which will be quoted as needed. ¶
- **[postgresql]** The psycopg2 dialect now uses psycopg2's "unicode extension" on all new connections, which allows all String/Text/etc. types to skip the need to post-process bytestrings into unicode (an expensive step due to its volume). Other dialects which return unicode natively (pg8000, zxjdbc) also skip unicode post-processing. ¶
- **[postgresql]** Added new ENUM type, which exists as a schema-level construct and extends the generic Enum type. Automatically associates itself with tables and their parent metadata to issue the appropriate CREATE TYPE/DROP TYPE commands as needed, supports unicode labels, supports reflection. ¶ References: #1511
- **[postgresql]** INTERVAL supports an optional "precision" argument corresponding to the argument that PG accepts. ¶
- **[postgresql]** using new dialect.initialize() feature to set up version-dependent behavior. ¶
- **[postgresql]** somewhat better support for % signs in table/column names; psycopg2 can't handle a bind parameter name of %(foobar)s however and SQLA doesn't want to add overhead just to treat that one non-existent use case. ¶ References: #1279
- **[postgresql]** Inserting NULL into a primary key + foreign key column will allow the "not null constraint" error to raise, not an attempt to execute a nonexistent "col_id_seq" sequence. ¶ References: #1516
- **[postgresql]** autoincrement SELECT statements, i.e. those which select from a procedure that modifies rows, now work with server-side cursor mode (the named cursor isn't used for such statements.) ¶

- **[postgresql]** postgresql dialect can properly detect pg "devel" version strings, i.e. "8.5devel" ¶ References: #1636
- **[postgresql]** The psycopg2 now respects the statement option "stream_results". This option overrides the connection setting "server_side_cursors". If true, server side cursors will be used for the statement. If false, they will not be used, even if "server_side_cursors" is true on the connection. ¶ References: #1619

mysql

- **[mysql]** New dialects: oursql, a new native dialect, MySQL Connector/Python, a native Python port of MySQLdb, and of course zxjdbc on Jython. ¶
- **[mysql]** VARCHAR/NVARCHAR will not render without a length, raises an error before passing to MySQL. Doesn't impact CAST since VARCHAR is not allowed in MySQL CAST anyway, the dialect renders CHAR/NCHAR in those cases. ¶
- **[mysql]** all the _detect_XXX() functions now run once underneath dialect.initialize() ¶
- **[mysql]** somewhat better support for % signs in table/column names; MySQLdb can't handle % signs in SQL when executemany() is used, and SQLA doesn't want to add overhead just to treat that one non-existent use case. ¶ References: #1279
- **[mysql]** the BINARY and MSBinary types now generate "BINARY" in all cases. Omitting the "length" parameter will generate "BINARY" with no length. Use BLOB to generate an unlengthed binary column. ¶
- **[mysql]** the "quoting='quoted'" argument to MSEnum/ENUM is deprecated. It's best to rely upon the automatic quoting. ¶
- **[mysql]** ENUM now subclasses the new generic Enum type, and also handles unicode values implicitly, if the given labelnames are unicode objects. ¶
- **[mysql]** a column of type TIMESTAMP now defaults to NULL if "nullable=False" is not passed to Column(), and no default is present. This is now consistent with all other types, and in the case of TIMESTAMP explicitly renders "NULL" due to MySQL's "switching" of default nullability for TIMESTAMP columns. ¶ References: #1539

sqlite

- **[sqlite]** DATE, TIME and DATETIME types can now take optional storage_format and regexp argument. storage_format can be used to store those types using a custom string format. regexp allows to use a custom regular expression to match string values from the database. ¶
- **[sqlite]** Time and DateTime types now use by a default a stricter regular expression to match strings from the database. Use the regexp argument if you are using data stored in a legacy format. ¶
- **[sqlite]** __legacy_microseconds__ on SQLite Time and DateTime types is not supported anymore. You should use the storage_format argument instead. ¶
- **[sqlite]** Date, Time and DateTime types are now stricter in what they accept as bind parameters: Date type only accepts date objects (and datetime ones, because they inherit from date), Time only accepts time objects, and DateTime only accepts date and datetime objects. ¶
- **[sqlite]** Table() supports a keyword argument "sqlite_autoincrement", which applies the SQLite keyword "AUTOINCREMENT" to the single integer primary key column when generating DDL. Will prevent generation of a separate PRIMARY KEY constraint. ¶ References: #1016

mssql

- **[mssql]** MSSQL + Pyodbc + FreeTDS now works for the most part, with possible exceptions regarding binary data as well as unicode schema identifiers. ¶
- **[mssql]** the "has_window_funcs" flag is removed. LIMIT/OFFSET usage will use ROW NUMBER as always, and if on an older version of SQL Server, the operation fails. The behavior is exactly the same except the error is raised by SQL server instead of the dialect, and no flag setting is required to enable it. ¶
- **[mssql]** the "auto_identity_insert" flag is removed. This feature always takes effect when an INSERT statement overrides a column that is known to have a sequence on it. As with "has_window_funcs", if the underlying driver doesn't support this, then you can't do this operation in any case, so there's no point in having a flag. ¶
- **[mssql]** using new dialect.initialize() feature to set up version-dependent behavior. ¶
- **[mssql]** removed references to sequence which is no longer used. implicit identities in mssql work the same as implicit sequences on any other dialects. Explicit sequences are enabled through the use of "default=Sequence()". See the MSSQL dialect documentation for more information. ¶

oracle

- **[oracle]** unit tests pass 100% with cx_oracle ! ¶
- **[oracle]** support for cx_Oracle's "native unicode" mode which does not require NLS_LANG to be set. Use the latest 5.0.2 or later of cx_oracle. ¶
- **[oracle]** an NCLOB type is added to the base types. ¶
- **[oracle]** use_ansi=False won't leak into the FROM/WHERE clause of a statement that's selecting from a subquery that also uses JOIN/OUTERJOIN. ¶
- **[oracle]** added native INTERVAL type to the dialect. This supports only the DAY TO SECOND interval type so far due to lack of support in cx_oracle for YEAR TO MONTH. ¶ References: #1467
- **[oracle]** usage of the CHAR type results in cx_oracle's FIXED_CHAR dbapi type being bound to statements. ¶
- **[oracle]** the Oracle dialect now features NUMBER which intends to act justlike Oracle's NUMBER type. It is the primary numeric type returned by table reflection and attempts to return Decimal()/float/int based on the precision/scale parameters. ¶ References: #885
- **[oracle]** func.char_length is a generic function for LENGTH ¶
- **[oracle]** ForeignKey() which includes onupdate=<value> will emit a warning, not emit ON UPDATE CASCADE which is unsupported by oracle ¶
- **[oracle]** the keys() method of RowProxy() now returns the result column names *normalized* to be SQLAlchemy case insensitive names. This means they will be lower case for case insensitive names, whereas the DBAPI would normally return them as UPPERCASE names. This allows row keys() to be compatible with further SQLAlchemy operations. ¶
- **[oracle]** using new dialect.initialize() feature to set up version-dependent behavior. ¶
- **[oracle]** using types.BigInteger with Oracle will generate NUMBER(19) ¶ References: #1125
- **[oracle]** "case sensitivity" feature will detect an all-lowercase case-sensitive column name during reflect and add "quote=True" to the generated Column, so that proper quoting is maintained. ¶

firebird

- **[firebird]** the keys() method of RowProxy() now returns the result column names *normalized* to be SQLAlchemy case insensitive names. This means they will be lower case for case insensitive names, whereas the DBAPI would normally return them as UPPERCASE names. This allows row keys() to be compatible with further SQLAlchemy operations. ¶
- **[firebird]** using new dialect.initialize() feature to set up version-dependent behavior. ¶
- **[firebird]** "case sensitivity" feature will detect an all-lowercase case-sensitive column name during reflect and add "quote=True" to the generated Column, so that proper quoting is maintained. ¶

misc

- **[release] [major]** For the full set of feature descriptions, see http://www.sqlalchemy.org/trac/wiki/06Migration . This document is a work in progress. ¶
- **[release] [major]** All bug fixes and feature enhancements from the most recent 0.5 version and below are also included within 0.6. ¶
- **[release] [major]** Platforms targeted now include Python 2.4/2.5/2.6, Python 3.1, Jython2.5. ¶
- **[engines]** transaction isolation level may be specified with create_engine(... isolation_level="...") ; available on postgresql and sqlite. ¶ References: #443
- **[engines]** Connection has execution_options(), generative method which accepts keywords that affect how the statement is executed w.r.t. the DBAPI. Currently supports "stream_results", causes psycopg2 to use a server side cursor for that statement, as well as "autocommit", which is the new location for the "autocommit" option from select() and text(). select() and text() also have .execution_options() as well as ORM Query(). ¶
- **[engines]** fixed the import for entrypoint-driven dialects to not rely upon silly tb_info trick to determine import error status. ¶ References: #1630
- **[engines]** added first() method to ResultProxy, returns first row and closes result set immediately. ¶
- **[engines]** RowProxy objects are now pickleable, i.e. the object returned by result.fetchone(), result.fetchall() etc. ¶
- **[engines]** RowProxy no longer has a close() method, as the row no longer maintains a reference to the parent. Call close() on the parent ResultProxy instead, or use autoclose. ¶
- **[engines]** ResultProxy internals have been overhauled to greatly reduce method call counts when fetching columns. Can provide a large speed improvement (up to more than 100%) when fetching large result sets. The improvement is larger when fetching columns that have no type-level processing applied and when using results as tuples (instead of as dictionaries). Many thanks to Elixir's Gaëtan de Menten for this dramatic improvement ! ¶ References: #1586
- **[engines]** Databases which rely upon postfetch of "last inserted id" to get at a generated sequence value (i.e. MySQL, MS-SQL) now work correctly when there is a composite primary key where the "autoincrement" column is not the first primary key column in the table. ¶
- **[engines]** the last_inserted_ids() method has been renamed to the descriptor "inserted_primary_key". ¶
- **[engines]** setting echo=False on create_engine() now sets the loglevel to WARN instead of NOTSET. This so that logging can be disabled for a particular engine even if logging for "sqlalchemy.engine" is enabled overall. Note that the default setting of "echo" is *None*. ¶ References: #1554
- **[engines]** ConnectionProxy now has wrapper methods for all transaction lifecycle events, including begin(), rollback(), commit() begin_nested(), begin_prepared(), prepare(), release_savepoint(), etc. ¶

- **[engines]** Connection pool logging now uses both INFO and DEBUG log levels for logging. INFO is for major events such as invalidated connections, DEBUG for all the acquire/return logging. *echo_pool* can be False, None, True or "debug" the same way as *echo* works. ¶
- **[engines]** All pyodbc-dialects now support extra pyodbc-specific kw arguments 'ansi', 'unicode_results', 'autocommit'. ¶ References: #1621
- **[engines]** the "threadlocal" engine has been rewritten and simplified and now supports SAVEPOINT operations. ¶
- **[engines]**

  deprecated or removed

  - result.last_inserted_ids() is deprecated. Use result.inserted_primary_key
  - dialect.get_default_schema_name(connection) is now public via dialect.default_schema_name.
  - the "connection" argument from engine.transaction() and engine.run_callable() is removed - Connection itself now has those methods. All four methods accept *args and **kwargs which are passed to the given callable, as well as the operating connection.

  ¶
- **[reflection/inspection]** Table reflection has been expanded and generalized into a new API called "sqlalchemy.engine.reflection.Inspector". The Inspector object provides fine-grained information about a wide variety of schema information, with room for expansion, including table names, column names, view definitions, sequences, indexes, etc. ¶
- **[reflection/inspection]** Views are now reflectable as ordinary Table objects. The same Table constructor is used, with the caveat that "effective" primary and foreign key constraints aren't part of the reflection results; these have to be specified explicitly if desired. ¶
- **[reflection/inspection]** The existing autoload=True system now uses Inspector underneath so that each dialect need only return "raw" data about tables and other objects - Inspector is the single place that information is compiled into Table objects so that consistency is at a maximum. ¶
- **[ddl]** the DDL system has been greatly expanded. the DDL() class now extends the more generic DDLElement(), which forms the basis of many new constructs:

  - CreateTable()
  - DropTable()
  - AddConstraint()
  - DropConstraint()
  - CreateIndex()
  - DropIndex()
  - CreateSequence()
  - DropSequence()

  These support "on" and "execute-at()" just like plain DDL() does. User-defined DDLElement subclasses can be created and linked to a compiler using the sqlalchemy.ext.compiler extension.

  ¶
- **[ddl]** The signature of the "on" callable passed to DDL() and DDLElement() is revised as follows:

  ddl the DDLElement object itself

  event the string event name.

target previously "schema_item", the Table or MetaData object triggering the event.

connection the Connection object in use for the operation.

****kw** keyword arguments. In the case of MetaData before/after create/drop, the list of Table objects for which CREATE/DROP DDL is to be issued is passed as the kw argument "tables". This is necessary for metadata-level DDL that is dependent on the presence of specific tables.

The "schema_item" attribute of DDL has been renamed to "target".

¶

- **[dialect] [refactor]** Dialect modules are now broken into database dialects plus DBAPI implementations. Connect URLs are now preferred to be specified using dialect+driver://..., i.e. "mysql+mysqldb://scott:tiger@localhost/test". See the 0.6 documentation for examples. ¶
- **[dialect] [refactor]** the setuptools entrypoint for external dialects is now called "sqlalchemy.dialects". ¶
- **[dialect] [refactor]** the "owner" keyword argument is removed from Table. Use "schema" to represent any namespaces to be prepended to the table name. ¶
- **[dialect] [refactor]** server_version_info becomes a static attribute. ¶
- **[dialect] [refactor]** dialects receive an initialize() event on initial connection to determine connection properties. ¶
- **[dialect] [refactor]** dialects receive a visit_pool event have an opportunity to establish pool listeners. ¶
- **[dialect] [refactor]** cached TypeEngine classes are cached per-dialect class instead of per-dialect. ¶
- **[dialect] [refactor]** new UserDefinedType should be used as a base class for new types, which preserves the 0.5 behavior of get_col_spec(). ¶
- **[dialect] [refactor]** The result_processor() method of all type classes now accepts a second argument "coltype", which is the DBAPI type argument from cursor.description. This argument can help some types decide on the most efficient processing of result values. ¶
- **[dialect] [refactor]** Deprecated Dialect.get_params() removed. ¶
- **[dialect] [refactor]** Dialect.get_rowcount() has been renamed to a descriptor "rowcount", and calls cursor.rowcount directly. Dialects which need to hardwire a rowcount in for certain calls should override the method to provide different behavior. ¶
- **[dialect] [refactor]** DefaultRunner and subclasses have been removed. The job of this object has been simplified and moved into ExecutionContext. Dialects which support sequences should add a *fire_sequence()* method to their execution context implementation. ¶ References: #1566
- **[dialect] [refactor]** Functions and operators generated by the compiler now use (almost) regular dispatch functions of the form "visit_<opname>" and "visit_<funcname>_fn" to provide customed processing. This replaces the need to copy the "functions" and "operators" dictionaries in compiler subclasses with straightforward visitor methods, and also allows compiler subclasses complete control over rendering, as the full _Function or _BinaryExpression object is passed in. ¶
- **[types]** The construction of types within dialects has been totally overhauled. Dialects now define publically available types as UPPERCASE names exclusively, and internal implementation types using underscore identifiers (i.e. are private). The system by which types are expressed in SQL and DDL has been moved to the compiler system. This has the effect that there are much fewer type objects within most dialects. A detailed document on this architecture for dialect authors is in lib/sqlalchemy/dialects/type_migration_guidelines.txt . ¶
- **[types]** Types no longer make any guesses as to default parameters. In particular, Numeric, Float, NUMERIC, FLOAT, DECIMAL don't generate any length or scale unless specified. ¶

- **[types]** types.Binary is renamed to types.LargeBinary, it only produces BLOB, BYTEA, or a similar "long binary" type. New base BINARY and VARBINARY types have been added to access these MySQL/MS-SQL specific types in an agnostic way. ¶ References: #1664
- **[types]** String/Text/Unicode types now skip the unicode() check on each result column value if the dialect has detected the DBAPI as returning Python unicode objects natively. This check is issued on first connect using "SELECT CAST 'some text' AS VARCHAR(10)" or equivalent, then checking if the returned object is a Python unicode. This allows vast performance increases for native-unicode DBAPIs, including pysqlite/sqlite3, psycopg2, and pg8000. ¶
- **[types]** Most types result processors have been checked for possible speed improvements. Specifically, the following generic types have been optimized, resulting in varying speed improvements: Unicode, PickleType, Interval, TypeDecorator, Binary. Also the following dbapi-specific implementations have been improved: Time, Date and DateTime on Sqlite, ARRAY on Postgresql, Time on MySQL, Numeric(as_decimal=False) on MySQL, oursql and pypostgresql, DateTime on cx_oracle and LOB-based types on cx_oracle. ¶
- **[types]** Reflection of types now returns the exact UPPERCASE type within types.py, or the UPPERCASE type within the dialect itself if the type is not a standard SQL type. This means reflection now returns more accurate information about reflected types. ¶
- **[types]** Added a new Enum generic type. Enum is a schema-aware object to support databases which require specific DDL in order to use enum or equivalent; in the case of PG it handles the details of *CREATE TYPE*, and on other databases without native enum support will by generate VARCHAR + an inline CHECK constraint to enforce the enum. ¶ References: #1511, #1109
- **[types]** The Interval type includes a "native" flag which controls if native INTERVAL types (postgresql + oracle) are selected if available, or not. "day_precision" and "second_precision" arguments are also added which propagate as appropriately to these native types. Related to. ¶ References: #1467
- **[types]** The Boolean type, when used on a backend that doesn't have native boolean support, will generate a CHECK constraint "col IN (0, 1)" along with the int/smallint- based column type. This can be switched off if desired with create_constraint=False. Note that MySQL has no native boolean *or* CHECK constraint support so this feature isn't available on that platform. ¶ References: #1589
- **[types]** PickleType now uses == for comparison of values when mutable=True, unless the "comparator" argument with a comparsion function is specified to the type. Objects being pickled will be compared based on identity (which defeats the purpose of mutable=True) if __eq__() is not overridden or a comparison function is not provided. ¶
- **[types]** The default "precision" and "scale" arguments of Numeric and Float have been removed and now default to None. NUMERIC and FLOAT will be rendered with no numeric arguments by default unless these values are provided. ¶
- **[types]** AbstractType.get_search_list() is removed - the games that was used for are no longer necessary. ¶
- **[types]** Added a generic BigInteger type, compiles to BIGINT or NUMBER(19). ¶ References: #1125
- **[types]** sqlsoup has been overhauled to explicitly support an 0.5 style session, using autocommit=False, autoflush=True. Default behavior of SQLSoup now requires the usual usage of commit() and rollback(), which have been added to its interface. An explcit Session or scoped_session can be passed to the constructor, allowing these arguments to be overridden. ¶
- **[types]** sqlsoup db.<sometable>.update() and delete() now call query(cls).update() and delete(), respectively. ¶
- **[types]** sqlsoup now has execute() and connection(), which call upon the Session methods of those names, ensuring that the bind is in terms of the SqlSoup object's bind. ¶
- **[types]** sqlsoup objects no longer have the 'query' attribute - it's not needed for sqlsoup's usage paradigm and it gets in the way of a column that is actually named 'query'. ¶

- **[types]** The signature of the proxy_factory callable passed to association_proxy is now (lazy_collection, creator, value_attr, association_proxy), adding a fourth argument that is the parent AssociationProxy argument. Allows serializability and subclassing of the built in collections. ¶ References: #1259
- **[types]** association_proxy now has basic comparator methods .any(), .has(), .contains(), ==, !=, thanks to Scott Torborg. ¶ References: #1372

6.2.7 0.5 Changelog

0.5.9

no release date

sql

- **[sql]** Fixed erroneous self_group() call in expression package. ¶ References: #1661

0.5.8

Released: Sat Jan 16 2010

sql

- **[sql]** The copy() method on Column now supports uninitialized, unnamed Column objects. This allows easy creation of declarative helpers which place common columns on multiple subclasses. ¶
- **[sql]** Default generators like Sequence() translate correctly across a copy() operation. ¶
- **[sql]** Sequence() and other DefaultGenerator objects are accepted as the value for the "default" and "onupdate" keyword arguments of Column, in addition to being accepted positionally. ¶
- **[sql]** Fixed a column arithmetic bug that affected column correspondence for cloned selectables which contain free-standing column expressions. This bug is generally only noticeable when exercising newer ORM behavior only available in 0.6 via, but is more correct at the SQL expression level as well. ¶ References: #1568, #1617

postgresql

- **[postgresql]** The extract() function, which was slightly improved in 0.5.7, needed a lot more work to generate the correct typecast (the typecasts appear to be necessary in PG's EXTRACT quite a lot of the time). The typecast is now generated using a rule dictionary based on PG's documentation for date/time/interval arithmetic. It also accepts text() constructs again, which was broken in 0.5.7. ¶ References: #1647

firebird

- **[firebird]** Recognize more errors as disconnections. ¶ References: #1646

0.5.7

Released: Sat Dec 26 2009

orm

- **[orm]** contains_eager() now works with the automatically generated subquery that results when you say "query(Parent).join(Parent.somejoinedsubclass)", i.e. when Parent joins to a joined-table-inheritance subclass. Previously contains_eager() would erroneously add the subclass table to the query separately producing a cartesian product. An example is in the ticket description. ¶ References: #1543
- **[orm]** query.options() now only propagate to loaded objects for potential further sub-loads only for options where such behavior is relevant, keeping various unserializable options like those generated by contains_eager() out of individual instance states. ¶ References: #1553
- **[orm]** Session.execute() now locates table- and mapper-specific binds based on a passed in expression which is an insert()/update()/delete() construct. ¶ References: #1054
- **[orm]** Session.merge() now properly overwrites a many-to-one or uselist=False attribute to None if the attribute is also None in the given object to be merged. ¶
- **[orm]** Fixed a needless select which would occur when merging transient objects that contained a null primary key identifier. ¶ References: #1618
- **[orm]** Mutable collection passed to the "extension" attribute of relation(), column_property() etc. will not be mutated or shared among multiple instrumentation calls, preventing duplicate extensions, such as backref populators, from being inserted into the list. ¶ References: #1585
- **[orm]** Fixed the call to get_committed_value() on CompositeProperty. ¶ References: #1504
- **[orm]** Fixed bug where Query would crash if a join() with no clear "left" side were called when a non-mapped column entity appeared in the columns list. ¶ References: #1602
- **[orm]** Fixed bug whereby composite columns wouldn't load properly when configured on a joined-table subclass, introduced in version 0.5.6 as a result of the fix for. thx to Scott Torborg. ¶ References: #1616, #1480
- **[orm]** The "use get" behavior of many-to-one relations, i.e. that a lazy load will fallback to the possibly cached query.get() value, now works across join conditions where the two compared types are not exactly the same class, but share the same "affinity" - i.e. Integer and SmallInteger. Also allows combinations of reflected and non-reflected types to work with 0.5 style type reflection, such as PGText/Text (note 0.6 reflects types as their generic versions). ¶ References: #1556
- **[orm]** Fixed bug in query.update() when passing Cls.attribute as keys in the value dict and using synchronize_session='expire' ('fetch' in 0.6). ¶ References: #1436

sql

- **[sql]** Fixed bug in two-phase transaction whereby commit() method didn't set the full state which allows subsequent close() call to succeed. ¶ References: #1603
- **[sql]** Fixed the "numeric" paramstyle, which apparently is the default paramstyle used by Informixdb. ¶
- **[sql]** Repeat expressions in the columns clause of a select are deduped based on the identity of each clause element, not the actual string. This allows positional elements to render correctly even if they all render identically, such as "qmark" style bind parameters. ¶ References: #1574
- **[sql]** The cursor associated with connection pool connections (i.e. _CursorFairy) now proxies *__iter__()* to the underlying cursor correctly. ¶ References: #1632
- **[sql]** types now support an "affinity comparison" operation, i.e. that an Integer/SmallInteger are "compatible", or a Text/String, PickleType/Binary, etc. Part of. ¶ References: #1556
- **[sql]** Fixed bug preventing alias() of an alias() from being cloned or adapted (occurs frequently in ORM operations). ¶ References: #1641

postgresql

- **[postgresql]** Added support for reflecting the DOUBLE PRECISION type, via a new postgres.PGDoublePrecision object. This is postgresql.DOUBLE_PRECISION in 0.6. ¶ References: #1085
- **[postgresql]** Added support for reflecting the INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND syntaxes of the INTERVAL type. ¶ References: #460
- **[postgresql]** Corrected the "has_sequence" query to take current schema, or explicit sequence-stated schema, into account. ¶ References: #1576
- **[postgresql]** Fixed the behavior of extract() to apply operator precedence rules to the "::" operator when applying the "timestamp" cast - ensures proper parenthesization. ¶ References: #1611

sqlite

- **[sqlite]** sqlite dialect properly generates CREATE INDEX for a table that is in an alternate schema. ¶ References: #1439

mssql

- **[mssql]** Changed the name of TrustedConnection to Trusted_Connection when constructing pyodbc connect arguments ¶ References: #1561

oracle

- **[oracle]** The "table_names" dialect function, used by MetaData .reflect(), omits "index overflow tables", a system table generated by Oracle when "index only tables" with overflow are used. These tables aren't accessible via SQL and can't be reflected. ¶ References: #1637

misc

- **[ext]** A column can be added to a joined-table declarative superclass after the class has been constructed (i.e. via class-level attribute assignment), and the column will be propagated down to subclasses. This is the reverse situation as that of, fixed in 0.5.6. ¶ References: #1570, #1523
- **[ext]** Fixed a slight inaccuracy in the sharding example. Comparing equivalence of columns in the ORM is best accomplished using col1.shares_lineage(col2). ¶ References: #1491
- **[ext]** Removed unused *load()* method from ShardedQuery. ¶ References: #1606

0.5.6

Released: Sat Sep 12 2009

orm

- **[orm]** Fixed bug whereby inheritance discriminator part of a composite primary key would fail on updates. Continuation of. ¶ References: #1300

- **[orm]** Fixed bug which disallowed one side of a many-to-many bidirectional reference to declare itself as "viewonly" ¶ References: #1507
- **[orm]** Added an assertion that prevents a @validates function or other AttributeExtension from loading an unloaded collection such that internal state may be corrupted. ¶ References: #1526
- **[orm]** Fixed bug which prevented two entities from mutually replacing each other's primary key values within a single flush() for some orderings of operations. ¶ References: #1519
- **[orm]** Fixed an obscure issue whereby a joined-table subclass with a self-referential eager load on the base class would populate the related object's "subclass" table with data from the "subclass" table of the parent. ¶ References: #1485
- **[orm]** relations() now have greater ability to be "overridden", meaning a subclass that explicitly specifies a relation() overriding that of the parent class will be honored during a flush. This is currently to support many-to-many relations from concrete inheritance setups. Outside of that use case, YMMV. ¶ References: #1477
- **[orm]** Squeezed a few more unnecessary "lazy loads" out of relation(). When a collection is mutated, many-to-one backrefs on the other side will not fire off to load the "old" value, unless "single_parent=True" is set. A direct assignment of a many-to-one still loads the "old" value in order to update backref collections on that value, which may be present in the session already, thus maintaining the 0.5 behavioral contract. ¶ References: #1483
- **[orm]** Fixed bug whereby a load/refresh of joined table inheritance attributes which were based on column_property() or similar would fail to evaluate. ¶ References: #1480
- **[orm]** Improved support for MapperProperty objects overriding that of an inherited mapper for non-concrete inheritance setups - attribute extensions won't randomly collide with each other. ¶ References: #1488
- **[orm]** UPDATE and DELETE do not support ORDER BY, LIMIT, OFFSET, etc. in standard SQL. Query.update() and Query.delete() now raise an exception if any of limit(), offset(), order_by(), group_by(), or distinct() have been called. ¶ References: #1487
- **[orm]** Added AttributeExtension to sqlalchemy.orm.__all__ ¶
- **[orm]** Improved error message when query() is called with a non-SQL /entity expression. ¶ References: #1476
- **[orm]** Using False or 0 as a polymorphic discriminator now works on the base class as well as a subclass. ¶ References: #1440
- **[orm]** Added enable_assertions(False) to Query which disables the usual assertions for expected state - used by Query subclasses to engineer custom state.. See http://www.sqlalchemy.org/trac/wiki/UsageRecipes/PreFilteredQuery for an example. ¶ References: #1424
- **[orm]** Fixed recursion issue which occurred if a mapped object's *__len__()* or *__nonzero__()* method resulted in state changes. ¶ References: #1501
- **[orm]** Fixed incorrect exception raise in Weak/StrongIdentityMap.add() ¶ References: #1506
- **[orm]** Fixed the error message for "could not find a FROM clause" in query.join() which would fail to issue correctly if the query was against a pure SQL construct. ¶ References: #1522
- **[orm]** Fixed a somewhat hypothetical issue which would result in the wrong primary key being calculated for a mapper using the old polymorphic_union function - but this is old stuff. ¶ References: #1486

sql

- **[sql]** Fixed column.copy() to copy defaults and onupdates. ¶ References: #1373
- **[sql]** Fixed a bug in extract() introduced in 0.5.4 whereby the string "field" argument was getting treated as a ClauseElement, causing various errors within more complex SQL transformations. ¶

- **[sql]** Unary expressions such as DISTINCT propagate their type handling to result sets, allowing conversions like unicode and such to take place. ¶ References: #1420
- **[sql]** Fixed bug in Table and Column whereby passing empty dict for "info" argument would raise an exception. ¶ References: #1482

oracle

- **[oracle]** Backported 0.6 fix for Oracle alias names not getting truncated. ¶ References: #1309

misc

- **[ext]** The collection proxies produced by associationproxy are now pickleable. A user-defined proxy_factory however is still not pickleable unless it defines __getstate__ and __setstate__. ¶ References: #1446
- **[ext]** Declarative will raise an informative exception if __table_args__ is passed as a tuple with no dict argument. Improved documentation. ¶ References: #1468
- **[ext]** Table objects declared in the MetaData can now be used in string expressions sent to primaryjoin/secondaryjoin/ secondary - the name is pulled from the MetaData of the declarative base. ¶ References: #1527
- **[ext]** A column can be added to a joined-table subclass after the class has been constructed (i.e. via class-level attribute assignment). The column is added to the underlying Table as always, but now the mapper will rebuild its "join" to include the new column, instead of raising an error about "no such column, use column_property() instead". ¶ References: #1523
- **[test]** Added examples into the test suite so they get exercised regularly and cleaned up a couple deprecation warnings. ¶

0.5.5

Released: Mon Jul 13 2009

general

- **[general]** unit tests have been migrated from unittest to nose. See README.unittests for information on how to run the tests. ¶ References: #970

orm

- **[orm]** The "foreign_keys" argument of relation() will now propagate automatically to the backref in the same way that primaryjoin and secondaryjoin do. For the extremely rare use case where the backref of a relation() has intentionally different "foreign_keys" configured, both sides now need to be configured explicitly (if they do in fact require this setting, see the next note...). ¶
- **[orm]** ...the only known (and really, really rare) use case where a different foreign_keys setting was used on the forwards/backwards side, a composite foreign key that partially points to its own columns, has been enhanced such that the fk->itself aspect of the relation won't be used to determine relation direction. ¶
- **[orm]** Session.mapper is now *deprecated*.

Call session.add() if you'd like a free-standing object to be part of your session. Otherwise, a DIY version of Session.mapper is now documented at http://www.sqlalchemy.org/trac/wiki/UsageRecipes/SessionAwareMapper The method will remain deprecated throughout 0.6. ¶

- **[orm]** Fixed Query being able to join() from individual columns of a joined-table subclass entity, i.e. query(SubClass.foo, SubcClass.bar).join(<anything>). In most cases, an error "Could not find a FROM clause to join from" would be raised. In a few others, the result would be returned in terms of the base class rather than the subclass - so applications which relied on this erroneous result need to be adjusted. ¶ References: #1431
- **[orm]** Fixed a bug involving contains_eager(), which would apply itself to a secondary (i.e. lazy) load in a particular rare case, producing cartesian products. improved the targeting of query.options() on secondary loads overall. ¶ References: #1461
- **[orm]** Fixed bug introduced in 0.5.4 whereby Composite types fail when default-holding columns are flushed. ¶
- **[orm]** Fixed another 0.5.4 bug whereby mutable attributes (i.e. PickleType) wouldn't be deserialized correctly when the whole object was serialized. ¶ References: #1426
- **[orm]** Fixed bug whereby session.is_modified() would raise an exception if any synonyms were in use. ¶
- **[orm]** Fixed potential memory leak whereby previously pickled objects placed back in a session would not be fully garbage collected unless the Session were explicitly closed out. ¶
- **[orm]** Fixed bug whereby list-based attributes, like pickletype and PGArray, failed to be merged() properly. ¶
- **[orm]** Repaired non-working attributes.set_committed_value function. ¶
- **[orm]** Trimmed the pickle format for InstanceState which should further reduce the memory footprint of pickled instances. The format should be backwards compatible with that of 0.5.4 and previous. ¶
- **[orm]** sqlalchemy.orm.join and sqlalchemy.orm.outerjoin are now added to __all__ in sqlalchemy.orm.*. ¶ References: #1463
- **[orm]** Fixed bug where Query exception raise would fail when a too-short composite primary key value were passed to get(). ¶ References: #1458

sql

- **[sql]** Removed an obscure feature of execute() (including connection, engine, Session) whereby a bindparam() construct can be sent as a key to the params dictionary. This usage is undocumented and is at the core of an issue whereby the bindparam() object created implicitly by a text() construct may have the same hash value as a string placed in the params dictionary and may result in an inappropriate match when computing the final bind parameters. Internal checks for this condition would add significant latency to the critical task of parameter rendering, so the behavior is removed. This is a backwards incompatible change for any application that may have been using this feature, however the feature has never been documented. ¶

misc

- **[engine/pool]** Implemented recreate() for StaticPool. ¶

0.5.4p2

Released: Tue May 26 2009

sql

- **[sql]** Repaired the printing of SQL exceptions which are not based on parameters or are not executemany() style. ¶

postgresql

- **[postgresql]** Deprecated the hardcoded TIMESTAMP function, which when used as func.TIMESTAMP(value) would render "TIMESTAMP value". This breaks on some platforms as PostgreSQL doesn't allow bind parameters to be used in this context. The hard-coded uppercase is also inappropriate and there's lots of other PG casts that we'd need to support. So instead, use text constructs i.e. select(["timestamp '12/05/09'"]). ¶

0.5.4p1

Released: Mon May 18 2009

orm

- **[orm]** Fixed an attribute error introduced in 0.5.4 which would occur when merge() was used with an incomplete object. ¶

0.5.4

Released: Sun May 17 2009

orm

- **[orm]** Significant performance enhancements regarding Sessions/flush() in conjunction with large mapper graphs, large numbers of objects:
  - Removed all* O(N) scanning behavior from the flush() process, i.e. operations that were scanning the full session, including an extremely expensive one that was erroneously assuming primary key values were changing when this was not the case.
    - one edge case remains which may invoke a full scan, if an existing primary key attribute is modified to a new value.
  - The Session's "weak referencing" behavior is now *full* - no strong references whatsoever are made to a mapped object or related items/collections in its __dict__. Backrefs and other cycles in objects no longer affect the Session's ability to lose all references to unmodified objects. Objects with pending changes still are maintained strongly until flush.

    The implementation also improves performance by moving the "resurrection" process of garbage collected items to only be relevant for mappings that map "mutable" attributes (i.e. PickleType, composite attrs). This removes overhead from the gc process and simplifies internal behavior.

    If a "mutable" attribute change is the sole change on an object which is then dereferenced, the mapper will not have access to other attribute state when the UPDATE is issued. This may present itself differently to some MapperExtensions.

    The change also affects the internal attribute API, but not the AttributeExtension interface nor any of the publically documented attribute functions.

  - The unit of work no longer genererates a graph of "dependency" processors for the full graph of mappers during flush(), instead creating such processors only for those mappers which represent objects with pending changes. This saves a tremendous number of method calls in the context of a large interconnected graph of mappers.
  - Cached a wasteful "table sort" operation that previously occurred multiple times per flush, also removing significant method call count from flush().
  - Other redundant behaviors have been simplified in mapper._save_obj().

  ¶ References: #1398

- **[orm]** Modified query_cls on DynamicAttributeImpl to accept a full mixin version of the AppenderQuery, which allows subclassing the AppenderMixin. ¶
- **[orm]** The "polymorphic discriminator" column may be part of a primary key, and it will be populated with the correct discriminator value. ¶ References: #1300
- **[orm]** Fixed the evaluator not being able to evaluate IS NULL clauses. ¶
- **[orm]** Fixed the "set collection" function on "dynamic" relations to initiate events correctly. Previously a collection could only be assigned to a pending parent instance, otherwise modified events would not be fired correctly. Set collection is now compatible with merge(), fixes. ¶ References: #1352
- **[orm]** Allowed pickling of PropertyOption objects constructed with instrumented descriptors; previously, pickle errors would occur when pickling an object which was loaded with a descriptor-based option, such as query.options(eagerload(MyClass.foo)). ¶
- **[orm]** Lazy loader will not use get() if the "lazy load" SQL clause matches the clause used by get(), but contains some parameters hardcoded. Previously the lazy strategy would fail with the get(). Ideally get() would be used with the hardcoded parameters but this would require further development. ¶ References: #1357
- **[orm]** MapperOptions and other state associated with query.options() is no longer bundled within callables associated with each lazy/deferred-loading attribute during a load. The options are now associated with the instance's state object just once when it's populated. This removes the need in most cases for per-instance/attribute loader objects, improving load speed and memory overhead for individual instances. ¶ References: #1391
- **[orm]** Fixed another location where autoflush was interfering with session.merge(). autoflush is disabled completely for the duration of merge() now. ¶ References: #1360
- **[orm]** Fixed bug which prevented "mutable primary key" dependency logic from functioning properly on a one-to-one relation(). ¶ References: #1406
- **[orm]** Fixed bug in relation(), introduced in 0.5.3, whereby a self referential relation from a base class to a joined-table subclass would not configure correctly. ¶
- **[orm]** Fixed obscure mapper compilation issue when inheriting mappers are used which would result in uninitialized attributes. ¶
- **[orm]** Fixed documentation for session weak_identity_map - the default value is True, indicating a weak referencing map in use. ¶
- **[orm]** Fixed a unit of work issue whereby the foreign key attribute on an item contained within a collection owned by an object being deleted would not be set to None if the relation() was self-referential. ¶ References: #1376
- **[orm]** Fixed Query.update() and Query.delete() failures with eagerloaded relations. ¶ References: #1378
- **[orm]** It is now an error to specify both columns of a binary primaryjoin condition in the foreign_keys or remote_side collection. Whereas previously it was just nonsensical, but would succeed in a non-deterministic way. ¶

sql

- **[sql]** Back-ported the "compiler" extension from SQLA 0.6. This is a standardized interface which allows the creation of custom ClauseElement subclasses and compilers. In particular it's handy as an alternative to text() when you'd like to build a construct that has database-specific compilations. See the extension docs for details. ¶
- **[sql]** Exception messages are truncated when the list of bound parameters is larger than 10, preventing enormous multi-page exceptions from filling up screens and logfiles for large executemany() statements. ¶ References: #1413
- **[sql]** `sqlalchemy.extract()` is now dialect sensitive and can extract components of timestamps idiomatically across the supported databases, including SQLite. ¶
- **[sql]** Fixed __repr__() and other _get_colspec() methods on ForeignKey constructed from __clause_element__() style construct (i.e. declarative columns). ¶ References: #1353

schema

- **[schema] [1341] [ticket: 594]** Added a quote_schema() method to the IdentifierPreparer class so that dialects can override how schemas get handled. This enables the MSSQL dialect to treat schemas as multipart identifiers, such as 'database.owner'. ¶

mysql

- **[mysql]** Reflecting a FOREIGN KEY construct will take into account a dotted schema.tablename combination, if the foreign key references a table in a remote schema. ¶ References: #1405

sqlite

- **[sqlite]** Corrected the SLBoolean type so that it properly treats only 1 as True. ¶ References: #1402
- **[sqlite]** Corrected the float type so that it correctly maps to a SLFloat type when being reflected. ¶ References: #1273

mssql

- **[mssql]** Modified how savepoint logic works to prevent it from stepping on non-savepoint oriented routines. Savepoint support is still very experimental. ¶
- **[mssql]** Added in reserved words for MSSQL that covers version 2008 and all prior versions. ¶ References: #1310
- **[mssql]** Corrected problem with information schema not working with a binary collation based database. Cleaned up information schema since it is only used by mssql now. ¶ References: #1343

misc

- **[extensions]** Fixed adding of deferred or other column properties to a declarative class. ¶ References: #1379

0.5.3

Released: Tue Mar 24 2009

orm

- **[orm]** The "objects" argument to session.flush() is deprecated. State which represents the linkage between a parent and child object does not support "flushed" status on one side of the link and not the other, so supporting this operation leads to misleading results. ¶ References: #1315
- **[orm]** Query now implements __clause_element__() which produces its selectable, which means a Query instance can be accepted in many SQL expressions, including col.in_(query), union(query1, query2), select([foo]).select_from(query), etc. ¶
- **[orm]** Query.join() can now construct multiple FROM clauses, if needed. Such as, query(A, B).join(A.x).join(B.y) might say SELECT A.*, B.* FROM A JOIN X, B JOIN Y. Eager loading can also tack its joins onto those multiple FROM clauses. ¶ References: #1337
- **[orm]** Fixed bug in dynamic_loader() where append/remove events after construction time were not being propagated to the UOW to pick up on flush(). ¶ References: #1347
- **[orm]** Fixed bug where column_prefix wasn't being checked before not mapping an attribute that already had class-level name present. ¶
- **[orm]** a session.expire() on a particular collection attribute will clear any pending backref additions as well, so that the next access correctly returns only what was present in the database. Presents some degree of a workaround for, although we are considering removing the flush([objects]) feature altogether. ¶ References: #1315
- **[orm]** Session.scalar() now converts raw SQL strings to text() the same way Session.execute() does and accepts same alternative **kw args. ¶
- **[orm]** improvements to the "determine direction" logic of relation() such that the direction of tricky situations like mapper(A.join(B)) -> relation-> mapper(B) can be determined. ¶
- **[orm]** When flushing partial sets of objects using session.flush([somelist]), pending objects which remain pending after the operation won't inadvertently be added as persistent. ¶ References: #1306
- **[orm]** Added "post_configure_attribute" method to InstrumentationManager, so that the "listen_for_events.py" example works again. ¶ References: #1314
- **[orm]** a forward and complementing backwards reference which are both of the same direction, i.e. ONETOMANY or MANYTOONE, is now detected, and an error message is raised. Saves crazy CircularDependencyErrors later on. ¶
- **[orm]** Fixed bugs in Query regarding simultaneous selection of multiple joined-table inheritance entities with common base classes:
  - previously the adaption applied to "B" on "A JOIN B" would be erroneously partially applied to "A".
  - comparisons on relations (i.e. A.related==someb) were not getting adapted when they should.
  - Other filterings, like query(A).join(A.bs).filter(B.foo=='bar'), were erroneously adapting "B.foo" as though it were an "A".

  ¶
- **[orm]** Fixed adaptation of EXISTS clauses via any(), has(), etc. in conjunction with an aliased object on the left and of_type() on the right. ¶ References: #1325

- **[orm]** Added an attribute helper method `set_committed_value` in sqlalchemy.orm.attributes. Given an object, attribute name, and value, will set the value on the object as part of its "committed" state, i.e. state that is understood to have been loaded from the database. Helps with the creation of homegrown collection loaders and such. ¶
- **[orm]** Query won't fail with weakref error when a non-mapper/class instrumented descriptor is passed, raises "Invalid column expression". ¶
- **[orm]** Query.group_by() properly takes into account aliasing applied to the FROM clause, such as with select_from(), using with_polymorphic(), or using from_self(). ¶

sql

- **[sql]** An alias() of a select() will convert to a "scalar subquery" when used in an unambiguously scalar context, i.e. it's used in a comparison operation. This applies to the ORM when using query.subquery() as well. ¶
- **[sql]** Fixed missing _label attribute on Function object, others when used in a select() with use_labels (such as when used in an ORM column_property()). ¶ References: #1302
- **[sql]** anonymous alias names now truncate down to the max length allowed by the dialect. More significant on DBs like Oracle with very small character limits. ¶ References: #1309
- **[sql]** the __selectable__() interface has been replaced entirely by __clause_element__(). ¶
- **[sql]** The per-dialect cache used by TypeEngine to cache dialect-specific types is now a WeakKeyDictionary. This to prevent dialect objects from being referenced forever for an application that creates an arbitrarily large number of engines or dialects. There is a small performance penalty which will be resolved in 0.6. ¶ References: #1299

postgresql

- **[postgresql]** Index reflection won't fail when an index with multiple expressions is encountered. ¶
- **[postgresql]** Added PGUuid and PGBit types to sqlalchemy.databases.postgres. ¶ References: #1327
- **[postgresql]** Refection of unknown PG types won't crash when those types are specified within a domain. ¶ References: #1327

sqlite

- **[sqlite]** Fixed SQLite reflection methods so that non-present cursor.description, which triggers an auto-cursor close, will be detected so that no results doesn't fail on recent versions of pysqlite which raise an error when fetchone() called with no rows present. ¶

mssql

- **[mssql]** Preliminary support for pymssql 1.0.1 ¶
- **[mssql]** Corrected issue on mssql where max_identifier_length was not being respected. ¶

misc

- **[extensions]** Fixed a recursive pickling issue in serializer, triggered by an EXISTS or other embedded FROM construct. ¶

- **[extensions]** Declarative locates the "inherits" class using a search through __bases__, to skip over mixins that are local to subclasses. ¶
- **[extensions]** Declarative figures out joined-table inheritance primary join condition even if "inherits" mapper argument is given explicitly. ¶
- **[extensions]** Declarative will properly interpret the "foreign_keys" argument on a backref() if it's a string. ¶
- **[extensions]** Declarative will accept a table-bound column as a property when used in conjunction with __table__, if the column is already present in __table__. The column will be remapped to the given key the same way as when added to the mapper() properties dict. ¶

0.5.2

Released: Sat Jan 24 2009

orm

- **[orm]** Further refined 0.5.1's warning about delete-orphan cascade placed on a many-to-many relation. First, the bad news: the warning will apply to both many-to-many as well as many-to-one relations. This is necessary since in both cases, SQLA does not scan the full set of potential parents when determining "orphan" status - for a persistent object it only detects an in-python de-association event to establish the object as an "orphan". Next, the good news: to support one-to-one via a foreign key or association table, or to support one-to-many via an association table, a new flag single_parent=True may be set which indicates objects linked to the relation are only meant to have a single parent. The relation will raise an error if multiple parent-association events occur within Python. ¶
- **[orm]** Adjusted the attribute instrumentation change from 0.5.1 to fully establish instrumentation for subclasses where the mapper was created after the superclass had already been fully instrumented. ¶ References: #1292
- **[orm]** Fixed bug in delete-orphan cascade whereby two one-to-one relations from two different parent classes to the same target class would prematurely expunge the instance. ¶
- **[orm]** Fixed an eager loading bug whereby self-referential eager loading would prevent other eager loads, self referential or not, from joining to the parent JOIN properly. Thanks to Alex K for creating a great test case. ¶
- **[orm]** session.expire() and related methods will not expire() unloaded deferred attributes. This prevents them from being needlessly loaded when the instance is refreshed. ¶
- **[orm]** query.join()/outerjoin() will now properly join an aliased() construct to the existing left side, even if query.from_self() or query.select_from(someselectable) has been called. ¶ References: #1293

sql

- **[sql]**

  Further fixes to the "percent signs and spaces in column/table names" functionality.

  ¶ References: #1284

mssql

- **[mssql]** Restored convert_unicode handling. Results were being passed on through without conversion. ¶ References: #1291
- **[mssql]** Really fixing the decimal handling this time.. ¶ References: #1282

- **[mssql] [Ticket:1289]** Modified table reflection code to use only kwargs when constructing tables. ¶

0.5.1

Released: Sat Jan 17 2009

orm

- **[orm]** Removed an internal join cache which could potentially leak memory when issuing query.join() repeatedly to ad-hoc selectables. ¶
- **[orm]** The "clear()", "save()", "update()", "save_or_update()" Session methods have been deprecated, replaced by "expunge_all()" and "add()". "expunge_all()" has also been added to ScopedSession. ¶
- **[orm]** Modernized the "no mapped table" exception and added a more explicit __table__/__tablename__ exception to declarative. ¶
- **[orm]** Concrete inheriting mappers now instrument attributes which are inherited from the superclass, but are not defined for the concrete mapper itself, with an InstrumentedAttribute that issues a descriptive error when accessed. ¶ References: #1237
- **[orm]** Added a new *relation()* keyword *back_populates*. This allows configuation of backreferences using explicit relations. This is required when creating bidirectional relations between a hierarchy of concrete mappers and another class. ¶ References: #1237, #781
- **[orm]** Test coverage added for *relation()* objects specified on concrete mappers. ¶ References: #1237
- **[orm]** Query.from_self() as well as query.subquery() both disable the rendering of eager joins inside the subquery produced. The "disable all eager joins" feature is available publically via a new query.enable_eagerloads() generative. ¶ References: #1276
- **[orm]** Added a rudimental series of set operations to Query that receive Query objects as arguments, including union(), union_all(), intersect(), except_(), insertsect_all(), except_all(). See the API documentation for Query.union() for examples. ¶
- **[orm]** Fixed bug that prevented Query.join() and eagerloads from attaching to a query that selected from a union or aliased union. ¶
- **[orm]** A short documentation example added for bidirectional relations specified on concrete mappers. ¶ References: #1237
- **[orm]** Mappers now instrument class attributes upon construction with the final InstrumentedAttribute object which remains persistent. The *_CompileOnAttr/__getattribute__()* methodology has been removed. The net effect is that Column-based mapped class attributes can now be used fully at the class level without invoking a mapper compilation operation, greatly simplifying typical usage patterns within declarative. ¶ References: #1269
- **[orm]** ColumnProperty (and front-end helpers such as `deferred`) no longer ignores unknown **keyword arguments. ¶
- **[orm]** Fixed a bug with the unitofwork's "row switch" mechanism, i.e. the conversion of INSERT/DELETE into an UPDATE, when combined with joined-table inheritance and an object which contained no defined values for the child table where an UPDATE with no SET clause would be rendered. ¶
- **[orm]** Using delete-orphan on a many-to-many relation is deprecated. This produces misleading or erroneous results since SQLA does not retrieve the full list of "parents" for m2m. To get delete-orphan behavior with an m2m table, use an explcit association class so that the individual association row is treated as a parent. ¶ References: #1281

- **[orm]** delete-orphan cascade always requires delete cascade. Specifying delete-orphan without delete now raises a deprecation warning. ¶ References: #1281

sql

- **[sql]** Improved the methodology to handling percent signs in column names from. Added more tests. MySQL and PostgreSQL dialects still do not issue correct CREATE TABLE statements for identifiers with percent signs in them. ¶ References: #1256

schema

- **[schema]** Index now accepts column-oriented InstrumentedAttributes (i.e. column-based mapped class attributes) as column arguments. ¶ References: #1214
- **[schema]** Column with no name (as in declarative) won't raise a NoneType error when its string output is requested (such as in a stack trace). ¶
- **[schema]** Fixed bug when overriding a Column with a ForeignKey on a reflected table, where derived columns (i.e. the "virtual" columns of a select, etc.) would inadvertently call upon schema-level cleanup logic intended only for the original column. ¶ References: #1278

mysql

- **[mysql]** Added the missing keywords from MySQL 4.1 so they get escaped properly. ¶

mssql

- **[mssql]** Corrected handling of large decimal values with more robust tests. Removed string manipulation on floats. ¶ References: #1280
- **[mssql]** Modified the do_begin handling in mssql to use the Cursor not the Connection so it is DBAPI compatible. ¶
- **[mssql]** Corrected SAVEPOINT support on adodbapi by changing the handling of savepoint_release, which is unsupported on mssql. ¶

misc

- **[declarative]** Can now specify Column objects on subclasses which have no table of their own (i.e. use single table inheritance). The columns will be appended to the base table, but only mapped by the subclass. ¶
- **[declarative]** For both joined and single inheriting subclasses, the subclass will only map those columns which are already mapped on the superclass and those explicit on the subclass. Other columns that are present on the *Table* will be excluded from the mapping by default, which can be disabled by passing a blank *exclude_properties* collection to the *__mapper_args__*. This is so that single-inheriting classes which define their own columns are the only classes to map those columns. The effect is actually a more organized mapping than you'd normally get with explicit *mapper()* calls unless you set up the *exclude_properties* arguments explicitly. ¶
- **[declarative]** It's an error to add new Column objects to a declarative class that specified an existing table using __table__. ¶

0.5.0

Released: Tue Jan 06 2009

general

- **[general]** Documentation has been converted to Sphinx. In particular, the generated API documentation has been constructed into a full blown "API Reference" section which organizes editorial documentation combined with generated docstrings. Cross linking between sections and API docs are vastly improved, a javascript-powered search feature is provided, and a full index of all classes, functions and members is provided. ¶
- **[general]** setup.py now imports setuptools only optionally. If not present, distutils is used. The new "pip" installer is recommended over easy_install as it installs in a more simplified way. ¶
- **[general]** added an extremely basic illustration of a PostGIS integration to the examples folder. ¶

orm

- **[orm]** Query.with_polymorphic() now accepts a third argument "discriminator" which will replace the value of mapper.polymorphic_on for that query. Mappers themselves no longer require polymorphic_on to be set, even if the mapper has a polymorphic_identity. When not set, the mapper will load non-polymorphically by default. Together, these two features allow a non-polymorphic concrete inheritance setup to use polymorphic loading on a per-query basis, since concrete setups are prone to many issues when used polymorphically in all cases. ¶
- **[orm]** dynamic_loader accepts a query_class= to customize the Query classes used for both the dynamic collection and the queries built from it. ¶
- **[orm]** query.order_by() accepts None which will remove any pending order_by state from the query, as well as cancel out any mapper/relation configured ordering. This is primarily useful for overriding the ordering specified on a dynamic_loader(). ¶ References: #1079
- **[orm]** Exceptions raised during compile_mappers() are now preserved to provide "sticky behavior" - if a hasattr() call on a pre-compiled mapped attribute triggers a failing compile and suppresses the exception, subsequent compilation is blocked and the exception will be reiterated on the next compile() call. This issue occurs frequently when using declarative. ¶
- **[orm]** property.of_type() is now recognized on a single-table inheriting target, when used in the context of prop.of_type(..).any()/has(), as well as query.join(prop.of_type(...)). ¶
- **[orm]** query.join() raises an error when the target of the join doesn't match the property-based attribute - while it's unlikely anyone is doing this, the SQLAlchemy author was guilty of this particular loosey-goosey behavior. ¶
- **[orm]** Fixed bug when using weak_instance_map=False where modified events would not be intercepted for a flush(). ¶ References: #1272
- **[orm]** Fixed some deep "column correspondence" issues which could impact a Query made against a selectable containing multiple versions of the same table, as well as unions and similar which contained the same table columns in different column positions at different levels. ¶ References: #1268
- **[orm]** Custom comparator classes used in conjunction with column_property(), relation() etc. can define new comparison methods on the Comparator, which will become available via __getattr__() on the InstrumentedAttribute. In the case of synonym() or comparable_property(), attributes are resolved first on the user-defined descriptor, then on the user-defined comparator. ¶
- **[orm]** Added ScopedSession.is_active accessor. ¶ References: #976
- **[orm]** Can pass mapped attributes and column objects as keys to query.update({}). ¶ References: #1262

- **[orm]** Mapped attributes passed to the values() of an expression level insert() or update() will use the keys of the mapped columns, not that of the mapped attribute. ¶
- **[orm]** Corrected problem with Query.delete() and Query.update() not working properly with bind parameters. ¶ References: #1242
- **[orm]** Query.select_from(), from_statement() ensure that the given argument is a FromClause, or Text/Select/Union, respectively. ¶
- **[orm]** Query() can be passed a "composite" attribute as a column expression and it will be expanded. Somewhat related to. ¶ References: #1253
- **[orm]** Query() is a little more robust when passed various column expressions such as strings, clauselists, text() constructs (which may mean it just raises an error more nicely). ¶
- **[orm]** first() works as expected with Query.from_statement(). ¶
- **[orm]** Fixed bug introduced in 0.5rc4 involving eager loading not functioning for properties which were added to a mapper post-compile using add_property() or equivalent. ¶
- **[orm]** Fixed bug where many-to-many relation() with viewonly=True would not correctly reference the link between secondary->remote. ¶
- **[orm]** Duplicate items in a list-based collection will be maintained when issuing INSERTs to a "secondary" table in a many-to-many relation. Assuming the m2m table has a unique or primary key constraint on it, this will raise the expected constraint violation instead of silently dropping the duplicate entries. Note that the old behavior remains for a one-to-many relation since collection entries in that case don't result in INSERT statements and SQLA doesn't manually police collections. ¶ References: #1232
- **[orm]** Query.add_column() can accept FromClause objects in the same manner as session.query() can. ¶
- **[orm]** Comparison of many-to-one relation to NULL is properly converted to IS NOT NULL based on not_(). ¶
- **[orm]** Extra checks added to ensure explicit primaryjoin/secondaryjoin are ClauseElement instances, to prevent more confusing errors later on. ¶ References: #1087
- **[orm]** Improved mapper() check for non-class classes. ¶ References: #1236
- **[orm]** comparator_factory argument is now documented and supported by all MapperProperty types, including column_property(), relation(), backref(), and synonym(). ¶ References: #5051
- **[orm]** Changed the name of PropertyLoader to RelationProperty, to be consistent with all the other names. PropertyLoader is still present as a synonym. ¶
- **[orm]** fixed "double iter()" call causing bus errors in shard API, removed errant result.close() left over from the 0.4 version. ¶ References: #1099, #1228
- **[orm]** made Session.merge cascades not trigger autoflush. Fixes merged instances getting prematurely inserted with missing values. ¶
- **[orm]** Two fixes to help prevent out-of-band columns from being rendered in polymorphic_union inheritance scenarios (which then causes extra tables to be rendered in the FROM clause causing cartesian products):
  - improvements to "column adaption" for a->b->c inheritance situations to better locate columns that are related to one another via multiple levels of indirection, rather than rendering the non-adapted column.
  - the "polymorphic discriminator" column is only rendered for the actual mapper being queried against. The column won't be "pulled in" from a subclass or superclass mapper since it's not needed.

  ¶
- **[orm]** Fixed shard_id argument on ShardedSession.execute(). ¶ References: #1072

sql

- **[sql]** RowProxy objects can be used in place of dictionary arguments sent to connection.execute() and friends. ¶ References: #935
- **[sql]** Columns can again contain percent signs within their names. ¶ References: #1256
- **[sql]** sqlalchemy.sql.expression.Function is now a public class. It can be subclassed to provide user-defined SQL functions in an imperative style, including with pre-established behaviors. The postgis.py example illustrates one usage of this. ¶
- **[sql]** PickleType now favors == comparison by default, if the incoming object (such as a dict) implements __eq__(). If the object does not implement __eq__() and mutable=True, a deprecation warning is raised. ¶
- **[sql]** Fixed the import weirdness in sqlalchemy.sql to not export __names__. ¶ References: #1215
- **[sql]** Using the same ForeignKey object repeatedly raises an error instead of silently failing later. ¶ References: #1238
- **[sql]** Added NotImplementedError for params() method on Insert/Update/Delete constructs. These items currently don't support this functionality, which also would be a little misleading compared to values(). ¶
- **[sql]** Reflected foreign keys will properly locate their referenced column, even if the column was given a "key" attribute different from the reflected name. This is achieved via a new flag on ForeignKey/ForeignKeyConstraint called "link_to_name", if True means the given name is the referred-to column's name, not its assigned key. ¶ References: #650
- **[sql]** select() can accept a ClauseList as a column in the same way as a Table or other selectable and the interior expressions will be used as column elements. ¶ References: #1253
- **[sql]** the "passive" flag on session.is_modified() is correctly propagated to the attribute manager. ¶
- **[sql]** union() and union_all() will not whack any order_by() that has been applied to the select()s inside. If you union() a select() with order_by() (presumably to support LIMIT/OFFSET), you should also call self_group() on it to apply parenthesis. ¶

mysql

- **[mysql]** "%" signs in text() constructs are automatically escaped to "%%". Because of the backwards incompatible nature of this change, a warning is emitted if '%%' is detected in the string. ¶
- **[mysql]** Fixed bug in exception raise when FK columns not present during reflection. ¶ References: #1241
- **[mysql]** Fixed bug involving reflection of a remote-schema table with a foreign key ref to another table in that schema. ¶

sqlite

- **[sqlite]** Table reflection now stores the actual DefaultClause value for the column. ¶ References: #1266
- **[sqlite]** bugfixes, behavioral changes ¶

mssql

- **[mssql]** Added in a new MSGenericBinary type. This maps to the Binary type so it can implement the specialized behavior of treating length specified types as fixed-width Binary types and non-length types as an unbound variable length Binary type. ¶

- **[mssql]** Added in new types: MSVarBinary and MSImage. ¶ References: #1249
- **[mssql]** Added in the MSReal, MSNText, MSSmallDateTime, MSTime, MSDateTimeOffset, and MSDateTime2 types ¶
- **[mssql]** Refactored the Date/Time types. The `smalldatetime` data type no longer truncates to a date only, and will now be mapped to the MSSmallDateTime type. ¶ References: #1254
- **[mssql]** Corrected an issue with Numerics to accept an int. ¶
- **[mssql]** Mapped `char_length` to the `LEN()` function. ¶
- **[mssql]** If an `INSERT` includes a subselect the `INSERT` is converted from an `INSERT INTO VALUES` construct to a `INSERT INTO SELECT` construct. ¶
- **[mssql]** If the column is part of a `primary_key` it will be `NOT NULL` since MSSQL doesn't allow `NULL` in primary_key columns. ¶
- **[mssql]** `MSBinary` now returns a `BINARY` instead of an `IMAGE`. This is a backwards incompatible change in that `BINARY` is a fixed length data type whereas `IMAGE` is a variable length data type. ¶ References: #1249
- **[mssql]** `get_default_schema_name` is now reflected from the database based on the user's default schema. This only works with MSSQL 2005 and later. ¶ References: #1258
- **[mssql]** Added collation support through the use of a new collation argument. This is supported on the following types: char, nchar, varchar, nvarchar, text, ntext. ¶ References: #1248
- **[mssql]** Changes to the connection string parameters favor DSN as the default specification for pyodbc. See the mssql.py docstring for detailed usage instructions. ¶
- **[mssql]** Added experimental support of savepoints. It currently does not work fully with sessions. ¶
- **[mssql]** Support for three levels of column nullability: NULL, NOT NULL, and the database's configured default. The default Column configuration (nullable=True) will now generate NULL in the DDL. Previously no specification was emitted and the database default would take effect (usually NULL, but not always). To explicitly request the database default, configure columns with nullable=None and no specification will be emitted in DDL. This is backwards incompatible behavior. ¶ References: #1243

oracle

- **[oracle]** Adjusted the format of create_xid() to repair two-phase commit. We now have field reports of Oracle two-phase commit working properly with this change. ¶
- **[oracle]** Added OracleNVarchar type, produces NVARCHAR2, and also subclasses Unicode so that convert_unicode=True by default. NVARCHAR2 reflects into this type automatically so these columns pass unicode on a reflected table with no explicit convert_unicode=True flags. ¶ References: #1233
- **[oracle]** Fixed bug which was preventing out params of certain types from being received; thanks a ton to huddlej at wwu.edu ! ¶ References: #1265

misc

- **[dialect]** Added a new description_encoding attribute on the dialect that is used for encoding the column name when processing the metadata. This usually defaults to utf-8. ¶
- **[engine/pool]** Connection.invalidate() checks for closed status to avoid attribute errors. ¶ References: #1246
- **[engine/pool]** NullPool supports reconnect on failure behavior. ¶ References: #1094
- **[engine/pool]** Added a mutex for the initial pool creation when using pool.manage(dbapi). This prevents a minor case of "dogpile" behavior which would otherwise occur upon a heavy load startup. ¶ References: #799

- **[engine/pool]** _execute_clauseelement() goes back to being a private method. Subclassing Connection is not needed now that ConnectionProxy is available. ¶
- **[documentation]** Tickets. ¶ References: #1149, #1200
- **[documentation]** Added note about create_session() defaults. ¶
- **[documentation]** Added section about metadata.reflect(). ¶
- **[documentation]** Updated *TypeDecorator* section. ¶
- **[documentation]** Rewrote the "threadlocal" strategy section of the docs due to recent confusion over this feature. ¶
- **[documentation]** Removed badly out of date 'polymorphic_fetch' and 'select_table' docs from inheritance, reworked the second half of "joined table inheritance". ¶
- **[documentation]** Documented *comparator_factory* kwarg, added new doc section "Custom Comparators". ¶
- **[postgres]** "%" signs in text() constructs are automatically escaped to "%%". Because of the backwards incompatible nature of this change, a warning is emitted if '%%' is detected in the string. ¶ References: #1267
- **[postgres]** Calling alias.execute() in conjunction with server_side_cursors won't raise AttributeError. ¶
- **[postgres]** Added Index reflection support to PostgreSQL, using a great patch we long neglected, submitted by Ken Kuhlman. ¶ References: #714
- **[associationproxy]** The association proxy properties are make themselves available at the class level, e.g. MyClass.aproxy. Previously this evaluated to None. ¶
- **[declarative]** The full list of arguments accepted as string by backref() includes 'primaryjoin', 'secondaryjoin', 'secondary', 'foreign_keys', 'remote_side', 'order_by'. ¶

0.5.0rc4

Released: Fri Nov 14 2008

general

- **[general]** global "propigate"->"propagate" change. ¶

orm

- **[orm]** Query.count() has been enhanced to do the "right thing" in a wider variety of cases. It can now count multiple-entity queries, as well as column-based queries. Note that this means if you say query(A, B).count() without any joining criterion, it's going to count the cartesian product of A*B. Any query which is against column-based entities will automatically issue "SELECT count(1) FROM (SELECT...)" so that the real row-count is returned, meaning a query such as query(func.count(A.name)).count() will return a value of one, since that query would return one row. ¶
- **[orm]** Lots of performance tuning. A rough guesstimate over various ORM operations places it 10% faster over 0.5.0rc3, 25-30% over 0.4.8. ¶
- **[orm]** bugfixes and behavioral changes ¶
- **[orm]** Adjustments to the enhanced garbage collection on InstanceState to better guard against errors due to lost state. ¶

- **[orm]** Query.get() returns a more informative error message when executed against multiple entities. ¶ References: #1220
- **[orm]** Restored NotImplementedError on Cls.relation.in_() ¶ References: #1140, #1221
- **[orm]** Fixed PendingDeprecationWarning involving order_by parameter on relation(). ¶ References: #1226

sql

- **[sql]** Removed the 'properties' attribute of the Connection object, Connection.info should be used. ¶
- **[sql]** Restored "active rowcount" fetch before ResultProxy autocloses the cursor. This was removed in 0.5rc3. ¶
- **[sql]** Rearranged the *load_dialect_impl()* method in *TypeDecorator* such that it will take effect even if the user-defined *TypeDecorator* uses another *TypeDecorator* as its impl. ¶

mssql

- **[mssql]** Lots of cleanup and fixes to correct problems with limit and offset. ¶
- **[mssql]** Correct situation where subqueries as part of a binary expression need to be translated to use the IN and NOT IN syntax. ¶
- **[mssql]** Fixed E Notation issue that prevented the ability to insert decimal values less than 1E-6. ¶ References: #1216
- **[mssql]** Corrected problems with reflection when dealing with schemas, particularly when those schemas are the default schema. ¶ References: #1217
- **[mssql]** Corrected problem with casting a zero length item to a varchar. It now correctly adjusts the CAST. ¶

misc

- **[access]** Added support for Currency type. ¶
- **[access]** Functions were not return their result. ¶ References: #1017
- **[access]** Corrected problem with joins. Access only support LEFT OUTER or INNER not just JOIN by itself. ¶ References: #1017
- **[ext]** Can now use a custom "inherit_condition" in __mapper_args__ when using declarative. ¶
- **[ext]** fixed string-based "remote_side", "order_by" and others not propagating correctly when used in backref(). ¶

0.5.0rc3

Released: Fri Nov 07 2008

orm

- **[orm]** Added two new hooks to SessionExtension: after_bulk_delete() and after_bulk_update(). after_bulk_delete() is called after a bulk delete() operation on a query. after_bulk_update() is called after a bulk update() operation on a query. ¶

- **[orm]** "not equals" comparisons of simple many-to-one relation to an instance will not drop into an EXISTS clause and will compare foreign key columns instead. ¶
- **[orm]** Removed not-really-working use cases of comparing a collection to an iterable. Use contains() to test for collection membership. ¶
- **[orm]** Improved the behavior of aliased() objects such that they more accurately adapt the expressions generated, which helps particularly with self-referential comparisons. ¶ References: #1171
- **[orm]** Fixed bug involving primaryjoin/secondaryjoin conditions constructed from class-bound attributes (as often occurs when using declarative), which later would be inappropriately aliased by Query, particularly with the various EXISTS based comparators. ¶
- **[orm]** Fixed bug when using multiple query.join() with an aliased-bound descriptor which would lose the left alias. ¶
- **[orm]** Improved weakref identity map memory management to no longer require mutexing, resurrects garbage collected instance on a lazy basis for an InstanceState with pending changes. ¶
- **[orm]** InstanceState object now removes circular references to itself upon disposal to keep it outside of cyclic garbage collection. ¶
- **[orm]** relation() won't hide unrelated ForeignKey errors inside of the "please specify primaryjoin" message when determining join condition. ¶
- **[orm]** Fixed bug in Query involving order_by() in conjunction with multiple aliases of the same class (will add tests in) ¶ References: #1218
- **[orm]** When using Query.join() with an explicit clause for the ON clause, the clause will be aliased in terms of the left side of the join, allowing scenarios like query(Source). from_self().join((Dest, Source.id==Dest.source_id)) to work properly. ¶
- **[orm]** polymorphic_union() function respects the "key" of each Column if they differ from the column's name. ¶
- **[orm]** Repaired support for "passive-deletes" on a many-to-one relation() with "delete" cascade. ¶ References: #1183
- **[orm]** Fixed bug in composite types which prevented a primary-key composite type from being mutated. ¶ References: #1213
- **[orm]** Added more granularity to internal attribute access, such that cascade and flush operations will not initialize unloaded attributes and collections, leaving them intact for a lazy-load later on. Backref events still initialize attrbutes and collections for pending instances. ¶ References: #1202

sql

- **[sql]** SQL compiler optimizations and complexity reduction. The call count for compiling a typical select() construct is 20% less versus 0.5.0rc2. ¶
- **[sql]** Dialects can now generate label names of adjustable length. Pass in the argument "label_length=<value>" to create_engine() to adjust how many characters max will be present in dynamically generated column labels, i.e. "somecolumn AS somelabel". Any value less than 6 will result in a label of minimal size, consisting of an underscore and a numeric counter. The compiler uses the value of dialect.max_identifier_length as a default. ¶ References: #1211
- **[sql]** Simplified the check for ResultProxy "autoclose without results" to be based solely on presence of cursor.description. All the regexp-based guessing about statements returning rows has been removed. ¶ References: #1212
- **[sql]** Direct execution of a union() construct will properly set up result-row processing. ¶ References: #1194

- **[sql]** The internal notion of an "OID" or "ROWID" column has been removed. It's basically not used by any dialect, and the possibility of its usage with psycopg2's cursor.lastrowid is basically gone now that INSERT..RETURNING is available. ¶
- **[sql]** Removed "default_order_by()" method on all FromClause objects. ¶
- **[sql]** Repaired the table.tometadata() method so that a passed-in schema argument is propagated to ForeignKey constructs. ¶
- **[sql]** Slightly changed behavior of IN operator for comparing to empty collections. Now results in inequality comparison against self. More portable, but breaks with stored procedures that aren't pure functions. ¶

mysql

- **[mysql]** Fixed foreign key reflection in the edge case where a Table's explicit schema= is the same as the schema (database) the connection is attached to. ¶
- **[mysql]** No longer expects include_columns in table reflection to be lower case. ¶

oracle

- **[oracle]** Wrote a docstring for Oracle dialect. Apparently that Ohloh "few source code comments" label is starting to sting :). ¶
- **[oracle]** Removed FIRST_ROWS() optimize flag when using LIMIT/OFFSET, can be reenabled with optimize_limits=True create_engine() flag. ¶ References: #536
- **[oracle]** bugfixes and behavioral changes ¶
- **[oracle]** Setting the auto_convert_lobs to False on create_engine() will also instruct the OracleBinary type to return the cx_oracle LOB object unchanged. ¶

misc

- **[ext]** Added a new extension sqlalchemy.ext.serializer. Provides Serializer/Deserializer "classes" which mirror Pickle/Unpickle, as well as dumps() and loads(). This serializer implements an "external object" pickler which keeps key context-sensitive objects, including engines, sessions, metadata, Tables/Columns, and mappers, outside of the pickle stream, and can later restore the pickle using any engine/metadata/session provider. This is used not for pickling regular object instances, which are pickleable without any special logic, but for pickling expression objects and full Query objects, such that all mapper/engine/session dependencies can be restored at unpickle time. ¶
- **[ext]** Fixed bug preventing declarative-bound "column" objects from being used in column_mapped_collection(). ¶ References: #1174
- **[misc]** util.flatten_iterator() func doesn't interpret strings with __iter__() methods as iterators, such as in pypy. ¶ References: #1077

0.5.0rc2

Released: Sun Oct 12 2008

orm

- **[orm]** Fixed bug involving read/write relation()s that contain literal or other non-column expressions within their primaryjoin condition equated to a foreign key column. ¶
- **[orm]** "non-batch" mode in mapper(), a feature which allows mapper extension methods to be called as each instance is updated/inserted, now honors the insert order of the objects given. ¶
- **[orm]** Fixed RLock-related bug in mapper which could deadlock upon reentrant mapper compile() calls, something that occurs when using declarative constructs inside of ForeignKey objects. ¶
- **[orm]** ScopedSession.query_property now accepts a query_cls factory, overriding the session's configured query_cls. ¶
- **[orm]** Fixed shared state bug interfering with ScopedSession.mapper's ability to apply default __init__ implementations on object subclasses. ¶
- **[orm]** Fixed up slices on Query (i.e. query[x:y]) to work properly for zero length slices, slices with None on either end. ¶ References: #1177
- **[orm]** Added an example illustrating Celko's "nested sets" as a SQLA mapping. ¶
- **[orm]** contains_eager() with an alias argument works even when the alias is embedded in a SELECT, as when sent to the Query via query.select_from(). ¶
- **[orm]** contains_eager() usage is now compatible with a Query that also contains a regular eager load and limit/offset, in that the columns are added to the Query-generated subquery. ¶ References: #1180
- **[orm]** session.execute() will execute a Sequence object passed to it (regression from 0.4). ¶
- **[orm]** Removed the "raiseerror" keyword argument from object_mapper() and class_mapper(). These functions raise in all cases if the given class/instance is not mapped. ¶
- **[orm]** Fixed session.transaction.commit() on a autocommit=False session not starting a new transaction. ¶
- **[orm]** Some adjustments to Session.identity_map's weak referencing behavior to reduce asynchronous GC side effects. ¶
- **[orm]** Adjustment to Session's post-flush accounting of newly "clean" objects to better protect against operating on objects as they're asynchronously gc'ed. ¶ References: #1182

sql

- **[sql]** column.in_(someselect) can now be used as a columns-clause expression without the subquery bleeding into the FROM clause ¶ References: #1074

mysql

- **[mysql]** Temporary tables are now reflectable. ¶

sqlite

- **[sqlite]** Overhauled SQLite date/time bind/result processing to use regular expressions and format strings, rather than strptime/strftime, to generically support pre-1900 dates, dates with microseconds. ¶ References: #968
- **[sqlite]** String's (and Unicode's, UnicodeText's, etc.) convert_unicode logic disabled in the sqlite dialect, to adjust for pysqlite 2.5.0's new requirement that only Python unicode objects are accepted; http://itsystementwicklung.de/pipermail/list-pysqlite/2008-March/000018.html ¶

oracle

- **[oracle]** Oracle will detect string-based statements which contain comments at the front before a SELECT as SELECT statements. ¶ References: #1187

0.5.0rc1

Released: Thu Sep 11 2008

orm

- **[orm]** Query now has delete() and update(values) methods. This allows to perform bulk deletes/updates with the Query object. ¶
- **[orm]** The RowTuple object returned by Query(*cols) now features keynames which prefer mapped attribute names over column keys, column keys over column names, i.e. Query(Class.foo, Class.bar) will have names "foo" and "bar" even if those are not the names of the underlying Column objects. Direct Column objects such as Query(table.c.col) will return the "key" attribute of the Column. ¶
- **[orm]** Added scalar() and value() methods to Query, each return a single scalar value. scalar() takes no arguments and is roughly equivalent to first()[0], value() takes a single column expression and is roughly equivalent to values(expr).next()[0]. ¶
- **[orm]** Improved the determination of the FROM clause when placing SQL expressions in the query() list of entities. In particular scalar subqueries should not "leak" their inner FROM objects out into the enclosing query. ¶
- **[orm]** Joins along a relation() from a mapped class to a mapped subclass, where the mapped subclass is configured with single table inheritance, will include an IN clause which limits the subtypes of the joined class to those requested, within the ON clause of the join. This takes effect for eager load joins as well as query.join(). Note that in some scenarios the IN clause will appear in the WHERE clause of the query as well since this discrimination has multiple trigger points. ¶
- **[orm]** AttributeExtension has been refined such that the event is fired before the mutation actually occurs. Additionally, the append() and set() methods must now return the given value, which is used as the value to be used in the mutation operation. This allows creation of validating AttributeListeners which raise before the action actually occurs, and which can change the given value into something else before its used. ¶
- **[orm]** column_property(), composite_property(), and relation() now accept a single or list of AttributeExtensions using the "extension" keyword argument. ¶
- **[orm]** query.order_by().get() silently drops the "ORDER BY" from the query issued by GET but does not raise an exception. ¶
- **[orm]** Added a Validator AttributeExtension, as well as a @validates decorator which is used in a similar fashion as @reconstructor, and marks a method as validating one or more mapped attributes. ¶
- **[orm]** class.someprop.in_() raises NotImplementedError pending the implementation of "in_" for relation ¶ References: #1140
- **[orm]** Fixed primary key update for many-to-many collections where the collection had not been loaded yet ¶ References: #1127
- **[orm]** Fixed bug whereby deferred() columns with a group in conjunction with an otherwise unrelated synonym() would produce an AttributeError during deferred load. ¶

- **[orm]** The before_flush() hook on SessionExtension takes place before the list of new/dirty/deleted is calculated for the final time, allowing routines within before_flush() to further change the state of the Session before the flush proceeds. ¶ References: #1128
- **[orm]** The "extension" argument to Session and others can now optionally be a list, supporting events sent to multiple SessionExtension instances. Session places SessionExtensions in Session.extensions. ¶
- **[orm]** Reentrant calls to flush() raise an error. This also serves as a rudimentary, but not foolproof, check against concurrent calls to Session.flush(). ¶
- **[orm]** Improved the behavior of query.join() when joining to joined-table inheritance subclasses, using explicit join criteria (i.e. not on a relation). ¶
- **[orm]** @orm.attributes.reconstitute and MapperExtension.reconstitute have been renamed to @orm.reconstructor and MapperExtension.reconstruct_instance ¶
- **[orm]** Fixed @reconstructor hook for subclasses which inherit from a base class. ¶ References: #1129
- **[orm]** The composite() property type now supports a __set_composite_values__() method on the composite class which is required if the class represents state using attribute names other than the column's keynames; default-generated values now get populated properly upon flush. Also, composites with attributes set to None compare correctly. ¶ References: #1132
- **[orm]** The 3-tuple of iterables returned by attributes.get_history() may now be a mix of lists and tuples. (Previously members were always lists.) ¶
- **[orm]** Fixed bug whereby changing a primary key attribute on an entity where the attribute's previous value had been expired would produce an error upon flush(). ¶ References: #1151
- **[orm]** Fixed custom instrumentation bug whereby get_instance_dict() was not called for newly constructed instances not loaded by the ORM. ¶
- **[orm]** Session.delete() adds the given object to the session if not already present. This was a regression bug from 0.4. ¶ References: #1150
- **[orm]** The *echo_uow* flag on *Session* is deprecated, and unit-of-work logging is now application-level only, not per-session level. ¶
- **[orm]** Removed conflicting *contains()* operator from *InstrumentedAttribute* which didn't accept *escape* kwaarg. ¶ References: #1153

sql

- **[sql]** Temporarily rolled back the "ORDER BY" enhancement from. This feature is on hold pending further development. ¶ References: #1068
- **[sql]** The exists() construct won't "export" its contained list of elements as FROM clauses, allowing them to be used more effectively in the columns clause of a SELECT. ¶
- **[sql]** and_() and or_() now generate a ColumnElement, allowing boolean expressions as result columns, i.e. select([and_(1, 0)]). ¶ References: #798
- **[sql]** Bind params now subclass ColumnElement which allows them to be selectable by orm.query (they already had most ColumnElement semantics). ¶
- **[sql]** Added select_from() method to exists() construct, which becomes more and more compatible with a regular select(). ¶
- **[sql]** Added func.min(), func.max(), func.sum() as "generic functions", which basically allows for their return type to be determined automatically. Helps with dates on SQLite, decimal types, others. ¶ References: #1160

- **[sql]** added decimal.Decimal as an "auto-detect" type; bind parameters and generic functions will set their type to Numeric when a Decimal is used. ¶

schema

- **[schema]** Added "sorted_tables" accessor to MetaData, which returns Table objects sorted in order of dependency as a list. This deprecates the MetaData.table_iterator() method. The "reverse=False" keyword argument has also been removed from util.sort_tables(); use the Python 'reversed' function to reverse the results. ¶ References: #1033
- **[schema]** The 'length' argument to all Numeric types has been renamed to 'scale'. 'length' is deprecated and is still accepted with a warning. ¶
- **[schema]** Dropped 0.3-compatibility for user defined types (convert_result_value, convert_bind_param). ¶

mysql

- **[mysql]** The 'length' argument to MSInteger, MSBigInteger, MSTinyInteger, MSSmallInteger and MSYear has been renamed to 'display_width'. ¶
- **[mysql]** Added MSMediumInteger type. ¶ References: #1146
- **[mysql]** the function func.utc_timestamp() compiles to UTC_TIMESTAMP, without the parenthesis, which seem to get in the way when using in conjunction with executemany(). ¶

oracle

- **[oracle]** limit/offset no longer uses ROW NUMBER OVER to limit rows, and instead uses subqueries in conjunction with a special Oracle optimization comment. Allows LIMIT/OFFSET to work in conjunction with DISTINCT. ¶ References: #536
- **[oracle]** has_sequence() now takes the current "schema" argument into account ¶ References: #1155
- **[oracle]** added BFILE to reflected type names ¶ References: #1121

misc

- **[declarative]** Fixed bug whereby mapper couldn't initialize if a composite primary key referenced another table that was not defined yet. ¶ References: #1161
- **[declarative]** Fixed exception throw which would occur when string-based primaryjoin condition was used in conjunction with backref. ¶

0.5.0beta3

Released: Mon Aug 04 2008

orm

- **[orm]** The "entity_name" feature of SQLAlchemy mappers has been removed. For rationale, see http://tinyurl.com/6nm2ne ¶

- **[orm]** the "autoexpire" flag on Session, sessionmaker(), and scoped_session() has been renamed to "expire_on_commit". It does not affect the expiration behavior of rollback(). ¶
- **[orm]** fixed endless loop bug which could occur within a mapper's deferred load of inherited attributes. ¶
- **[orm]** a legacy-support flag "_enable_transaction_accounting" flag added to Session which when False, disables all transaction-level object accounting, including expire on rollback, expire on commit, new/deleted list maintenance, and autoflush on begin. ¶
- **[orm]** The 'cascade' parameter to relation() accepts None as a value, which is equivalent to no cascades. ¶
- **[orm]** A critical fix to dynamic relations allows the "modified" history to be properly cleared after a flush(). ¶
- **[orm]** user-defined @properties on a class are detected and left in place during mapper initialization. This means that a table-bound column of the same name will not be mapped at all if a @property is in the way (and the column is not remapped to a different name), nor will an instrumented attribute from an inherited class be applied. The same rules apply for names excluded using the include_properties/exclude_properties collections. ¶
- **[orm]** Added a new SessionExtension hook called after_attach(). This is called at the point of attachment for objects via add(), add_all(), delete(), and merge(). ¶
- **[orm]** A mapper which inherits from another, when inheriting the columns of its inherited mapper, will use any reassigned property names specified in that inheriting mapper. Previously, if "Base" had reassigned "base_id" to the name "id", "SubBase(Base)" would still get an attribute called "base_id". This could be worked around by explicitly stating the column in each submapper as well but this is fairly unworkable and also impossible when using declarative. ¶ References: #1111
- **[orm]** Fixed a series of potential race conditions in Session whereby asynchronous GC could remove unmodified, no longer referenced items from the session as they were present in a list of items to be processed, typically during session.expunge_all() and dependent methods. ¶
- **[orm]** Some improvements to the _CompileOnAttr mechanism which should reduce the probability of "Attribute x was not replaced during compile" warnings. (this generally applies to SQLA hackers, like Elixir devs). ¶
- **[orm]** Fixed bug whereby the "unsaved, pending instance" FlushError raised for a pending orphan would not take superclass mappers into account when generating the list of relations responsible for the error. ¶

sql

- **[sql]** func.count() with no arguments renders as COUNT(*), equivalent to func.count(text('*')). ¶
- **[sql]** simple label names in ORDER BY expressions render as themselves, and not as a re-statement of their corresponding expression. This feature is currently enabled only for SQLite, MySQL, and PostgreSQL. It can be enabled on other dialects as each is shown to support this behavior. ¶ References: #1068

mysql

- **[mysql]** Quoting of MSEnum values for use in CREATE TABLE is now optional & will be quoted on demand as required. (Quoting was always optional for use with existing tables.) ¶ References: #1110

misc

- **[ext]** Class-bound attributes sent as arguments to relation()'s remote_side and foreign_keys parameters are now accepted, allowing them to be used with declarative. Additionally fixed bugs involving order_by being specified as a class-bound attribute in conjunction with eager loading. ¶

- **[ext]** declarative initialization of Columns adjusted so that non-renamed columns initialize in the same way as a non declarative mapper. This allows an inheriting mapper to set up its same-named "id" columns in particular such that the parent "id" column is favored over the child column, reducing database round trips when this value is requested. ¶

0.5.0beta2

Released: Mon Jul 14 2008

orm

- **[orm]** In addition to expired attributes, deferred attributes also load if their data is present in the result set. ¶ References: #870
- **[orm]** session.refresh() raises an informative error message if the list of attributes does not include any column-based attributes. ¶
- **[orm]** query() raises an informative error message if no columns or mappers are specified. ¶
- **[orm]** lazy loaders now trigger autoflush before proceeding. This allows expire() of a collection or scalar relation to function properly in the context of autoflush. ¶
- **[orm]** column_property() attributes which represent SQL expressions or columns that are not present in the mapped tables (such as those from views) are automatically expired after an INSERT or UPDATE, assuming they have not been locally modified, so that they are refreshed with the most recent data upon access. ¶ References: #887
- **[orm]** Fixed explicit, self-referential joins between two joined-table inheritance mappers when using query.join(cls, aliased=True). ¶ References: #1082
- **[orm]** Fixed query.join() when used in conjunction with a columns-only clause and an SQL-expression ON clause in the join. ¶
- **[orm]** The "allow_column_override" flag from mapper() has been removed. This flag is virtually always misunderstood. Its specific functionality is available via the include_properties/exclude_properties mapper arguments. ¶
- **[orm]** Repaired *__str__()* method on Query. ¶ References: #1066
- **[orm]** Session.bind gets used as a default even when table/mapper specific binds are defined. ¶

sql

- **[sql]** Added new match() operator that performs a full-text search. Supported on PostgreSQL, SQLite, MySQL, MS-SQL, and Oracle backends. ¶

schema

- **[schema]** Added prefixes option to *Table* that accepts a list of strings to insert after CREATE in the CREATE TABLE statement. ¶ References: #1075
- **[schema]** Unicode, UnicodeText types now set "assert_unicode" and "convert_unicode" by default, but accept overriding **kwargs for these values. ¶

sqlite

- **[sqlite]** Modified SQLite's representation of "microseconds" to match the output of str(somedatetime), i.e. in that the microseconds are represented as fractional seconds in string format. This makes SQLA's SQLite date type compatible with datetimes that were saved directly using Pysqlite (which just calls str()). Note that this is incompatible with the existing microseconds values in a SQLA 0.4 generated SQLite database file.

  To get the old behavior globally:

  > from sqlalchemy.databases.sqlite import DateTimeMixin DateTimeMixin.__legacy_microseconds__ = True

  To get the behavior on individual DateTime types:

  > t = sqlite.SLDateTime() t.__legacy_microseconds__ = True

  Then use "t" as the type on the Column. ¶ References: #1090

- **[sqlite]** SQLite Date, DateTime, and Time types only accept Python datetime objects now, not strings. If you'd like to format dates as strings yourself with SQLite, use a String type. If you'd like them to return datetime objects anyway despite their accepting strings as input, make a TypeDecorator around String - SQLA doesn't encourage this pattern. ¶

misc

- **[extensions]** Declarative supports a __table_args__ class variable, which is either a dictionary, or tuple of the form (arg1, arg2, ..., {kwarg1:value, ...}) which contains positional + kw arguments to be passed to the Table constructor. ¶ References: #1096

0.5.0beta1

Released: Thu Jun 12 2008

general

- **[general]** global "propigate"->"propagate" change. ¶

orm

- **[orm]** polymorphic_union() function respects the "key" of each Column if they differ from the column's name. ¶
- **[orm]** Fixed 0.4-only bug preventing composite columns from working properly with inheriting mappers ¶ References: #1199
- **[orm]** Fixed RLock-related bug in mapper which could deadlock upon reentrant mapper compile() calls, something that occurs when using declarative constructs inside of ForeignKey objects. Ported from 0.5. ¶
- **[orm]** Fixed bug in composite types which prevented a primary-key composite type from being mutated. ¶ References: #1213
- **[orm]** Added ScopedSession.is_active accessor. ¶ References: #976
- **[orm]** Class-bound accessor can be used as the argument to relation() order_by. ¶ References: #939
- **[orm]** Fixed shard_id argument on ShardedSession.execute(). ¶ References: #1072

sql

- **[sql]** Connection.invalidate() checks for closed status to avoid attribute errors. ¶ References: #1246
- **[sql]** NullPool supports reconnect on failure behavior. ¶ References: #1094
- **[sql]** The per-dialect cache used by TypeEngine to cache dialect-specific types is now a WeakKeyDictionary. This to prevent dialect objects from being referenced forever for an application that creates an arbitrarily large number of engines or dialects. There is a small performance penalty which will be resolved in 0.6. ¶ References: #1299
- **[sql]** Fixed SQLite reflection methods so that non-present cursor.description, which triggers an auto-cursor close, will be detected so that no results doesn't fail on recent versions of pysqlite which raise an error when fetchone() called with no rows present. ¶

mysql

- **[mysql]** Fixed bug in exception raise when FK columns not present during reflection. ¶ References: #1241

oracle

- **[oracle]** Fixed bug which was preventing out params of certain types from being received; thanks a ton to huddlej at wwu.edu ! ¶ References: #1265

firebird

- **[firebird]** Added support for returning values from inserts (2.0+ only), updates and deletes (2.1+ only). ¶

misc

- The "__init__" trigger/decorator added by mapper now attempts to exactly mirror the argument signature of the original __init__. The pass-through for '_sa_session' is no longer implicit- you must allow for this keyword argument in your constructor. ¶
- ClassState is renamed to ClassManager. ¶
- Classes may supply their own InstrumentationManager by providing a __sa_instrumentation_manager__ property. ¶
- Custom instrumentation may use any mechanism to associate a ClassManager with a class and an InstanceState with an instance. Attributes on those objects are still the default association mechanism used by SQLAlchemy's native instrumentation. ¶
- Moved entity_name, _sa_session_id, and _instance_key from the instance object to the instance state. These values are still available in the old way, which is now deprecated, using descriptors attached to the class. A deprecation warning will be issued when accessed. ¶
- The _prepare_instrumentation alias for prepare_instrumentation has been removed. ¶
- sqlalchemy.exceptions has been renamed to sqlalchemy.exc. The module may be imported under either name. ¶
- ORM-related exceptions are now defined in sqlalchemy.orm.exc. ConcurrentModificationError, FlushError, and UnmappedColumnError compatibility aliases are installed in sqlalchemy.exc during the import of sqlalchemy.orm. ¶

- sqlalchemy.logging has been renamed to sqlalchemy.log. ¶
- The transitional sqlalchemy.log.SADeprecationWarning alias for the warning's definition in sqlalchemy.exc has been removed. ¶
- exc.AssertionError has been removed and usage replaced with Python's built-in AssertionError. ¶
- The behavior of MapperExtensions attached to multiple, entity_name= primary mappers for a single class has been altered. The first mapper() defined for a class is the only mapper eligible for the MapperExtension 'instrument_class', 'init_instance' and 'init_failed' events. This is backwards incompatible; previously the extensions of last mapper defined would receive these events. ¶
- **[postgres]** Added Index reflection support to Postgres, using a great patch we long neglected, submitted by Ken Kuhlman. ¶ References: #714

6.2.8 0.4 Changelog

0.4.8

Released: Sun Oct 12 2008

orm

- **[orm]** Fixed bug regarding inherit_condition passed with "A=B" versus "B=A" leading to errors ¶ References: #1039
- **[orm]** Changes made to new, dirty and deleted collections in SessionExtension.before_flush() will take effect for that flush. ¶
- **[orm]** Added label() method to InstrumentedAttribute to establish forwards compatibility with 0.5. ¶

sql

- **[sql]** column.in_(someselect) can now be used as a columns-clause expression without the subquery bleeding into the FROM clause ¶ References: #1074

mysql

- **[mysql]** Added MSMediumInteger type. ¶ References: #1146

sqlite

- **[sqlite]** Supplied a custom strftime() function which handles dates before 1900. ¶ References: #968
- **[sqlite]** String's (and Unicode's, UnicodeText's, etc.) convert_unicode logic disabled in the sqlite dialect, to adjust for pysqlite 2.5.0's new requirement that only Python unicode objects are accepted; http://itsystementwicklung.de/pipermail/list-pysqlite/2008-March/000018.html ¶

oracle

- **[oracle]** has_sequence() now takes schema name into account ¶ References: #1155
- **[oracle]** added BFILE to the list of reflected types ¶ References: #1121

0.4.7p1

Released: Thu Jul 31 2008

orm

- **[orm]** Added "add()" and "add_all()" to scoped_session methods. Workaround for 0.4.7:

```
from sqlalchemy.orm.scoping import ScopedSession, instrument
setattr(ScopedSession, "add", instrument("add"))
setattr(ScopedSession, "add_all", instrument("add_all"))
```

  ¶
- **[orm]** Fixed non-2.3 compatible usage of set() and generator expression within relation(). ¶

0.4.7

Released: Sat Jul 26 2008

orm

- **[orm]** The contains() operator when used with many-to-many will alias() the secondary (association) table so that multiple contains() calls will not conflict with each other ¶ References: #1058
- **[orm]** fixed bug preventing merge() from functioning in conjunction with a comparable_property() ¶
- **[orm]** the enable_typechecks=False setting on relation() now only allows subtypes with inheriting mappers. Totally unrelated types, or subtypes not set up with mapper inheritance against the target mapper are still not allowed. ¶
- **[orm]** Added is_active flag to Sessions to detect when a transaction is in progress. This flag is always True with a "transactional" (in 0.5 a non-"autocommit") Session. ¶ References: #976

sql

- **[sql]** Fixed bug when calling select([literal('foo')]) or select([bindparam('foo')]). ¶

schema

- **[schema]** create_all(), drop_all(), create(), drop() all raise an error if the table name or schema name contains more characters than that dialect's configured character limit. Some DB's can handle too-long table names during usage, and SQLA can handle this as well. But various reflection/ checkfirst-during-create scenarios fail since we are looking for the name within the DB's catalog tables. ¶ References: #571

- **[schema]** The index name generated when you say "index=True" on a Column is truncated to the length appropriate for the dialect. Additionally, an Index with a too- long name cannot be explicitly dropped with Index.drop(), similar to. ¶ References: #571, #820

mysql

- **[mysql]** Added 'CALL' to the list of SQL keywords which return result rows. ¶

oracle

- **[oracle]** Oracle get_default_schema_name() "normalizes" the name before returning, meaning it returns a lower-case name when the identifier is detected as case insensitive. ¶
- **[oracle]** creating/dropping tables takes schema name into account when searching for the existing table, so that tables in other owner namespaces with the same name do not conflict ¶ References: #709
- **[oracle]** Cursors now have "arraysize" set to 50 by default on them, the value of which is configurable using the "arraysize" argument to create_engine() with the Oracle dialect. This to account for cx_oracle's default setting of "1", which has the effect of many round trips being sent to Oracle. This actually works well in conjunction with BLOB/CLOB-bound cursors, of which there are any number available but only for the life of that row request (so BufferedColumnRow is still needed, but less so). ¶ References: #1062
- **[oracle]**

  sqlite

  - add SLFloat type, which matches the SQLite REAL type affinity. Previously, only SLNumeric was provided which fulfills NUMERIC affinity, but that's not the same as REAL.

  ¶

misc

- **[postgres]** Repaired server_side_cursors to properly detect text() clauses. ¶
- **[postgres]** Added PGCidr type. ¶ References: #1092

0.4.6

Released: Sat May 10 2008

orm

- **[orm]** Fix to the recent relation() refactoring which fixes exotic viewonly relations which join between local and remote table multiple times, with a common column shared between the joins. ¶
- **[orm]** Also re-established viewonly relation() configurations that join across multiple tables. ¶
- **[orm]** Added experimental relation() flag to help with primaryjoins across functions, etc., _local_remote_pairs=[tuples]. This complements a complex primaryjoin condition allowing you to provide the individual column pairs which comprise the relation's local and remote sides. Also improved lazy load SQL generation to handle placing bind params inside of functions and other expressions. (partial progress towards) ¶ References: #610

- **[orm]** repaired single table inheritance such that you can single-table inherit from a joined-table inherting mapper without issue. ¶ References: #1036
- **[orm]** Fixed "concatenate tuple" bug which could occur with Query.order_by() if clause adaption had taken place. ¶ References: #1027
- **[orm]** Removed ancient assertion that mapped selectables require "alias names" - the mapper creates its own alias now if none is present. Though in this case you need to use the class, not the mapped selectable, as the source of column attributes - so a warning is still issued. ¶
- **[orm]** fixes to the "exists" function involving inheritance (any(), has(), ~contains()); the full target join will be rendered into the EXISTS clause for relations that link to subclasses. ¶
- **[orm]** restored usage of append_result() extension method for primary query rows, when the extension is present and only a single- entity result is being returned. ¶
- **[orm]** Also re-established viewonly relation() configurations that join across multiple tables. ¶
- **[orm]** removed ancient assertion that mapped selectables require "alias names" - the mapper creates its own alias now if none is present. Though in this case you need to use the class, not the mapped selectable, as the source of column attributes - so a warning is still issued. ¶
- **[orm]** refined mapper._save_obj() which was unnecessarily calling __ne__() on scalar values during flush ¶ References: #1015
- **[orm]** added a feature to eager loading whereby subqueries set as column_property() with explicit label names (which is not necessary, btw) will have the label anonymized when the instance is part of the eager join, to prevent conflicts with a subquery or column of the same name on the parent object. ¶ References: #1019
- **[orm]** set-based collections |=, -=, ^= and &= are stricter about their operands and only operate on sets, frozensets or subclasses of the collection type. Previously, they would accept any duck-typed set. ¶
- **[orm]** added an example dynamic_dict/dynamic_dict.py, illustrating a simple way to place dictionary behavior on top of a dynamic_loader. ¶

sql

- **[sql]** Added COLLATE support via the .collate(<collation>) expression operator and collate(<expr>, <collation>) sql function. ¶
- **[sql]** Fixed bug with union() when applied to non-Table connected select statements ¶
- **[sql]** improved behavior of text() expressions when used as FROM clauses, such as select().select_from(text("sometext")) ¶ References: #1014
- **[sql]** Column.copy() respects the value of "autoincrement", fixes usage with Migrate ¶ References: #1021

mssql

- **[mssql]** Added "odbc_autotranslate" parameter to engine / dburi parameters. Any given string will be passed through to the ODBC connection string as:

  "AutoTranslate=%s" % odbc_autotranslate

  ¶ References: #1005
- **[mssql]** Added "odbc_options" parameter to engine / dburi parameters. The given string is simply appended to the SQLAlchemy-generated odbc connection string.

  This should obviate the need of adding a myriad of ODBC options in the future. ¶

firebird

- **[firebird]** Handle the "SUBSTRING(:string FROM :start FOR :length)" builtin. ¶

misc

- **[declarative] [extension]** Joined table inheritance mappers use a slightly relaxed function to create the "inherit condition" to the parent table, so that other foreign keys to not-yet-declared Table objects don't trigger an error. ¶
- **[declarative] [extension]** fixed reentrant mapper compile hang when a declared attribute is used within ForeignKey, ie. ForeignKey(MyOtherClass.someattribute) ¶
- **[engines]** Pool listeners can now be provided as a dictionary of callables or a (possibly partial) duck-type of PoolListener, your choice. ¶
- **[engines]** added "rollback_returned" option to Pool which will disable the rollback() issued when connections are returned. This flag is only safe to use with a database which does not support transactions (i.e. MySQL/MyISAM). ¶
- **[ext]** set-based association proxies |=, -=, ^= and &= are stricter about their operands and only operate on sets, frozensets or other association proxies. Previously, they would accept any duck-typed set. ¶

0.4.5

Released: Fri Apr 04 2008

orm

- **[orm]** A small change in behavior to session.merge() - existing objects are checked for based on primary key attributes, not necessarily _instance_key. So the widely requested capability, that:

  x = MyObject(id=1) x = sess.merge(x)

  will in fact load MyObject with id #1 from the database if present, is now available. merge() still copies the state of the given object to the persistent one, so an example like the above would typically have copied "None" from all attributes of "x" onto the persistent copy. These can be reverted using session.expire(x). ¶
- **[orm]** Also fixed behavior in merge() whereby collection elements present on the destination but not the merged collection were not being removed from the destination. ¶
- **[orm]** Added a more aggressive check for "uncompiled mappers", helps particularly with declarative layer ¶ References: #995
- **[orm]** The methodology behind "primaryjoin"/"secondaryjoin" has been refactored. Behavior should be slightly more intelligent, primarily in terms of error messages which have been pared down to be more readable. In a slight number of scenarios it can better resolve the correct foreign key than before. ¶
- **[orm]** Added comparable_property(), adds query Comparator behavior to regular, unmanaged Python properties ¶
- **[orm] ['machines'] [Company.employees.of_type(Engineer)]** the functionality of query.with_polymorphic() has been added to mapper() as a configuration option.

  It's set via several forms: with_polymorphic='*' with_polymorphic=[mappers] with_polymorphic=('*', selectable) with_polymorphic=([mappers], selectable)

This controls the default polymorphic loading strategy for inherited mappers. When a selectable is not given, outer joins are created for all joined-table inheriting mappers requested. Note that the auto-create of joins is not compatible with concrete table inheritance.

The existing select_table flag on mapper() is now deprecated and is synonymous with with_polymorphic('*', select_table). Note that the underlying "guts" of select_table have been completely removed and replaced with the newer, more flexible approach.

The new approach also automatically allows eager loads to work for subclasses, if they are present, for example:

```
sess.query(Company).options(
 eagerload_all(
))
```

to load Company objects, their employees, and the 'machines' collection of employees who happen to be Engineers. A "with_polymorphic" Query option should be introduced soon as well which would allow per-Query control of with_polymorphic() on relations. ¶

- **[orm]** added two "experimental" features to Query, "experimental" in that their specific name/behavior is not carved in stone just yet: _values() and _from_self(). We'd like feedback on these.
  - _values(*columns) is given a list of column expressions, and returns a new Query that only returns those columns. When evaluated, the return value is a list of tuples just like when using add_column() or add_entity(), the only difference is that "entity zero", i.e. the mapped class, is not included in the results. This means it finally makes sense to use group_by() and having() on Query, which have been sitting around uselessly until now.

    A future change to this method may include that its ability to join, filter and allow other options not related to a "resultset" are removed, so the feedback we're looking for is how people want to use _values()...i.e. at the very end, or do people prefer to continue generating after it's called.
  - _from_self() compiles the SELECT statement for the Query (minus any eager loaders), and returns a new Query that selects from that SELECT. So basically you can query from a Query without needing to extract the SELECT statement manually. This gives meaning to operations like query[3:5]._from_self().filter(some criterion). There's not much controversial here except that you can quickly create highly nested queries that are less efficient, and we want feedback on the naming choice.

  ¶
- **[orm]** query.order_by() and query.group_by() will accept multiple arguments using *args (like select() already does). ¶
- **[orm]** Added some convenience descriptors to Query: query.statement returns the full SELECT construct, query.whereclause returns just the WHERE part of the SELECT construct. ¶
- **[orm]** Fixed/covered case when using a False/0 value as a polymorphic discriminator. ¶
- **[orm]** Fixed bug which was preventing synonym() attributes from being used with inheritance ¶
- **[orm]** Fixed SQL function truncation of trailing underscores ¶ References: #996
- **[orm]** When attributes are expired on a pending instance, an error will not be raised when the "refresh" action is triggered and no result is found. ¶
- **[orm]** Session.execute can now find binds from metadata ¶
- **[orm]** Adjusted the definition of "self-referential" to be any two mappers with a common parent (this affects whether or not aliased=True is required when joining with Query). ¶
- **[orm]** Made some fixes to the "from_joinpoint" argument to query.join() so that if the previous join was aliased and this one isn't, the join still happens successfully. ¶

- **[orm]**

  Assorted "cascade deletes" fixes:

  - Fixed "cascade delete" operation of dynamic relations, which had only been implemented for foreign-key nulling behavior in 0.4.2 and not actual cascading deletes
  - Delete cascade without delete-orphan cascade on a many-to-one will not delete orphans which were disconnected from the parent before session.delete() is called on the parent (one-to-many already had this).
  - Delete cascade with delete-orphan will delete orphans whether or not it remains attached to its also-deleted parent.
  - delete-orphan casacde is properly detected on relations that are present on superclasses when using inheritance.

  ¶ References: #895

- **[orm]** Fixed order_by calculation in Query to properly alias mapper-config'ed order_by when using select_from() ¶
- **[orm]** Refactored the diffing logic that kicks in when replacing one collection with another into collections.bulk_replace, useful to anyone building multi-level collections. ¶
- **[orm]** Cascade traversal algorithm converted from recursive to iterative to support deep object graphs. ¶

sql

- **[sql]** schema-qualified tables now will place the schemaname ahead of the tablename in all column expressions as well as when generating column labels. This prevents cross- schema name collisions in all cases ¶ References: #999
- **[sql]** can now allow selects which correlate all FROM clauses and have no FROM themselves. These are typically used in a scalar context, i.e. SELECT x, (SELECT x WHERE y) FROM table. Requires explicit correlate() call. ¶
- **[sql]** 'name' is no longer a required constructor argument for Column(). It (and .key) may now be deferred until the column is added to a Table. ¶
- **[sql]** like(), ilike(), contains(), startswith(), endswith() take an optional keyword argument "escape=<somestring>", which is set as the escape character using the syntax "x LIKE y ESCAPE '<somestring>'". ¶ References: #791, #993
- **[sql]** random() is now a generic sql function and will compile to the database's random implementation, if any. ¶
- **[sql]** update().values() and insert().values() take keyword arguments. ¶
- **[sql]** Fixed an issue in select() regarding its generation of FROM clauses, in rare circumstances two clauses could be produced when one was intended to cancel out the other. Some ORM queries with lots of eager loads might have seen this symptom. ¶
- **[sql]** The case() function now also takes a dictionary as its whens parameter. It also interprets the "THEN" expressions as values by default, meaning case([(x==y, "foo")]) will interpret "foo" as a bound value, not a SQL expression. use text(expr) for literal SQL expressions in this case. For the criterion itself, these may be literal strings only if the "value" keyword is present, otherwise SA will force explicit usage of either text() or literal(). ¶

mysql

- **[mysql]** The connection.info keys the dialect uses to cache server settings have changed and are now namespaced. ¶

mssql

- **[mssql]** Reflected tables will now automatically load other tables which are referenced by Foreign keys in the auto-loaded table,. ¶ References: #979
- **[mssql]** Added executemany check to skip identity fetch,. ¶ References: #916
- **[mssql]** Added stubs for small date type. ¶ References: #884
- **[mssql]** Added a new 'driver' keyword parameter for the pyodbc dialect. Will substitute into the ODBC connection string if given, defaults to 'SQL Server'. ¶
- **[mssql]** Added a new 'max_identifier_length' keyword parameter for the pyodbc dialect. ¶
- **[mssql]** Improvements to pyodbc + Unix. If you couldn't get that combination to work before, please try again. ¶

oracle

- **[oracle]** The "owner" keyword on Table is now deprecated, and is exactly synonymous with the "schema" keyword. Tables can now be reflected with alternate "owner" attributes, explicitly stated on the Table object or not using "schema". ¶
- **[oracle]** All of the "magic" searching for synonyms, DBLINKs etc. during table reflection are disabled by default unless you specify "oracle_resolve_synonyms=True" on the Table object. Resolving synonyms necessarily leads to some messy guessing which we'd rather leave off by default. When the flag is set, tables and related tables will be resolved against synonyms in all cases, meaning if a synonym exists for a particular table, reflection will use it when reflecting related tables. This is stickier behavior than before which is why it's off by default. ¶
- **[oracle]** The "owner" keyword on Table is now deprecated, and is exactly synonymous with the "schema" keyword. Tables can now be reflected with alternate "owner" attributes, explicitly stated on the Table object or not using "schema". ¶
- **[oracle]** All of the "magic" searching for synonyms, DBLINKs etc. during table reflection are disabled by default unless you specify "oracle_resolve_synonyms=True" on the Table object. Resolving synonyms necessarily leads to some messy guessing which we'd rather leave off by default. When the flag is set, tables and related tables will be resolved against synonyms in all cases, meaning if a synonym exists for a particular table, reflection will use it when reflecting related tables. This is stickier behavior than before which is why it's off by default. ¶

misc

- **[declarative] [extension]** The "synonym" function is now directly usable with "declarative". Pass in the decorated property using the "descriptor" keyword argument, e.g.: somekey = synonym('_somekey', descriptor=property(g, s)) ¶
- **[declarative] [extension]** The "deferred" function is usable with "declarative". Simplest usage is to declare deferred and Column together, e.g.: data = deferred(Column(Text)) ¶

- **[declarative] [extension]** Declarative also gained @synonym_for(...) and @comparable_using(...), front-ends for synonym and comparable_property. ¶
- **[declarative] [extension]** Improvements to mapper compilation when using declarative; already-compiled mappers will still trigger compiles of other uncompiled mappers when used ¶ References: #995
- **[declarative] [extension]** Declarative will complete setup for Columns lacking names, allows a more DRY syntax.

  class Foo(Base): __tablename__ = 'foos' id = Column(Integer, primary_key=True)

  ¶
- **[declarative] [extension]** inheritance in declarative can be disabled when sending "inherits=None" to __mapper_args__. ¶
- **[declarative] [extension]** declarative_base() takes optional kwarg "mapper", which is any callable/class/method that produces a mapper, such as declarative_base(mapper=scopedsession.mapper). This property can also be set on individual declarative classes using the "__mapper_cls__" property. ¶
- **[postgres]** Got PG server side cursors back into shape, added fixed unit tests as part of the default test suite. Added better uniqueness to the cursor ID ¶ References: #1001

0.4.4

Released: Wed Mar 12 2008

orm

- **[orm]** any(), has(), contains(), ~contains(), attribute level == and != now work properly with self-referential relations - the clause inside the EXISTS is aliased on the "remote" side to distinguish it from the parent table. This applies to single table self-referential as well as inheritance-based self-referential. ¶
- **[orm]** Repaired behavior of == and != operators at the relation() level when compared against NULL for one-to-one relations ¶ References: #985
- **[orm]** Fixed bug whereby session.expire() attributes were not loading on an polymorphically-mapped instance mapped by a select_table mapper. ¶
- **[orm]** Added query.with_polymorphic() - specifies a list of classes which descend from the base class, which will be added to the FROM clause of the query. Allows subclasses to be used within filter() criterion as well as eagerly loads the attributes of those subclasses. ¶
- **[orm]** Your cries have been heard: removing a pending item from an attribute or collection with delete-orphan expunges the item from the session; no FlushError is raised. Note that if you session.save()'ed the pending item explicitly, the attribute/collection removal still knocks it out. ¶
- **[orm]** session.refresh() and session.expire() raise an error when called on instances which are not persistent within the session ¶
- **[orm]** Fixed potential generative bug when the same Query was used to generate multiple Query objects using join(). ¶
- **[orm]** Fixed bug which was introduced in 0.4.3, whereby loading an already-persistent instance mapped with joined table inheritance would trigger a useless "secondary" load from its joined table, when using the default "select" polymorphic_fetch. This was due to attributes being marked as expired during its first load and not getting unmarked from the previous "secondary" load. Attributes are now unexpired based on presence in __dict__ after any load or commit operation succeeds. ¶

- **[orm]** Deprecated Query methods apply_sum(), apply_max(), apply_min(), apply_avg(). Better methodologies are coming.... ¶
- **[orm]** relation() can accept a callable for its first argument, which returns the class to be related. This is in place to assist declarative packages to define relations without classes yet being in place. ¶
- **[orm]** Added a new "higher level" operator called "of_type()": used in join() as well as with any() and has(), qualifies the subclass which will be used in filter criterion, e.g.:

  query.filter(Company.employees.of_type(Engineer). any(Engineer.name=='foo'))

  or

  query.join(Company.employees.of_type(Engineer)). filter(Engineer.name=='foo')

  ¶
- **[orm]** Preventive code against a potential lost-reference bug in flush(). ¶
- **[orm]** Expressions used in filter(), filter_by() and others, when they make usage of a clause generated from a relation using the identity of a child object (e.g., filter(Parent.child==<somechild>)), evaluate the actual primary key value of <somechild> at execution time so that the autoflush step of the Query can complete, thereby populating the PK value of <somechild> in the case that <somechild> was pending. ¶
- **[orm]** setting the relation()-level order by to a column in the many-to-many "secondary" table will now work with eager loading, previously the "order by" wasn't aliased against the secondary table's alias. ¶
- **[orm]** Synonyms riding on top of existing descriptors are now full proxies to those descriptors. ¶

sql

- **[sql]** Can again create aliases of selects against textual FROM clauses. ¶ References: #975
- **[sql]** The value of a bindparam() can be a callable, in which case it's evaluated at statement execution time to get the value. ¶
- **[sql]** Added exception wrapping/reconnect support to result set fetching. Reconnect works for those databases that raise a catchable data error during results (i.e. doesn't work on MySQL) ¶ References: #978
- **[sql]** Implemented two-phase API for "threadlocal" engine, via engine.begin_twophase(), engine.prepare() ¶ References: #936
- **[sql]** Fixed bug which was preventing UNIONS from being cloneable. ¶ References: #986
- **[sql]** Added "bind" keyword argument to insert(), update(), delete() and DDL(). The .bind property is now assignable on those statements as well as on select(). ¶
- **[sql]** Insert statements can now be compiled with extra "prefix" words between INSERT and INTO, for vendor extensions like MySQL's INSERT IGNORE INTO table. ¶

misc

- **[dialects]** Invalid SQLite connection URLs now raise an error. ¶
- **[dialects]** postgres TIMESTAMP renders correctly ¶ References: #981
- **[dialects]** postgres PGArray is a "mutable" type by default; when used with the ORM, mutable-style equality/copy-on-write techniques are used to test for changes. ¶
- **[extensions]** a new super-small "declarative" extension has been added, which allows Table and mapper() configuration to take place inline underneath a class declaration. This extension differs from ActiveMapper

and Elixir in that it does not redefine any SQLAlchemy semantics at all; literal Column, Table and relation() constructs are used to define the class behavior and table definition. ¶

0.4.3

Released: Thu Feb 14 2008

general

- **[general]** Fixed a variety of hidden and some not-so-hidden compatibility issues for Python 2.3, thanks to new support for running the full test suite on 2.3. ¶
- **[general]** Warnings are now issued as type exceptions.SAWarning. ¶

orm

- **[orm]** Every Session.begin() must now be accompanied by a corresponding commit() or rollback() unless the session is closed with Session.close(). This also includes the begin() which is implicit to a session created with transactional=True. The biggest change introduced here is that when a Session created with transactional=True raises an exception during flush(), you must call Session.rollback() or Session.close() in order for that Session to continue after an exception. ¶
- **[orm]** Fixed merge() collection-doubling bug when merging transient entities with backref'ed collections. ¶ References: #961
- **[orm]** merge(dont_load=True) does not accept transient entities, this is in continuation with the fact that merge(dont_load=True) does not accept any "dirty" objects either. ¶
- **[orm]** Added standalone "query" class attribute generated by a scoped_session. This provides MyClass.query without using Session.mapper. Use via:

  ```
  MyClass.query = Session.query_property()
  ```

  ¶
- **[orm]** The proper error message is raised when trying to access expired instance attributes with no session present ¶
- **[orm]** dynamic_loader() / lazy="dynamic" now accepts and uses the order_by parameter in the same way in which it works with relation(). ¶
- **[orm]** Added expire_all() method to Session. Calls expire() for all persistent instances. This is handy in conjunction with... ¶
- **[orm]** Instances which have been partially or fully expired will have their expired attributes populated during a regular Query operation which affects those objects, preventing a needless second SQL statement for each instance. ¶
- **[orm]** Dynamic relations, when referenced, create a strong reference to the parent object so that the query still has a parent to call against even if the parent is only created (and otherwise dereferenced) within the scope of a single expression. ¶ References: #938
- **[orm]** Added a mapper() flag "eager_defaults". When set to True, defaults that are generated during an INSERT or UPDATE operation are post-fetched immediately, instead of being deferred until later. This mimics the old 0.3 behavior. ¶

- **[orm]** query.join() can now accept class-mapped attributes as arguments. These can be used in place or in any combination with strings. In particular this allows construction of joins to subclasses on a polymorphic relation, i.e.:

  query(Company).join(['employees', Engineer.name])

  ¶

- **[orm] [people.join(engineer))] [('employees'] [Engineer.name]** query.join() can also accept tuples of attribute name/some selectable as arguments. This allows construction of joins *from* subclasses of a polymorphic relation, i.e.:

  query(Company).join(

)

  ¶

- **[orm]** General improvements to the behavior of join() in conjunction with polymorphic mappers, i.e. joining from/to polymorphic mappers and properly applying aliases. ¶
- **[orm]** Fixed/improved behavior when a mapper determines the natural "primary key" of a mapped join, it will more effectively reduce columns which are equivalent via foreign key relation. This affects how many arguments need to be sent to query.get(), among other things. ¶ References: #933
- **[orm]** The lazy loader can now handle a join condition where the "bound" column (i.e. the one that gets the parent id sent as a bind parameter) appears more than once in the join condition. Specifically this allows the common task of a relation() which contains a parent-correlated subquery, such as "select only the most recent child item". ¶ References: #946
- **[orm]** Fixed bug in polymorphic inheritance where an incorrect exception is raised when base polymorphic_on column does not correspond to any columns within the local selectable of an inheriting mapper more than one level deep ¶
- **[orm]** Fixed bug in polymorphic inheritance which made it difficult to set a working "order_by" on a polymorphic mapper. ¶
- **[orm]** Fixed a rather expensive call in Query that was slowing down polymorphic queries. ¶
- **[orm]** "Passive defaults" and other "inline" defaults can now be loaded during a flush() call if needed; in particular, this allows constructing relations() where a foreign key column references a server-side-generated, non-primary-key column. ¶ References: #954
- **[orm]**

  Additional Session transaction fixes/changes:

  - Fixed bug with session transaction management: parent transactions weren't started on the connection when adding a connection to a nested transaction.
  - session.transaction now always refers to the innermost active transaction, even when commit/rollback are called directly on the session transaction object.
  - Two-phase transactions can now be prepared.
  - When preparing a two-phase transaction fails on one connection, all the connections are rolled back.
  - session.close() didn't close all transactions when nested transactions were used.
  - rollback() previously erroneously set the current transaction directly to the parent of the transaction that could be rolled back to. Now it rolls back the next transaction up that can handle it, but sets the current transaction to its parent and inactivates the transactions in between. Inactive transactions can only be rolled back or closed, any other call results in an error.
  - autoflush for commit() wasn't flushing for simple subtransactions.

  - unitofwork flush didn't close the failed transaction when the session was not in a transaction and committing the transaction failed.

¶

- **[orm]** Miscellaneous tickets: ¶ References: #964, #940

sql

- **[sql]** Added "schema.DDL", an executable free-form DDL statement. DDLs can be executed in isolation or attached to Table or MetaData instances and executed automatically when those objects are created and/or dropped. ¶
- **[sql]** Table columns and constraints can be overridden on a an existing table (such as a table that was already reflected) using the 'useexisting=True' flag, which now takes into account the arguments passed along with it. ¶
- **[sql]** Added a callable-based DDL events interface, adds hooks before and after Tables and MetaData create and drop. ¶
- **[sql]** Added generative where(<criterion>) method to delete() and update() constructs which return a new object with criterion joined to existing criterion via AND, just like select().where(). ¶
- **[sql]** Added "ilike()" operator to column operations. Compiles to ILIKE on postgres, lower(x) LIKE lower(y) on all others. ¶ References: #727
- **[sql]** Added "now()" as a generic function; on SQLite, Oracle and MSSQL compiles as "CURRENT_TIMESTAMP"; "now()" on all others. ¶ References: #943
- **[sql]** The startswith(), endswith(), and contains() operators now concatenate the wildcard operator with the given operand in SQL, i.e. "'%' || <bindparam>" in all cases, accept text('something') operands properly ¶ References: #962
- **[sql]** cast() accepts text('something') and other non-literal operands properly ¶ References: #962
- **[sql]** fixed bug in result proxy where anonymously generated column labels would not be accessible using their straight string name ¶
- **[sql]** Deferrable constraints can now be defined. ¶
- **[sql]** Added "autocommit=True" keyword argument to select() and text(), as well as generative autocommit() method on select(); for statements which modify the database through some user-defined means other than the usual INSERT/UPDATE/ DELETE etc. This flag will enable "autocommit" behavior during execution if no transaction is in progress. ¶ References: #915
- **[sql]** The '.c.' attribute on a selectable now gets an entry for every column expression in its columns clause. Previously, "unnamed" columns like functions and CASE statements weren't getting put there. Now they will, using their full string representation if no 'name' is available. ¶
- **[sql]** a CompositeSelect, i.e. any union(), union_all(), intersect(), etc. now asserts that each selectable contains the same number of columns. This conforms to the corresponding SQL requirement. ¶
- **[sql]** The anonymous 'label' generated for otherwise unlabeled functions and expressions now propagates outwards at compile time for expressions like select([select([func.foo()])]). ¶
- **[sql]** Building on the above ideas, CompositeSelects now build up their ".c." collection based on the names present in the first selectable only; corresponding_column() now works fully for all embedded selectables. ¶
- **[sql]** Oracle and others properly encode SQL used for defaults like sequences, etc., even if no unicode idents are used since identifier preparer may return a cached unicode identifier. ¶

- **[sql]** Column and clause comparisons to datetime objects on the left hand side of the expression now work (d < table.c.col). (datetimes on the RHS have always worked, the LHS exception is a quirk of the datetime implementation.) ¶

misc

- **[dialects]** Better support for schemas in SQLite (linked in by ATTACH DATABASE ... AS name). In some cases in the past, schema names were omitted from generated SQL for SQLite. This is no longer the case. ¶
- **[dialects]** table_names on SQLite now picks up temporary tables as well. ¶
- **[dialects]** Auto-detect an unspecified MySQL ANSI_QUOTES mode during reflection operations, support for changing the mode midstream. Manual mode setting is still required if no reflection is used. ¶
- **[dialects]** Fixed reflection of TIME columns on SQLite. ¶
- **[dialects]** Finally added PGMacAddr type to postgres ¶ References: #580
- **[dialects]** Reflect the sequence associated to a PK field (typically with a BEFORE INSERT trigger) under Firebird ¶
- **[dialects]** Oracle assembles the correct columns in the result set column mapping when generating a LIMIT/OFFSET subquery, allows columns to map properly to result sets even if long-name truncation kicks in ¶ References: #941
- **[dialects]** MSSQL now includes EXEC in the _is_select regexp, which should allow row-returning stored procedures to be used. ¶
- **[dialects]** MSSQL now includes an experimental implementation of LIMIT/OFFSET using the ANSI SQL row_number() function, so it requires MSSQL-2005 or higher. To enable the feature, add "has_window_funcs" to the keyword arguments for connect, or add "?has_window_funcs=1" to your dburi query arguments. ¶
- **[ext]** Changed ext.activemapper to use a non-transactional session for the objectstore. ¶
- **[ext]** Fixed output order of "['a'] + obj.proxied" binary operation on association-proxied lists. ¶

0.4.2p3

Released: Wed Jan 09 2008

general

- **[general]** sub version numbering scheme changed to suite setuptools version number rules; easy_install -u should now get this version over 0.4.2. ¶

orm

- **[orm]** fixed bug with session.dirty when using "mutable scalars" (such as PickleTypes) ¶
- **[orm]** added a more descriptive error message when flushing on a relation() that has non-locally-mapped columns in its primary or secondary join condition ¶
- **[orm]** suppressing *all* errors in InstanceState.__cleanup() now. ¶
- **[orm]** fixed an attribute history bug whereby assigning a new collection to a collection-based attribute which already had pending changes would generate incorrect history ¶ References: #922

- **[orm]** fixed delete-orphan cascade bug whereby setting the same object twice to a scalar attribute could log it as an orphan ¶ References: #925
- **[orm]** Fixed cascades on a += assignment to a list-based relation. ¶
- **[orm]** synonyms can now be created against props that don't exist yet, which are later added via add_property(). This commonly includes backrefs. (i.e. you can make synonyms for backrefs without worrying about the order of operations) ¶ References: #919
- **[orm]** fixed bug which could occur with polymorphic "union" mapper which falls back to "deferred" loading of inheriting tables ¶
- **[orm]** the "columns" collection on a mapper/mapped class (i.e. 'c') is against the mapped table, not the select_table in the case of polymorphic "union" loading (this shouldn't be noticeable). ¶
- **[orm]** fixed fairly critical bug whereby the same instance could be listed more than once in the unitofwork.new collection; most typically reproduced when using a combination of inheriting mappers and ScopedSession.mapper, as the multiple __init__ calls per instance could save() the object with distinct _state objects ¶
- **[orm]** added very rudimentary yielding iterator behavior to Query. Call query.yield_per(<number of rows>) and evaluate the Query in an iterative context; every collection of N rows will be packaged up and yielded. Use this method with extreme caution since it does not attempt to reconcile eagerly loaded collections across result batch boundaries, nor will it behave nicely if the same instance occurs in more than one batch. This means that an eagerly loaded collection will get cleared out if it's referenced in more than one batch, and in all cases attributes will be overwritten on instances that occur in more than one batch. ¶
- **[orm]** Fixed in-place set mutation operators for set collections and association proxied sets. ¶ References: #920

sql

- **[sql]** Text type is properly exported now and does not raise a warning on DDL create; String types with no length only raise warnings during CREATE TABLE ¶ References: #912
- **[sql]** new UnicodeText type is added, to specify an encoded, unlengthed Text type ¶
- **[sql]** fixed bug in union() so that select() statements which don't derive from FromClause objects can be unioned ¶
- **[sql]** changed name of TEXT to Text since its a "generic" type; TEXT name is deprecated until 0.5. The "upgrading" behavior of String to Text when no length is present is also deprecated until 0.5; will issue a warning when used for CREATE TABLE statements (String with no length for SQL expression purposes is still fine) ¶ References: #912
- **[sql]** generative select.order_by(None) / group_by(None) was not managing to reset order by/group by criterion, fixed ¶ References: #924

misc

- **[dialects]** Fixed reflection of mysql empty string column defaults. ¶
- **[ext]** '+', '*', '+=' and '*=' support for association proxied lists. ¶
- **[dialects]** mssql - narrowed down the test for "date"/"datetime" in MSDate/ MSDateTime subclasses so that incoming "datetime" objects don't get mis-interpreted as "date" objects and vice versa. ¶ References: #923
- **[dialects]** Fixed the missing call to subtype result processor for the PGArray type. ¶ References: #913

0.4.2

Released: Wed Jan 02 2008

orm

- **[orm]** a major behavioral change to collection-based backrefs: they no longer trigger lazy loads ! "reverse" adds and removes are queued up and are merged with the collection when it is actually read from and loaded; but do not trigger a load beforehand. For users who have noticed this behavior, this should be much more convenient than using dynamic relations in some cases; for those who have not, you might notice your apps using a lot fewer queries than before in some situations. ¶ References: #871
- **[orm]** mutable primary key support is added. primary key columns can be changed freely, and the identity of the instance will change upon flush. In addition, update cascades of foreign key referents (primary key or not) along relations are supported, either in tandem with the database's ON UPDATE CASCADE (required for DB's like Postgres) or issued directly by the ORM in the form of UPDATE statements, by setting the flag "passive_cascades=False". ¶
- **[orm]** inheriting mappers now inherit the MapperExtensions of their parent mapper directly, so that all methods for a particular MapperExtension are called for subclasses as well. As always, any MapperExtension can return either EXT_CONTINUE to continue extension processing or EXT_STOP to stop processing. The order of mapper resolution is: <extensions declared on the classes mapper> <extensions declared on the classes' parent mapper> <globally declared extensions>.

  Note that if you instantiate the same extension class separately and then apply it individually for two mappers in the same inheritance chain, the extension will be applied twice to the inheriting class, and each method will be called twice.

  To apply a mapper extension explicitly to each inheriting class but have each method called only once per operation, use the same instance of the extension for both mappers. ¶ References: #490
- **[orm]** MapperExtension.before_update() and after_update() are now called symmetrically; previously, an instance that had no modified column attributes (but had a relation() modification) could be called with before_update() but not after_update() ¶ References: #907
- **[orm]** columns which are missing from a Query's select statement now get automatically deferred during load. ¶
- **[orm]** mapped classes which extend "object" and do not provide an __init__() method will now raise TypeError if non-empty *args or **kwargs are present at instance construction time (and are not consumed by any extensions such as the scoped_session mapper), consistent with the behavior of normal Python classes ¶ References: #908
- **[orm]** fixed Query bug when filter_by() compares a relation against None ¶ References: #899
- **[orm]** improved support for pickling of mapped entities. Per-instance lazy/deferred/expired callables are now serializable so that they serialize and deserialize with _state. ¶
- **[orm]** new synonym() behavior: an attribute will be placed on the mapped class, if one does not exist already, in all cases. if a property already exists on the class, the synonym will decorate the property with the appropriate comparison operators so that it can be used in column expressions just like any other mapped attribute (i.e. usable in filter(), etc.) the "proxy=True" flag is deprecated and no longer means anything. Additionally, the flag "map_column=True" will automatically generate a ColumnProperty corresponding to the name of the synonym, i.e.: 'somename':synonym('_somename', map_column=True) will map the column named 'somename' to the attribute '_somename'. See the example in the mapper docs. ¶ References: #801
- **[orm]** Query.select_from() now replaces all existing FROM criterion with the given argument; the previous behavior of constructing a list of FROM clauses was generally not useful as is required filter() calls to create

join criterion, and new tables introduced within filter() already add themselves to the FROM clause. The new behavior allows not just joins from the main table, but select statements as well. Filter criterion, order bys, eager load clauses will be "aliased" against the given statement. ¶

- **[orm]** this month's refactoring of attribute instrumentation changes the "copy-on-load" behavior we've had since midway through 0.3 with "copy-on-modify" in most cases. This takes a sizable chunk of latency out of load operations and overall does less work as only attributes which are actually modified get their "committed state" copied. Only "mutable scalar" attributes (i.e. a pickled object or other mutable item), the reason for the copy-on-load change in the first place, retain the old behavior. ¶
- **[orm] [attrname]** a slight behavioral change to attributes is, del'ing an attribute does *not* cause the lazyloader of that attribute to fire off again; the "del" makes the effective value of the attribute "None". To re-trigger the "loader" for an attribute, use session.expire(instance,). ¶
- **[orm]** query.filter(SomeClass.somechild == None), when comparing a many-to-one property to None, properly generates "id IS NULL" including that the NULL is on the right side. ¶
- **[orm]** query.order_by() takes into account aliased joins, i.e. query.join('orders', aliased=True).order_by(Order.id) ¶
- **[orm]** eagerload(), lazyload(), eagerload_all() take an optional second class-or-mapper argument, which will select the mapper to apply the option towards. This can select among other mappers which were added using add_entity(). ¶
- **[orm]** eagerloading will work with mappers added via add_entity(). ¶
- **[orm]** added "cascade delete" behavior to "dynamic" relations just like that of regular relations. if passive_deletes flag (also just added) is not set, a delete of the parent item will trigger a full load of the child items so that they can be deleted or updated accordingly. ¶
- **[orm]** also with dynamic, implemented correct count() behavior as well as other helper methods. ¶
- **[orm]** fix to cascades on polymorphic relations, such that cascades from an object to a polymorphic collection continue cascading along the set of attributes specific to each element in the collection. ¶
- **[orm]** query.get() and query.load() do not take existing filter or other criterion into account; these methods *always* look up the given id in the database or return the current instance from the identity map, disregarding any existing filter, join, group_by or other criterion which has been configured. ¶ References: #893
- **[orm]** added support for version_id_col in conjunction with inheriting mappers. version_id_col is typically set on the base mapper in an inheritance relationship where it takes effect for all inheriting mappers. ¶ References: #883
- **[orm]** relaxed rules on column_property() expressions having labels; any ColumnElement is accepted now, as the compiler auto-labels non-labeled ColumnElements now. a selectable, like a select() statement, still requires conversion to ColumnElement via as_scalar() or label(). ¶
- **[orm]** fixed backref bug where you could not del instance.attr if attr was None ¶
- **[orm]** several ORM attributes have been removed or made private: mapper.get_attr_by_column(), mapper.set_attr_by_column(), mapper.pks_by_table, mapper.cascade_callable(), MapperProperty.cascade_callable(), mapper.canload(), mapper.save_obj(), mapper.delete_obj(), mapper._mapper_registry, attributes.AttributeManager ¶
- **[orm]** Assigning an incompatible collection type to a relation attribute now raises TypeError instead of sqlalchemy's ArgumentError. ¶
- **[orm]** Bulk assignment of a MappedCollection now raises an error if a key in the incoming dictionary does not match the key that the collection's keyfunc would use for that value. ¶ References: #886
- **[orm] [newval2] [newval1]** Custom collections can now specify a @converter method to translate objects used in "bulk" assignment into a stream of values, as in:

```
obj.col =
# or
obj.dictcol = {'foo': newval1, 'bar': newval2}
```

The MappedCollection uses this hook to ensure that incoming key/value pairs are sane from the collection's perspective. ¶

- **[orm]** fixed endless loop issue when using lazy="dynamic" on both sides of a bi-directional relationship ¶ References: #872
- **[orm]** more fixes to the LIMIT/OFFSET aliasing applied with Query + eagerloads, in this case when mapped against a select statement ¶ References: #904
- **[orm]** fix to self-referential eager loading such that if the same mapped instance appears in two or more distinct sets of columns in the same result set, its eagerly loaded collection will be populated regardless of whether or not all of the rows contain a set of "eager" columns for that collection. this would also show up as a KeyError when fetching results with join_depth turned on. ¶
- **[orm]** fixed bug where Query would not apply a subquery to the SQL when LIMIT was used in conjunction with an inheriting mapper where the eager loader was only in the parent mapper. ¶
- **[orm]** clarified the error message which occurs when you try to update() an instance with the same identity key as an instance already present in the session. ¶
- **[orm]** some clarifications and fixes to merge(instance, dont_load=True). fixed bug where lazy loaders were getting disabled on returned instances. Also, we currently do not support merging an instance which has uncommitted changes on it, in the case that dont_load=True is used....this will now raise an error. This is due to complexities in merging the "committed state" of the given instance to correctly correspond to the newly copied instance, as well as other modified state. Since the use case for dont_load=True is caching, the given instances shouldn't have any uncommitted changes on them anyway. We also copy the instances over without using any events now, so that the 'dirty' list on the new session remains unaffected. ¶
- **[orm]** fixed bug which could arise when using session.begin_nested() in conjunction with more than one level deep of enclosing session.begin() statements ¶
- **[orm]** fixed session.refresh() with instance that has custom entity_name ¶ References: #914

sql

- **[sql]** generic functions ! we introduce a database of known SQL functions, such as current_timestamp, coalesce, and create explicit function objects representing them. These objects have constrained argument lists, are type aware, and can compile in a dialect-specific fashion. So saying func.char_length("foo", "bar") raises an error (too many args), func.coalesce(datetime.date(2007, 10, 5), datetime.date(2005, 10, 15)) knows that its return type is a Date. We only have a few functions represented so far but will continue to add to the system ¶ References: #615
- **[sql]** auto-reconnect support improved; a Connection can now automatically reconnect after its underlying connection is invalidated, without needing to connect() again from the engine. This allows an ORM session bound to a single Connection to not need a reconnect. Open transactions on the Connection must be rolled back after an invalidation of the underlying connection else an error is raised. Also fixed bug where disconnect detect was not being called for cursor(), rollback(), or commit(). ¶
- **[sql]** added new flag to String and create_engine(), assert_unicode=(True|False|'warn'|None). Defaults to *False* or *None* on create_engine() and String, *'warn'* on the Unicode type. When *True*, results in all unicode conversion operations raising an exception when a non-unicode bytestring is passed as a bind parameter. 'warn' results in a warning. It is strongly advised that all unicode-aware applications make proper use of Python unicode objects (i.e. u'hello' and not 'hello') so that data round trips accurately. ¶

- **[sql]** generation of "unique" bind parameters has been simplified to use the same "unique identifier" mechanisms as everything else. This doesn't affect user code, except any code that might have been hardcoded against the generated names. Generated bind params now have the form "<paramname>_<num>", whereas before only the second bind of the same name would have this form. ¶
- **[sql]** select().as_scalar() will raise an exception if the select does not have exactly one expression in its columns clause. ¶
- **[sql]** bindparam() objects themselves can be used as keys for execute(), i.e. statement.execute({bind1:'foo', bind2:'bar'}) ¶
- **[sql]** added new methods to TypeDecorator, process_bind_param() and process_result_value(), which automatically take advantage of the processing of the underlying type. Ideal for using with Unicode or Pickletype. TypeDecorator should now be the primary way to augment the behavior of any existing type including other TypeDecorator subclasses such as PickleType. ¶
- **[sql]** selectables (and others) will issue a warning when two columns in their exported columns collection conflict based on name. ¶
- **[sql]** tables with schemas can still be used in sqlite, firebird, schema name just gets dropped ¶ References: #890
- **[sql]** changed the various "literal" generation functions to use an anonymous bind parameter. not much changes here except their labels now look like ":param_1", ":param_2" instead of ":literal" ¶
- **[sql]** column labels in the form "tablename.columname", i.e. with a dot, are now supported. ¶
- **[sql]** from_obj keyword argument to select() can be a scalar or a list. ¶

firebird

- **[firebird] [backend]** does properly reflect domains (partially fixing) and PassiveDefaults ¶ References: #410
- **[firebird] [3562] [backend]** reverted to use default poolclass (was set to SingletonThreadPool in 0.4.0 for test purposes) ¶
- **[firebird] [backend]** map func.length() to 'char_length' (easily overridable with the UDF 'strlen' on old versions of Firebird) ¶

misc

- **[dialects]** sqlite SLDate type will not erroneously render "microseconds" portion of a datetime or time object. ¶
- **[dialects]**

  oracle

  - added disconnect detection support for Oracle
  - some cleanup to binary/raw types so that cx_oracle.LOB is detected on an ad-hoc basis

  ¶ References: #902
- **[dialects]**

  MSSQL

  - PyODBC no longer has a global "set nocount on".
  - Fix non-identity integer PKs on autload
  - Better support for convert_unicode

  - Less strict date conversion for pyodbc/adodbapi
  - Schema-qualified tables / autoload

¶ References: #824, #839, #842, #901

0.4.1

Released: Sun Nov 18 2007

orm

- **[orm]** eager loading with LIMIT/OFFSET applied no longer adds the primary table joined to a limited subquery of itself; the eager loads now join directly to the subquery which also provides the primary table's columns to the result set. This eliminates a JOIN from all eager loads with LIMIT/OFFSET. ¶ References: #843
- **[orm]** session.refresh() and session.expire() now support an additional argument "attribute_names", a list of individual attribute keynames to be refreshed or expired, allowing partial reloads of attributes on an already-loaded instance. ¶ References: #802
- **[orm]** added op() operator to instrumented attributes; i.e. User.name.op('ilike')('%somename%') ¶ References: #767
- **[orm]** Mapped classes may now define __eq__, __hash__, and __nonzero__ methods with arbitrary semantics. The orm now handles all mapped instances on an identity-only basis. (e.g. 'is' vs '==') ¶ References: #676
- **[orm]** the "properties" accessor on Mapper is removed; it now throws an informative exception explaining the usage of mapper.get_property() and mapper.iterate_properties ¶
- **[orm]** added having() method to Query, applies HAVING to the generated statement in the same way as filter() appends to the WHERE clause. ¶
- **[orm]** The behavior of query.options() is now fully based on paths, i.e. an option such as eagerload_all('x.y.z.y.x') will apply eagerloading to only those paths, i.e. and not 'x.y.x'; eagerload('children.children') applies only to exactly two-levels deep, etc. ¶ References: #777
- **[orm]** PickleType will compare using == when set up with mutable=False, and not the *is* operator. To use *is* or any other comparator, send in a custom comparison function using PickleType(comparator=my_custom_comparator). ¶
- **[orm]** query doesn't throw an error if you use distinct() and an order_by() containing UnaryExpressions (or other) together ¶ References: #848
- **[orm]** order_by() expressions from joined tables are properly added to columns clause when using distinct() ¶ References: #786
- **[orm]** fixed error where Query.add_column() would not accept a class-bound attribute as an argument; Query also raises an error if an invalid argument was sent to add_column() (at instances() time) ¶ References: #858
- **[orm]** added a little more checking for garbage-collection dereferences in InstanceState.__cleanup() to reduce "gc ignored" errors on app shutdown ¶
- **[orm]** The session API has been solidified: ¶
- **[orm]** It's an error to session.save() an object which is already persistent ¶ References: #840
- **[orm]** It's an error to session.delete() an object which is *not* persistent. ¶
- **[orm]** session.update() and session.delete() raise an error when updating or deleting an instance that is already in the session with a different identity. ¶

- **[orm]** The session checks more carefully when determining "object X already in another session"; e.g. if you pickle a series of objects and unpickle (i.e. as in a Pylons HTTP session or similar), they can go into a new session without any conflict ¶
- **[orm]** merge() includes a keyword argument "dont_load=True". setting this flag will cause the merge operation to not load any data from the database in response to incoming detached objects, and will accept the incoming detached object as though it were already present in that session. Use this to merge detached objects from external caching systems into the session. ¶
- **[orm]** Deferred column attributes no longer trigger a load operation when the attribute is assigned to. In those cases, the newly assigned value will be present in the flushes' UPDATE statement unconditionally. ¶
- **[orm]** Fixed a truncation error when re-assigning a subset of a collection (obj.relation = obj.relation[1:]) ¶ References: #834
- **[orm]** De-cruftified backref configuration code, backrefs which step on existing properties now raise an error ¶ References: #832
- **[orm]** Improved behavior of add_property() etc., fixed involving synonym/deferred. ¶ References: #831
- **[orm]** Fixed clear_mappers() behavior to better clean up after itself. ¶
- **[orm]** Fix to "row switch" behavior, i.e. when an INSERT/DELETE is combined into a single UPDATE; many-to-many relations on the parent object update properly. ¶ References: #841
- **[orm]** Fixed __hash__ for association proxy- these collections are unhashable, just like their mutable Python counterparts. ¶
- **[orm]** Added proxying of save_or_update, __contains__ and __iter__ methods for scoped sessions. ¶
- **[orm]** fixed very hard-to-reproduce issue where by the FROM clause of Query could get polluted by certain generative calls ¶ References: #852

sql

- **[sql]** the "shortname" keyword parameter on bindparam() has been deprecated. ¶
- **[sql]** Added contains operator (generates a "LIKE %<other>%" clause). ¶
- **[sql]** anonymous column expressions are automatically labeled. e.g. select([x* 5]) produces "SELECT x * 5 AS anon_1". This allows the labelname to be present in the cursor.description which can then be appropriately matched to result-column processing rules. (we can't reliably use positional tracking for result-column matches since text() expressions may represent multiple columns). ¶
- **[sql]** operator overloading is now controlled by TypeEngine objects - the one built-in operator overload so far is String types overloading '+' to be the string concatenation operator. User-defined types can also define their own operator overloading by overriding the adapt_operator(self, op) method. ¶
- **[sql]** untyped bind parameters on the right side of a binary expression will be assigned the type of the left side of the operation, to better enable the appropriate bind parameter processing to take effect ¶ References: #819
- **[sql]** Removed regular expression step from most statement compilations. Also fixes ¶ References: #833
- **[sql]** Fixed empty (zero column) sqlite inserts, allowing inserts on autoincrementing single column tables. ¶
- **[sql]** Fixed expression translation of text() clauses; this repairs various ORM scenarios where literal text is used for SQL expressions ¶
- **[sql]** Removed ClauseParameters object; compiled.params returns a regular dictionary now, as well as result.last_inserted_params() / last_updated_params(). ¶

- **[sql]** Fixed INSERT statements w.r.t. primary key columns that have SQL-expression based default generators on them; SQL expression executes inline as normal but will not trigger a "postfetch" condition for the column, for those DB's who provide it via cursor.lastrowid ¶
- **[sql]** func. objects can be pickled/unpickled ¶ References: #844
- **[sql]** rewrote and simplified the system used to "target" columns across selectable expressions. On the SQL side this is represented by the "corresponding_column()" method. This method is used heavily by the ORM to "adapt" elements of an expression to similar, aliased expressions, as well as to target result set columns originally bound to a table or selectable to an aliased, "corresponding" expression. The new rewrite features completely consistent and accurate behavior. ¶
- **[sql]** Added a field ("info") for storing arbitrary data on schema items ¶ References: #573
- **[sql]** The "properties" collection on Connections has been renamed "info" to match schema's writable collections. Access is still available via the "properties" name until 0.5. ¶
- **[sql]** fixed the close() method on Transaction when using strategy='threadlocal' ¶
- **[sql]** fix to compiled bind parameters to not mistakenly populate None ¶ References: #853
- **[sql]** <Engine|Connection>._execute_clauseelement becomes a public method Connectable.execute_clauseelement ¶

misc

- **[dialects]** Added experimental support for MaxDB (versions >= 7.6.03.007 only). ¶
- **[dialects]** oracle will now reflect "DATE" as an OracleDateTime column, not OracleDate ¶
- **[dialects]** added awareness of schema name in oracle table_names() function, fixes metadata.reflect(schema='someschema') ¶ References: #847
- **[dialects]** MSSQL anonymous labels for selection of functions made deterministic ¶
- **[dialects]** sqlite will reflect "DECIMAL" as a numeric column. ¶
- **[dialects]** Made access dao detection more reliable ¶ References: #828
- **[dialects]** Renamed the Dialect attribute 'preexecute_sequences' to 'preexecute_pk_sequences'. An attribute porxy is in place for out-of-tree dialects using the old name. ¶
- **[dialects]** Added test coverage for unknown type reflection. Fixed sqlite/mysql handling of type reflection for unknown types. ¶
- **[dialects]** Added REAL for mysql dialect (for folks exploiting the REAL_AS_FLOAT sql mode). ¶
- **[dialects]** mysql Float, MSFloat and MSDouble constructed without arguments now produce no-argument DDL, e.g.'FLOAT'. ¶
- **[misc]** Removed unused util.hash(). ¶

0.4.0

Released: Wed Oct 17 2007

- (see 0.4.0beta1 for the start of major changes against 0.3, as well as http://www.sqlalchemy.org/trac/wiki/WhatsNewIn04) ¶
- Added initial Sybase support (mxODBC so far) ¶ References: #785
- Added partial index support for PostgreSQL. Use the postgres_where keyword on the Index. ¶

- string-based query param parsing/config file parser understands wider range of string values for booleans ¶ References: #817
- backref remove object operation doesn't fail if the other-side collection doesn't contain the item, supports noload collections ¶ References: #813
- removed __len__ from "dynamic" collection as it would require issuing a SQL "count()" operation, thus forcing all list evaluations to issue redundant SQL ¶ References: #818
- inline optimizations added to locate_dirty() which can greatly speed up repeated calls to flush(), as occurs with autoflush=True ¶ References: #816
- The IdentifierPreprarer's _requires_quotes test is now regex based. Any out-of-tree dialects that provide custom sets of legal_characters or illegal_initial_characters will need to move to regexes or override _requires_quotes. ¶
- Firebird has supports_sane_rowcount and supports_sane_multi_rowcount set to False due to ticket #370 (right way). ¶
- **Improvements and fixes on Firebird reflection:**
  - FBDialect now mimics OracleDialect, regarding case-sensitivity of TABLE and COLUMN names (see 'case_sensitive remotion' topic on this current file).
  - FBDialect.table_names() doesn't bring system tables (ticket:796).
  - FB now reflects Column's nullable property correctly.

  ¶
- Fixed SQL compiler's awareness of top-level column labels as used in result-set processing; nested selects which contain the same column names don't affect the result or conflict with result-column metadata. ¶
- query.get() and related functions (like many-to-one lazyloading) use compile-time-aliased bind parameter names, to prevent name conflicts with bind parameters that already exist in the mapped selectable. ¶
- Fixed three- and multi-level select and deferred inheritance loading (i.e. abc inheritance with no select_table). ¶ References: #795
- Ident passed to id_chooser in shard.py always a list. ¶
- The no-arg ResultProxy._row_processor() is now the class attribute *_process_row*. ¶
- Added support for returning values from inserts and updates for PostgreSQL 8.2+. ¶ References: #797
- PG reflection, upon seeing the default schema name being used explicitly as the "schema" argument in a Table, will assume that this is the user's desired convention, and will explicitly set the "schema" argument in foreign-key-related reflected tables, thus making them match only with Table constructors that also use the explicit "schema" argument (even though its the default schema). In other words, SA assumes the user is being consistent in this usage. ¶
- fixed sqlite reflection of BOOL/BOOLEAN ¶ References: #808
- Added support for UPDATE with LIMIT on mysql. ¶
- null foreign key on a m2o doesn't trigger a lazyload ¶ References: #803
- oracle does not implicitly convert to unicode for non-typed result sets (i.e. when no TypeEngine/String/Unicode type is even being used; previously it was detecting DBAPI types and converting regardless). should fix ¶ References: #800
- fix to anonymous label generation of long table/column names ¶ References: #806
- Firebird dialect now uses SingletonThreadPool as poolclass. ¶
- Firebird now uses dialect.preparer to format sequences names ¶

- Fixed breakage with postgres and multiple two-phase transactions. Two-phase commits and rollbacks didn't automatically end up with a new transaction as the usual dbapi commits/rollbacks do. ¶ References: #810
- Added an option to the _ScopedExt mapper extension to not automatically save new objects to session on object initialization. ¶
- fixed Oracle non-ansi join syntax ¶
- PickleType and Interval types (on db not supporting it natively) are now slightly faster. ¶
- Added Float and Time types to Firebird (FBFloat and FBTime). Fixed BLOB SUB_TYPE for TEXT and Binary types. ¶
- Changed the API for the in_ operator. in_() now accepts a single argument that is a sequence of values or a selectable. The old API of passing in values as varargs still works but is deprecated. ¶

0.4.0beta6

Released: Thu Sep 27 2007

- The Session identity map is now *weak referencing* by default, use weak_identity_map=False to use a regular dict. The weak dict we are using is customized to detect instances which are "dirty" and maintain a temporary strong reference to those instances until changes are flushed. ¶
- Mapper compilation has been reorganized such that most compilation occurs upon mapper construction. This allows us to have fewer calls to mapper.compile() and also to allow class-based properties to force a compilation (i.e. User.addresses == 7 will compile all mappers; this is). The only caveat here is that an inheriting mapper now looks for its inherited mapper upon construction; so mappers within inheritance relationships need to be constructed in inheritance order (which should be the normal case anyway). ¶ References: #758
- added "FETCH" to the keywords detected by Postgres to indicate a result-row holding statement (i.e. in addition to "SELECT"). ¶
- Added full list of SQLite reserved keywords so that they get escaped properly. ¶
- Tightened up the relationship between the Query's generation of "eager load" aliases, and Query.instances() which actually grabs the eagerly loaded rows. If the aliases were not specifically generated for that statement by EagerLoader, the EagerLoader will not take effect when the rows are fetched. This prevents columns from being grabbed accidentally as being part of an eager load when they were not meant for such, which can happen with textual SQL as well as some inheritance situations. It's particularly important since the "anonymous aliasing" of columns uses simple integer counts now to generate labels. ¶
- Removed "parameters" argument from clauseelement.compile(), replaced with "column_keys". The parameters sent to execute() only interact with the insert/update statement compilation process in terms of the column names present but not the values for those columns. Produces more consistent execute/executemany behavior, simplifies things a bit internally. ¶
- Added 'comparator' keyword argument to PickleType. By default, "mutable" PickleType does a "deep compare" of objects using their dumps() representation. But this doesn't work for dictionaries. Pickled objects which provide an adequate __eq__() implementation can be set up with "PickleType(comparator=operator.eq)" ¶ References: #560
- Added session.is_modified(obj) method; performs the same "history" comparison operation as occurs within a flush operation; setting include_collections=False gives the same result as is used when the flush determines whether or not to issue an UPDATE for the instance's row. ¶
- Added "schema" argument to Sequence; use this with Postgres /Oracle when the sequence is located in an alternate schema. Implements part of, should fix. ¶ References: #584, #761
- Fixed reflection of the empty string for mysql enums. ¶

- Changed MySQL dialect to use the older LIMIT <offset>, <limit> syntax instead of LIMIT <l> OFFSET <o> for folks using 3.23. ¶ References: #794
- Added 'passive_deletes="all"' flag to relation(), disables all nulling-out of foreign key attributes during a flush where the parent object is deleted. ¶
- Column defaults and onupdates, executing inline, will add parenthesis for subqueries and other parenthesis-requiring expressions ¶
- The behavior of String/Unicode types regarding that they auto-convert to TEXT/CLOB when no length is present now occurs *only* for an exact type of String or Unicode with no arguments. If you use VARCHAR or NCHAR (subclasses of String/Unicode) with no length, they will be interpreted by the dialect as VARCHAR/NCHAR; no "magic" conversion happens there. This is less surprising behavior and in particular this helps Oracle keep string-based bind parameters as VARCHARs and not CLOBs. ¶ References: #793
- Fixes to ShardedSession to work with deferred columns. ¶ References: #771
- User-defined shard_chooser() function must accept "clause=None" argument; this is the ClauseElement passed to session.execute(statement) and can be used to determine correct shard id (since execute() doesn't take an instance.) ¶
- Adjusted operator precedence of NOT to match '==' and others, so that ~(x <operator> y) produces NOT (x <op> y), which is better compatible with older MySQL versions.. This doesn't apply to "~(x==y)" as it does in 0.3 since ~(x==y) compiles to "x != y", but still applies to operators like BETWEEN. ¶ References: #764
- Other tickets:,,. ¶ References: #757, #768, #779, #728

0.4.0beta5

no release date

- Connection pool fixes; the better performance of beta4 remains but fixes "connection overflow" and other bugs which were present (like). ¶ References: #754
- Fixed bugs in determining proper sync clauses from custom inherit conditions. ¶ References: #769
- Extended 'engine_from_config' coercion for QueuePool size / overflow. ¶ References: #763
- mysql views can be reflected again. ¶ References: #748
- AssociationProxy can now take custom getters and setters. ¶
- Fixed malfunctioning BETWEEN in orm queries. ¶
- Fixed OrderedProperties pickling ¶ References: #762
- SQL-expression defaults and sequences now execute "inline" for all non-primary key columns during an INSERT or UPDATE, and for all columns during an executemany()-style call. inline=True flag on any insert/update statement also forces the same behavior with a single execute(). result.postfetch_cols() is a collection of columns for which the previous single insert or update statement contained a SQL-side default expression. ¶
- Fixed PG executemany() behavior. ¶ References: #759
- postgres reflects tables with autoincrement=False for primary key columns which have no defaults. ¶
- postgres no longer wraps executemany() with individual execute() calls, instead favoring performance. "rowcount"/"concurrency" checks with deleted items (which use executemany) are disabled with PG since psycopg2 does not report proper rowcount for executemany(). ¶
- **[tickets] [fixed]** ¶ References: #742
- **[tickets] [fixed]** ¶ References: #748
- **[tickets] [fixed]** ¶ References: #760

- **[tickets] [fixed]** ¶ References: #762
- **[tickets] [fixed]** ¶ References: #763

0.4.0beta4

Released: Wed Aug 22 2007

- Tidied up what ends up in your namespace when you 'from sqlalchemy import *': ¶
- 'table' and 'column' are no longer imported. They remain available by direct reference (as in 'sql.table' and 'sql.column') or a glob import from the sql package. It was too easy to accidentally use a sql.expressions.table instead of schema.Table when just starting out with SQLAlchemy, likewise column. ¶
- Internal-ish classes like ClauseElement, FromClause, NullTypeEngine, etc., are also no longer imported into your namespace ¶
- The 'Smallinteger' compatibility name (small i!) is no longer imported, but remains in schema.py for now. SmallInteger (big I!) is still imported. ¶
- The connection pool uses a "threadlocal" strategy internally to return the same connection already bound to a thread, for "contextual" connections; these are the connections used when you do a "connectionless" execution like insert().execute(). This is like a "partial" version of the "threadlocal" engine strategy but without the thread-local transaction part of it. We're hoping it reduces connection pool overhead as well as database usage. However, if it proves to impact stability in a negative way, we'll roll it right back. ¶
- Fix to bind param processing such that "False" values (like blank strings) still get processed/encoded. ¶
- Fix to select() "generative" behavior, such that calling column(), select_from(), correlate(), and with_prefix() does not modify the original select object ¶ References: #752
- Added a "legacy" adapter to types, such that user-defined TypeEngine and TypeDecorator classes which define convert_bind_param() and/or convert_result_value() will continue to function. Also supports calling the super() version of those methods. ¶
- Added session.prune(), trims away instances cached in a session that are no longer referenced elsewhere. (A utility for strong-ref identity maps). ¶
- Added close() method to Transaction. Closes out a transaction using rollback if it's the outermost transaction, otherwise just ends without affecting the outer transaction. ¶
- Transactional and non-transactional Session integrates better with bound connection; a close() will ensure that connection transactional state is the same as that which existed on it before being bound to the Session. ¶
- Modified SQL operator functions to be module-level operators, allowing SQL expressions to be pickleable. ¶ References: #735
- Small adjustment to mapper class.__init__ to allow for Py2.6 object.__init__() behavior. ¶
- Fixed 'prefix' argument for select() ¶
- Connection.begin() no longer accepts nested=True, this logic is now all in begin_nested(). ¶
- Fixes to new "dynamic" relation loader involving cascades ¶
- **[tickets] [fixed]** ¶ References: #735
- **[tickets] [fixed]** ¶ References: #752

0.4.0beta3

Released: Thu Aug 16 2007

- SQL types optimization: ¶
- New performance tests show a combined mass-insert/mass-select test as having 68% fewer function calls than the same test run against 0.3. ¶
- General performance improvement of result set iteration is around 10-20%. ¶
- In types.AbstractType, convert_bind_param() and convert_result_value() have migrated to callable-returning bind_processor() and result_processor() methods. If no callable is returned, no pre/post processing function is called. ¶
- Hooks added throughout base/sql/defaults to optimize the calling of bind aram/result processors so that method call overhead is minimized. ¶
- Support added for executemany() scenarios such that unneeded "last row id" logic doesn't kick in, parameters aren't excessively traversed. ¶
- Added 'inherit_foreign_keys' arg to mapper(). ¶
- Added support for string date passthrough in sqlite. ¶
- **[tickets] [fixed]** ¶ References: #738
- **[tickets] [fixed]** ¶ References: #739
- **[tickets] [fixed]** ¶ References: #743
- **[tickets] [fixed]** ¶ References: #744

0.4.0beta2

Released: Tue Aug 14 2007

oracle

- **[oracle] [improvements.]** Auto-commit after LOAD DATA INFILE for mysql. ¶
- **[oracle] [improvements.]** A rudimental SessionExtension class has been added, allowing user-defined functionality to take place at flush(), commit(), and rollback() boundaries. ¶
- **[oracle] [improvements.]** Added engine_from_config() function for helping to create_engine() from an .ini style config. ¶
- **[oracle] [improvements.]** base_mapper() becomes a plain attribute. ¶
- **[oracle] [improvements.]** session.execute() and scalar() can search for a Table with which to bind from using the given ClauseElement. ¶
- **[oracle] [improvements.]** Session automatically extrapolates tables from mappers with binds, also uses base_mapper so that inheritance hierarchies bind automatically. ¶
- **[oracle] [improvements.]** Moved ClauseVisitor traversal back to inlined non-recursive. ¶

misc

- **[tickets] [fixed]** ¶ References: #730
- **[tickets] [fixed]** ¶ References: #732
- **[tickets] [fixed]** ¶ References: #733
- **[tickets] [fixed]** ¶ References: #734

0.4.0beta1

Released: Sun Aug 12 2007

orm

- **[orm]** Speed! Along with recent speedups to ResultProxy, total number of function calls significantly reduced for large loads. ¶
- **[orm]** test/perf/masseagerload.py reports 0.4 as having the fewest number of function calls across all SA versions (0.1, 0.2, and 0.3). ¶
- **[orm]** New collection_class api and implementation. Collections are now instrumented via decorations rather than proxying. You can now have collections that manage their own membership, and your class instance will be directly exposed on the relation property. The changes are transparent for most users. ¶ References: #213
- **[orm]** InstrumentedList (as it was) is removed, and relation properties no longer have 'clear()', '.data', or any other added methods beyond those provided by the collection type. You are free, of course, to add them to a custom class. ¶
- **[orm]** __setitem__-like assignments now fire remove events for the existing value, if any. ¶
- **[orm]** dict-likes used as collection classes no longer need to change __iter__ semantics- itervalues() is used by default instead. This is a backwards incompatible change. ¶
- **[orm]** Subclassing dict for a mapped collection is no longer needed in most cases. orm.collections provides canned implementations that key objects by a specified column or a custom function of your choice. ¶
- **[orm]** Collection assignment now requires a compatible type- assigning None to clear a collection or assigning a list to a dict collection will now raise an argument error. ¶
- **[orm]** AttributeExtension moved to interfaces, and .delete is now .remove The event method signature has also been swapped around. ¶
- **[orm]** Major overhaul for Query: ¶
- **[orm]** All selectXXX methods are deprecated. Generative methods are now the standard way to do things, i.e. filter(), filter_by(), all(), one(), etc. Deprecated methods are docstring'ed with their new replacements. ¶
- **[orm]** Class-level properties are now usable as query elements... no more '.c.'! "Class.c.propname" is now superseded by "Class.propname". All clause operators are supported, as well as higher level operators such as Class.prop==<some instance> for scalar attributes, Class.prop.contains(<some instance>) and Class.prop.any(<some expression>) for collection-based attributes (all are also negatable). Table-based column expressions as well as columns mounted on mapped classes via 'c' are of course still fully available and can be freely mixed with the new attributes. ¶ References: #643
- **[orm]** Removed ancient query.select_by_attributename() capability. ¶

- **[orm]** The aliasing logic used by eager loading has been generalized, so that it also adds full automatic aliasing support to Query. It's no longer necessary to create an explicit Alias to join to the same tables multiple times; *even for self-referential relationships*.
  - join() and outerjoin() take arguments "aliased=True". Yhis causes their joins to be built on aliased tables; subsequent calls to filter() and filter_by() will translate all table expressions (yes, real expressions using the original mapped Table) to be that of the Alias for the duration of that join() (i.e. until reset_joinpoint() or another join() is called).
  - join() and outerjoin() take arguments "id=<somestring>". When used with "aliased=True", the id can be referenced by add_entity(cls, id=<somestring>) so that you can select the joined instances even if they're from an alias.
  - join() and outerjoin() now work with self-referential relationships! Using "aliased=True", you can join as many levels deep as desired, i.e. query.join(['children', 'children'], aliased=True); filter criterion will be against the rightmost joined table

  ¶
- **[orm]** Added query.populate_existing(), marks the query to reload all attributes and collections of all instances touched in the query, including eagerly-loaded entities. ¶ References: #660
- **[orm]** Added eagerload_all(), allows eagerload_all('x.y.z') to specify eager loading of all properties in the given path. ¶
- **[orm]** Major overhaul for Session: ¶
- **[orm]** New function which "configures" a session called "sessionmaker()". Send various keyword arguments to this function once, returns a new class which creates a Session against that stereotype. ¶
- **[orm]** SessionTransaction removed from "public" API. You now can call begin()/ commit()/rollback() on the Session itself. ¶
- **[orm]** Session also supports SAVEPOINT transactions; call begin_nested(). ¶
- **[orm]** Session supports two-phase commit behavior when vertically or horizontally partitioning (i.e., using more than one engine). Use twophase=True. ¶
- **[orm]** Session flag "transactional=True" produces a session which always places itself into a transaction when first used. Upon commit(), rollback() or close(), the transaction ends; but begins again on the next usage. ¶
- **[orm]** Session supports "autoflush=True". This issues a flush() before each query. Use in conjunction with transactional, and you can just save()/update() and then query, the new objects will be there. Use commit() at the end (or flush() if non-transactional) to flush remaining changes. ¶
- **[orm]** New scoped_session() function replaces SessionContext and assignmapper. Builds onto "sessionmaker()" concept to produce a class whos Session() construction returns the thread-local session. Or, call all Session methods as class methods, i.e. Session.save(foo); Session.commit(). just like the old "objectstore" days. ¶
- **[orm]** Added new "binds" argument to Session to support configuration of multiple binds with sessionmaker() function. ¶
- **[orm]** A rudimental SessionExtension class has been added, allowing user-defined functionality to take place at flush(), commit(), and rollback() boundaries. ¶
- **[orm]** Query-based relation()s available with dynamic_loader(). This is a *writable* collection (supporting append() and remove()) which is also a live Query object when accessed for reads. Ideal for dealing with very large collections where only partial loading is desired. ¶
- **[orm]** flush()-embedded inline INSERT/UPDATE expressions. Assign any SQL expression, like "sometable.c.column + 1", to an instance's attribute. Upon flush(), the mapper detects the expression and em-

beds it directly in the INSERT or UPDATE statement; the attribute gets deferred on the instance so it loads the new value the next time you access it. ¶

- **[orm]** A rudimental sharding (horizontal scaling) system is introduced. This system uses a modified Session which can distribute read and write operations among multiple databases, based on user-defined functions defining the "sharding strategy". Instances and their dependents can be distributed and queried among multiple databases based on attribute values, round-robin approaches or any other user-defined system. ¶ References: #618
- **[orm]** Eager loading has been enhanced to allow even more joins in more places. It now functions at any arbitrary depth along self-referential and cyclical structures. When loading cyclical structures, specify "join_depth" on relation() indicating how many times you'd like the table to join to itself; each level gets a distinct table alias. The alias names themselves are generated at compile time using a simple counting scheme now and are a lot easier on the eyes, as well as of course completely deterministic. ¶ References: #659
- **[orm]** Added composite column properties. This allows you to create a type which is represented by more than one column, when using the ORM. Objects of the new type are fully functional in query expressions, comparisons, query.get() clauses, etc. and act as though they are regular single-column scalars... except they're not! Use the function composite(cls, *columns) inside of the mapper's "properties" dict, and instances of cls will be created/mapped to a single attribute, comprised of the values corresponding to *columns. ¶ References: #211
- **[orm]** Improved support for custom column_property() attributes which feature correlated subqueries, works better with eager loading now. ¶
- **[orm]** Primary key "collapse" behavior; the mapper will analyze all columns in its given selectable for primary key "equivalence", that is, columns which are equivalent via foreign key relationship or via an explicit inherit_condition. primarily for joined-table inheritance scenarios where different named PK columns in inheriting tables should "collapse" into a single-valued (or fewer-valued) primary key. Fixes things like. ¶ References: #611
- **[orm]** Joined-table inheritance will now generate the primary key columns of all inherited classes against the root table of the join only. This implies that each row in the root table is distinct to a single instance. If for some rare reason this is not desirable, explicit primary_key settings on individual mappers will override it. ¶
- **[orm]** When "polymorphic" flags are used with joined-table or single-table inheritance, all identity keys are generated against the root class of the inheritance hierarchy; this allows query.get() to work polymorphically using the same caching semantics as a non-polymorphic get. Note that this currently does not work with concrete inheritance. ¶
- **[orm]** Secondary inheritance loading: polymorphic mappers can be constructed *without* a select_table argument. inheriting mappers whose tables were not represented in the initial load will issue a second SQL query immediately, once per instance (i.e. not very efficient for large lists), in order to load the remaining columns. ¶
- **[orm]** Secondary inheritance loading can also move its second query into a column-level "deferred" load, via the "polymorphic_fetch" argument, which can be set to 'select' or 'deferred' ¶
- **[orm]** It's now possible to map only a subset of available selectable columns onto mapper properties, using include_columns/exclude_columns.. ¶ References: #696
- **[orm]** Added undefer_group() MapperOption, sets a set of "deferred" columns joined by a "group" to load as "undeferred". ¶
- **[orm]** Rewrite of the "deterministic alias name" logic to be part of the SQL layer, produces much simpler alias and label names more in the style of Hibernate ¶

sql

- **[sql]** Speed! Clause compilation as well as the mechanics of SQL constructs have been streamlined and simplified to a significant degree, for a 20-30% improvement of the statement construction/compilation overhead of 0.3. ¶
- **[sql]** All "type" keyword arguments, such as those to bindparam(), column(), Column(), and func.<something>(), renamed to "type_". Those objects still name their "type" attribute as "type". ¶
- **[sql]** case_sensitive=(True|False) setting removed from schema items, since checking this state added a lot of method call overhead and there was no decent reason to ever set it to False. Table and column names which are all lower case will be treated as case-insensitive (yes we adjust for Oracle's UPPERCASE style too). ¶

mysql

- **[mysql]** Table and column names loaded via reflection are now Unicode. ¶
- **[mysql]** All standard column types are now supported, including SET. ¶
- **[mysql]** Table reflection can now be performed in as little as one round-trip. ¶
- **[mysql]** ANSI and ANSI_QUOTES sql modes are now supported. ¶
- **[mysql]** Indexes are now reflected. ¶

oracle

- **[oracle]** Very rudimental support for OUT parameters added; use sql.outparam(name, type) to set up an OUT parameter, just like bindparam(); after execution, values are available via result.out_parameters dictionary. ¶ References: #507

misc

- **[transactions]** Added context manager (with statement) support for transactions. ¶
- **[transactions]** Added support for two phase commit, works with mysql and postgres so far. ¶
- **[transactions]** Added a subtransaction implementation that uses savepoints. ¶
- **[transactions]** Added support for savepoints. ¶
- **[metadata]** Tables can be reflected from the database en-masse without declaring them in advance. MetaData(engine, reflect=True) will load all tables present in the database, or use metadata.reflect() for finer control. ¶
- **[metadata]** DynamicMetaData has been renamed to ThreadLocalMetaData ¶
- **[metadata]** The ThreadLocalMetaData constructor now takes no arguments. ¶
- **[metadata]** BoundMetaData has been removed- regular MetaData is equivalent ¶
- **[metadata]** Numeric and Float types now have an "asdecimal" flag; defaults to True for Numeric, False for Float. When True, values are returned as decimal.Decimal objects; when False, values are returned as float(). The defaults of True/False are already the behavior for PG and MySQL's DBAPI modules. ¶ References: #646
- **[metadata]** New SQL operator implementation which removes all hardcoded operators from expression structures and moves them into compilation; allows greater flexibility of operator compilation; for example, "+"

compiles to "||" when used in a string context, or "concat(a,b)" on MySQL; whereas in a numeric context it compiles to "+". Fixes. ¶ References: #475

- **[metadata]** "Anonymous" alias and label names are now generated at SQL compilation time in a completely deterministic fashion... no more random hex IDs ¶
- **[metadata]** Significant architectural overhaul to SQL elements (ClauseElement). All elements share a common "mutability" framework which allows a consistent approach to in-place modifications of elements as well as generative behavior. Improves stability of the ORM which makes heavy usage of mutations to SQL expressions. ¶
- **[metadata]** select() and union()'s now have "generative" behavior. Methods like order_by() and group_by() return a *new* instance - the original instance is left unchanged. Non-generative methods remain as well. ¶
- **[metadata]** The internals of select/union vastly simplified- all decision making regarding "is subquery" and "correlation" pushed to SQL generation phase. select() elements are now *never* mutated by their enclosing containers or by any dialect's compilation process ¶ References: #569, #52
- **[metadata]** select(scalar=True) argument is deprecated; use select(..).as_scalar(). The resulting object obeys the full "column" interface and plays better within expressions. ¶
- **[metadata]** Added select().with_prefix('foo') allowing any set of keywords to be placed before the columns clause of the SELECT ¶ References: #504
- **[metadata]** Added array slice support to row[<index>] ¶ References: #686
- **[metadata]** Result sets make a better attempt at matching the DBAPI types present in cursor.description to the TypeEngine objects defined by the dialect, which are then used for result-processing. Note this only takes effect for textual SQL; constructed SQL statements always have an explicit type map. ¶
- **[metadata]** Result sets from CRUD operations close their underlying cursor immediately and will also auto-close the connection if defined for the operation; this allows more efficient usage of connections for successive CRUD operations with less chance of "dangling connections". ¶
- **[metadata]** Column defaults and onupdate Python functions (i.e. passed to ColumnDefault) may take zero or one arguments; the one argument is the ExecutionContext, from which you can call "context.parameters[someparam]" to access the other bind parameter values affixed to the statement. The connection used for the execution is available as well so that you can pre-execute statements. ¶ References: #559
- **[metadata]** Added "explcit" create/drop/execute support for sequences (i.e. you can pass a "connectable" to each of those methods on Sequence). ¶
- **[metadata]** Better quoting of identifiers when manipulating schemas. ¶
- **[metadata]** Standardized the behavior for table reflection where types can't be located; NullType is substituted instead, warning is raised. ¶
- **[metadata]** ColumnCollection (i.e. the 'c' attribute on tables) follows dictionary semantics for "__contains__" ¶ References: #606
- **[engines]** Speed! The mechanics of result processing and bind parameter processing have been overhauled, streamlined and optimized to issue as little method calls as possible. Bench tests for mass INSERT and mass rowset iteration both show 0.4 to be over twice as fast as 0.3, using 68% fewer function calls. ¶
- **[engines]** You can now hook into the pool lifecycle and run SQL statements or other logic at new each DBAPI connection, pool check-out and check-in. ¶
- **[engines]** Connections gain a .properties collection, with contents scoped to the lifetime of the underlying DBAPI connection ¶
- **[engines]** Removed auto_close_cursors and disallow_open_cursors arguments from Pool; reduces overhead as cursors are normally closed by ResultProxy and Connection. ¶

- **[extensions]** proxyengine is temporarily removed, pending an actually working replacement. ¶
- **[extensions]** SelectResults has been replaced by Query. SelectResults / SelectResultsExt still exist but just return a slightly modified Query object for backwards-compatibility. join_to() method from SelectResults isn't present anymore, need to use join(). ¶
- **[postgres]** Added PGArray datatype for using postgres array datatypes. ¶

6.2.9 0.3 Changelog

0.3.11

Released: Sun Oct 14 2007

orm

- **[orm]** added a check for joining from A->B using join(), along two different m2m tables. this raises an error in 0.3 but is possible in 0.4 when aliases are used. ¶ References: #687
- **[orm]** fixed small exception throw bug in Session.merge() ¶
- **[orm]** fixed bug where mapper, being linked to a join where one table had no PK columns, would not detect that the joined table had no PK. ¶
- **[orm]** fixed bugs in determining proper sync clauses from custom inherit conditions ¶ References: #769
- **[orm]** backref remove object operation doesn't fail if the other-side collection doesn't contain the item, supports noload collections ¶ References: #813

engine

- **[engine]** fixed another occasional race condition which could occur when using pool with threadlocal setting ¶

sql

- **[sql]** tweak DISTINCT precedence for clauses like *func.count(t.c.col.distinct())* ¶
- **[sql]** Fixed detection of internal '$' characters in :bind$params ¶ References: #719
- **[sql]** don't assume join criterion consists only of column objects ¶ References: #768
- **[sql]** adjusted operator precedence of NOT to match '==' and others, so that ~(x==y) produces NOT (x=y), which is compatible with MySQL < 5.0 (doesn't like "NOT x=y") ¶ References: #764

mysql

- **[mysql]** fixed specification of YEAR columns when generating schema ¶

sqlite

- **[sqlite]** passthrough for stringified dates ¶

mssql

- **[mssql]** added support for TIME columns (simulated using DATETIME) ¶ References: #679
- **[mssql]** added support for BIGINT, MONEY, SMALLMONEY, UNIQUEIDENTIFIER and SQL_VARIANT ¶ References: #721
- **[mssql]** index names are now quoted when dropping from reflected tables ¶ References: #684
- **[mssql]** can now specify a DSN for PyODBC, using a URI like mssql:///?dsn=bob ¶

oracle

- **[oracle]** removed LONG_STRING, LONG_BINARY from "binary" types, so type objects don't try to read their values as LOB. ¶ References: #622, #751

firebird

- **[firebird]** supports_sane_rowcount() set to False due to ticket #370 (right way). ¶
- **[firebird]** fixed reflection of Column's nullable property. ¶

misc

- **[postgres]** when reflecting tables from alternate schemas, the "default" placed upon the primary key, i.e. usually a sequence name, has the "schema" name unconditionally quoted, so that schema names which need quoting are fine. its slightly unnecessary for schema names which don't need quoting but not harmful. ¶

0.3.10

Released: Fri Jul 20 2007

general

- **[general]** a new mutex that was added in 0.3.9 causes the pool_timeout feature to fail during a race condition; threads would raise TimeoutError immediately with no delay if many threads push the pool into overflow at the same time. this issue has been fixed. ¶

orm

- **[orm]** cleanup to connection-bound sessions, SessionTransaction ¶

sql

- **[sql]** got connection-bound metadata to work with implicit execution ¶
- **[sql]** foreign key specs can have any chararcter in their identifiers ¶ References: #667
- **[sql]** added commutativity-awareness to binary clause comparisons to each other, improves ORM lazy load optimization ¶ References: #664

misc

- **[postgres]** fixed max identifier length (63) ¶ References: #571

0.3.9

Released: Sun Jul 15 2007

general

- **[general]** better error message for NoSuchColumnError ¶ References: #607
- **[general]** finally figured out how to get setuptools version in, available as sqlalchemy.__version__ ¶ References: #428
- **[general]** the various "engine" arguments, such as "engine", "connectable", "engine_or_url", "bind_to", etc. are all present, but deprecated. they all get replaced by the single term "bind". you also set the "bind" of MetaData using metadata.bind = <engine or connection> ¶

orm

- **[orm]** forwards-compatibility with 0.4: added one(), first(), and all() to Query. almost all Query functionality from 0.4 is present in 0.3.9 for forwards-compat purposes. ¶
- **[orm]** reset_joinpoint() really really works this time, promise ! lets you re-join from the root: query.join(['a', 'b']).filter(<crit>).reset_joinpoint().join(['a', 'c']).filter(<some other crit>).all() in 0.4 all join() calls start from the "root" ¶
- **[orm]** added synchronization to the mapper() construction step, to avoid thread collisions when pre-existing mappers are compiling in a different thread ¶ References: #613
- **[orm]** a warning is issued by Mapper when two primary key columns of the same name are munged into a single attribute. this happens frequently when mapping to joins (or inheritance). ¶
- **[orm]** synonym() properties are fully supported by all Query joining/ with_parent operations ¶ References: #598
- **[orm]** fixed very stupid bug when deleting items with many-to-many uselist=False relations ¶
- **[orm]** remember all that stuff about polymorphic_union ? for joined table inheritance ? Funny thing... You sort of don't need it for joined table inheritance, you can just string all the tables together via outerjoin(). The UNION still applies if concrete tables are involved, though (since nothing to join them on). ¶
- **[orm]** small fix to eager loading to better work with eager loads to polymorphic mappers that are using a straight "outerjoin" clause ¶

sql

- **[sql]** ForeignKey to a table in a schema that's not the default schema requires the schema to be explicit; i.e. ForeignKey('alt_schema.users.id') ¶
- **[sql]** MetaData can now be constructed with an engine or url as the first argument, just like BoundMetaData ¶
- **[sql]** BoundMetaData is now deprecated, and MetaData is a direct substitute. ¶

- **[sql]** DynamicMetaData has been renamed to ThreadLocalMetaData. the DynamicMetaData name is deprecated and is an alias for ThreadLocalMetaData or a regular MetaData if threadlocal=False ¶
- **[sql]** composite primary key is represented as a non-keyed set to allow for composite keys consisting of cols with the same name; occurs within a Join. helps inheritance scenarios formulate correct PK. ¶
- **[sql]** improved ability to get the "correct" and most minimal set of primary key columns from a join, equating foreign keys and otherwise equated columns. this is also mostly to help inheritance scenarios formulate the best choice of primary key columns. ¶ References: #185
- **[sql]** added 'bind' argument to Sequence.create()/drop(), ColumnDefault.execute() ¶
- **[sql]** columns can be overridden in a reflected table with a "key" attribute different than the column's name, including for primary key columns ¶ References: #650
- **[sql]** fixed "ambiguous column" result detection, when dupe col names exist in a result ¶ References: #657
- **[sql]** some enhancements to "column targeting", the ability to match a column to a "corresponding" column in another selectable. this affects mostly ORM ability to map to complex joins ¶
- **[sql]** MetaData and all SchemaItems are safe to use with pickle. slow table reflections can be dumped into a pickled file to be reused later. Just reconnect the engine to the metadata after unpickling. ¶ References: #619
- **[sql]** added a mutex to QueuePool's "overflow" calculation to prevent a race condition that can bypass max_overflow ¶
- **[sql]** fixed grouping of compound selects to give correct results. will break on sqlite in some cases, but those cases were producing incorrect results anyway, sqlite doesn't support grouped compound selects ¶ References: #623
- **[sql]** fixed precedence of operators so that parenthesis are correctly applied ¶ References: #620
- **[sql]** calling <column>.in_() (i.e. with no arguments) will return "CASE WHEN (<column> IS NULL) THEN NULL ELSE 0 END = 1)", so that NULL or False is returned in all cases, rather than throwing an error ¶ References: #545
- **[sql]** fixed "where"/"from" criterion of select() to accept a unicode string in addition to regular string - both convert to text() ¶
- **[sql]** added standalone distinct() function in addition to column.distinct() ¶ References: #558
- **[sql]** result.last_inserted_ids() should return a list that is identically sized to the primary key constraint of the table. values that were "passively" created and not available via cursor.lastrowid will be None. ¶
- **[sql]** long-identifier detection fixed to use > rather than >= for max ident length ¶ References: #589
- **[sql]** fixed bug where selectable.corresponding_column(selectable.c.col) would not return selectable.c.col, if the selectable is a join of a table and another join involving the same table. messed up ORM decision making ¶ References: #593
- **[sql]** added Interval type to types.py ¶ References: #595

mysql

- **[mysql]** fixed catching of some errors that imply a dropped connection ¶ References: #625
- **[mysql]** fixed escaping of the modulo operator ¶ References: #624
- **[mysql]** added 'fields' to reserved words ¶ References: #590
- **[mysql]** various reflection enhancement/fixes ¶

sqlite

- **[sqlite]** rearranged dialect initialization so it has time to warn about pysqlite1 being too old. ¶
- **[sqlite]** sqlite better handles datetime/date/time objects mixed and matched with various Date/Time/DateTime columns ¶
- **[sqlite]** string PK column inserts don't get overwritten with OID ¶ References: #603

mssql

- **[mssql]** fix port option handling for pyodbc ¶ References: #634
- **[mssql]** now able to reflect start and increment values for identity columns ¶
- **[mssql]** preliminary support for using scope_identity() with pyodbc ¶

oracle

- **[oracle]** datetime fixes: got subsecond TIMESTAMP to work, added OracleDate which supports types.Date with only year/month/day ¶ References: #604
- **[oracle]** added dialect flag "auto_convert_lobs", defaults to True; will cause any LOB objects detected in a result set to be forced into OracleBinary so that the LOB is read() automatically, if no typemap was present (i.e., if a textual execute() was issued). ¶
- **[oracle]** mod operator '%' produces MOD ¶ References: #624
- **[oracle]** converts cx_oracle datetime objects to Python datetime.datetime when Python 2.3 used ¶ References: #542
- **[oracle]** fixed unicode conversion in Oracle TEXT type ¶

misc

- **[ext]** iteration over dict association proxies is now dict-like, not InstrumentedList-like (e.g. over keys instead of values) ¶
- **[ext]** association proxies no longer bind tightly to source collections, and are constructed with a thunk instead ¶ References: #597
- **[ext]** added selectone_by() to assignmapper ¶
- **[postgres]** fixed escaping of the modulo operator ¶ References: #624
- **[postgres]** added support for reflection of domains ¶ References: #570
- **[postgres]** types which are missing during reflection resolve to Null type instead of raising an error ¶
- **[postgres]** the fix in "schema" above fixes reflection of foreign keys from an alt-schema table to a public schema table ¶

0.3.8

Released: Sat Jun 02 2007

orm

- **[orm]** added reset_joinpoint() method to Query, moves the "join point" back to the starting mapper. 0.4 will change the behavior of join() to reset the "join point" in all cases so this is an interim method. for forwards compatibility, ensure joins across multiple relations are specified using a single join(), i.e. join(['a', 'b', 'c']). ¶
- **[orm]** fixed bug in query.instances() that wouldn't handle more than on additional mapper or one additional column. ¶
- **[orm]** "delete-orphan" no longer implies "delete". ongoing effort to separate the behavior of these two operations. ¶
- **[orm]** many-to-many relationships properly set the type of bind params for delete operations on the association table ¶
- **[orm]** many-to-many relationships check that the number of rows deleted from the association table by a delete operation matches the expected results ¶
- **[orm]** session.get() and session.load() propagate **kwargs through to query ¶
- **[orm]** fix to polymorphic query which allows the original polymorphic_union to be embedded into a correlated subquery ¶ References: #577
- **[orm]** fix to select_by(<propname>=<object instance>) -style joins in conjunction with many-to-many relationships, bug introduced in r2556 ¶
- **[orm]** the "primary_key" argument to mapper() is propagated to the "polymorphic" mapper. primary key columns in this list get normalized to that of the mapper's local table. ¶
- **[orm]** restored logging of "lazy loading clause" under sa.orm.strategies logger, got removed in 0.3.7 ¶
- **[orm]** improved support for eagerloading of properties off of mappers that are mapped to select() statements; i.e. eagerloader is better at locating the correct selectable with which to attach its LEFT OUTER JOIN. ¶

sql

- **[sql]** _Label class overrides compare_self to return its ultimate object. meaning, if you say someexpr.label('foo') == 5, it produces the correct "someexpr == 5". ¶
- **[sql]** _Label propagates "_hide_froms()" so that scalar selects behave more properly with regards to FROM clause #574 ¶
- **[sql]** fix to long name generation when using oid_column as an order by (oids used heavily in mapper queries) ¶
- **[sql]** significant speed improvement to ResultProxy, pre-caches TypeEngine dialect implementations and saves on function calls per column ¶
- **[sql]** parenthesis are applied to clauses via a new _Grouping construct. uses operator precedence to more intelligently apply parenthesis to clauses, provides cleaner nesting of clauses (doesn't mutate clauses placed in other clauses, i.e. no 'parens' flag) ¶
- **[sql]** added 'modifier' keyword, works like func.<foo> except does not add parenthesis. e.g. select([modifier.DISTINCT(...)]) etc. ¶
- **[sql]** removed "no group by's in a select that's part of a UNION" restriction ¶ References: #578

mysql

- **[mysql]** Nearly all MySQL column types are now supported for declaration and reflection. Added NCHAR, NVARCHAR, VARBINARY, TINYBLOB, LONGBLOB, YEAR ¶
- **[mysql]** The sqltypes.Binary passthrough now always builds a BLOB, avoiding problems with very old database versions ¶
- **[mysql]** support for column-level CHARACTER SET and COLLATE declarations, as well as ASCII, UNICODE, NATIONAL and BINARY shorthand. ¶

firebird

- **[firebird]** set max identifier length to 31 ¶
- **[firebird]** supports_sane_rowcount() set to False due to ticket #370. versioned_id_col feature wont work in FB. ¶
- **[firebird]** some execution fixes ¶
- **[firebird]** new association proxy implementation, implementing complete proxies to list, dict and set-based relation collections ¶
- **[firebird]** added orderinglist, a custom list class that synchronizes an object attribute with that object's position in the list ¶
- **[firebird]** small fix to SelectResultsExt to not bypass itself during select(). ¶
- **[firebird]** added filter(), filter_by() to assignmapper ¶

misc

- **[engines]** added detach() to Connection, allows underlying DBAPI connection to be detached from its pool, closing on dereference/close() instead of being reused by the pool. ¶
- **[engines]** added invalidate() to Connection, immediately invalidates the Connection and its underlying DBAPI connection. ¶

0.3.7

Released: Sun Apr 29 2007

orm

- **[orm]** fixed critical issue when, after options(eagerload()) is used, the mapper would then always apply query "wrapping" behavior for all subsequent LIMIT/OFFSET/DISTINCT queries, even if no eager loading was applied on those subsequent queries. ¶
- **[orm]** added query.with_parent(someinstance) method. searches for target instance using lazy join criterion from parent instance. takes optional string "property" to isolate the desired relation. also adds static Query.query_from_parent(instance, property) version. ¶ References: #541
- **[orm]** improved query.XXX_by(someprop=someinstance) querying to use similar methodology to with_parent, i.e. using the "lazy" clause which prevents adding the remote instance's table to the SQL, thereby making more complex conditions possible ¶ References: #554

- **[orm]** added generative versions of aggregates, i.e. sum(), avg(), etc. to query. used via query.apply_max(), apply_sum(), etc. #552 ¶
- **[orm]** fix to using distinct() or distinct=True in combination with join() and similar ¶
- **[orm]** corresponding to label/bindparam name generation, eager loaders generate deterministic names for the aliases they create using md5 hashes. ¶
- **[orm]** improved/fixed custom collection classes when giving it "set"/ "sets.Set" classes or subclasses (was still looking for append() methods on them during lazy loads) ¶
- **[orm]** restored old "column_property()" ORM function (used to be called "column()") to force any column expression to be added as a property on a mapper, particularly those that aren't present in the mapped selectable. this allows "scalar expressions" of any kind to be added as relations (though they have issues with eager loads). ¶
- **[orm]** fix to many-to-many relationships targeting polymorphic mappers ¶ References: #533
- **[orm]** making progress with session.merge() as well as combining its usage with entity_name ¶ References: #543
- **[orm]** the usual adjustments to relationships between inheriting mappers, in this case establishing relation()s to subclass mappers where the join conditions come from the superclass' table ¶

sql

- **[sql]** keys() of result set columns are not lowercased, come back exactly as they're expressed in cursor.description. note this causes colnames to be all caps in oracle. ¶
- **[sql]** preliminary support for unicode table names, column names and SQL statements added, for databases which can support them. Works with sqlite and postgres so far. Mysql *mostly* works except the has_table() function does not work. Reflection works too. ¶
- **[sql]** the Unicode type is now a direct subclass of String, which now contains all the "convert_unicode" logic. This helps the variety of unicode situations that occur in db's such as MS-SQL to be better handled and allows subclassing of the Unicode datatype. ¶ References: #522
- **[sql]** ClauseElements can be used in in_() clauses now, such as bind parameters, etc. #476 ¶
- **[sql]** reverse operators implemented for *CompareMixin* elements, allows expressions like "5 + somecolumn" etc. #474 ¶
- **[sql]** the "where" criterion of an update() and delete() now correlates embedded select() statements against the table being updated or deleted. this works the same as nested select() statement correlation, and can be disabled via the correlate=False flag on the embedded select(). ¶
- **[sql]** column labels are now generated in the compilation phase, which means their lengths are dialect-dependent. So on oracle a label that gets truncated to 30 chars will go out to 63 characters on postgres. Also, the true labelname is always attached as the accessor on the parent Selectable so there's no need to be aware of the "truncated" label names. ¶ References: #512
- **[sql]** column label and bind param "truncation" also generate deterministic names now, based on their ordering within the full statement being compiled. this means the same statement will produce the same string across application restarts and allowing DB query plan caching to work better. ¶
- **[sql]** the "mini" column labels generated when using subqueries, which are to work around glitchy SQLite behavior that doesn't understand "foo.id" as equivalent to "id", are now only generated in the case that those named columns are selected from (part of) ¶ References: #513
- **[sql]** the label() method on ColumnElement will properly propagate the TypeEngine of the base element out to the label, including a label() created from a scalar=True select() statement. ¶

- **[sql]** MS-SQL better detects when a query is a subquery and knows not to generate ORDER BY phrases for those ¶ References: #513
- **[sql]** fix for fetchmany() "size" argument being positional in most dbapis ¶ References: #505
- **[sql]** sending None as an argument to func.<something> will produce an argument of NULL ¶
- **[sql]** query strings in unicode URLs get keys encoded to ascii for **kwargs compat ¶
- **[sql]** slight tweak to raw execute() change to also support tuples for positional parameters, not just lists ¶ References: #523
- **[sql]** fix to case() construct to propagate the type of the first WHEN condition as the return type of the case statement ¶

mysql

- **[mysql]** support for SSL arguments given as inline within URL query string, prefixed with "ssl_", courtesy terjeros@gmail.com. ¶
- **[mysql] [<schemaname>]** mysql uses "DESCRIBE.<tablename>", catching exceptions if table doesn't exist, in order to determine if a table exists. this supports unicode table names as well as schema names. tested with MySQL5 but should work with 4.1 series as well. (#557) ¶

sqlite

- **[sqlite]** removed silly behavior where sqlite would reflect UNIQUE indexes as part of the primary key (?!) ¶

mssql

- **[mssql]** pyodbc is now the preferred DB-API for MSSQL, and if no module is specifically requested, will be loaded first on a module probe. ¶
- **[mssql]** The @@SCOPE_IDENTITY is now used instead of @@IDENTITY. This behavior may be overridden with the engine_connect "use_scope_identity" keyword parameter, which may also be specified in the dburi. ¶

oracle

- **[oracle]** small fix to allow successive compiles of the same SELECT object which features LIMIT/OFFSET. oracle dialect needs to modify the object to have ROW_NUMBER OVER and wasn't performing the full series of steps on successive compiles. ¶

misc

- **[engines]** warnings module used for issuing warnings (instead of logging) ¶
- **[engines]** cleanup of DBAPI import strategies across all engines ¶ References: #480
- **[engines]** refactoring of engine internals which reduces complexity, number of codepaths; places more state inside of ExecutionContext to allow more dialect control of cursor handling, result sets. ResultProxy totally refactored and also has two versions of "buffered" result sets used for different purposes. ¶
- **[engines]** server side cursor support fully functional in postgres. ¶ References: #514

- **[engines]** improved framework for auto-invalidation of connections that have lost their underlying database, via dialect-specific detection of exceptions corresponding to that database's disconnect related error messages. Additionally, when a "connection no longer open" condition is detected, the entire connection pool is discarded and replaced with a new instance. #516 ¶
- **[engines]** the dialects within sqlalchemy.databases become a setuptools entry points. loading the built-in database dialects works the same as always, but if none found will fall back to trying pkg_resources to load an external module ¶ References: #521
- **[engines]** Engine contains a "url" attribute referencing the url.URL object used by create_engine(). ¶
- **[informix]** informix support added ! courtesy James Zhang, who put a ton of effort in. ¶
- **[extensions]** big fix to AssociationProxy so that multiple AssociationProxy objects can be associated with a single association collection. ¶
- **[extensions]** assign_mapper names methods according to their keys (i.e. __name__) #551 ¶

0.3.6

Released: Fri Mar 23 2007

orm

- **[orm]** the full featureset of the SelectResults extension has been merged into a new set of methods available off of Query. These methods all provide "generative" behavior, whereby the Query is copied and a new one returned with additional criterion added. The new methods include:
  - filter() - applies select criterion to the query
  - filter_by() - applies "by"-style criterion to the query
  - avg() - return the avg() function on the given column
  - join() - join to a property (or across a list of properties)
  - outerjoin() - like join() but uses LEFT OUTER JOIN
  - limit()/offset() - apply LIMIT/OFFSET range-based access which applies limit/offset: session.query(Foo)[3:5]
  - distinct() - apply DISTINCT
  - list() - evaluate the criterion and return results

  no incompatible changes have been made to Query's API and no methods have been deprecated. Existing methods like select(), select_by(), get(), get_by() all execute the query at once and return results like they always did. join_to()/join_via() are still there although the generative join()/outerjoin() methods are easier to use. ¶
- **[orm]** the return value for multiple mappers used with instances() now returns a cartesian product of the requested list of mappers, represented as a list of tuples. this corresponds to the documented behavior. So that instances match up properly, the "uniquing" is disabled when this feature is used. ¶
- **[orm]** Query has add_entity() and add_column() generative methods. these will add the given mapper/class or ColumnElement to the query at compile time, and apply them to the instances() method. the user is responsible for constructing reasonable join conditions (otherwise you can get full cartesian products). result set is the list of tuples, non-uniqued. ¶
- **[orm]** strings and columns can also be sent to the *args of instances() where those exact result columns will be part of the result tuples. ¶

- **[orm]** a full select() construct can be passed to query.select() (which worked anyway), but also query.selectfirst(), query.selectone() which will be used as is (i.e. no query is compiled). works similarly to sending the results to instances(). ¶
- **[orm]** eager loading will not "aliasize" "order by" clauses that were placed in the select statement by something other than the eager loader itself, to fix possibility of dupe columns as illustrated in. however, this means you have to be more careful with the columns placed in the "order by" of Query.select(), that you have explicitly named them in your criterion (i.e. you cant rely on the eager loader adding them in for you) ¶ References: #495
- **[orm]** added a handy multi-use "identity_key()" method to Session, allowing the generation of identity keys for primary key values, instances, and rows, courtesy Daniel Miller ¶
- **[orm]** many-to-many table will be properly handled even for operations that occur on the "backref" side of the operation ¶ References: #249
- **[orm]** added "refresh-expire" cascade. allows refresh() and expire() calls to propagate along relationships. ¶ References: #492
- **[orm]** more fixes to polymorphic relations, involving proper lazy-clause generation on many-to-one relationships to polymorphic mappers. also fixes to detection of "direction", more specific targeting of columns that belong to the polymorphic union vs. those that don't. ¶ References: #493
- **[orm]** some fixes to relationship calcs when using "viewonly=True" to pull in other tables into the join condition which arent parent of the relationship's parent/child mappings ¶
- **[orm]** flush fixes on cyclical-referential relationships that contain references to other instances outside of the cyclical chain, when some of the objects in the cycle are not actually part of the flush ¶
- **[orm]** put an aggressive check for "flushing object A with a collection of B's, but you put a C in the collection" error condition - **even if C is a subclass of B**, unless B's mapper loads polymorphically. Otherwise, the collection will later load a "B" which should be a "C" (since its not polymorphic) which breaks in bi-directional relationships (i.e. C has its A, but A's backref will lazyload it as a different instance of type "B") This check is going to bite some of you who do this without issues, so the error message will also document a flag "enable_typechecks=False" to disable this checking. But be aware that bi-directional relationships in particular become fragile without this check. ¶ References: #500

sql

- **[sql]** bindparam() names are now repeatable! specify two distinct bindparam()s with the same name in a single statement, and the key will be shared. proper positional/named args translate at compile time. for the old behavior of "aliasing" bind parameters with conflicting names, specify "unique=True" - this option is still used internally for all the auto-genererated (value-based) bind parameters. ¶
- **[sql]** slightly better support for bind params as column clauses, either via bindparam() or via literal(), i.e. select([literal('foo')]) ¶
- **[sql]** MetaData can bind to an engine either via "url" or "engine" kwargs to constructor, or by using connect() method. BoundMetaData is identical to MetaData except engine_or_url param is required. DynamicMetaData is the same and provides thread-local connections be default. ¶
- **[sql]** exists() becomes useable as a standalone selectable, not just in a WHERE clause, i.e. exists([columns], criterion).select() ¶
- **[sql]** correlated subqueries work inside of ORDER BY, GROUP BY ¶
- **[sql]** fixed function execution with explicit connections, i.e. conn.execute(func.dosomething()) ¶
- **[sql]** use_labels flag on select() wont auto-create labels for literal text column elements, since we can make no assumptions about the text. to create labels for literal columns, you can say "somecol AS somelabel", or use literal_column("somecol").label("somelabel") ¶

- **[sql]** quoting wont occur for literal columns when they are "proxied" into the column collection for their selectable (is_literal flag is propagated). literal columns are specified via literal_column("somestring"). ¶
- **[sql]** added "fold_equivalents" boolean argument to Join.select(), which removes 'duplicate' columns from the resulting column clause that are known to be equivalent based on the join condition. this is of great usage when constructing subqueries of joins which Postgres complains about if duplicate column names are present. ¶
- **[sql]** fixed use_alter flag on ForeignKeyConstraint ¶ References: #503
- **[sql]** fixed usage of 2.4-only "reversed" in topological.py ¶ References: #506
- **[sql]** for hackers, refactored the "visitor" system of ClauseElement and SchemaItem so that the traversal of items is controlled by the ClauseVisitor itself, using the method visitor.traverse(item). accept_visitor() methods can still be called directly but will not do any traversal of child items. ClauseElement/SchemaItem now have a configurable get_children() method to return the collection of child elements for each parent object. This allows the full traversal of items to be clear and unambiguous (as well as loggable), with an easy method of limiting a traversal (just pass flags which are picked up by appropriate get_children() methods). ¶ References: #501
- **[sql]** the "else_" parameter to the case statement now properly works when set to zero. ¶

mysql

- **[mysql]** added a catchall **kwargs to MSString, to help reflection of obscure types (like "varchar() binary" in MS 4.0) ¶
- **[mysql]** added explicit MSTimeStamp type which takes effect when using types.TIMESTAMP. ¶

oracle

- **[oracle]** got binary working for any size input ! cx_oracle works fine, it was my fault as BINARY was being passed and not BLOB for setinputsizes (also unit tests weren't even setting input sizes). ¶
- **[oracle]** also fixed CLOB read/write on a separate changeset. ¶
- **[oracle]** auto_setinputsizes defaults to True for Oracle, fixed cases where it improperly propagated bad types. ¶

misc

- **[extensions]** options() method on SelectResults now implemented "generatively" like the rest of the SelectResults methods. But you're going to just use Query now anyway. ¶ References: #472
- **[extensions]** query() method is added by assignmapper. this helps with navigating to all the new generative methods on Query. ¶
- **[ms-sql]**

  removed seconds input on DATE column types (probably should remove the time altogether)

  ¶
- **[ms-sql]** null values in float fields no longer raise errors ¶
- **[ms-sql]** LIMIT with OFFSET now raises an error (MS-SQL has no OFFSET support) ¶
- **[ms-sql]** added an facility to use the MSSQL type VARCHAR(max) instead of TEXT for large unsized string fields. Use the new "text_as_varchar" to turn it on. ¶ References: #509

- **[ms-sql]** ORDER BY clauses without a LIMIT are now stripped in subqueries, as MS-SQL forbids this usage ¶
- **[ms-sql]** cleanup of module importing code; specifiable DB-API module; more explicit ordering of module preferences. ¶ References: #480

0.3.5

Released: Thu Feb 22 2007

orm

- **[orm] [bugs]** another refactoring to relationship calculation. Allows more accurate ORM behavior with relationships from/to/between mappers, particularly polymorphic mappers, also their usage with Query, SelectResults. tickets include,,. ¶ References: #441, #448, #439
- **[orm] [bugs]** removed deprecated method of specifying custom collections on classes; you must now use the "collection_class" option. the old way was beginning to produce conflicts when people used assign_mapper(), which now patches an "options" method, in conjunction with a relationship named "options". (relationships take precedence over monkeypatched assign_mapper methods). ¶
- **[orm] [bugs]** extension() query option propagates to Mapper._instance() method so that all loading-related methods get called ¶ References: #454
- **[orm] [bugs]** eager relation to an inheriting mapper wont fail if no rows returned for the relationship. ¶
- **[orm] [bugs]** eager relation loading bug fixed for eager relation on multiple descendant classes ¶ References: #486
- **[orm] [bugs]** fix for very large topological sorts, courtesy ants.aasma at gmail ¶ References: #423
- **[orm] [bugs]** eager loading is slightly more strict about detecting "self-referential" relationships, specifically between polymorphic mappers. this results in an "eager degrade" to lazy loading. ¶
- **[orm] [bugs]** improved support for complex queries embedded into "where" criterion for query.select() ¶ References: #449
- **[orm] [bugs]** mapper options like eagerload(), lazyload(), deferred(), will work for "synonym()" relationships ¶ References: #485
- **[orm] [bugs]** fixed bug where cascade operations incorrectly included deleted collection items in the cascade ¶ References: #445
- **[orm] [bugs]** fixed relationship deletion error when one-to-many child item is moved to a new parent in a single unit of work ¶ References: #478
- **[orm] [bugs]** fixed relationship deletion error where parent/child with a single column as PK/FK on the child would raise a "blank out the primary key" error, if manually deleted or "delete" cascade without "delete-orphan" was used ¶
- **[orm] [bugs]** fix to deferred so that load operation doesn't mistakenly occur when only PK col attributes are set ¶
- **[orm] [enhancements]** implemented foreign_keys argument to mapper. use in conjunction with primaryjoin/secondaryjoin arguments to specify/override foreign keys defined on the Table instance. ¶ References: #385
- **[orm] [enhancements]** contains_eager('foo') automatically implies eagerload('foo') ¶

- **[orm] [enhancements]** added "alias" argument to contains_eager(). use it to specify the string name or Alias instance of an alias used in the query for the eagerly loaded child items. easier to use than "decorator" ¶
- **[orm] [enhancements]** added "contains_alias()" option for result set mapping to an alias of the mapped table ¶
- **[orm] [enhancements]** added support for py2.5 "with" statement with SessionTransaction ¶ References: #468

sql

- **[sql]** the value of "case_sensitive" defaults to True now, regardless of the casing of the identifier, unless specifically set to False. this is because the object might be label'ed as something else which does contain mixed case, and propigating "case_sensitive=False" breaks that. Other fixes to quoting when using labels and "fake" column objects ¶
- **[sql]** added a "supports_execution()" method to ClauseElement, so that individual kinds of clauses can express if they are appropriate for executing...such as, you can execute a "select", but not a "Table" or a "Join". ¶
- **[sql]** fixed argument passing to straight textual execute() on engine, connection. can handle *args or a list instance for positional, **kwargs or a dict instance for named args, or a list of list or dicts to invoke executemany() ¶
- **[sql]** small fix to BoundMetaData to accept unicode or string URLs ¶
- **[sql]** fixed named PrimaryKeyConstraint generation courtesy andrija at gmail ¶ References: #466
- **[sql]** fixed generation of CHECK constraints on columns ¶ References: #464
- **[sql]** fixes to tometadata() operation to propagate Constraints at column and table level ¶

mysql

- **[mysql]** fix to reflection on older DB's that might return array() type for "show variables like" statements ¶

mssql

- **[mssql]** preliminary support for pyodbc (Yay!) ¶ References: #419
- **[mssql]** better support for NVARCHAR types added ¶ References: #298
- **[mssql]** fix for commit logic on pymssql ¶
- **[mssql]** fix for query.get() with schema ¶ References: #456
- **[mssql]** fix for non-integer relationships ¶ References: #473
- **[mssql]** DB-API module now selectable at run-time ¶ References: #419
- **[mssql] [415] [tickets:422] [481]** now passes many more unit tests ¶
- **[mssql]** better unittest compatibility with ANSI functions ¶ References: #479
- **[mssql]** improved support for implicit sequence PK columns with auto-insert ¶ References: #415
- **[mssql]** fix for blank password in adodbapi ¶ References: #371
- **[mssql]** fixes to get unit tests working with pyodbc ¶ References: #481
- **[mssql]** fix to auto_identity_insert on db-url query ¶
- **[mssql]** added query_timeout to db-url query parms. currently works only for pymssql ¶

- **[mssql]** tested with pymssql 0.8.0 (which is now LGPL) ¶

oracle

- **[oracle]** when returning "rowid" as the ORDER BY column or in use with ROW_NUMBER OVER, oracle dialect checks the selectable its being applied to and will switch to table PK if not applicable, i.e. for a UNION. checking for DISTINCT, GROUP BY (other places that rowid is invalid) still a TODO. allows polymorphic mappings to function. ¶ References: #436
- **[oracle]** sequences on a non-pk column will properly fire off on INSERT ¶
- **[oracle]** added PrefetchingResultProxy support to pre-fetch LOB columns when they are known to be present, fixes ¶ References: #435
- **[oracle]** implemented reflection of tables based on synonyms, including across dblinks ¶ References: #379
- **[oracle]** issues a log warning when a related table cant be reflected due to certain permission errors ¶ References: #363

misc

- **[postgres]** better reflection of sequences for alternate-schema Tables ¶ References: #442
- **[postgres]** sequences on a non-pk column will properly fire off on INSERT ¶
- **[postgres]** added PGInterval type, PGInet type ¶ References: #460, #444
- **[extensions]** added distinct() method to SelectResults. generally should only make a difference when using count(). ¶
- **[extensions]** added options() method to SelectResults, equivalent to query.options() ¶ References: #472
- **[extensions]** added optional __table_opts__ dictionary to ActiveMapper, will send kw options to Table objects ¶ References: #462
- **[extensions]** added selectfirst(), selectfirst_by() to assign_mapper ¶ References: #467

0.3.4

Released: Tue Jan 23 2007

general

- **[general]** global "insure"->"ensure" change. in US english "insure" is actually largely interchangeable with "ensure" (so says the dictionary), so I'm not completely illiterate, but its definitely sub-optimal to "ensure" which is non-ambiguous. ¶

orm

- **[orm]** poked the first hole in the can of worms: saying query.select_by(somerelationname=someinstance) will create the join of the primary key columns represented by "somerelationname"'s mapper to the actual primary key in "someinstance". ¶

- **[orm]** reworked how relations interact with "polymorphic" mappers, i.e. mappers that have a select_table as well as polymorphic flags. better determination of proper join conditions, interaction with user- defined join conditions, and support for self-referential polymorphic mappers. ¶
- **[orm]** related to polymorphic mapping relations, some deeper error checking when compiling relations, to detect an ambiguous "primaryjoin" in the case that both sides of the relationship have foreign key references in the primary join condition. also tightened down conditions used to locate "relation direction", associating the "foreignkey" of the relationship with the "primaryjoin" ¶
- **[orm]** a little bit of improvement to the concept of a "concrete" inheritance mapping, though that concept is not well fleshed out yet (added test case to support concrete mappers on top of a polymorphic base). ¶
- **[orm]** fix to "proxy=True" behavior on synonym() ¶
- **[orm]** fixed bug where delete-orphan basically didn't work with many-to-many relationships, backref presence generally hid the symptom ¶ References: #427
- **[orm]** added a mutex to the mapper compilation step. ive been reluctant to add any kind of threading anything to SA but this is one spot that its really needed since mappers are typically "global", and while their state does not change during normal operation, the initial compilation step does modify internal state significantly, and this step usually occurs not at module-level initialization time (unless you call compile()) but at first-request time ¶
- **[orm]** basic idea of "session.merge()" actually implemented. needs more testing. ¶
- **[orm]** added "compile_mappers()" function as a shortcut to compiling all mappers ¶
- **[orm]** fix to MapperExtension create_instance so that entity_name properly associated with new instance ¶
- **[orm]** speed enhancements to ORM object instantiation, eager loading of rows ¶
- **[orm]** invalid options sent to 'cascade' string will raise an exception ¶ References: #406
- **[orm]** fixed bug in mapper refresh/expire whereby eager loaders didn't properly re-populate item lists ¶ References: #407
- **[orm]** fix to post_update to ensure rows are updated even for non insert/delete scenarios ¶ References: #413
- **[orm]** added an error message if you actually try to modify primary key values on an entity and then flush it ¶ References: #412

sql

- **[sql]** added "fetchmany()" support to ResultProxy ¶
- **[sql]** added support for column "key" attribute to be useable in row[<key>]/row.<key> ¶
- **[sql]** changed "BooleanExpression" to subclass from "BinaryExpression", so that boolean expressions can also follow column-clause behaviors (i.e. label(), etc). ¶
- **[sql]** trailing underscores are trimmed from func.<xxx> calls, such as func.if_() ¶
- **[sql]** fix to correlation of subqueries when the column list of the select statement is constructed with individual calls to append_column(); this fixes an ORM bug whereby nested select statements were not getting correlated with the main select generated by the Query object. ¶
- **[sql]** another fix to subquery correlation so that a subquery which has only one FROM element will *not* correlate that single element, since at least one FROM element is required in a query. ¶
- **[sql]** default "timezone" setting is now False. this corresponds to Python's datetime behavior as well as Postgres' timestamp/time types (which is the only timezone-sensitive dialect at the moment) ¶ References: #414

- **[sql]** the "op()" function is now treated as an "operation", rather than a "comparison". the difference is, an operation produces a BinaryExpression from which further operations can occur whereas comparison produces the more restrictive BooleanExpression ¶
- **[sql]** trying to redefine a reflected primary key column as non-primary key raises an error ¶
- **[sql]** type system slightly modified to support TypeDecorators that can be overridden by the dialect (ok, that's not very clear, it allows the mssql tweak below to be possible) ¶

mysql

- **[mysql]** mysql is inconsistent with what kinds of quotes it uses in foreign keys during a SHOW CREATE TABLE, reflection updated to accommodate for all three styles ¶ References: #420
- **[mysql]** mysql table create options work on a generic passthru now, i.e. Table(..., mysql_engine='InnoDB', mysql_collate="latin1_german2_ci", mysql_auto_increment="5", mysql_<somearg>...), helps ¶ References: #418

mssql

- **[mssql]** added an NVarchar type (produces NVARCHAR), also MSUnicode which provides Unicode-translation for the NVarchar regardless of dialect convert_unicode setting. ¶

oracle

- **[oracle]** *slight* support for binary, but still need to figure out how to insert reasonably large values (over 4K). requires auto_setinputsizes=True sent to create_engine(), rows must be fully fetched individually, etc. ¶

firebird

- **[firebird]** order of constraint creation puts primary key first before all other constraints; required for firebird, not a bad idea for others ¶ References: #408
- **[firebird]** Firebird fix to autoload multifield foreign keys ¶ References: #409
- **[firebird]** Firebird NUMERIC type properly handles a type without precision ¶ References: #409

misc

- **[postgres]** fix to the initial checkfirst for tables to take current schema into account ¶ References: #424
- **[postgres]** postgres has an optional "server_side_cursors=True" flag which will utilize server side cursors. these are appropriate for fetching only partial results and are necessary for working with very large unbounded result sets. While we'd like this to be the default behavior, different environments seem to have different results and the causes have not been isolated so we are leaving the feature off by default for now. Uses an apparently undocumented psycopg2 behavior recently discovered on the psycopg mailing list. ¶
- **[postgres]** added "BIGSERIAL" support for postgres table with PGBigInteger/autoincrement ¶
- **[postgres]** fixes to postgres reflection to better handle when schema names are present; thanks to jason (at) ncsmags.com ¶ References: #402
- **[extensions]** added "validate=False" argument to assign_mapper, if True will ensure that only mapped attributes are named ¶ References: #426

- **[extensions]** assign_mapper gets "options", "instances" functions added (i.e. MyClass.instances()) ¶

0.3.3

Released: Fri Dec 15 2006

- string-based FROM clauses fixed, i.e. select(..., from_obj=["sometext"]) ¶
- fixes to passive_deletes flag, lazy=None (noload) flag ¶
- added example/docs for dealing with large collections ¶
- added object_session() method to sqlalchemy namespace ¶
- fixed QueuePool bug whereby its better able to reconnect to a database that was not reachable (thanks to SÃ©bastien Lelong), also fixed dispose() method ¶
- patch that makes MySQL rowcount work correctly! ¶ References: #396
- fix to MySQL catch of 2006/2014 errors to properly re-raise OperationalError exception ¶

0.3.2

Released: Sun Dec 10 2006

- major connection pool bug fixed. fixes MySQL out of sync errors, will also prevent transactions getting rolled back accidentally in all DBs ¶ References: #387
- major speed enhancements vs. 0.3.1, to bring speed back to 0.2.8 levels ¶
- made conditional dozens of debug log calls that were time-intensive to generate log messages ¶
- fixed bug in cascade rules whereby the entire object graph could be unnecessarily cascaded on the save/update cascade ¶
- various speedups in attributes module ¶
- identity map in Session is by default *no longer weak referencing*. to have it be weak referencing, use create_session(weak_identity_map=True) fixes ¶ References: #388
- MySQL detects errors 2006 (server has gone away) and 2014 (commands out of sync) and invalidates the connection on which it occurred. ¶
- MySQL bool type fix: ¶ References: #307
- postgres reflection fixes: ¶ References: #382, #349
- added keywords for EXCEPT, INTERSECT, EXCEPT ALL, INTERSECT ALL ¶ References: #247
- assign_mapper in assignmapper extension returns the created mapper ¶ References: #2110
- added label() function to Select class, when scalar=True is used to create a scalar subquery i.e. "select x, y, (select max(foo) from table) AS foomax from table" ¶
- added onupdate and ondelete keyword arguments to ForeignKey; propagate to underlying ForeignKeyConstraint if present. (don't propagate in the other direction, however) ¶
- fix to session.update() to preserve "dirty" status of incoming object ¶
- sending a selectable to an IN via the in_() function no longer creates a "union" out of multiple selects; only one selectable to a the in_() function is allowed now (make a union yourself if union is needed) ¶
- improved support for disabling save-update cascade via cascade="none" etc. ¶

- added "remote_side" argument to relation(), used only with self-referential mappers to force the direction of the parent/child relationship. replaces the usage of the "foreignkey" parameter for "switching" the direction. "foreignkey" argument is deprecated for all uses and will eventually be replaced by an argument dedicated to ForeignKey specification on mappers. ¶

0.3.1

Released: Mon Nov 13 2006

orm

- **[orm]** the "delete" cascade will load in all child objects, if they were not loaded already. this can be turned off (i.e. the old behavior) by setting passive_deletes=True on a relation(). ¶
- **[orm]** adjustments to reworked eager query generation to not fail on circular eager-loaded relationships (like backrefs) ¶
- **[orm]** fixed bug where eagerload() (nor lazyload()) option didn't properly instruct the Query whether or not to use "nesting" when producing a LIMIT query. ¶
- **[orm]** fixed bug in circular dependency sorting at flush time; if object A contained a cyclical many-to-one relationship to object B, and object B was just attached to object A, *but* object B itself wasn't changed, the many-to-one synchronize of B's primary key attribute to A's foreign key attribute wouldn't occur. ¶ References: #360
- **[orm]** implemented from_obj argument for query.count, improves count function on selectresults ¶ References: #325
- **[orm]** added an assertion within the "cascade" step of ORM relationships to check that the class of object attached to a parent object is appropriate (i.e. if A.items stores B objects, raise an error if a C is appended to A.items) ¶
- **[orm]** new extension sqlalchemy.ext.associationproxy, provides transparent "association object" mappings. new example examples/association/proxied_association.py illustrates. ¶
- **[orm]** improvement to single table inheritance to load full hierarchies beneath the target class ¶
- **[orm]** fix to subtle condition in topological sort where a node could appear twice, for ¶ References: #362
- **[orm]** additional rework to topological sort, refactoring, for ¶ References: #365
- **[orm]** "delete-orphan" for a certain type can be set on more than one parent class; the instance is an "orphan" only if its not attached to *any* of those parents ¶

misc

- **[engine/pool]** some new Pool utility classes, updated docs ¶
- **[engine/pool]** "use_threadlocal" on Pool defaults to False (same as create_engine) ¶
- **[engine/pool]** fixed direct execution of Compiled objects ¶
- **[engine/pool]** create_engine() reworked to be strict about incoming **kwargs. all keyword arguments must be consumed by one of the dialect, connection pool, and engine constructors, else a TypeError is thrown which describes the full set of invalid kwargs in relation to the selected dialect/pool/engine configuration. ¶
- **[databases/types]** MySQL catches exception on "describe" and reports as NoSuchTableError ¶
- **[databases/types]** further fixes to sqlite booleans, weren't working as defaults ¶

- **[databases/types]** fix to postgres sequence quoting when using schemas ¶

0.3.0

Released: Sun Oct 22 2006

general

- **[general]** logging is now implemented via standard python "logging" module. "echo" keyword parameters are still functional but set/unset log levels for their respective classes/instances. all logging can be controlled directly through the Python API by setting INFO and DEBUG levels for loggers in the "sqlalchemy" namespace. class-level logging is under "sqlalchemy.<module>.<classname>", instance-level logging under "sqlalchemy.<module>.<classname>.0x..<00-FF>". Test suite includes "–log-info" and "–log-debug" arguments which work independently of –verbose/–quiet. Logging added to orm to allow tracking of mapper configurations, row iteration. ¶
- **[general]** the documentation-generation system has been overhauled to be much simpler in design and more integrated with Markdown ¶

orm

- **[orm]** attribute tracking modified to be more intelligent about detecting changes, particularly with mutable types. TypeEngine objects now take a greater role in defining how to compare two scalar instances, including the addition of a MutableType mixin which is implemented by PickleType. unit-of-work now tracks the "dirty" list as an expression of all persistent objects where the attribute manager detects changes. The basic issue that's fixed is detecting changes on PickleType objects, but also generalizes type handling and "modified" object checking to be more complete and extensible. ¶
- **[orm]** a wide refactoring to "attribute loader" and "options" architectures. ColumnProperty and PropertyLoader define their loading behaivor via switchable "strategies", and MapperOptions no longer use mapper/property copying in order to function; they are instead propagated via QueryContext and SelectionContext objects at query/instances time. All of the internal copying of mappers and properties that was used to handle inheritance as well as options() has been removed; the structure of mappers and properties is much simpler than before and is clearly laid out in the new 'interfaces' module. ¶
- **[orm]** related to the mapper/property overhaul, internal refactoring to mapper instances() method to use a SelectionContext object to track state during the operation. SLIGHT API BREAKAGE: the append_result() and populate_instances() methods on MapperExtension have a slightly different method signature now as a result of the change; hoping that these methods are not in widespread use as of yet. ¶
- **[orm]** instances() method moved to Query now, backwards-compatible version remains on Mapper. ¶
- **[orm]** added contains_eager() MapperOption, used in conjunction with instances() to specify properties that should be eagerly loaded from the result set, using their plain column names by default, or translated given an custom row-translation function. ¶
- **[orm]** more rearrangements of unit-of-work commit scheme to better allow dependencies within circular flushes to work properly...updated task traversal/logging implementation ¶
- **[orm]** polymorphic mappers (i.e. using inheritance) now produces INSERT statements in order of tables across all inherited classes ¶ References: #321
- **[orm]** added an automatic "row switch" feature to mapping, which will detect a pending instance/deleted instance pair with the same identity key and convert the INSERT/DELETE to a single UPDATE ¶
- **[orm]** "association" mappings simplified to take advantage of automatic "row switch" feature ¶

- **[orm]** "custom list classes" is now implemented via the "collection_class" keyword argument to relation(). the old way still works but is deprecated ¶ References: #212
- **[orm]** added "viewonly" flag to relation(), allows construction of relations that have no effect on the flush() process. ¶
- **[orm]** added "lockmode" argument to base Query select/get functions, including "with_lockmode" function to get a Query copy that has a default locking mode. Will translate "read"/"update" arguments into a for_update argument on the select side. ¶ References: #292
- **[orm]** implemented "version check" logic in Query/Mapper, used when version_id_col is in effect and query.with_lockmode() is used to get() an instance that's already loaded ¶
- **[orm]** post_update behavior improved; does a better job at not updating too many rows, updates only required columns ¶ References: #208
- **[orm]** adjustments to eager loading so that its "eager chain" is kept separate from the normal mapper setup, thereby preventing conflicts with lazy loader operation, fixes ¶ References: #308
- **[orm]** fix to deferred group loading ¶
- **[orm]** session.flush() wont close a connection it opened ¶ References: #346
- **[orm]** added "batch=True" flag to mapper; if False, save_obj will fully save one object at a time including calls to before_XXXX and after_XXXX ¶
- **[orm]** added "column_prefix=None" argument to mapper; prepends the given string (typically '_') to column-based attributes automatically set up from the mapper's Table ¶
- **[orm]** specifying joins in the from_obj argument of query.select() will replace the main table of the query, if the table is somewhere within the given from_obj. this makes it possible to produce custom joins and outerjoins in queries without the main table getting added twice. ¶ References: #315
- **[orm]** eagerloading is adjusted to more thoughtfully attach its LEFT OUTER JOINs to the given query, looking for custom "FROM" clauses that may have already been set up. ¶
- **[orm]** added join_to and outerjoin_to transformative methods to SelectResults, to build up join/outerjoin conditions based on property names. also added select_from to explicitly set from_obj parameter. ¶
- **[orm]** removed "is_primary" flag from mapper. ¶

sql

- **[sql] [construction]** changed "for_update" parameter to accept False/True/"nowait" and "read", the latter two of which are interpreted only by Oracle and Mysql ¶ References: #292
- **[sql] [construction]** added extract() function to sql dialect (SELECT extract(field FROM expr)) ¶
- **[sql] [construction]** BooleanExpression includes new "negate" argument to specify the appropriate negation operator if one is available. ¶
- **[sql] [construction]** calling a negation on an "IN" or "IS" clause will result in "NOT IN", "IS NOT" (as opposed to NOT (x IN y)). ¶
- **[sql] [construction]** Function objects know what to do in a FROM clause now. their behavior should be the same, except now you can also do things like select(['*'], from_obj=[func.my_function()]) to get multiple columns from the result, or even use sql.column() constructs to name the return columns ¶ References: #172

schema

- **[schema]** a fair amount of cleanup to the schema package, removal of ambiguous methods, methods that are no longer needed. slightly more constrained usage, greater emphasis on explicitness ¶
- **[schema]** the "primary_key" attribute of Table and other selectables becomes a setlike ColumnCollection object; is ordered but not numerically indexed. a comparison clause between two pks that are derived from the same underlying tables (i.e. such as two Alias objects) can be generated via table1.primary_key==table2.primary_key ¶
- **[schema]** ForeignKey(Constraint) supports "use_alter=True", to create/drop a foreign key via ALTER. this allows circular foreign key relationships to be set up. ¶
- **[schema]** append_item() methods removed from Table and Column; preferably construct Table/Column/related objects inline, but if needed use append_column(), append_foreign_key(), append_constraint(), etc. ¶
- **[schema]** table.create() no longer returns the Table object, instead has no return value. the usual case is that tables are created via metadata, which is preferable since it will handle table dependencies. ¶
- **[schema]** added UniqueConstraint (goes at Table level), CheckConstraint (goes at Table or Column level). ¶
- **[schema]** index=False/unique=True on Column now creates a UniqueConstraint, index=True/unique=False creates a plain Index, index=True/unique=True on Column creates a unique Index. 'index' and 'unique' keyword arguments to column are now boolean only; for explcit names and groupings of indexes or unique constraints, use the UniqueConstraint/Index constructs explicitly. ¶
- **[schema]** added autoincrement=True to Column; will disable schema generation of SERIAL/AUTO_INCREMENT/identity seq for postgres/mysql/mssql if explicitly set to False ¶
- **[schema]** TypeEngine objects now have methods to deal with copying and comparing values of their specific type. Currently used by the ORM, see below. ¶
- **[schema]** fixed condition that occurred during reflection when a primary key column was explciitly overridden, where the PrimaryKeyConstraint would get both the reflected and the programmatic column doubled up ¶
- **[schema]** the "foreign_key" attribute on Column and ColumnElement in general is deprecated, in favor of the "foreign_keys" list/set-based attribute, which takes into account multiple foreign keys on one column. "foreign_key" will return the first element in the "foreign_keys" list/set or None if the list is empty. ¶

sqlite

- **[sqlite]** sqlite boolean datatype converts False/True to 0/1 by default ¶
- **[sqlite]** fixes to Date/Time (SLDate/SLTime) types; works as good as postgres now ¶ References: #335

oracle

- **[oracle]** Oracle has experimental support for cx_Oracle.TIMESTAMP, which requires a setinputsizes() call on the cursor that is now enabled via the 'auto_setinputsizes' flag to the oracle dialect. ¶

firebird

- **[firebird]** aliases do not use "AS" ¶
- **[firebird]** correctly raises NoSuchTableError when reflecting non-existent table ¶

misc

- **[ms-sql]** fixes bug 261 (table reflection broken for MS-SQL case-sensitive databases) ¶
- **[ms-sql]** can now specify port for pymssql ¶
- **[ms-sql]** introduces new "auto_identity_insert" option for auto-switching between "SET IDENTITY_INSERT" mode when values specified for IDENTITY columns ¶
- **[ms-sql]** now supports multi-column foreign keys ¶
- **[ms-sql]** fix to reflecting date/datetime columns ¶
- **[ms-sql]** NCHAR and NVARCHAR type support added ¶
- **[connections/pooling/execution]** connection pool tracks open cursors and automatically closes them if connection is returned to pool with cursors still opened. Can be affected by options which cause it to raise an error instead, or to do nothing. fixes issues with MySQL, others ¶
- **[connections/pooling/execution]** fixed bug where Connection wouldn't lose its Transaction after commit/rollback ¶
- **[connections/pooling/execution]** added scalar() method to ComposedSQLEngine, ResultProxy ¶
- **[connections/pooling/execution]** ResultProxy will close() the underlying cursor when the ResultProxy itself is closed. this will auto-close cursors for ResultProxy objects that have had all their rows fetched (or had scalar() called). ¶
- **[connections/pooling/execution]** ResultProxy.fetchall() internally uses DBAPI fetchall() for better efficiency, added to mapper iteration as well (courtesy Michael Twomey) ¶

6.2.10 0.2 Changelog

0.2.8

Released: Tue Sep 05 2006

- cleanup on connection methods + documentation. custom DBAPI arguments specified in query string, 'connect_args' argument to 'create_engine', or custom creation function via 'creator' function to 'create_engine'. ¶
- added "recycle" argument to Pool, is "pool_recycle" on create_engine, defaults to 3600 seconds; connections after this age will be closed and replaced with a new one, to handle db's that automatically close stale connections ¶ References: #274
- changed "invalidate" semantics with pooled connection; will instruct the underlying connection record to reconnect the next time its called. "invalidate" will also automatically be called if any error is thrown in the underlying call to connection.cursor(). this will hopefully allow the connection pool to reconnect to a database that had been stopped and started without restarting the connecting application ¶ References: #121
- eesh ! the tutorial doctest was broken for quite some time. ¶
- add_property() method on mapper does a "compile all mappers" step in case the given property references a non-compiled mapper (as it did in the case of the tutorial !) ¶
- check for pg sequence already existing before create ¶ References: #277
- if a contextual session is established via MapperExtension.get_session (as it is using the sessioncontext plugin, etc), a lazy load operation will use that session by default if the parent object is not persistent with a session already. ¶

- lazy loads will not fire off for an object that does not have a database identity (why? see http://www.sqlalchemy.org/trac/wiki/WhyDontForeignKeysLoadData) ¶
- unit-of-work does a better check for "orphaned" objects that are part of a "delete-orphan" cascade, for certain conditions where the parent isn't available to cascade from. ¶
- mappers can tell if one of their objects is an "orphan" based on interactions with the attribute package. this check is based on a status flag maintained for each relationship when objects are attached and detached from each other. ¶
- it is now invalid to declare a self-referential relationship with "delete-orphan" (as the abovementioned check would make them impossible to save) ¶
- improved the check for objects being part of a session when the unit of work seeks to flush() them as part of a relationship.. ¶
- statement execution supports using the same BindParam object more than once in an expression; simplified handling of positional parameters. nice job by Bill Noon figuring out the basic idea. ¶ References: #280
- postgres reflection moved to use pg_schema tables, can be overridden with use_information_schema=True argument to create_engine. ¶ References: #60, #71
- added case_sensitive argument to MetaData, Table, Column, determines itself automatically based on if a parent schemaitem has a non-None setting for the flag, or if not, then whether the identifier name is all lower case or not. when set to True, quoting is applied to identifiers with mixed or uppercase identifiers. quoting is also applied automatically in all cases to identifiers that are known to be reserved words or contain other non-standard characters. various database dialects can override all of this behavior, but currently they are all using the default behavior. tested with postgres, mysql, sqlite, oracle. needs more testing with firebird, ms-sql. part of the ongoing work with ¶ References: #155
- unit tests updated to run without any pysqlite installed; pool test uses a mock DBAPI ¶
- urls support escaped characters in passwords ¶ References: #281
- added limit/offset to UNION queries (though not yet in oracle) ¶
- added "timezone=True" flag to DateTime and Time types. postgres so far will convert this to "TIME[STAMP] (WITH|WITHOUT) TIME ZONE", so that control over timezone presence is more controllable (psycopg2 returns datetimes with tzinfo's if available, which can create confusion against datetimes that don't). ¶
- fix to using query.count() with distinct, **kwargs with SelectResults count() ¶ References: #287
- deregister Table from MetaData when autoload fails; ¶ References: #289
- import of py2.5s sqlite3 ¶ References: #293
- unicode fix for startswith()/endswith() ¶ References: #296

0.2.7

Released: Sat Aug 12 2006

- quoting facilities set up so that database-specific quoting can be turned on for individual table, schema, and column identifiers when used in all queries/creates/drops. Enabled via "quote=True" in Table or Column, as well as "quote_schema=True" in Table. Thanks to Aaron Spike for the excellent efforts. ¶
- assignmapper was setting is_primary=True, causing all sorts of mayhem by not raising an error when redundant mappers were set up, fixed ¶
- added allow_null_pks option to Mapper, allows rows where some primary key columns are null (i.e. when mapping to outer joins etc) ¶

- modifcation to unitofwork to not maintain ordering within the "new" list or within the UOWTask "objects" list; instead, new objects are tagged with an ordering identifier as they are registered as new with the session, and the INSERT statements are then sorted within the mapper save_obj. the INSERT ordering has basically been pushed all the way to the end of the flush cycle. that way the various sorts and organizations occurring within UOWTask (particularly the circular task sort) don't have to worry about maintaining order (which they weren't anyway) ¶
- fixed reflection of foreign keys to autoload the referenced table if it was not loaded already ¶
- – pass URL query string arguments to connect() function

  ¶ References: #256
- – oracle boolean type

  ¶ References: #257
- custom primary/secondary join conditions in a relation *will* be propagated to backrefs by default. specifying a backref() will override this behavior. ¶
- better check for ambiguous join conditions in sql.Join; propagates to a better error message in PropertyLoader (i.e. relation()/backref()) for when the join condition can't be reasonably determined. ¶
- sqlite creates ForeignKeyConstraint objects properly upon table reflection. ¶
- adjustments to pool stemming from changes made for. overflow counter should only be decremented if the connection actually succeeded. added a test script to attempt testing this. ¶ References: #224
- fixed mysql reflection of default values to be PassiveDefault ¶
- added reflected 'tinyint', 'mediumint' type to MS-SQL. ¶ References: #263, #264
- SingletonThreadPool has a size and does a cleanup pass, so that only a given number of thread-local connections stay around (needed for sqlite applications that dispose of threads en masse) ¶
- fixed small pickle bug(s) with lazy loaders ¶ References: #267, #265
- fixed possible error in mysql reflection where certain versions return an array instead of string for SHOW CREATE TABLE call ¶
- fix to lazy loads when mapping to joins ¶ References: #1770
- all create()/drop() calls have a keyword argument of "connectable". "engine" is deprecated. ¶
- fixed ms-sql connect() to work with adodbapi ¶
- added "nowait" flag to Select() ¶
- inheritance check uses issubclass() instead of direct __mro__ check to make sure class A inherits from B, allowing mapper inheritance to more flexibly correspond to class inheritance ¶ References: #271
- SelectResults will use a subselect, when calling an aggregate (i.e. max, min, etc.) on a SelectResults that has an ORDER BY clause ¶ References: #252
- fixes to types so that database-specific types more easily used; fixes to mysql text types to work with this methodology ¶ References: #269
- some fixes to sqlite date type organization ¶
- added MSTinyInteger to MS-SQL ¶ References: #263

0.2.6

Released: Thu Jul 20 2006

- big overhaul to schema to allow truly composite primary and foreign key constraints, via new ForeignKeyConstraint and PrimaryKeyConstraint objects. Existing methods of primary/foreign key creation have not been changed but use these new objects behind the scenes. table creation and reflection is now more table oriented rather than column oriented. ¶ References: #76
- overhaul to MapperExtension calling scheme, wasn't working very well previously ¶
- tweaks to ActiveMapper, supports self-referential relationships ¶
- slight rearrangement to objectstore (in activemapper/threadlocal) so that the SessionContext is referenced by '.context' instead of subclassed directly. ¶
- activemapper will use threadlocal's objectstore if the mod is activated when activemapper is imported ¶
- small fix to URL regexp to allow filenames with '@' in them ¶
- fixes to Session expunge/update/etc...needs more cleanup. ¶
- select_table mappers *still* weren't always compiling ¶
- fixed up Boolean datatype ¶
- added count()/count_by() to list of methods proxied by assignmapper; this also adds them to activemapper ¶
- connection exceptions wrapped in DBAPIError ¶
- ActiveMapper now supports autoloading column definitions from the database if you supply a __autoload__ = True attribute in your mapping inner-class. Currently this does not support reflecting any relationships. ¶
- deferred column load could screw up the connection status in a flush() under some circumstances, this was fixed ¶
- expunge() was not working with cascade, fixed. ¶
- potential endless loop in cascading operations fixed. ¶
- added "synonym()" function, applied to properties to have a propname the same as another, for the purposes of overriding props and allowing the original propname to be accessible in select_by(). ¶
- fix to typing in clause construction which specifically helps type issues with polymorphic_union (CAST/ColumnClause propagates its type to proxy columns) ¶
- mapper compilation work ongoing, someday it'll work....moved around the initialization of MapperProperty objects to be after all mappers are created to better handle circular compilations. do_init() method is called on all properties now which are more aware of their "inherited" status if so. ¶
- eager loads explicitly disallowed on self-referential relationships, or relationships to an inheriting mapper (which is also self-referential) ¶
- reduced bind param size in query._get to appease the picky oracle ¶ References: #244
- added 'checkfirst' argument to table.create()/table.drop(), as well as table.exists() ¶ References: #234
- some other ongoing fixes to inheritance ¶ References: #245
- attribute/backref/orphan/history-tracking tweaks as usual... ¶

0.2.5

Released: Sat Jul 08 2006

- fixed endless loop bug in select_by(), if the traversal hit two mappers that referenced each other ¶
- upgraded all unittests to insert './lib/' into sys.path, working around new setuptools PYTHONPATH-killing behavior ¶

- further fixes with attributes/dependencies/etc.... ¶
- improved error handling for when DynamicMetaData is not connected ¶
- MS-SQL support largely working (tested with pymssql) ¶
- ordering of UPDATE and DELETE statements within groups is now in order of primary key values, for more deterministic ordering ¶
- after_insert/delete/update mapper extensions now called per object, not per-object-per-table ¶
- further fixes/refactorings to mapper compilation ¶

0.2.4

Released: Tue Jun 27 2006

- try/except when the mapper sets init.__name__ on a mapped class, supports python 2.3 ¶
- fixed bug where threadlocal engine would still autocommit despite a transaction in progress ¶
- lazy load and deferred load operations require the parent object to be in a Session to do the operation; whereas before the operation would just return a blank list or None, it now raises an exception. ¶
- Session.update() is slightly more lenient if the session to which the given object was formerly attached to was garbage collected; otherwise still requires you explicitly remove the instance from the previous Session. ¶
- fixes to mapper compilation, checking for more error conditions ¶
- small fix to eager loading combined with ordering/limit/offset ¶
- utterly remarkable: added a single space between 'CREATE TABLE' and '(<the rest of it>' since *that's how MySQL indicates a non- reserved word tablename.....* ¶ References: #206
- more fixes to inheritance, related to many-to-many relations properly saving ¶
- fixed bug when specifying explicit module to mysql dialect ¶
- when QueuePool times out it raises a TimeoutError instead of erroneously making another connection ¶
- Queue.Queue usage in pool has been replaced with a locally modified version (works in py2.3/2.4!) that uses a threading.RLock for a mutex. this is to fix a reported case where a ConnectionFairy's __del__() method got called within the Queue's get() method, which then returns its connection to the Queue via the put() method, causing a reentrant hang unless threading.RLock is used. ¶
- postgres will not place SERIAL keyword on a primary key column if it has a foreign key constraint ¶
- cursor() method on ConnectionFairy allows db-specific extension arguments to be propagated ¶ References: #221
- lazy load bind params properly propagate column type ¶ References: #225
- new MySQL types: MSEnum, MSTinyText, MSMediumText, MSLongText, etc. more support for MS-specific length/precision params in numeric types patch courtesy Mike Bernson ¶
- some fixes to connection pool invalidate() ¶ References: #224

0.2.3

Released: Sat Jun 17 2006

- overhaul to mapper compilation to be deferred. this allows mappers to be constructed in any order, and their relationships to each other are compiled when the mappers are first used. ¶

- fixed a pretty big speed bottleneck in cascading behavior particularly when backrefs were in use ¶
- the attribute instrumentation module has been completely rewritten; its now a large degree simpler and clearer, slightly faster. the "history" of an attribute is no longer micromanaged with each change and is instead part of a "CommittedState" object created when the instance is first loaded. HistoryArraySet is gone, the behavior of list attributes is now more open ended (i.e. they're not sets anymore). ¶
- py2.4 "set" construct used internally, falls back to sets.Set when "set" not available/ordering is needed. ¶
- fix to transaction control, so that repeated rollback() calls don't fail (was failing pretty badly when flush() would raise an exception in a larger try/except transaction block) ¶
- "foreignkey" argument to relation() can also be a list. fixed auto-foreignkey detection ¶ References: #151
- fixed bug where tables with schema names weren't getting indexed in the MetaData object properly ¶
- fixed bug where Column with redefined "key" property wasn't getting type conversion happening in the ResultProxy ¶ References: #207
- fixed 'port' attribute of URL to be an integer if present ¶
- fixed old bug where if a many-to-many table mapped as "secondary" had extra columns, delete operations didn't work ¶
- bugfixes for mapping against UNION queries ¶
- fixed incorrect exception class thrown when no DB driver present ¶
- added NonExistentTable exception thrown when reflecting a table that doesn't exist ¶ References: #138
- small fix to ActiveMapper regarding one-to-one backrefs, other refactorings ¶
- overridden constructor in mapped classes gets __name__ and __doc__ from the original class ¶
- fixed small bug in selectresult.py regarding mapper extension ¶ References: #200
- small tweak to cascade_mappers, not very strongly supported function at the moment ¶
- some fixes to between(), column.between() to propagate typing information better ¶ References: #202
- if an object fails to be constructed, is not added to the session ¶ References: #203
- CAST function has been made into its own clause object with its own compilation function in ansicompiler; allows MySQL to silently ignore most CAST calls since MySQL seems to only support the standard CAST syntax with Date types. MySQL-compatible CAST support for strings, ints, etc. a TODO ¶

0.2.2

Released: Mon Jun 05 2006

- big improvements to polymorphic inheritance behavior, enabling it to work with adjacency list table structures ¶ References: #190
- major fixes and refactorings to inheritance relationships overall, more unit tests ¶
- fixed "echo_pool" flag on create_engine() ¶
- fix to docs, removed incorrect info that close() is unsafe to use with threadlocal strategy (its totally safe !) ¶
- create_engine() can take URLs as string or unicode ¶ References: #188
- firebird support partially completed; thanks to James Ralston and Brad Clements for their efforts. ¶
- Oracle url translation was broken, fixed, will feed host/port/sid into cx_oracle makedsn() if 'database' field is present, else uses straight TNS name from the 'host' field ¶

- fix to using unicode criterion for query.get()/query.load() ¶
- count() function on selectables now uses table primary key or first column instead of "1" for criterion, also uses label "rowcount" instead of "count". ¶
- got rudimental "mapping to multiple tables" functionality cleaned up, more correctly documented ¶
- restored global_connect() function, attaches to a DynamicMetaData instance called "default_metadata". leaving MetaData arg to Table out will use the default metadata. ¶
- fixes to session cascade behavior, entity_name propigation ¶
- reorganized unittests into subdirectories ¶
- more fixes to threadlocal connection nesting patterns ¶

0.2.1

Released: Mon May 29 2006

- "pool" argument to create_engine() properly propagates ¶
- fixes to URL, raises exception if not parsed, does not pass blank fields along to the DB connect string (a string such as user:host@/db was breaking on postgres) ¶
- small fixes to Mapper when it inserts and tries to get new primary key values back ¶
- rewrote half of TLEngine, the ComposedSQLEngine used with 'strategy="threadlocal"'. it now properly implements engine.begin()/ engine.commit(), which nest fully with connection.begin()/trans.commit(). added about six unittests. ¶
- major "duh" in pool.Pool, forgot to put back the WeakValueDictionary. unittest which was supposed to check for this was also silently missing it. fixed unittest to ensure that ConnectionFairy properly falls out of scope. ¶
- placeholder dispose() method added to SingletonThreadPool, doesn't do anything yet ¶
- rollback() is automatically called when an exception is raised, but only if there's no transaction in process (i.e. works more like autocommit). ¶
- fixed exception raise in sqlite if no sqlite module present ¶
- added extra example detail for association object doc ¶
- Connection adds checks for already being closed ¶

0.2.0

Released: Sat May 27 2006

- overhaul to Engine system so that what was formerly the SQLEngine is now a ComposedSQLEngine which consists of a variety of components, including a Dialect, ConnectionProvider, etc. This impacted all the db modules as well as Session and Mapper. ¶
- create_engine now takes only RFC-1738-style strings: driver://user:password@host:port/database ¶
- total rewrite of connection-scoping methodology, Connection objects can now execute clause elements directly, added explicit "close" as well as support throughout Engine/ORM to handle closing properly, no longer relying upon __del__ internally to return connections to the pool. ¶ References: #152
- overhaul to Session interface and scoping. uses hibernate-style methods, including query(class), save(), save_or_update(), etc. no threadlocal scope is installed by default. Provides a binding interface to specific Engines and/or Connections so that underlying Schema objects do not need to be bound to an Engine. Added a basic SessionTransaction object that can simplistically aggregate transactions across multiple engines. ¶

- overhaul to mapper's dependency and "cascade" behavior; dependency logic factored out of properties.py into a separate module "dependency.py". "cascade" behavior is now explicitly controllable, proper implementation of "delete", "delete-orphan", etc. dependency system can now determine at flush time if a child object has a parent or not so that it makes better decisions on how that child should be updated in the DB with regards to deletes. ¶
- overhaul to Schema to build upon MetaData object instead of an Engine. Entire SQL/Schema system can be used with no Engines whatsoever, executed solely by an explicit Connection object. the "bound" methodlogy exists via the BoundMetaData for schema objects. ProxyEngine is generally not needed anymore and is replaced by DynamicMetaData. ¶
- true polymorphic behavior implemented, fixes ¶ References: #167
- "oid" system has been totally moved into compile-time behavior; if they are used in an order_by where they are not available, the order_by doesn't get compiled, fixes ¶ References: #147
- overhaul to packaging; "mapping" is now "orm", "objectstore" is now "session", the old "objectstore" namespace gets loaded in via the "threadlocal" mod if used ¶
- mods now called in via "import <modname>". extensions favored over mods as mods are globally-monkeypatching ¶
- fix to add_property so that it propagates properties to inheriting mappers ¶ References: #154
- backrefs create themselves against primary mapper of its originating property, priamry/secondary join arguments can be specified to override. helps their usage with polymorphic mappers ¶
- "table exists" function has been implemented ¶ References: #31
- "create_all/drop_all" added to MetaData object ¶ References: #98
- improvements and fixes to topological sort algorithm, as well as more unit tests ¶
- tutorial page added to docs which also can be run with a custom doctest runner to ensure its properly working. docs generally overhauled to deal with new code patterns ¶
- many more fixes, refactorings. ¶
- migration guide is available on the Wiki at http://www.sqlalchemy.org/trac/wiki/02Migration ¶

6.2.11 0.1 Changelog

0.1.7

Released: Fri May 05 2006

- some fixes to topological sort algorithm ¶
- added DISTINCT ON support to Postgres (just supply distinct=[col1,col2..]) ¶
- added __mod__ (% operator) to sql expressions ¶
- "order_by" mapper property inherited from inheriting mapper ¶
- fix to column type used when mapper UPDATES/DELETEs ¶
- with convert_unicode=True, reflection was failing, has been fixed ¶
- types types types! still weren't working....have to use TypeDecorator again :(¶
- mysql binary type converts array output to buffer, fixes PickleType ¶
- fixed the attributes.py memory leak once and for all ¶
- unittests are qualified based on the databases that support each one ¶

- fixed bug where column defaults would clobber VALUES clause of insert objects ¶
- fixed bug where table def w/ schema name would force engine connection ¶
- fix for parenthesis to work correctly with subqueries in INSERT/UPDATE ¶
- HistoryArraySet gets extend() method ¶
- fixed lazyload support for other comparison operators besides = ¶
- lazyload fix where two comparisons in the join condition point to the samem column ¶
- added "construct_new" flag to mapper, will use __new__ to create instances instead of __init__ (standard in 0.2) ¶
- added selectresults.py to SVN, missed it last time ¶
- tweak to allow a many-to-many relationship from a table to itself via an association table ¶
- small fix to "translate_row" function used by polymorphic example ¶
- create_engine uses cgi.parse_qsl to read query string (out the window in 0.2) ¶
- tweaks to CAST operator ¶
- fixed function names LOCAL_TIME/LOCAL_TIMESTAMP -> LOCALTIME/LOCALTIMESTAMP ¶
- fixed order of ORDER BY/HAVING in compile ¶

0.1.6

Released: Wed Apr 12 2006

- support for MS-SQL added courtesy Rick Morrison, Runar Petursson ¶
- the latest SQLSoup from J. Ellis ¶
- ActiveMapper has preliminary support for inheritance (Jeff Watkins) ¶
- added a "mods" system which allows pluggable modules that modify/augment core functionality, using the function "install_mods(*modnames)". ¶
- added the first "mod", SelectResults, which modifies mapper selects to return generators that turn ranges into LIMIT/OFFSET queries (Jonas Borgstr? ¶
- factored out querying capabilities of Mapper into a separate Query object which is Session-centric. this improves the performance of mapper.using(session) and makes other things possible. ¶
- objectstore/Session refactored, the official way to save objects is now via the flush() method. The begin/commit functionality of Session is factored into LegacySession which is still established as the default behavior, until the 0.2 series. ¶
- types system is bound to an engine at query compile time, not schema construction time. this simplifies the types system as well as the ProxyEngine. ¶
- added 'version_id' keyword argument to mapper. this keyword should reference a Column object with type Integer, preferably non-nullable, which will be used on the mapped table to track version numbers. this number is incremented on each save operation and is specifed in the UPDATE/DELETE conditions so that it factors into the returned row count, which results in a ConcurrencyError if the value received is not the expected count. ¶
- added 'entity_name' keyword argument to mapper. a mapper is now associated with a class via the class object as well as an optional entity_name parameter, which is a string defaulting to None. any number of primary mappers can be created for a class, qualified by the entity name. instances of those classes will issue all of their load and save operations through their entity_name-qualified mapper, and maintain separate a identity in the identity map for an otherwise equilvalent object. ¶

- overhaul to the attributes system. code has been clarified, and also fixed to support proper polymorphic behavior on object attributes. ¶
- added "for_update" flag to Select objects ¶
- some fixes for backrefs ¶
- fix for postgres1 DateTime type ¶
- documentation pages mostly switched over to Markdown syntax ¶

0.1.5

Released: Mon Mar 27 2006

- added SQLSession concept to SQLEngine. this object keeps track of retrieving a connection from the connection pool as well as an in-progress transaction. methods push_session() and pop_session() added to SQLEngine which push/pop a new SQLSession onto the engine, allowing operation upon a second connection "nested" within the previous one, allowing nested transactions. Other tricks are sure to come later regarding SQLSession. ¶
- added nest_on argument to objectstore.Session. This is a single SQLEngine or list of engines for which push_session()/pop_session() will be called each time this Session becomes the active session (via objectstore.push_session() or equivalent). This allows a unit of work Session to take advantage of the nested transaction feature without explicitly calling push_session/pop_session on the engine. ¶
- factored apart objectstore/unitofwork to separate "Session scoping" from "uow commit heavy lifting" ¶
- added populate_instance() method to MapperExtension. allows an extension to modify the population of object attributes. this method can call the populate_instance() method on another mapper to proxy the attribute population from one mapper to another; some row translation logic is also built in to help with this. ¶
- fixed Oracle8-compatibility "use_ansi" flag which converts JOINs to comparisons with the = and (+) operators, passes basic unittests ¶
- tweaks to Oracle LIMIT/OFFSET support ¶
- Oracle reflection uses ALL_** views instead of USER_** to get larger list of stuff to reflect from ¶
- fixes to Oracle foreign key reflection ¶ References: #105
- objectstore.commit(obj1, obj2,...) adds an extra step to seek out private relations on properties and delete child objects, even though its not a global commit ¶
- lots and lots of fixes to mappers which use inheritance, strengthened the concept of relations on a mapper being made towards the "local" table for that mapper, not the tables it inherits. allows more complex compositional patterns to work with lazy/eager loading. ¶
- added support for mappers to inherit from others based on the same table, just specify the same table as that of both parent/child mapper. ¶
- some minor speed improvements to the attributes system with regards to instantiating and populating new objects. ¶
- fixed MySQL binary unit test ¶
- INSERTs can receive clause elements as VALUES arguments, not just literal values ¶
- support for calling multi-tokened functions, i.e. schema.mypkg.func() ¶
- added J. Ellis' SQLSoup module to extensions package ¶

- added "polymorphic" examples illustrating methods to load multiple object types from one mapper, the second of which uses the new populate_instance() method. small improvements to mapper, UNION construct to help the examples along ¶
- improvements/fixes to session.refresh()/session.expire() (which may have been called "invalidate" earlier..) ¶
- added session.expunge() which totally removes an object from the current session ¶
- added *args, **kwargs pass-thru to engine.transaction(func) allowing easier creation of transactionalizing decorator functions ¶
- added iterator interface to ResultProxy: "for row in result:..." ¶
- added assertion to tx = session.begin(); tx.rollback(); tx.begin(), i.e. cant use it after a rollback() ¶
- added date conversion on bind parameter fix to SQLite enabling dates to work with pysqlite1 ¶
- improvements to subqueries to more intelligently construct their FROM clauses ¶ References: #116
- added PickleType to types. ¶
- fixed two bugs with column labels with regards to bind parameters: bind param keynames they are now generated from a column "label" in all relevant cases to take advantage of excess-name-length rules, and checks for a peculiar collision against a column named the same as "tablename_colname" added ¶
- major overhaul to unit of work documentation, other documentation sections. ¶
- fixed attributes bug where if an object is committed, its lazy-loaded list got blown away if it hadn't been loaded ¶
- added unique_connection() method to engine, connection pool to return a connection that is not part of the thread-local context or any current transaction ¶
- added invalidate() function to pooled connection. will remove the connection from the pool. still need work for engines to auto-reconnect to a stale DB though. ¶
- added distinct() function to column elements so you can do func.count(mycol.distinct()) ¶
- added "always_refresh" flag to Mapper, creates a mapper that will always refresh the attributes of objects it gets/selects from the DB, overwriting any changes made. ¶

0.1.4

Released: Mon Mar 13 2006

- create_engine() now uses genericized parameters; host/hostname, db/dbname/database, password/passwd, etc. for all engine connections. makes engine URIs much more "universal" ¶
- added support for SELECT statements embedded into a column clause, using the flag "scalar=True" ¶
- another overhaul to EagerLoading when used in conjunction with mappers that inherit; improvements to eager loads figuring out their aliased queries correctly, also relations set up against a mapper with inherited mappers will create joins against the table that is specific to the mapper itself (i.e. and not any tables that are inherited/are further down the inheritance chain), this can be overridden by using custom primary/secondary joins. ¶
- added J.Ellis patch to mapper.py so that selectone() throws an exception if query returns more than one object row, selectfirst() to not throw the exception. also adds selectfirst_by (synonymous with get_by) and selectone_by ¶
- added onupdate parameter to Column, will exec SQL/python upon an update statement.Also adds "for_update=True" to all DefaultGenerator subclasses ¶
- added support for Oracle table reflection contributed by Andrija Zaric; still some bugs to work out regarding composite primary keys/dictionary selection ¶

- checked in an initial Firebird module, awaiting testing. ¶
- added sql.ClauseParameters dictionary object as the result for compiled.get_params(), does late-typeprocessing of bind parameters so that the original values are easier to access ¶
- more docs for indexes, column defaults, connection pooling, engine construction ¶
- overhaul to the construction of the types system. uses a simpler inheritance pattern so that any of the generic types can be easily subclassed, with no need for TypeDecorator. ¶
- added "convert_unicode=False" parameter to SQLEngine, will cause all String types to perform unicode encoding/decoding (makes Strings act like Unicodes) ¶
- added 'encoding="utf8"' parameter to engine. the given encoding will be used for all encode/decode calls within Unicode types as well as Strings when convert_unicode=True. ¶
- improved support for mapping against UNIONs, added polymorph.py example to illustrate multi-class mapping against a UNION ¶
- fix to SQLite LIMIT/OFFSET syntax ¶
- fix to Oracle LIMIT syntax ¶
- added backref() function, allows backreferences to have keyword arguments that will be passed to the backref. ¶
- Sequences and ColumnDefault objects can do execute()/scalar() standalone ¶
- SQL functions (i.e. func.foo()) can do execute()/scalar() standalone ¶
- fix to SQL functions so that the ANSI-standard functions, i.e. current_timestamp etc., do not specify parenthesis. all other functions do. ¶
- added settattr_clean and append_clean to SmartProperty, which set attributes without triggering a "dirty" event or any history. used as: myclass.prop1.setattr_clean(myobject, 'hi') ¶
- improved support to column defaults when used by mappers; mappers will pull pre-executed defaults from statement's executed bind parameters (pre-conversion) to populate them into a saved object's attributes; if any PassiveDefaults have fired off, will instead post-fetch the row from the DB to populate the object. ¶
- added 'get_session().invalidate(*obj)' method to objectstore, instances will refresh() themselves upon the next attribute access. ¶
- improvements to SQL func calls including an "engine" keyword argument so they can be execute()d or scalar()ed standalone, also added func accessor to SQLEngine ¶
- fix to MySQL4 custom table engines, i.e. TYPE instead of ENGINE ¶
- slightly enhanced logging, includes timestamps and a somewhat configurable formatting system, in lieu of a full-blown logging system ¶
- improvements to the ActiveMapper class from the TG gang, including many-to-many relationships ¶
- added Double and TinyInt support to mysql ¶

0.1.3

Released: Thu Mar 02 2006

- completed "post_update" feature, will add a second update statement before inserts and after deletes in order to reconcile a relationship without any dependencies being created; used when persisting two rows that are dependent on each other ¶

- completed mapper.using(session) function, localized per-object Session functionality; objects can be declared and manipulated as local to any user-defined Session ¶
- fix to Oracle "row_number over" clause with multiple tables ¶
- mapper.get() was not selecting multiple-keyed objects if the mapper's table was a join, such as in an inheritance relationship, this is fixed. ¶
- overhaul to sql/schema packages so that the sql package can run all on its own, producing selects, inserts, etc. without any engine dependencies. builds upon new TableClause/ColumnClause lexical objects. Schema's Table/Column objects are the "physical" subclasses of them. simplifies schema/sql relationship, extensions (like proxyengine), and speeds overall performance by a large margin. removes the entire getattr() behavior that plagued 0.1.1. ¶
- refactoring of how the mapper "synchronizes" data between two objects into a separate module, works better with properties attached to a mapper that has an additional inheritance relationship to one of the related tables, also the same methodology used to synchronize parent/child objects now used by mapper to synchronize between inherited and inheriting mappers. ¶
- made objectstore "check for out-of-identitymap" more aggressive, will perform the check when object attributes are modified or the object is deleted ¶
- Index object fully implemented, can be constructed standalone, or via "index" and "unique" arguments on Columns. ¶
- added "convert_unicode" flag to SQLEngine, will treat all String/CHAR types as Unicode types, with raw-byte/utf-8 translation on the bind parameter and result set side. ¶
- postgres maintains a list of ANSI functions that must have no parenthesis so function calls with no arguments work consistently ¶
- tables can be created with no engine specified. this will default their engine to a module-scoped "default engine" which is a ProxyEngine. this engine can be connected via the function "global_connect". ¶
- added "refresh(*obj)" method to objectstore / Session to reload the attributes of any set of objects from the database unconditionally ¶

0.1.2

Released: Fri Feb 24 2006

- fixed a recursive call in schema that was somehow running 994 times then returning normally. broke nothing, slowed down everything. thanks to jpellerin for finding this. ¶

0.1.1

Released: Thu Feb 23 2006

- small fix to Function class so that expressions with a func.foo() use the type of the Function object (i.e. the left side) as the type of the boolean expression, not the other side which is more of a moving target (changeset 1020). ¶
- creating self-referring mappers with backrefs slightly easier (but still not that easy - changeset 1019) ¶
- fixes to one-to-one mappings (changeset 1015) ¶
- psycopg1 date/time issue with None fixed (changeset 1005) ¶
- two issues related to postgres, which doesn't want to give you the "lastrowid" since oids are deprecated:

  - postgres database-side defaults that are on primary key cols *do* execute explicitly beforehand, even though that's not the idea of a PassiveDefault. this is because sequences on columns get reflected as PassiveDefaults, but need to be explicitly executed on a primary key col so we know what we just inserted.
  - if you did add a row that has a bunch of database-side defaults on it, and the PassiveDefault thing was working the old way, i.e. they just execute on the DB side, the "cant get the row back without an OID" exception that occurred also will not happen unless someone (usually the ORM) explicitly asks for it.

  ¶

- fixed a glitch with engine.execute_compiled where it was making a second ResultProxy that just got thrown away. ¶
- began to implement newer logic in object properities. you can now say myclass.attr.property, which will give you the PropertyLoader corresponding to that attribute, i.e. myclass.mapper.props['attr'] ¶
- eager loading has been internally overhauled to use aliases at all times. more complicated chains of eager loads can now be created without any need for explicit "use aliases"-type instructions. EagerLoader code is also much simpler now. ¶
- a new somewhat experimental flag "use_update" added to relations, indicates that this relationship should be handled by a second UPDATE statement, either after a primary INSERT or before a primary DELETE. handles circular row dependencies. ¶
- added exceptions module, all raised exceptions (except for some KeyError/AttributeError exceptions) descend from these classes. ¶
- fix to date types with MySQL, returned timedelta converted to datetime.time ¶
- two-phase objectstore.commit operations (i.e. begin/commit) now return a transactional object (SessionTrans), to more clearly indicate transaction boundaries. ¶
- Index object with create/drop support added to schema ¶
- fix to postgres, where it will explicitly pre-execute a PassiveDefault on a table if it is a primary key column, pursuant to the ongoing "we cant get inserted rows back from postgres" issue ¶
- change to information_schema query that gets back postgres table defs, now uses explicit JOIN keyword, since one user had faster performance with 8.1 ¶
- fix to engine.process_defaults so it works correctly with a table that has different column name/column keys (changset 982) ¶
- a column can only be attached to one table - this is now asserted ¶
- postgres time types descend from Time type ¶
- fix to alltests so that it runs types test (now named testtypes) ¶
- fix to Join object so that it correctly exports its foreign keys (cs 973) ¶
- creating relationships against mappers that use inheritance fixed (cs 973) ¶

6.3 Older Migration Guides

6.3.1 What's New in SQLAlchemy 1.0?

About this Document

This document describes changes between SQLAlchemy version 0.9, undergoing maintenance releases as of May, 2014, and SQLAlchemy version 1.0, released in April, 2015.

Document last updated: June 9, 2015

Introduction

This guide introduces what's new in SQLAlchemy version 1.0, and also documents changes which affect users migrating their applications from the 0.9 series of SQLAlchemy to 1.0.

Please carefully review the sections on behavioral changes for potentially backwards-incompatible changes in behavior.

New Features and Improvements - ORM

New Session Bulk INSERT/UPDATE API

A new series of `Session` methods which provide hooks directly into the unit of work's facility for emitting INSERT and UPDATE statements has been created. When used correctly, this expert-oriented system can allow ORM-mappings to be used to generate bulk insert and update statements batched into executemany groups, allowing the statements to proceed at speeds that rival direct use of the Core.

See also:

Bulk Operations - introduction and full documentation

#3100

New Performance Example Suite

Inspired by the benchmarking done for the *Bulk Operations* feature as well as for the *How can I profile a SQLAlchemy powered application?* section of the FAQ, a new example section has been added which features several scripts designed to illustrate the relative performance profile of various Core and ORM techniques. The scripts are organized into use cases, and are packaged under a single console interface such that any combination of demonstrations can be run, dumping out timings, Python profile results and/or RunSnake profile displays.

See also:

Performance

"Baked" Queries

The "baked" query feature is an unusual new approach which allows for straightforward construction an invocation of `Query` objects using caching, which upon successive calls features vastly reduced Python function call overhead (over 75%). By specifying a `Query` object as a series of lambdas which are only invoked once, a query as a pre-compiled unit begins to be feasable:

```
from sqlalchemy.ext import baked
from sqlalchemy import bindparam

bakery = baked.bakery()

def search_for_user(session, username, email=None):

    baked_query = bakery(lambda session: session.query(User))
    baked_query += lambda q: q.filter(User.name == bindparam('username'))
```

```
    baked_query += lambda q: q.order_by(User.id)

    if email:
        baked_query += lambda q: q.filter(User.email == bindparam('email'))

    result = baked_query(session).params(username=username, email=email).all()

    return result
```

See also:

Baked Queries

#3054

Improvements to declarative mixins, `@declared_attr` and related features

The declarative system in conjunction with *declared_attr* has been overhauled to support new capabilities.

A function decorated with *declared_attr* is now called only **after** any mixin-based column copies are generated. This means the function can call upon mixin-established columns and will receive a reference to the correct *Column* object:

```
class HasFooBar(object):
    foobar = Column(Integer)

    @declared_attr
    def foobar_prop(cls):
        return column_property('foobar: ' + cls.foobar)

class SomeClass(HasFooBar, Base):
    __tablename__ = 'some_table'
    id = Column(Integer, primary_key=True)
```

Above, `SomeClass.foobar_prop` will be invoked against `SomeClass`, and `SomeClass.foobar` will be the final *Column* object that is to be mapped to `SomeClass`, as opposed to the non-copied object present directly on `HasFooBar`, even though the columns aren't mapped yet.

The *declared_attr* function now **memoizes** the value that's returned on a per-class basis, so that repeated calls to the same attribute will return the same value. We can alter the example to illustrate this:

```
class HasFooBar(object):
    @declared_attr
    def foobar(cls):
        return Column(Integer)

    @declared_attr
    def foobar_prop(cls):
        return column_property('foobar: ' + cls.foobar)

class SomeClass(HasFooBar, Base):
    __tablename__ = 'some_table'
    id = Column(Integer, primary_key=True)
```

Previously, `SomeClass` would be mapped with one particular copy of the `foobar` column, but the `foobar_prop` by calling upon `foobar` a second time would produce a different column. The value of `SomeClass.foobar` is

now memoized during declarative setup time, so that even before the attribute is mapped by the mapper, the interim column value will remain consistent no matter how many times the *declared_attr* is called upon.

The two behaviors above should help considerably with declarative definition of many types of mapper properties that derive from other attributes, where the *declared_attr* function is called upon from other *declared_attr* functions locally present before the class is actually mapped.

For a pretty slim edge case where one wishes to build a declarative mixin that establishes distinct columns per subclass, a new modifier *declared_attr.cascading* is added. With this modifier, the decorated function will be invoked individually for each class in the mapped inheritance hierarchy. While this is already the behavior for special attributes such as `__table_args__` and `__mapper_args__`, for columns and other properties the behavior by default assumes that attribute is affixed to the base class only, and just inherited from subclasses. With *declared_attr.cascading*, individual behaviors can be applied:

```
class HasSomeAttribute(object):
    @declared_attr.cascading
    def some_id(cls):
        if has_inherited_table(cls):
            return Column(ForeignKey('myclass.id'), primary_key=True)
        else:
            return Column(Integer, primary_key=True)

        return Column('id', Integer, primary_key=True)

class MyClass(HasSomeAttribute, Base):
    ""
    # ...

class MySubClass(MyClass):
    ""
    # ...
```

See also:

Mixing in Columns in Inheritance Scenarios

Finally, the *AbstractConcreteBase* class has been reworked so that a relationship or other mapper property can be set up inline on the abstract base:

```
from sqlalchemy import Column, Integer, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import (declarative_base, declared_attr,
    AbstractConcreteBase)

Base = declarative_base()

class Something(Base):
    __tablename__ = u'something'
    id = Column(Integer, primary_key=True)


class Abstract(AbstractConcreteBase, Base):
    id = Column(Integer, primary_key=True)

    @declared_attr
    def something_id(cls):
        return Column(ForeignKey(Something.id))

    @declared_attr
```

```
    def something(cls):
        return relationship(Something)


class Concrete(Abstract):
    __tablename__ = u'cca'
    __mapper_args__ = {'polymorphic_identity': 'cca', 'concrete': True}
```

The above mapping will set up a table `cca` with both an `id` and a `something_id` column, and `Concrete` will also have a relationship `something`. The new feature is that `Abstract` will also have an independently configured relationship `something` that builds against the polymorphic union of the base.

#3150 #2670 #3149 #2952 #3050

ORM full object fetches 25% faster

The mechanics of the `loading.py` module as well as the identity map have undergone several passes of inlining, refactoring, and pruning, so that a raw load of rows now populates ORM-based objects around 25% faster. Assuming a 1M row table, a script like the following illustrates the type of load that's improved the most:

```
import time
from sqlalchemy import Integer, Column, create_engine, Table
from sqlalchemy.orm import Session
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Foo(Base):
    __table__ = Table(
        'foo', Base.metadata,
        Column('id', Integer, primary_key=True),
        Column('a', Integer(), nullable=False),
        Column('b', Integer(), nullable=False),
        Column('c', Integer(), nullable=False),
    )

engine = create_engine(
    'mysql+mysqldb://scott:tiger@localhost/test', echo=True)

sess = Session(engine)

now = time.time()

# avoid using all() so that we don't have the overhead of building
# a large list of full objects in memory
for obj in sess.query(Foo).yield_per(100).limit(1000000):
    pass

print("Total time: %d" % (time.time() - now))
```

Local MacBookPro results bench from 19 seconds for 0.9 down to 14 seconds for 1.0. The *`Query.yield_per()`* call is always a good idea when batching huge numbers of rows, as it prevents the Python interpreter from having to allocate a huge amount of memory for all objects and their instrumentation at once. Without the *`Query.yield_per()`*, the above script on the MacBookPro is 31 seconds on 0.9 and 26 seconds on 1.0, the extra time spent setting up very large memory buffers.

New KeyedTuple implementation dramatically faster

We took a look into the *KeyedTuple* implementation in the hopes of improving queries like this:

```
rows = sess.query(Foo.a, Foo.b, Foo.c).all()
```

The *KeyedTuple* class is used rather than Python's `collections.namedtuple()`, because the latter has a very complex type-creation routine that benchmarks much slower than *KeyedTuple*. However, when fetching hundreds of thousands of rows, `collections.namedtuple()` quickly overtakes *KeyedTuple* which becomes dramatically slower as instance invocation goes up. What to do? A new type that hedges between the approaches of both. Benching all three types for "size" (number of rows returned) and "num" (number of distinct queries), the new "lightweight keyed tuple" either outperforms both, or lags very slightly behind the faster object, based on which scenario. In the "sweet spot", where we are both creating a good number of new types as well as fetching a good number of rows, the lightweight object totally smokes both namedtuple and KeyedTuple:

```
-----------------
size=10 num=10000                 # few rows, lots of queries
namedtuple: 3.60302400589         # namedtuple falls over
keyedtuple: 0.255059957504        # KeyedTuple very fast
lw keyed tuple: 0.582715034485    # lw keyed trails right on KeyedTuple
-----------------
size=100 num=1000                 # <--- sweet spot
namedtuple: 0.365247011185
keyedtuple: 0.24896979332
lw keyed tuple: 0.0889317989349   # lw keyed blows both away!
-----------------
size=10000 num=100
namedtuple: 0.572599887848
keyedtuple: 2.54251694679
lw keyed tuple: 0.613876104355
-----------------
size=1000000 num=10               # few queries, lots of rows
namedtuple: 5.79669594765         # namedtuple very fast
keyedtuple: 28.856498003          # KeyedTuple falls over
lw keyed tuple: 6.74346804619     # lw keyed trails right on namedtuple
```

#3176

Significant Improvements in Structural Memory Use

Structural memory use has been improved via much more significant use of `__slots__` for many internal objects. This optimization is particularly geared towards the base memory size of large applications that have lots of tables and columns, and reduces memory size for a variety of high-volume objects including event listening internals, comparator objects and parts of the ORM attribute and loader strategy system.

A bench that makes use of heapy measure the startup size of Nova illustrates a difference of about 3.7 fewer megs, or 46%, taken up by SQLAlchemy's objects, associated dictionaries, as well as weakrefs, within a basic import of "nova.db.sqlalchemy.models":

```
# reported by heapy, summation of SQLAlchemy objects +
# associated dicts + weakref-related objects with core of Nova imported:

    Before: total count 26477 total bytes 7975712
    After: total count 18181 total bytes 4236456
```

```
# reported for the Python module space overall with the
# core of Nova imported:

    Before: Partition of a set of 355558 objects. Total size = 61661760 bytes.
    After: Partition of a set of 346034 objects. Total size = 57808016 bytes.
```

UPDATE statements are now batched with executemany() in a flush

UPDATE statements can now be batched within an ORM flush into more performant executemany() call, similarly to how INSERT statements can be batched; this will be invoked within flush based on the following criteria:

- two or more UPDATE statements in sequence involve the identical set of columns to be modified.
- The statement has no embedded SQL expressions in the SET clause.
- The mapping does not use a `version_id_col`, or the backend dialect supports a "sane" rowcount for an executemany() operation; most DBAPIs support this correctly now.

Session.get_bind() handles a wider variety of inheritance scenarios

The `Session.get_bind()` method is invoked whenever a query or unit of work flush process seeks to locate the database engine that corresponds to a particular class. The method has been improved to handle a variety of inheritance-oriented scenarios, including:

- Binding to a Mixin or Abstract Class:

```
class MyClass(SomeMixin, Base):
    __tablename__ = 'my_table'
    # ...

session = Session(binds={SomeMixin: some_engine})
```

- Binding to inherited concrete subclasses individually based on table:

```
class BaseClass(Base):
    __tablename__ = 'base'

    # ...

class ConcreteSubClass(BaseClass):
    __tablename__ = 'concrete'

    # ...

    __mapper_args__ = {'concrete': True}


session = Session(binds={
    base_table: some_engine,
    concrete_table: some_other_engine
})
```

#3035

Session.get_bind() will receive the Mapper in all relevant Query cases

A series of issues were repaired where the *Session.get_bind()* would not receive the primary *Mapper* of the *Query*, even though this mapper was readily available (the primary mapper is the single mapper, or alternatively the first mapper, that is associated with a *Query* object).

The *Mapper* object, when passed to *Session.get_bind()*, is typically used by sessions that make use of the *Session.binds* parameter to associate mappers with a series of engines (although in this use case, things frequently "worked" in most cases anyway as the bind would be located via the mapped table object), or more specifically implement a user-defined *Session.get_bind()* method that provies some pattern of selecting engines based on mappers, such as horizontal sharding or a so-called "routing" session that routes queries to different backends.

These scenarios include:

- *Query.count()*:

```
session.query(User).count()
```

- *Query.update()* and *Query.delete()*, both for the UPDATE/DELETE statement as well as for the SELECT used by the "fetch" strategy:

```
session.query(User).filter(User.id == 15).update(
        {"name": "foob"}, synchronize_session='fetch')

session.query(User).filter(User.id == 15).delete(
        synchronize_session='fetch')
```

- Queries against individual columns:

```
session.query(User.id, User.name).all()
```

- SQL functions and other expressions against indirect mappings such as *column_property*:

```
class User(Base):
    # ...

    score = column_property(func.coalesce(self.tables.users.c.name, None)))

session.query(func.max(User.score)).scalar()
```

#3227 #3242 #1326

.info dictionary improvements

The `InspectionAttr.info` collection is now available on every kind of object that one would retrieve from the *Mapper.all_orm_descriptors* collection. This includes *hybrid_property* and *association_proxy()*. However, as these objects are class-bound descriptors, they must be accessed **separately** from the class to which they are attached in order to get at the attribute. Below this is illustrated using the *Mapper.all_orm_descriptors* namespace:

```
class SomeObject(Base):
    # ...

    @hybrid_property
```

```
    def some_prop(self):
        return self.value + 5


inspect(SomeObject).all_orm_descriptors.some_prop.info['foo'] = 'bar'
```

It is also available as a constructor argument for all *SchemaItem* objects (e.g. *ForeignKey*, *UniqueConstraint* etc.) as well as remaining ORM constructs such as *orm.synonym()*.

#2971

#2963

ColumnProperty constructs work a lot better with aliases, order_by

A variety of issues regarding *column_property()* have been fixed, most specifically with regards to the *aliased()* construct as well as the "order by label" logic introduced in 0.9 (see *Label constructs can now render as their name alone in an ORDER BY*).

Given a mapping like the following:

```
class A(Base):
    __tablename__ = 'a'

    id = Column(Integer, primary_key=True)

class B(Base):
    __tablename__ = 'b'

    id = Column(Integer, primary_key=True)
    a_id = Column(ForeignKey('a.id'))


A.b = column_property(
        select([func.max(B.id)]).where(B.a_id == A.id).correlate(A)
    )
```

A simple scenario that included "A.b" twice would fail to render correctly:

```
print sess.query(A, a1).order_by(a1.b)
```

This would order by the wrong column:

```
SELECT a.id AS a_id, (SELECT max(b.id) AS max_1 FROM b
WHERE b.a_id = a.id) AS anon_1, a_1.id AS a_1_id,
(SELECT max(b.id) AS max_2
FROM b WHERE b.a_id = a_1.id) AS anon_2
FROM a, a AS a_1 ORDER BY anon_1
```

New output:

```
SELECT a.id AS a_id, (SELECT max(b.id) AS max_1
FROM b WHERE b.a_id = a.id) AS anon_1, a_1.id AS a_1_id,
(SELECT max(b.id) AS max_2
FROM b WHERE b.a_id = a_1.id) AS anon_2
FROM a, a AS a_1 ORDER BY anon_2
```

There were also many scenarios where the "order by" logic would fail to order by label, for example if the mapping were "polymorphic":

```
class A(Base):
    __tablename__ = 'a'

    id = Column(Integer, primary_key=True)
    type = Column(String)

    __mapper_args__ = {'polymorphic_on': type, 'with_polymorphic': '*'}
```

The order_by would fail to use the label, as it would be anonymized due to the polymorphic loading:

```
SELECT a.id AS a_id, a.type AS a_type, (SELECT max(b.id) AS max_1
FROM b WHERE b.a_id = a.id) AS anon_1
FROM a ORDER BY (SELECT max(b.id) AS max_2
FROM b WHERE b.a_id = a.id)
```

Now that the order by label tracks the anonymized label, this now works:

```
SELECT a.id AS a_id, a.type AS a_type, (SELECT max(b.id) AS max_1
FROM b WHERE b.a_id = a.id) AS anon_1
FROM a ORDER BY anon_1
```

Included in these fixes are a variety of heisenbugs that could corrupt the state of an `aliased()` construct such that the labeling logic would again fail; these have also been fixed.

#3148 #3188

New Features and Improvements - Core

Select/Query LIMIT / OFFSET may be specified as an arbitrary SQL expression

The *`Select.limit()`* and *`Select.offset()`* methods now accept any SQL expression, in addition to integer values, as arguments. The ORM *`Query`* object also passes through any expression to the underlying *`Select`* object. Typically this is used to allow a bound parameter to be passed, which can be substituted with a value later:

```
sel = select([table]).limit(bindparam('mylimit')).offset(bindparam('myoffset'))
```

Dialects which don't support non-integer LIMIT or OFFSET expressions may continue to not support this behavior; third party dialects may also need modification in order to take advantage of the new behavior. A dialect which currently uses the `._limit` or `._offset` attributes will continue to function for those cases where the limit/offset was specified as a simple integer value. However, when a SQL expression is specified, these two attributes will instead raise a *`CompileError`* on access. A third-party dialect which wishes to support the new feature should now call upon the `._limit_clause` and `._offset_clause` attributes to receive the full SQL expression, rather than the integer value.

The `use_alter` flag on `ForeignKeyConstraint` is (usually) no longer needed

The *`MetaData.create_all()`* and *`MetaData.drop_all()`* methods will now make use of a system that automatically renders an ALTER statement for foreign key constraints that are involved in mutually-dependent cycles between tables, without the need to specify *`ForeignKeyConstraint.use_alter`*. Additionally, the foreign key constraints no longer need to have a name in order to be created via ALTER; only the DROP operation requires a name. In the case of a DROP, the feature will ensure that only constraints which have explicit names are actually

included as ALTER statements. In the case of an unresolvable cycle within a DROP, the system emits a succinct and clear error message now if the DROP cannot proceed.

The `ForeignKeyConstraint.use_alter` and `ForeignKey.use_alter` flags remain in place, and continue to have the same effect of establishing those constraints for which ALTER is required during a CREATE/DROP scenario.

As of version 1.0.1, special logic takes over in the case of SQLite, which does not support ALTER, in the case that during a DROP, the given tables have an unresolvable cycle; in this case a warning is emitted, and the tables are dropped with **no** ordering, which is usually fine on SQLite unless constraints are enabled. To resolve the warning and proceed with at least a partial ordering on a SQLite database, particuarly one where constraints are enabled, re-apply "use_alter" flags to those `ForeignKey` and `ForeignKeyConstraint` objects which should be explicitly omitted from the sort.

See also:

Creating/Dropping Foreign Key Constraints via ALTER - full description of the new behavior.

#3282

ResultProxy "auto close" is now a "soft" close

For many releases, the `ResultProxy` object has always been automatically closed out at the point at which all result rows have been fetched. This was to allow usage of the object without the need to call upon `ResultProxy.close()` explicitly; as all DBAPI resources had been freed, the object was safe to discard. However, the object maintained a strict "closed" behavior, which meant that any subsequent calls to `ResultProxy.fetchone()`, `ResultProxy.fetchmany()` or `ResultProxy.fetchall()` would now raise a `ResourceClosedError`:

```
>>> result = connection.execute(stmt)
>>> result.fetchone()
(1, 'x')
>>> result.fetchone()
None  # indicates no more rows
>>> result.fetchone()
exception: ResourceClosedError
```

This behavior is inconsistent vs. what pep-249 states, which is that you can call upon the fetch methods repeatedly even after results are exhausted. It also interferes with behavior for some implementations of result proxy, such as the `BufferedColumnResultProxy` used by the cx_oracle dialect for certain datatypes.

To solve this, the "closed" state of the `ResultProxy` has been broken into two states; a "soft close" which does the majority of what "close" does, in that it releases the DBAPI cursor and in the case of a "close with result" object will also release the connection, and a "closed" state which is everything included by "soft close" as well as establishing the fetch methods as "closed". The `ResultProxy.close()` method is now never called implicitly, only the `ResultProxy._soft_close()` method which is non-public:

```
>>> result = connection.execute(stmt)
>>> result.fetchone()
(1, 'x')
>>> result.fetchone()
None  # indicates no more rows
>>> result.fetchone()
None  # still None
>>> result.fetchall()
[]
>>> result.close()
```

```
>>> result.fetchone()
exception: ResourceClosedError  # *now* it raises
```

#3330 #3329

CHECK Constraints now support the `%(column_0_name)s` token in naming conventions

The `%(column_0_name)s` will derive from the first column found in the expression of a *CheckConstraint*:

```
metadata = MetaData(
    naming_convention={"ck": "ck_%(table_name)s_%(column_0_name)s"}
)

foo = Table('foo', metadata,
    Column('value', Integer),
)

CheckConstraint(foo.c.value > 5)
```

Will render:

```
CREATE TABLE foo (
    value INTEGER,
    CONSTRAINT ck_foo_value CHECK (value > 5)
)
```

The combination of naming conventions with the constraint produced by a *SchemaType* such as *Boolean* or *Enum* will also now make use of all CHECK constraint conventions.

See also:

Naming CHECK Constraints

Configuring Naming for Boolean, Enum, and other schema types

#3299

Constraints referring to unattached Columns can auto-attach to the Table when their referred columns are attached

Since at least version 0.8, a *Constraint* has had the ability to "auto-attach" itself to a *Table* based on being passed table-attached columns:

```
from sqlalchemy import Table, Column, MetaData, Integer, UniqueConstraint

m = MetaData()

t = Table('t', m,
    Column('a', Integer),
    Column('b', Integer)
)

uq = UniqueConstraint(t.c.a, t.c.b)  # will auto-attach to Table

assert uq in t.constraints
```

In order to assist with some cases that tend to come up with declarative, this same auto-attachment logic can now function even if the *Column* objects are not yet associated with the *Table*; additional events are established such that when those *Column* objects are associated, the *Constraint* is also added:

```
from sqlalchemy import Table, Column, MetaData, Integer, UniqueConstraint

m = MetaData()

a = Column('a', Integer)
b = Column('b', Integer)

uq = UniqueConstraint(a, b)

t = Table('t', m, a, b)

assert uq in t.constraints  # constraint auto-attached
```

The above feature was a late add as of version 1.0.0b3. A fix as of version 1.0.4 for #3411 ensures that this logic does not occur if the *Constraint* refers to a mixture of *Column* objects and string column names; as we do not yet have tracking for the addition of names to a *Table*:

```
from sqlalchemy import Table, Column, MetaData, Integer, UniqueConstraint

m = MetaData()

a = Column('a', Integer)
b = Column('b', Integer)

uq = UniqueConstraint(a, 'b')

t = Table('t', m, a, b)

# constraint *not* auto-attached, as we do not have tracking
# to locate when a name 'b' becomes available on the table
assert uq not in t.constraints
```

Above, the attachment event for column "a" to table "t" will fire off before column "b" is attached (as "a" is stated in the *Table* constructor before "b"), and the constraint will fail to locate "b" if it were to attempt an attachment. For consistency, if the constraint refers to any string names, the autoattach-on-column-attach logic is skipped.

The original auto-attach logic of course remains in place, if the *Table* already contains all the target *Column* objects at the time the *Constraint* is constructed:

```
from sqlalchemy import Table, Column, MetaData, Integer, UniqueConstraint

m = MetaData()

a = Column('a', Integer)
b = Column('b', Integer)


t = Table('t', m, a, b)

uq = UniqueConstraint(a, 'b')

# constraint auto-attached normally as in older versions
assert uq in t.constraints
```

#3341 #3411

INSERT FROM SELECT now includes Python and SQL-expression defaults

Insert.from_select() now includes Python and SQL-expression defaults if otherwise unspecified; the limitation where non-server column defaults aren't included in an INSERT FROM SELECT is now lifted and these expressions are rendered as constants into the SELECT statement:

```
from sqlalchemy import Table, Column, MetaData, Integer, select, func

m = MetaData()

t = Table(
    't', m,
    Column('x', Integer),
    Column('y', Integer, default=func.somefunction()))

stmt = select([t.c.x])
print t.insert().from_select(['x'], stmt)
```

Will render:

```
INSERT INTO t (x, y) SELECT t.x, somefunction() AS somefunction_1
FROM t
```

The feature can be disabled using *Insert.from_select.include_defaults*.

Column server defaults now render literal values

The "literal binds" compiler flag is switched on when a *DefaultClause*, set up by *Column.server_default* is present as a SQL expression to be compiled. This allows literals embedded in SQL to render correctly, such as:

```
from sqlalchemy import Table, Column, MetaData, Text
from sqlalchemy.schema import CreateTable
from sqlalchemy.dialects.postgresql import ARRAY, array
from sqlalchemy.dialects import postgresql

metadata = MetaData()

tbl = Table("derp", metadata,
    Column("arr", ARRAY(Text),
                server_default=array(["foo", "bar", "baz"])),
)

print(CreateTable(tbl).compile(dialect=postgresql.dialect()))
```

Now renders:

```
CREATE TABLE derp (
    arr TEXT[] DEFAULT ARRAY['foo', 'bar', 'baz']
)
```

Previously, the literal values `"foo", "bar", "baz"` would render as bound parameters, which are useless in DDL.

#3087

UniqueConstraint is now part of the Table reflection process

A *Table* object populated using `autoload=True` will now include *UniqueConstraint* constructs as well as *Index* constructs. This logic has a few caveats for Postgresql and Mysql:

Postgresql Postgresql has the behavior such that when a UNIQUE constraint is created, it implicitly creates a UNIQUE INDEX corresponding to that constraint as well. The *Inspector.get_indexes()* and the *Inspector.get_unique_constraints()* methods will continue to **both** return these entries distinctly, where *Inspector.get_indexes()* now features a token `duplicates_constraint` within the index entry indicating the corresponding constraint when detected. However, when performing full table reflection using `Table(..., autoload=True)`, the *Index* construct is detected as being linked to the *UniqueConstraint*, and is **not** present within the `Table.indexes` collection; only the *UniqueConstraint* will be present in the `Table.constraints` collection. This deduplication logic works by joining to the `pg_constraint` table when querying `pg_index` to see if the two constructs are linked.

MySQL MySQL does not have separate concepts for a UNIQUE INDEX and a UNIQUE constraint. While it supports both syntaxes when creating tables and indexes, it does not store them any differently. The *Inspector.get_indexes()* and the *Inspector.get_unique_constraints()* methods will continue to **both** return an entry for a UNIQUE index in MySQL, where *Inspector.get_unique_constraints()* features a new token `duplicates_index` within the constraint entry indicating that this is a dupe entry corresponding to that index. However, when performing full table reflection using `Table(..., autoload=True)`, the *UniqueConstraint* construct is **not** part of the fully reflected *Table* construct under any circumstances; this construct is always represented by a *Index* with the `unique=True` setting present in the `Table.indexes` collection.

See also:

Postgresql Index Reflection

MySQL Unique Constraints and Reflection

#3184

New systems to safely emit parameterized warnings

For a long time, there has been a restriction that warning messages could not refer to data elements, such that a particular function might emit an infinite number of unique warnings. The key place this occurs is in the `Unicode type received non-unicode bind param value` warning. Placing the data value in this message would mean that the Python `__warningregistry__` for that module, or in some cases the Python-global `warnings.onceregistry`, would grow unbounded, as in most warning scenarios, one of these two collections is populated with every distinct warning message.

The change here is that by using a special `string` type that purposely changes how the string is hashed, we can control that a large number of parameterized messages are hashed only on a small set of possible hash values, such that a warning such as `Unicode type received non-unicode bind param value` can be tailored to be emitted only a specific number of times; beyond that, the Python warnings registry will begin recording them as duplicates.

To illustrate, the following test script will show only ten warnings being emitted for ten of the parameter sets, out of a total of 1000:

```
from sqlalchemy import create_engine, Unicode, select, cast
import random
import warnings
```

```
e = create_engine("sqlite://")

# Use the "once" filter (which is also the default for Python
# warnings).  Exactly ten of these warnings will
# be emitted; beyond that, the Python warnings registry will accumulate
# new values as dupes of one of the ten existing.
warnings.filterwarnings("once")

for i in range(1000):
    e.execute(select([cast(
        ('foo_%d' % random.randint(0, 1000000)).encode('ascii'), Unicode)]))
```

The format of the warning here is:

```
/path/lib/sqlalchemy/sql/sqltypes.py:186: SAWarning: Unicode type received
  non-unicode bind param value 'foo_4852'. (this warning may be
  suppressed after 10 occurrences)
```

#3178

Key Behavioral Changes - ORM

query.update() now resolves string names into mapped attribute names

The documentation for *Query.update()* states that the given `values` dictionary is "a dictionary with attributes names as keys", implying that these are mapped attribute names. Unfortunately, the function was designed more in mind to receive attributes and SQL expressions and not as much strings; when strings were passed, these strings would be passed through straight to the core update statement without any resolution as far as how these names are represented on the mapped class, meaning the name would have to match that of a table column exactly, not how an attribute of that name was mapped onto the class.

The string names are now resolved as attribute names in earnest:

```
class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    name = Column('user_name', String(50))
```

Above, the column `user_name` is mapped as `name`. Previously, a call to *Query.update()* that was passed strings would have to have been called as follows:

```
session.query(User).update({'user_name': 'moonbeam'})
```

The given string is now resolved against the entity:

```
session.query(User).update({'name': 'moonbeam'})
```

It is typically preferable to use the attribute directly, to avoid any ambiguity:

```
session.query(User).update({User.name: 'moonbeam'})
```

The change also indicates that synonyms and hybrid attributes can be referred to by string name as well:

```
class User(Base):
    __tablename__ = 'user'

    id = Column(Integer, primary_key=True)
    name = Column('user_name', String(50))

    @hybrid_property
    def fullname(self):
        return self.name

session.query(User).update({'fullname': 'moonbeam'})
```

#3228

Warnings emitted when comparing objects with None values to relationships

This change is new as of 1.0.1. Some users are performing queries that are essentially of this form:

```
session.query(Address).filter(Address.user == User(id=None))
```

This pattern is not currently supported in SQLAlchemy. For all versions, it emits SQL resembling:

```
SELECT address.id AS address_id, address.user_id AS address_user_id,
address.email_address AS address_email_address
FROM address WHERE ? = address.user_id
(None,)
```

Note above, there is a comparison `WHERE ? = address.user_id` where the bound value `?` is receving `None`, or `NULL` in SQL. **This will always return False in SQL**. The comparison here would in theory generate SQL as follows:

```
SELECT address.id AS address_id, address.user_id AS address_user_id,
address.email_address AS address_email_address
FROM address WHERE address.user_id IS NULL
```

But right now, **it does not**. Applications which are relying upon the fact that "NULL = NULL" produces False in all cases run the risk that someday, SQLAlchemy might fix this issue to generate "IS NULL", and the queries will then produce different results. Therefore with this kind of operation, you will see a warning:

```
SAWarning: Got None for value of column user.id; this is unsupported
for a relationship comparison and will not currently produce an
IS comparison (but may in a future release)
```

Note that this pattern was broken in most cases for release 1.0.0 including all of the betas; a value like `SYMBOL('NEVER_SET')` would be generated. This issue has been fixed, but as a result of identifying this pattern, the warning is now there so that we can more safely repair this broken behavior (now captured in #3373) in a future release.

#3371

A "negated contains or equals" relationship comparison will use the current value of attributes, not the database value

This change is new as of 1.0.1; while we would have preferred for this to be in 1.0.0, it only became apparent as a result of #3371.

Given a mapping:

```
class A(Base):
    __tablename__ = 'a'
    id = Column(Integer, primary_key=True)

class B(Base):
    __tablename__ = 'b'
    id = Column(Integer, primary_key=True)
    a_id = Column(ForeignKey('a.id'))
    a = relationship("A")
```

Given `A`, with primary key of 7, but which we changed to be 10 without flushing:

```
s = Session(autoflush=False)
a1 = A(id=7)
s.add(a1)
s.commit()

a1.id = 10
```

A query against a many-to-one relationship with this object as the target will use the value 10 in the bound parameters:

```
s.query(B).filter(B.a == a1)
```

Produces:

```
SELECT b.id AS b_id, b.a_id AS b_a_id
FROM b
WHERE ? = b.a_id
(10,)
```

However, before this change, the negation of this criteria would **not** use 10, it would use 7, unless the object were flushed first:

```
s.query(B).filter(B.a != a1)
```

Produces (in 0.9 and all versions prior to 1.0.1):

```
SELECT b.id AS b_id, b.a_id AS b_a_id
FROM b
WHERE b.a_id != ? OR b.a_id IS NULL
(7,)
```

For a transient object, it would produce a broken query:

```
SELECT b.id, b.a_id
FROM b
WHERE b.a_id != :a_id_1 OR b.a_id IS NULL
{u'a_id_1': symbol('NEVER_SET')}
```

This inconsistency has been repaired, and in all queries the current attribute value, in this example `10`, will now be used.

#3374

Changes to attribute events and other operations regarding attributes that have no pre-existing value

In this change, the default return value of `None` when accessing an object is now returned dynamically on each access, rather than implicitly setting the attribute's state with a special "set" operation when it is first accessed. The visible result of this change is that `obj.__dict__` is not implicitly modified on get, and there are also some minor behavioral changes for *`attributes.get_history()`* and related functions.

Given an object with no state:

```
>>> obj = Foo()
```

It has always been SQLAlchemy's behavior such that if we access a scalar or many-to-one attribute that was never set, it is returned as `None`:

```
>>> obj.someattr
None
```

This value of `None` is in fact now part of the state of `obj`, and is not unlike as though we had set the attribute explicitly, e.g. `obj.someattr = None`. However, the "set on get" here would behave differently as far as history and events. It would not emit any attribute event, and additionally if we view history, we see this:

```
>>> inspect(obj).attrs.someattr.history
History(added=(), unchanged=[None], deleted=())   # 0.9 and below
```

That is, it's as though the attribute were always `None` and were never changed. This is explicitly different from if we had set the attribute first instead:

```
>>> obj = Foo()
>>> obj.someattr = None
>>> inspect(obj).attrs.someattr.history
History(added=[None], unchanged=(), deleted=())  # all versions
```

The above means that the behavior of our "set" operation can be corrupted by the fact that the value was accessed via "get" earlier. In 1.0, this inconsistency has been resolved, by no longer actually setting anything when the default "getter" is used.

```
>>> obj = Foo()
>>> obj.someattr
None
>>> inspect(obj).attrs.someattr.history
History(added=(), unchanged=(), deleted=())  # 1.0
>>> obj.someattr = None
>>> inspect(obj).attrs.someattr.history
History(added=[None], unchanged=(), deleted=())
```

The reason the above behavior hasn't had much impact is because the INSERT statement in relational databases considers a missing value to be the same as NULL in most cases. Whether SQLAlchemy received a history event for a particular attribute set to None or not would usually not matter; as the difference between sending None/NULL or not wouldn't have an impact. However, as #3060 (described here in *Priority of attribute changes on relationship-bound attributes vs. FK-bound may appear to change*) illustrates, there are some seldom edge cases where we do in fact want

to positively have `None` set. Also, allowing the attribute event here means it's now possible to create "default value" functions for ORM mapped attributes.

As part of this change, the generation of the implicit "None" is now disabled for other situations where this used to occur; this includes when an attribute set operation on a many-to-one is received; previously, the "old" value would be "None" if it had been not set otherwise; it now will send the value `orm.attributes.NEVER_SET`, which is a value that may be sent to an attribute listener now. This symbol may also be received when calling on mapper utility functions such as *`Mapper.primary_key_from_instance()`*; if the primary key attributes have no setting at all, whereas the value would be `None` before, it will now be the `orm.attributes.NEVER_SET` symbol, and no change to the object's state occurs.

#3061

Priority of attribute changes on relationship-bound attributes vs. FK-bound may appear to change

As a side effect of #3060, setting a relationship-bound attribute to `None` is now a tracked history event which refers to the intention of persisting `None` to that attribute. As it has always been the case that setting a relationship-bound attribute will trump direct assignment to the foreign key attributes, a change in behavior can be seen here when assigning None. Given a mapping:

```
class A(Base):
    __tablename__ = 'table_a'

    id = Column(Integer, primary_key=True)

class B(Base):
    __tablename__ = 'table_b'

    id = Column(Integer, primary_key=True)
    a_id = Column(ForeignKey('table_a.id'))
    a = relationship(A)
```

In 1.0, the relationship-bound attribute takes precedence over the FK-bound attribute in all cases, whether or not the value we assign is a reference to an `A` object or is `None`. In 0.9, the behavior is inconsistent and only takes effect if a value is assigned; the None is not considered:

```
a1 = A(id=1)
a2 = A(id=2)
session.add_all([a1, a2])
session.flush()

b1 = B()
b1.a = a1   # we expect a_id to be '1'; takes precedence in 0.9 and 1.0

b2 = B()
b2.a = None  # we expect a_id to be None; takes precedence only in 1.0

b1.a_id = 2
b2.a_id = 2

session.add_all([b1, b2])
session.commit()

assert b1.a is a1  # passes in both 0.9 and 1.0
assert b2.a is None  # passes in 1.0, in 0.9 it's a2
```

#3060

session.expunge() will fully detach an object that's been deleted

The behavior of *Session.expunge()* had a bug that caused an inconsistency in behavior regarding deleted objects. The *object_session()* function as well as the *InstanceState.session* attribute would still report object as belonging to the *Session* subsequent to the expunge:

```
u1 = sess.query(User).first()
sess.delete(u1)

sess.flush()

assert u1 not in sess
assert inspect(u1).session is sess  # this is normal before commit

sess.expunge(u1)

assert u1 not in sess
assert inspect(u1).session is None  # would fail
```

Note that it is normal for `u1 not in sess` to be True while `inspect(u1).session` still refers to the session, while the transaction is ongoing subsequent to the delete operation and *Session.expunge()* has not been called; the full detachment normally completes once the transaction is committed. This issue would also impact functions that rely on *Session.expunge()* such as *make_transient()*.

#3139

Joined/Subquery eager loading explicitly disallowed with yield_per

In order to make the *Query.yield_per()* method easier to use, an exception is raised if any subquery eager loaders, or joined eager loaders that would use collections, are to take effect when yield_per is used, as these are currently not compatible with yield-per (subquery loading could be in theory, however). When this error is raised, the *lazyload()* option can be sent with an asterisk:

```
q = sess.query(Object).options(lazyload('*')).yield_per(100)
```

or use *Query.enable_eagerloads()*:

```
q = sess.query(Object).enable_eagerloads(False).yield_per(100)
```

The *lazyload()* option has the advantage that additional many-to-one joined loader options can still be used:

```
q = sess.query(Object).options(
    lazyload('*'), joinedload("some_manytoone")).yield_per(100)
```

Changes and fixes in handling of duplicate join targets

Changes here encompass bugs where an unexpected and inconsistent behavior would occur in some scenarios when joining to an entity twice, or to multple single-table entities against the same table, without using a relationship-based ON clause, as well as when joining multiple times to the same target relationship.

Starting with a mapping as:

```
from sqlalchemy import Integer, Column, String, ForeignKey
from sqlalchemy.orm import Session, relationship
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class A(Base):
    __tablename__ = 'a'
    id = Column(Integer, primary_key=True)
    bs = relationship("B")

class B(Base):
    __tablename__ = 'b'
    id = Column(Integer, primary_key=True)
    a_id = Column(ForeignKey('a.id'))
```

A query that joins to `A.bs` twice:

```
print s.query(A).join(A.bs).join(A.bs)
```

Will render:

```
SELECT a.id AS a_id
FROM a JOIN b ON a.id = b.a_id
```

The query deduplicates the redundant `A.bs` because it is attempting to support a case like the following:

```
s.query(A).join(A.bs).\
    filter(B.foo == 'bar').\
    reset_joinpoint().join(A.bs, B.cs).filter(C.bar == 'bat')
```

That is, the `A.bs` is part of a "path". As part of #3367, arriving at the same endpoint twice without it being part of a larger path will now emit a warning:

```
SAWarning: Pathed join target A.bs has already been joined to; skipping
```

The bigger change involves when joining to an entity without using a relationship-bound path. If we join to `B` twice:

```
print s.query(A).join(B, B.a_id == A.id).join(B, B.a_id == A.id)
```

In 0.9, this would render as follows:

```
SELECT a.id AS a_id
FROM a JOIN b ON b.a_id = a.id JOIN b AS b_1 ON b_1.a_id = a.id
```

This is problematic since the aliasing is implicit and in the case of different ON clauses can lead to unpredictable results.

In 1.0, no automatic aliasing is applied and we get:

```
SELECT a.id AS a_id
FROM a JOIN b ON b.a_id = a.id JOIN b ON b.a_id = a.id
```

This will raise an error from the database. While it might be nice if the "duplicate join target" acted identically if we joined both from redundant relationships vs. redundant non-relationship based targets, for now we are only changing the behavior in the more serious case where implicit aliasing would have occurred previously, and only emitting a warning in the relationship case. Ultimately, joining to the same thing twice without any aliasing to disambiguate should raise an error in all cases.

The change also has an impact on single-table inheritance targets. Using a mapping as follows:

```
from sqlalchemy import Integer, Column, String, ForeignKey
from sqlalchemy.orm import Session, relationship
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class A(Base):
    __tablename__ = "a"

    id = Column(Integer, primary_key=True)
    type = Column(String)

    __mapper_args__ = {'polymorphic_on': type, 'polymorphic_identity': 'a'}


class ASub1(A):
    __mapper_args__ = {'polymorphic_identity': 'asub1'}


class ASub2(A):
    __mapper_args__ = {'polymorphic_identity': 'asub2'}


class B(Base):
    __tablename__ = 'b'

    id = Column(Integer, primary_key=True)

    a_id = Column(Integer, ForeignKey("a.id"))

    a = relationship("A", primaryjoin="B.a_id == A.id", backref='b')

s = Session()

print s.query(ASub1).join(B, ASub1.b).join(ASub2, B.a)

print s.query(ASub1).join(B, ASub1.b).join(ASub2, ASub2.id == B.a_id)
```

The two queries at the bottom are equivalent, and should both render the identical SQL:

```
SELECT a.id AS a_id, a.type AS a_type
FROM a JOIN b ON b.a_id = a.id JOIN a ON b.a_id = a.id AND a.type IN (:type_1)
WHERE a.type IN (:type_2)
```

The above SQL is invalid, as it renders "a" within the FROM list twice. However, the implicit aliasing bug would occur with the second query only and render this instead:

```
SELECT a.id AS a_id, a.type AS a_type
FROM a JOIN b ON b.a_id = a.id JOIN a AS a_1
```

```
ON a_1.id = b.a_id AND a_1.type IN (:type_1)
WHERE a_1.type IN (:type_2)
```

Where above, the second join to "a" is aliased. While this seems convenient, it's not how single-inheritance queries work in general and is misleading and inconsistent.

The net effect is that applications which were relying on this bug will now have an error raised by the database. The solution is to use the expected form. When referring to multiple subclasses of a single-inheritance entity in a query, you must manually use aliases to disambiguate the table, as all the subclasses normally refer to the same table:

```
asub2_alias = aliased(ASub2)

print s.query(ASub1).join(B, ASub1.b).join(asub2_alias, B.a.of_type(asub2_alias))
```

#3233 #3367

Deferred Columns No Longer Implicitly Undefer

Mapped attributes marked as deferred without explicit undeferral will now remain "deferred" even if their column is otherwise present in the result set in some way. This is a performance enhancement in that an ORM load no longer spends time searching for each deferred column when the result set is obtained. However, for an application that has been relying upon this, an explicit `undefer()` or similar option should now be used, in order to prevent a SELECT from being emitted when the attribute is accessed.

Deprecated ORM Event Hooks Removed

The following ORM event hooks, some of which have been deprecated since 0.5, have been removed: `translate_row`, `populate_instance`, `append_result`, `create_instance`. The use cases for these hooks originated in the very early 0.1 / 0.2 series of SQLAlchemy and have long since been unnecessary. In particular, the hooks were largely unusable as the behavioral contracts within these events was strongly linked to the surrounding internals, such as how an instance needs to be created and initialized as well as how columns are located within an ORM-generated row. The removal of these hooks greatly simplifies the mechanics of ORM object loading.

API Change for new Bundle feature when custom row loaders are used

The new `Bundle` object of 0.9 has a small change in API, when the `create_row_processor()` method is overridden on a custom class. Previously, the sample code looked like:

```
from sqlalchemy.orm import Bundle

class DictBundle(Bundle):
    def create_row_processor(self, query, procs, labels):
        """Override create_row_processor to return values as dictionaries"""
        def proc(row, result):
            return dict(
                        zip(labels, (proc(row, result) for proc in procs))
                    )
        return proc
```

The unused `result` member is now removed:

```
from sqlalchemy.orm import Bundle

class DictBundle(Bundle):
    def create_row_processor(self, query, procs, labels):
        """Override create_row_processor to return values as dictionaries"""
        def proc(row):
            return dict(
                        zip(labels, (proc(row) for proc in procs))
                    )
        return proc
```

See also:

Column Bundles

Right inner join nesting now the default for joinedload with innerjoin=True

The behavior of `joinedload.innerjoin` as well as `relationship.innerjoin` is now to use "nested" inner joins, that is, right-nested, as the default behavior when an inner join joined eager load is chained to an outer join eager load. In order to get the old behavior of chaining all joined eager loads as outer join when an outer join is present, use `innerjoin="unnested"`.

As introduced in *Right-nested inner joins available in joined eager loads* from version 0.9, the behavior of `innerjoin="nested"` is that an inner join eager load chained to an outer join eager load will use a right-nested join. `"nested"` is now implied when using `innerjoin=True`:

```
query(User).options(
    joinedload("orders", innerjoin=False).joinedload("items", innerjoin=True))
```

With the new default, this will render the FROM clause in the form:

```
FROM users LEFT OUTER JOIN (orders JOIN items ON <onclause>) ON <onclause>
```

That is, using a right-nested join for the INNER join so that the full result of `users` can be returned. The use of an INNER join is more efficient than using an OUTER join, and allows the `joinedload.innerjoin` optimization parameter to take effect in all cases.

To get the older behavior, use `innerjoin="unnested"`:

```
query(User).options(
    joinedload("orders", innerjoin=False).joinedload("items", innerjoin="unnested"))
```

This will avoid right-nested joins and chain the joins together using all OUTER joins despite the innerjoin directive:

```
FROM users LEFT OUTER JOIN orders ON <onclause> LEFT OUTER JOIN items ON <onclause>
```

As noted in the 0.9 notes, the only database backend that has difficulty with right-nested joins is SQLite; SQLAlchemy as of 0.9 converts a right-nested join into a subquery as a join target on SQLite.

See also:

Right-nested inner joins available in joined eager loads - description of the feature as introduced in 0.9.4.

#3008

Subqueries no longer applied to uselist=False joined eager loads

Given a joined eager load like the following:

```
class A(Base):
    __tablename__ = 'a'
    id = Column(Integer, primary_key=True)
    b = relationship("B", uselist=False)


class B(Base):
    __tablename__ = 'b'
    id = Column(Integer, primary_key=True)
    a_id = Column(ForeignKey('a.id'))

s = Session()
print(s.query(A).options(joinedload(A.b)).limit(5))
```

SQLAlchemy considers the relationship `A.b` to be a "one to many, loaded as a single value", which is essentially a "one to one" relationship. However, joined eager loading has always treated the above as a situation where the main query needs to be inside a subquery, as would normally be needed for a collection of B objects where the main query has a LIMIT applied:

```
SELECT anon_1.a_id AS anon_1_a_id, b_1.id AS b_1_id, b_1.a_id AS b_1_a_id
FROM (SELECT a.id AS a_id
FROM a LIMIT :param_1) AS anon_1
LEFT OUTER JOIN b AS b_1 ON anon_1.a_id = b_1.a_id
```

However, since the relationship of the inner query to the outer one is that at most only one row is shared in the case of `uselist=False` (in the same way as a many-to-one), the "subquery" used with LIMIT + joined eager loading is now dropped in this case:

```
SELECT a.id AS a_id, b_1.id AS b_1_id, b_1.a_id AS b_1_a_id
FROM a LEFT OUTER JOIN b AS b_1 ON a.id = b_1.a_id
LIMIT :param_1
```

In the case that the LEFT OUTER JOIN returns more than one row, the ORM has always emitted a warning here and ignored addtional results for `uselist=False`, so the results in that error situation should not change.

#3249

query.update() / query.delete() raises if used with join(), select_from(), from_self()

A warning is emitted in SQLAlchemy 0.9.10 (not yet released as of June 9, 2015) when the *Query.update()* or *Query.delete()* methods are invoked against a query which has also called upon *Query.join()*, *Query.outerjoin()*, *Query.select_from()* or *Query.from_self()*. These are unsupported use cases which silently fail in the 0.9 series up until 0.9.10 where it emits a warning. In 1.0, these cases raise an exception.

#3349

query.update() with `synchronize_session='evaluate'` raises on multi-table update

The "evaluator" for *Query.update()* won't work with multi-table updates, and needs to be set to `synchronize_session=False` or `synchronize_session='fetch'` when multiple tables are present.

The new behavior is that an explicit exception is now raised, with a message to change the synchronize setting. This is upgraded from a warning emitted as of 0.9.7.

#3117

Resurrect Event has been Removed

The "resurrect" ORM event has been removed entirely. This event ceased to have any function since version 0.8 removed the older "mutable" system from the unit of work.

Change to single-table-inheritance criteria when using from_self(), count()

Given a single-table inheritance mapping, such as:

```
class Widget(Base):
    __table__ = 'widget_table'

class FooWidget(Widget):
    pass
```

Using `Query.from_self()` or `Query.count()` against a subclass would produce a subquery, but then add the "WHERE" criteria for subtypes to the outside:

```
sess.query(FooWidget).from_self().all()
```

rendering:

```
SELECT
    anon_1.widgets_id AS anon_1_widgets_id,
    anon_1.widgets_type AS anon_1_widgets_type
FROM (SELECT widgets.id AS widgets_id, widgets.type AS widgets_type,
FROM widgets) AS anon_1
WHERE anon_1.widgets_type IN (?)
```

The issue with this is that if the inner query does not specify all columns, then we can't add the WHERE clause on the outside (it actually tries, and produces a bad query). This decision apparently goes way back to 0.6.5 with the note "may need to make more adjustments to this". Well, those adjustments have arrived! So now the above query will render:

```
SELECT
    anon_1.widgets_id AS anon_1_widgets_id,
    anon_1.widgets_type AS anon_1_widgets_type
FROM (SELECT widgets.id AS widgets_id, widgets.type AS widgets_type,
FROM widgets
WHERE widgets.type IN (?)) AS anon_1
```

So that queries that don't include "type" will still work!:

```
sess.query(FooWidget.id).count()
```

Renders:

```
SELECT count(*) AS count_1
FROM (SELECT widgets.id AS widgets_id
FROM widgets
WHERE widgets.type IN (?)) AS anon_1
```

#3177

single-table-inheritance criteria added to all ON clauses unconditionally

When joining to a single-table inheritance subclass target, the ORM always adds the "single table criteria" when joining on a relationship. Given a mapping as:

```
class Widget(Base):
    __tablename__ = 'widget'
    id = Column(Integer, primary_key=True)
    type = Column(String)
    related_id = Column(ForeignKey('related.id'))
    related = relationship("Related", backref="widget")
    __mapper_args__ = {'polymorphic_on': type}


class FooWidget(Widget):
    __mapper_args__ = {'polymorphic_identity': 'foo'}


class Related(Base):
    __tablename__ = 'related'
    id = Column(Integer, primary_key=True)
```

It's been the behavior for quite some time that a JOIN on the relationship will render a "single inheritance" clause for the type:

```
s.query(Related).join(FooWidget, Related.widget).all()
```

SQL output:

```
SELECT related.id AS related_id
FROM related JOIN widget ON related.id = widget.related_id AND widget.type IN (:type_1)
```

Above, because we joined to a subclass `FooWidget`, *`Query.join()`* knew to add the `AND widget.type IN ('foo')` criteria to the ON clause.

The change here is that the `AND widget.type IN()` criteria is now appended to *any* ON clause, not just those generated from a relationship, including one that is explicitly stated:

```
# ON clause will now render as
# related.id = widget.related_id AND widget.type IN (:type_1)
s.query(Related).join(FooWidget, FooWidget.related_id == Related.id).all()
```

As well as the "implicit" join when no ON clause of any kind is stated:

```
# ON clause will now render as
# related.id = widget.related_id AND widget.type IN (:type_1)
s.query(Related).join(FooWidget).all()
```

Previously, the ON clause for these would not include the single-inheritance criteria. Applications that are already adding this criteria to work around this will want to remove its explicit use, though it should continue to work fine if the criteria happens to be rendered twice in the meantime.

See also:

Changes and fixes in handling of duplicate join targets

#3222

Key Behavioral Changes - Core

Warnings emitted when coercing full SQL fragments into text()

Since SQLAlchemy's inception, there has always been an emphasis on not getting in the way of the usage of plain text. The Core and ORM expression systems were intended to allow any number of points at which the user can just use plain text SQL expressions, not just in the sense that you can send a full SQL string to `Connection.execute()`, but that you can send strings with SQL expressions into many functions, such as `Select.where()`, `Query.filter()`, and `Select.order_by()`.

Note that by "SQL expressions" we mean a **full fragment of a SQL string**, such as:

```
# the argument sent to where() is a full SQL expression
stmt = select([sometable]).where("somecolumn = 'value'")
```

and we are **not talking about string arguments**, that is, the normal behavior of passing string values that become parameterized:

```
# This is a normal Core expression with a string argument -
# we aren't talking about this!!
stmt = select([sometable]).where(sometable.c.somecolumn == 'value')
```

The Core tutorial has long featured an example of the use of this technique, using a `select()` construct where virtually all components of it are specified as straight strings. However, despite this long-standing behavior and example, users are apparently surprised that this behavior exists, and when asking around the community, I was unable to find any user that was in fact *not* surprised that you can send a full string into a method like `Query.filter()`.

So the change here is to encourage the user to qualify textual strings when composing SQL that is partially or fully composed from textual fragments. When composing a select as below:

```
stmt = select(["a", "b"]).where("a = b").select_from("sometable")
```

The statement is built up normally, with all the same coercions as before. However, one will see the following warnings emitted:

```
SAWarning: Textual column expression 'a' should be explicitly declared
with text('a'), or use column('a') for more specificity
(this warning may be suppressed after 10 occurrences)

SAWarning: Textual column expression 'b' should be explicitly declared
with text('b'), or use column('b') for more specificity
(this warning may be suppressed after 10 occurrences)

SAWarning: Textual SQL expression 'a = b' should be explicitly declared
as text('a = b') (this warning may be suppressed after 10 occurrences)

SAWarning: Textual SQL FROM expression 'sometable' should be explicitly
```

```
declared as text('sometable'), or use table('sometable') for more
specificity (this warning may be suppressed after 10 occurrences)
```

These warnings attempt to show exactly where the issue is by displaying the parameters as well as where the string was received. The warnings make use of the *Session.get_bind() handles a wider variety of inheritance scenarios* so that parameterized warnings can be emitted safely without running out of memory, and as always, if one wishes the warnings to be exceptions, the Python Warnings Filter should be used:

```
import warnings
warnings.simplefilter("error")   # all warnings raise an exception
```

Given the above warnings, our statement works just fine, but to get rid of the warnings we would rewrite our statement as follows:

```
from sqlalchemy import select, text
stmt = select([
        text("a"),
        text("b")
    ]).where(text("a = b")).select_from(text("sometable"))
```

and as the warnings suggest, we can give our statement more specificity about the text if we use `column()` and `table()`:

```
from sqlalchemy import select, text, column, table

stmt = select([column("a"), column("b")]).\
    where(text("a = b")).select_from(table("sometable"))
```

Where note also that `table()` and `column()` can now be imported from "sqlalchemy" without the "sql" part.

The behavior here applies to `select()` as well as to key methods on `Query`, including `Query.filter()`, `Query.from_statement()` and `Query.having()`.

ORDER BY and GROUP BY are special cases There is one case where usage of a string has special meaning, and as part of this change we have enhanced its functionality. When we have a `select()` or `Query` that refers to some column name or named label, we might want to GROUP BY and/or ORDER BY known columns or labels:

```
stmt = select([
    user.c.name,
    func.count(user.c.id).label("id_count")
]).group_by("name").order_by("id_count")
```

In the above statement we expect to see "ORDER BY id_count", as opposed to a re-statement of the function. The string argument given is actively matched to an entry in the columns clause during compilation, so the above statement would produce as we expect, without warnings (though note that the `"name"` expression has been resolved to `users.name`!):

```
SELECT users.name, count(users.id) AS id_count
FROM users GROUP BY users.name ORDER BY id_count
```

However, if we refer to a name that cannot be located, then we get the warning again, as below:

```
stmt = select([
        user.c.name,
        func.count(user.c.id).label("id_count")
    ]).order_by("some_label")
```

The output does what we say, but again it warns us:

```
SAWarning: Can't resolve label reference 'some_label'; converting to
text() (this warning may be suppressed after 10 occurrences)

SELECT users.name, count(users.id) AS id_count
FROM users ORDER BY some_label
```

The above behavior applies to all those places where we might want to refer to a so-called "label reference"; ORDER BY and GROUP BY, but also within an OVER clause as well as a DISTINCT ON clause that refers to columns (e.g. the Postgresql syntax).

We can still specify any arbitrary expression for ORDER BY or others using `text()`:

```
stmt = select([users]).order_by(text("some special expression"))
```

The upshot of the whole change is that SQLAlchemy now would like us to tell it when a string is sent that this string is explicitly a `text()` construct, or a column, table, etc., and if we use it as a label name in an order by, group by, or other expression, SQLAlchemy expects that the string resolves to something known, else it should again be qualified with `text()` or similar.

#2992

Python-side defaults invoked for each row invidually when using a multivalued insert

Support for Python-side column defaults when using the multi-valued version of `Insert.values()` were essentially not implemented, and would only work "by accident" in specific situations, when the dialect in use was using a non-positional (e.g. named) style of bound parameter, and when it was not necessary that a Python-side callable be invoked for each row.

The feature has been overhauled so that it works more similarly to that of an "executemany" style of invocation:

```
import itertools

counter = itertools.count(1)
t = Table(
    'my_table', metadata,
    Column('id', Integer, default=lambda: next(counter)),
    Column('data', String)
)

conn.execute(t.insert().values([
    {"data": "d1"},
    {"data": "d2"},
    {"data": "d3"},
]))
```

The above example will invoke `next(counter)` for each row individually as would be expected:

```
INSERT INTO my_table (id, data) VALUES (?, ?), (?, ?), (?, ?)
(1, 'd1', 2, 'd2', 3, 'd3')
```

Previously, a positional dialect would fail as a bind would not be generated for additional positions:

```
Incorrect number of bindings supplied. The current statement uses 6,
and there are 4 supplied.
[SQL: u'INSERT INTO my_table (id, data) VALUES (?, ?), (?, ?), (?, ?)']
[parameters: (1, 'd1', 'd2', 'd3')]
```

And with a "named" dialect, the same value for "id" would be re-used in each row (hence this change is backwards-incompatible with a system that relied on this):

```
INSERT INTO my_table (id, data) VALUES (:id, :data_0), (:id, :data_1), (:id, :data_2)
{u'data_2': 'd3', u'data_1': 'd2', u'data_0': 'd1', 'id': 1}
```

The system will also refuse to invoke a "server side" default as inline-rendered SQL, since it cannot be guaranteed that a server side default is compatible with this. If the VALUES clause renders for a specific column, then a Python-side value is required; if an omitted value only refers to a server-side default, an exception is raised:

```
t = Table(
    'my_table', metadata,
    Column('id', Integer, primary_key=True),
    Column('data', String, server_default='some default')
)

conn.execute(t.insert().values([
    {"data": "d1"},
    {"data": "d2"},
    {},
]))
```

will raise:

```
sqlalchemy.exc.CompileError: INSERT value for column my_table.data is
explicitly rendered as a boundparameter in the VALUES clause; a
Python-side value or SQL expression is required
```

Previously, the value "d1" would be copied into that of the third row (but again, only with named format!):

```
INSERT INTO my_table (data) VALUES (:data_0), (:data_1), (:data_0)
{u'data_1': 'd2', u'data_0': 'd1'}
```

#3288

Event listeners can not be added or removed from within that event's runner

Removal of an event listener from inside that same event itself would modify the elements of a list during iteration, which would cause still-attached event listeners to silently fail to fire. To prevent this while still maintaining performance, the lists have been replaced with `collections.deque()`, which does not allow any additions or removals during iteration, and instead raises `RuntimeError`.

#3163

The INSERT...FROM SELECT construct now implies `inline=True`

Using *Insert.from_select()* now implies `inline=True` on *insert()*. This helps to fix a bug where an INSERT...FROM SELECT construct would inadvertently be compiled as "implicit returning" on supporting backends, which would cause breakage in the case of an INSERT that inserts zero rows (as implicit returning expects a row), as well as arbitrary return data in the case of an INSERT that inserts multiple rows (e.g. only the first row of many). A similar change is also applied to an INSERT..VALUES with multiple parameter sets; implicit RETURNING will no longer emit for this statement either. As both of these constructs deal with varible numbers of rows, the *ResultProxy.inserted_primary_key* accessor does not apply. Previously, there was a documentation note that one may prefer `inline=True` with INSERT..FROM SELECT as some databases don't support returning and therefore can't do "implicit" returning, but there's no reason an INSERT...FROM SELECT needs implicit returning in any case. Regular explicit *Insert.returning()* should be used to return variable numbers of result rows if inserted data is needed.

#3169

`autoload_with` now implies `autoload=True`

A *Table* can be set up for reflection by passing *Table.autoload_with* alone:

```
my_table = Table('my_table', metadata, autoload_with=some_engine)
```

#3027

DBAPI exception wrapping and handle_error() event improvements

SQLAlchemy's wrapping of DBAPI exceptions was not taking place in the case where a *Connection* object was invalidated, and then tried to reconnect and encountered an error; this has been resolved.

Additionally, the recently added *ConnectionEvents.handle_error()* event is now invoked for errors that occur upon initial connect, upon reconnect, and when *create_engine()* is used given a custom connection function via *create_engine.creator*.

The *ExceptionContext* object has a new datamember *ExceptionContext.engine* that will always refer to the *Engine* in use, in those cases when the *Connection* object is not available (e.g. on initial connect).

#3266

ForeignKeyConstraint.columns is now a ColumnCollection

`ForeignKeyConstraint.columns` was previously a plain list containing either strings or *Column* objects, depending on how the *ForeignKeyConstraint* was constructed and whether it was associated with a table. The collection is now a *ColumnCollection*, and is only initialized after the *ForeignKeyConstraint* is associated with a *Table*. A new accessor *ForeignKeyConstraint.column_keys* is added to unconditionally return string keys for the local set of columns regardless of how the object was constructed or its current state.

MetaData.sorted_tables accessor is "deterministic"

The sorting of tables resulting from the *MetaData.sorted_tables* accessor is "deterministic"; the ordering should be the same in all cases regardless of Python hashing. This is done by first sorting the tables by name before passing them to the topological algorithm, which maintains that ordering as it iterates.

Note that this change does **not** yet apply to the ordering applied when emitting *MetaData.create_all()* or *MetaData.drop_all()*.

#3084

null(), false() and true() constants are no longer singletons

These three constants were changed to return a "singleton" value in 0.9; unfortunately, that would lead to a query like the following to not render as expected:

```
select([null(), null()])
```

rendering only `SELECT NULL AS anon_1`, because the two *null()* constructs would come out as the same `NULL` object, and SQLAlchemy's Core model is based on object identity in order to determine lexical significance. The change in 0.9 had no importance other than the desire to save on object overhead; in general, an unnamed construct needs to stay lexically unique so that it gets labeled uniquely.

#3170

SQLite/Oracle have distinct methods for temporary table/view name reporting

The *Inspector.get_table_names()* and *Inspector.get_view_names()* methods in the case of SQLite/Oracle would also return the names of temporary tables and views, which is not provided by any other dialect (in the case of MySQL at least it is not even possible). This logic has been moved out to two new methods *Inspector.get_temp_table_names()* and *Inspector.get_temp_view_names()*.

Note that reflection of a specific named temporary table or temporary view, either by `Table('name', autoload=True)` or via methods like *Inspector.get_columns()* continues to function for most if not all dialects. For SQLite specifically, there is a bug fix for UNIQUE constraint reflection from temp tables as well, which is #3203.

#3204

Dialect Improvements and Changes - Postgresql

Overhaul of ENUM type create/drop rules

The rules for Postgresql *postgresql.ENUM* have been made more strict with regards to creating and dropping of the TYPE.

An *postgresql.ENUM* that is created **without** being explicitly associated with a *MetaData* object will be created *and* dropped corresponding to *Table.create()* and *Table.drop()*:

```
table = Table('sometable', metadata,
    Column('some_enum', ENUM('a', 'b', 'c', name='myenum'))
)

table.create(engine)  # will emit CREATE TYPE and CREATE TABLE
table.drop(engine)  # will emit DROP TABLE and DROP TYPE - new for 1.0
```

This means that if a second table also has an enum named 'myenum', the above DROP operation will now fail. In order to accomodate the use case of a common shared enumerated type, the behavior of a metadata-associated enumeration has been enhanced.

An *postgresql.ENUM* that is created **with** being explicitly associated with a *MetaData* object will *not* be created *or* dropped corresponding to *Table.create()* and *Table.drop()*, with the exception of *Table.create()* called with the `checkfirst=True` flag:

```
my_enum = ENUM('a', 'b', 'c', name='myenum', metadata=metadata)

table = Table('sometable', metadata,
    Column('some_enum', my_enum)
)

# will fail: ENUM 'my_enum' does not exist
table.create(engine)

# will check for enum and emit CREATE TYPE
table.create(engine, checkfirst=True)

table.drop(engine)  # will emit DROP TABLE, *not* DROP TYPE

metadata.drop_all(engine) # will emit DROP TYPE

metadata.create_all(engine) # will emit CREATE TYPE
```

#3319

New Postgresql Table options

Added support for PG table options TABLESPACE, ON COMMIT, WITH(OUT) OIDS, and INHERITS, when rendering DDL via the *Table* construct.

See also:

PostgreSQL Table Options

#2051

New get_enums() method with Postgresql Dialect

The *inspect()* method returns a *PGInspector* object in the case of Postgresql, which includes a new *PGInspector.get_enums()* method that returns information on all available `ENUM` types:

```
from sqlalchemy import inspect, create_engine

engine = create_engine("postgresql+psycopg2://host/dbname")
insp = inspect(engine)
print(insp.get_enums())
```

See also:

PGInspector.get_enums()

Postgresql Dialect reflects Materialized Views, Foreign Tables

Changes are as follows:

- the `Table` construct with `autoload=True` will now match a name that exists in the database as a materialized view or foreign table.

- *Inspector.get_view_names()* will return plain and materialized view names.
- *Inspector.get_table_names()* does **not** change for Postgresql, it continues to return only the names of plain tables.
- A new method *PGInspector.get_foreign_table_names()* is added which will return the names of tables that are specifically marked as "foreign" in the Postgresql schema tables.

The change to reflection involves adding `'m'` and `'f'` to the list of qualifiers we use when querying `pg_class.relkind`, but this change is new in 1.0.0 to avoid any backwards-incompatible surprises for those running 0.9 in production.

#2891

Postgresql `has_table()` now works for temporary tables

This is a simple fix such that "has table" for temporary tables now works, so that code like the following may proceed:

```
from sqlalchemy import *

metadata = MetaData()
user_tmp = Table(
    "user_tmp", metadata,
    Column("id", INT, primary_key=True),
    Column('name', VARCHAR(50)),
    prefixes=['TEMPORARY']
)

e = create_engine("postgresql://scott:tiger@localhost/test", echo='debug')
with e.begin() as conn:
    user_tmp.create(conn, checkfirst=True)

    # checkfirst will succeed
    user_tmp.create(conn, checkfirst=True)
```

The very unlikely case that this behavior will cause a non-failing application to behave differently, is because Postgresql allows a non-temporary table to silently overwrite a temporary table. So code like the following will now act completely differently, no longer creating the real table following the temporary table:

```
from sqlalchemy import *

metadata = MetaData()
user_tmp = Table(
    "user_tmp", metadata,
    Column("id", INT, primary_key=True),
    Column('name', VARCHAR(50)),
    prefixes=['TEMPORARY']
)

e = create_engine("postgresql://scott:tiger@localhost/test", echo='debug')
with e.begin() as conn:
    user_tmp.create(conn, checkfirst=True)

    m2 = MetaData()
    user = Table(
        "user_tmp", m2,
        Column("id", INT, primary_key=True),
        Column('name', VARCHAR(50)),
```

```
    )

    # in 0.9, *will create* the new table, overwriting the old one.
    # in 1.0, *will not create* the new table
    user.create(conn, checkfirst=True)
```

#3264

Postgresql FILTER keyword

The SQL standard FILTER keyword for aggregate functions is now supported by Postgresql as of 9.4. SQLAlchemy allows this using `FunctionElement.filter()`:

```
func.count(1).filter(True)
```

See also:

`FunctionElement.filter()`

`FunctionFilter`

PG8000 dialect supports client side encoding

The `create_engine.encoding` parameter is now honored by the pg8000 dialect, using on connect handler which emits `SET CLIENT_ENCODING` matching the selected encoding.

PG8000 native JSONB support

Support for PG8000 versions greater than 1.10.1 has been added, where JSONB is supported natively.

Support for psycopg2cffi Dialect on Pypy

Support for the pypy psycopg2cffi dialect is added.

See also:

`sqlalchemy.dialects.postgresql.psycopg2cffi`

Dialect Improvements and Changes - MySQL

MySQL TIMESTAMP Type now renders NULL / NOT NULL in all cases

The MySQL dialect has always worked around MySQL's implicit NOT NULL default associated with TIMESTAMP columns by emitting NULL for such a type, if the column is set up with `nullable=True`. However, MySQL 5.6.6 and above features a new flag explicit_defaults_for_timestamp which repairs MySQL's non-standard behavior to make it behave like any other type; to accommodate this, SQLAlchemy now emits NULL/NOT NULL unconditionally for all TIMESTAMP columns.

See also:

TIMESTAMP Columns and NULL

#3155

MySQL SET Type Overhauled to support empty sets, unicode, blank value handling

The *mysql.SET* type historically not included a system of handling blank sets and empty values separately; as different drivers had different behaviors for treatment of empty strings and empty-string-set representations, the SET type tried only to hedge between these behaviors, opting to treat the empty set as `set(['])` as is still the current behavior for the MySQL-Connector-Python DBAPI. Part of the rationale here was that it was otherwise impossible to actually store a blank string within a MySQL SET, as the driver gives us back strings with no way to discern between `set(['])` and `set()`. It was left to the user to determine if `set(['])` actually meant "empty set" or not.

The new behavior moves the use case for the blank string, which is an unusual case that isn't even documented in MySQL's documentation, into a special case, and the default behavior of *mysql.SET* is now:

- to treat the empty string `''` as returned by MySQL-python into the empty set `set()`;
- to convert the single-blank value set `set(['])` returned by MySQL-Connector-Python into the empty set `set()`;
- To handle the case of a set type that actually wishes includes the blank value `''` in its list of possible values, a new feature (required in this use case) is implemented whereby the set value is persisted and loaded as a bitwise integer value; the flag *mysql.SET.retrieve_as_bitwise* is added in order to enable this.

Using the *mysql.SET.retrieve_as_bitwise* flag allows the set to be persisted and retrieved with no ambiguity of values. Theoretically this flag can be turned on in all cases, as long as the given list of values to the type matches the ordering exactly as declared in the database; it only makes the SQL echo output a bit more unusual.

The default behavior of *mysql.SET* otherwise remains the same, roundtripping values using strings. The string-based behavior now supports unicode fully including MySQL-python with use_unicode=0.

#3283

MySQL internal "no such table" exceptions not passed to event handlers

The MySQL dialect will now disable *ConnectionEvents.handle_error()* events from firing for those statements which it uses internally to detect if a table exists or not. This is achieved using an execution option `skip_user_error_events` that disables the handle error event for the scope of that execution. In this way, user code that rewrites exceptions doesn't need to worry about the MySQL dialect or other dialects that occasionally need to catch SQLAlchemy specific exceptions.

Changed the default value of `raise_on_warnings` for MySQL-Connector

Changed the default value of "raise_on_warnings" to False for MySQL-Connector. This was set at True for some reason. The "buffered" flag unfortunately must stay at True as MySQLconnector does not allow a cursor to be closed unless all results are fully fetched.

#2515

MySQL boolean symbols "true", "false" work again

0.9's overhaul of the IS/IS NOT operators as well as boolean types in #2682 disallowed the MySQL dialect from making use of the "true" and "false" symbols in the context of "IS" / "IS NOT". Apparently, even though MySQL has no "boolean" type, it supports IS / IS NOT when the special "true" and "false" symbols are used, even though these are otherwise synonymous with "1" and "0" (and IS/IS NOT don't work with the numerics).

So the change here is that the MySQL dialect remains "non native boolean", but the `true()` and `false()` symbols again produce the keywords "true" and "false", so that an expression like `column.is_(true())` again works on MySQL.

#3186

The match() operator now returns an agnostic MatchType compatible with MySQL's floating point return value

The return type of a `ColumnOperators.match()` expression is now a new type called `MatchType`. This is a subclass of `Boolean`, that can be intercepted by the dialect in order to produce a different result type at SQL execution time.

Code like the following will now function correctly and return floating points on MySQL:

```
>>> connection.execute(
...    select([
...        matchtable.c.title.match('Agile Ruby Programming').label('ruby'),
...        matchtable.c.title.match('Dive Python').label('python'),
...        matchtable.c.title
...    ]).order_by(matchtable.c.id)
... )
[
    (2.0, 0.0, 'Agile Web Development with Ruby On Rails'),
    (0.0, 2.0, 'Dive Into Python'),
    (2.0, 0.0, "Programming Matz's Ruby"),
    (0.0, 0.0, 'The Definitive Guide to Django'),
    (0.0, 1.0, 'Python in a Nutshell')
]
```

#3263

Drizzle Dialect is now an External Dialect

The dialect for Drizzle is now an external dialect, available at https://bitbucket.org/zzzeek/sqlalchemy-drizzle. This dialect was added to SQLAlchemy right before SQLAlchemy was able to accommodate third party dialects well; going forward, all databases that aren't within the "ubiquitous use" category are third party dialects. The dialect's implementation hasn't changed and is still based on the MySQL + MySQLdb dialects within SQLAlchemy. The dialect is as of yet unreleased and in "attic" status; however it passes the majority of tests and is generally in decent working order, if someone wants to pick up on polishing it.

Dialect Improvements and Changes - SQLite

SQLite named and unnamed UNIQUE and FOREIGN KEY constraints will inspect and reflect

UNIQUE and FOREIGN KEY constraints are now fully reflected on SQLite both with and without names. Previously, foreign key names were ignored and unnamed unique constraints were skipped. In particular this will help with Alembic's new SQLite migration features.

To achieve this, for both foreign keys and unique constraints, the result of PRAGMA foreign_keys, index_list, and index_info is combined with regular expression parsing of the CREATE TABLE statement overall to form a complete picture of the names of constraints, as well as differentiating UNIQUE constraints that were created as UNIQUE vs. unnamed INDEXes.

#3244

#3261

Dialect Improvements and Changes - SQL Server

PyODBC driver name is required with hostname-based SQL Server connections

Connecting to SQL Server with PyODBC using a DSN-less connection, e.g. with an explicit hostname, now requires a driver name - SQLAlchemy will no longer attempt to guess a default:

```
engine = create_engine("mssql+pyodbc://scott:tiger@myhost:port/databasename?driver=SQL+Server+Native-
```

SQLAlchemy's previously hardcoded default of "SQL Server" is obsolete on Windows, and SQLAlchemy cannot be tasked with guessing the best driver based on operation system/driver detection. Using a DSN is always preferred when using ODBC to avoid this issue entirely.

#3182

SQL Server 2012 large text / binary types render as VARCHAR, NVARCHAR, VARBINARY

The rendering of the `Text`, `UnicodeText`, and `LargeBinary` types has been changed for SQL Server 2012 and greater, with options to control the behavior completely, based on deprecation guidelines from Microsoft. See *Large Text/Binary Type Deprecation* for details.

Dialect Improvements and Changes - Oracle

Improved support for CTEs in Oracle

CTE support has been fixed up for Oracle, and there is also a new feature `CTE.with_suffixes()` that can assist with Oracle's special directives:

```
included_parts = select([
    part.c.sub_part, part.c.part, part.c.quantity
]).where(part.c.part == "p1").\
    cte(name="included_parts", recursive=True).\
    suffix_with(
        "search depth first by part set ord1",
        "cycle part set y_cycle to 1 default 0", dialect='oracle')
```

#3220

New Oracle Keywords for DDL

Keywords such as COMPRESS, ON COMMIT, BITMAP:

Oracle Table Options

Oracle Specific Index Options

6.3.2 What's New in SQLAlchemy 0.9?

About this Document

This document describes changes between SQLAlchemy version 0.8, undergoing maintenance releases as of May, 2013, and SQLAlchemy version 0.9, which had its first production release on December 30, 2013.

Document last updated: June 10, 2015

Introduction

This guide introduces what's new in SQLAlchemy version 0.9, and also documents changes which affect users migrating their applications from the 0.8 series of SQLAlchemy to 0.9.

Please carefully review *Behavioral Changes - ORM* and *Behavioral Changes - Core* for potentially backwards-incompatible changes.

Platform Support

Targeting Python 2.6 and Up Now, Python 3 without 2to3

The first achievement of the 0.9 release is to remove the dependency on the 2to3 tool for Python 3 compatibility. To make this more straightforward, the lowest Python release targeted now is 2.6, which features a wide degree of cross-compatibility with Python 3. All SQLAlchemy modules and unit tests are now interpreted equally well with any Python interpreter from 2.6 forward, including the 3.1 and 3.2 interpreters.

#2671

C Extensions Supported on Python 3

The C extensions have been ported to support Python 3 and now build in both Python 2 and Python 3 environments.

#2161

Behavioral Changes - ORM

Composite attributes are now returned as their object form when queried on a per-attribute basis

Using a `Query` in conjunction with a composite attribute now returns the object type maintained by that composite, rather than being broken out into individual columns. Using the mapping setup at *Composite Column Types*:

```
>>> session.query(Vertex.start, Vertex.end).\
...     filter(Vertex.start == Point(3, 4)).all()
[(Point(x=3, y=4), Point(x=5, y=6))]
```

This change is backwards-incompatible with code that expects the individual attribute to be expanded into individual columns. To get that behavior, use the `.clauses` accessor:

```
>>> session.query(Vertex.start.clauses, Vertex.end.clauses).\
...     filter(Vertex.start == Point(3, 4)).all()
[(3, 4, 5, 6)]
```

See also:

Column Bundles for ORM queries

#2824

`Query.select_from()` no longer applies the clause to corresponding entities

The *`Query.select_from()`* method has been popularized in recent versions as a means of controlling the first thing that a *`Query`* object "selects from", typically for the purposes of controlling how a JOIN will render.

Consider the following example against the usual `User` mapping:

```
select_stmt = select([User]).where(User.id == 7).alias()

q = session.query(User).\
            join(select_stmt, User.id == select_stmt.c.id).\
            filter(User.name == 'ed')
```

The above statement predictably renders SQL like the following:

```
SELECT "user".id AS user_id, "user".name AS user_name
FROM "user" JOIN (SELECT "user".id AS id, "user".name AS name
FROM "user"
WHERE "user".id = :id_1) AS anon_1 ON "user".id = anon_1.id
WHERE "user".name = :name_1
```

If we wanted to reverse the order of the left and right elements of the JOIN, the documentation would lead us to believe we could use *`Query.select_from()`* to do so:

```
q = session.query(User).\
        select_from(select_stmt).\
        join(User, User.id == select_stmt.c.id).\
        filter(User.name == 'ed')
```

However, in version 0.8 and earlier, the above use of *`Query.select_from()`* would apply the `select_stmt` to **replace** the `User` entity, as it selects from the `user` table which is compatible with `User`:

```
-- SQLAlchemy 0.8 and earlier...
SELECT anon_1.id AS anon_1_id, anon_1.name AS anon_1_name
FROM (SELECT "user".id AS id, "user".name AS name
FROM "user"
WHERE "user".id = :id_1) AS anon_1 JOIN "user" ON anon_1.id = anon_1.id
WHERE anon_1.name = :name_1
```

The above statement is a mess, the ON clause refers `anon_1.id = anon_1.id`, our WHERE clause has been replaced with `anon_1` as well.

This behavior is quite intentional, but has a different use case from that which has become popular for *`Query.select_from()`*. The above behavior is now available by a new method known as *`Query.select_entity_from()`*. This is a lesser used behavior that in modern SQLAlchemy is roughly equivalent to selecting from a customized *`aliased()`* construct:

```
select_stmt = select([User]).where(User.id == 7)
user_from_stmt = aliased(User, select_stmt.alias())

q = session.query(user_from_stmt).filter(user_from_stmt.name == 'ed')
```

So with SQLAlchemy 0.9, our query that selects from `select_stmt` produces the SQL we expect:

```
-- SQLAlchemy 0.9
SELECT "user".id AS user_id, "user".name AS user_name
FROM (SELECT "user".id AS id, "user".name AS name
FROM "user"
WHERE "user".id = :id_1) AS anon_1 JOIN "user" ON "user".id = id
WHERE "user".name = :name_1
```

The *Query.select_entity_from()* method will be available in SQLAlchemy **0.8.2**, so applications which rely on the old behavior can transition to this method first, ensure all tests continue to function, then upgrade to 0.9 without issue.

#2736

`viewonly=True` on `relationship()` prevents history from taking effect

The `viewonly` flag on *relationship()* is applied to prevent changes to the target attribute from having any effect within the flush process. This is achieved by eliminating the attribute from being considered during the flush. However, up until now, changes to the attribute would still register the parent object as "dirty" and trigger a potential flush. The change is that the `viewonly` flag now prevents history from being set for the target attribute as well. Attribute events like backrefs and user-defined events still continue to function normally.

The change is illustrated as follows:

```
from sqlalchemy import Column, Integer, ForeignKey, create_engine
from sqlalchemy.orm import backref, relationship, Session
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import inspect

Base = declarative_base()

class A(Base):
    __tablename__ = 'a'
    id = Column(Integer, primary_key=True)

class B(Base):
    __tablename__ = 'b'

    id = Column(Integer, primary_key=True)
    a_id = Column(Integer, ForeignKey('a.id'))
    a = relationship("A", backref=backref("bs", viewonly=True))

e = create_engine("sqlite://")
Base.metadata.create_all(e)

a = A()
b = B()

sess = Session(e)
sess.add_all([a, b])
sess.commit()

b.a = a

assert b in sess.dirty
```

```
# before 0.9.0
# assert a in sess.dirty
# assert inspect(a).attrs.bs.history.has_changes()

# after 0.9.0
assert a not in sess.dirty
assert not inspect(a).attrs.bs.history.has_changes()
```

#2833

Association Proxy SQL Expression Improvements and Fixes

The == and != operators as implemented by an association proxy that refers to a scalar value on a scalar relationship now produces a more complete SQL expression, intended to take into account the "association" row being present or not when the comparison is against `None`.

Consider this mapping:

```
class A(Base):
    __tablename__ = 'a'

    id = Column(Integer, primary_key=True)

    b_id = Column(Integer, ForeignKey('b.id'), primary_key=True)
    b = relationship("B")
    b_value = association_proxy("b", "value")

class B(Base):
    __tablename__ = 'b'
    id = Column(Integer, primary_key=True)
    value = Column(String)
```

Up through 0.8, a query like the following:

```
s.query(A).filter(A.b_value == None).all()
```

would produce:

```
SELECT a.id AS a_id, a.b_id AS a_b_id
FROM a
WHERE EXISTS (SELECT 1
FROM b
WHERE b.id = a.b_id AND b.value IS NULL)
```

In 0.9, it now produces:

```
SELECT a.id AS a_id, a.b_id AS a_b_id
FROM a
WHERE (EXISTS (SELECT 1
FROM b
WHERE b.id = a.b_id AND b.value IS NULL)) OR a.b_id IS NULL
```

The difference being, it not only checks `b.value`, it also checks if `a` refers to no `b` row at all. This will return different results versus prior versions, for a system that uses this type of comparison where some parent rows have no association row.

More critically, a correct expression is emitted for `A.b_value != None`. In 0.8, this would return `True` for `A` rows that had no `b`:

```
SELECT a.id AS a_id, a.b_id AS a_b_id
FROM a
WHERE NOT (EXISTS (SELECT 1
FROM b
WHERE b.id = a.b_id AND b.value IS NULL))
```

Now in 0.9, the check has been reworked so that it ensures the A.b_id row is present, in addition to `B.value` being non-NULL:

```
SELECT a.id AS a_id, a.b_id AS a_b_id
FROM a
WHERE EXISTS (SELECT 1
FROM b
WHERE b.id = a.b_id AND b.value IS NOT NULL)
```

In addition, the `has()` operator is enhanced such that you can call it against a scalar column value with no criterion only, and it will produce criteria that checks for the association row being present or not:

```
s.query(A).filter(A.b_value.has()).all()
```

output:

```
SELECT a.id AS a_id, a.b_id AS a_b_id
FROM a
WHERE EXISTS (SELECT 1
FROM b
WHERE b.id = a.b_id)
```

This is equivalent to `A.b.has()`, but allows one to query against `b_value` directly.

#2751

Association Proxy Missing Scalar returns None

An association proxy from a scalar attribute to a scalar will now return `None` if the proxied object isn't present. This is consistent with the fact that missing many-to-ones return None in SQLAlchemy, so should the proxied value. E.g.:

```
from sqlalchemy import *
from sqlalchemy.orm import *
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.ext.associationproxy import association_proxy

Base = declarative_base()

class A(Base):
    __tablename__ = 'a'

    id = Column(Integer, primary_key=True)
    b = relationship("B", uselist=False)

    bname = association_proxy("b", "name")

class B(Base):
```

```
    __tablename__ = 'b'

    id = Column(Integer, primary_key=True)
    a_id = Column(Integer, ForeignKey('a.id'))
    name = Column(String)

a1 = A()

# this is how m2o's always have worked
assert a1.b is None

# but prior to 0.9, this would raise AttributeError,
# now returns None just like the proxied value.
assert a1.bname is None
```

#2810

attributes.get_history() will query from the DB by default if value not present

A bugfix regarding *attributes.get_history()* allows a column-based attribute to query out to the database for an unloaded value, assuming the `passive` flag is left at its default of `PASSIVE_OFF`. Previously, this flag would not be honored. Additionally, a new method *AttributeState.load_history()* is added to complement the *AttributeState.history* attribute, which will emit loader callables for an unloaded attribute.

This is a small change demonstrated as follows:

```
from sqlalchemy import Column, Integer, String, create_engine, inspect
from sqlalchemy.orm import Session, attributes
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class A(Base):
    __tablename__ = 'a'
    id = Column(Integer, primary_key=True)
    data = Column(String)

e = create_engine("sqlite://", echo=True)
Base.metadata.create_all(e)

sess = Session(e)

a1 = A(data='a1')
sess.add(a1)
sess.commit()  # a1 is now expired

# history doesn't emit loader callables
assert inspect(a1).attrs.data.history == (None, None, None)

# in 0.8, this would fail to load the unloaded state.
assert attributes.get_history(a1, 'data') == ((), ['a1',], ())

# load_history() is now equiavlent to get_history() with
# passive=PASSIVE_OFF ^ INIT_OK
assert inspect(a1).attrs.data.load_history() == ((), ['a1',], ())
```

#2787

Behavioral Changes - Core

Type objects no longer accept ignored keyword arguments

Up through the 0.8 series, most type objects accepted arbitrary keyword arguments which were silently ignored:

```
from sqlalchemy import Date, Integer

# storage_format argument here has no effect on any backend;
# it needs to be on the SQLite-specific type
d = Date(storage_format="%(day)02d.%(month)02d.%(year)04d")

# display_width argument here has no effect on any backend;
# it needs to be on the MySQL-specific type
i = Integer(display_width=5)
```

This was a very old bug for which a deprecation warning was added to the 0.8 series, but because nobody ever runs Python with the "-W" flag, it was mostly never seen:

```
$ python -W always::DeprecationWarning ~/dev/sqlalchemy/test.py
/Users/classic/dev/sqlalchemy/test.py:5: SADeprecationWarning: Passing arguments to
type object constructor <class 'sqlalchemy.types.Date'> is deprecated
  d = Date(storage_format="%(day)02d.%(month)02d.%(year)04d")
/Users/classic/dev/sqlalchemy/test.py:9: SADeprecationWarning: Passing arguments to
type object constructor <class 'sqlalchemy.types.Integer'> is deprecated
  i = Integer(display_width=5)
```

As of the 0.9 series the "catch all" constructor is removed from *TypeEngine*, and these meaningless arguments are no longer accepted.

The correct way to make use of dialect-specific arguments such as `storage_format` and `display_width` is to use the appropriate dialect-specific types:

```
from sqlalchemy.dialects.sqlite import DATE
from sqlalchemy.dialects.mysql import INTEGER

d = DATE(storage_format="%(day)02d.%(month)02d.%(year)04d")

i = INTEGER(display_width=5)
```

What about the case where we want the dialect-agnostic type also? We use the *TypeEngine.with_variant()* method:

```
from sqlalchemy import Date, Integer
from sqlalchemy.dialects.sqlite import DATE
from sqlalchemy.dialects.mysql import INTEGER

d = Date().with_variant(
        DATE(storage_format="%(day)02d.%(month)02d.%(year)04d"),
        "sqlite"
    )

i = Integer().with_variant(
        INTEGER(display_width=5),
        "mysql"
    )
```

TypeEngine.with_variant() isn't new, it was added in SQLAlchemy 0.7.2. So code that is running on the 0.8 series can be corrected to use this approach and tested before upgrading to 0.9.

None can no longer be used as a "partial AND" constructor

None can no longer be used as the "backstop" to form an AND condition piecemeal. This pattern was not a documented pattern even though some SQLAlchemy internals made use of it:

```
condition = None

for cond in conditions:
    condition = condition & cond

if condition is not None:
    stmt = stmt.where(condition)
```

The above sequence, when conditions is non-empty, will on 0.9 produce SELECT .. WHERE <condition> AND NULL. The None is no longer implicitly ignored, and is instead consistent with when None is interpreted in other contexts besides that of a conjunction.

The correct code for both 0.8 and 0.9 should read:

```
from sqlalchemy.sql import and_

if conditions:
    stmt = stmt.where(and_(*conditions))
```

Another variant that works on all backends on 0.9, but on 0.8 only works on backends that support boolean constants:

```
from sqlalchemy.sql import true

condition = true()

for cond in conditions:
    condition = cond & condition

stmt = stmt.where(condition)
```

On 0.8, this will produce a SELECT statement that always has AND true in the WHERE clause, which is not accepted by backends that don't support boolean constants (MySQL, MSSQL). On 0.9, the true constant will be dropped within an and_() conjunction.

See also:

Improved rendering of Boolean constants, NULL constants, conjunctions

The "password" portion of a create_engine() no longer considers the + sign as an encoded space

For whatever reason, the Python function unquote_plus() was applied to the "password" field of a URL, which is an incorrect application of the encoding rules described in RFC 1738 in that it escaped spaces as plus signs. The stringiciation of a URL now only encodes ":", "@", or "/" and nothing else, and is now applied to both the username and password fields (previously it only applied to the password). On parsing, encoded characters are converted, but plus signs and spaces are passed through as is:

```
# password: "pass word + other:words"
dbtype://user:pass word + other%3Awords@host/dbname

# password: "apples/oranges"
dbtype://username:apples%2Foranges@hostspec/database

# password: "apples@oranges@@"
dbtype://username:apples%40oranges%40%40@hostspec/database

# password: '', username is "username@"
dbtype://username%40:@hostspec/database
```

#2873

The precedence rules for COLLATE have been changed

Previously, an expression like the following:

```
print (column('x') == 'somevalue').collate("en_EN")
```

would produce an expression like this:

```
-- 0.8 behavior
(x = :x_1) COLLATE en_EN
```

The above is misunderstood by MSSQL and is generally not the syntax suggested for any database. The expression will now produce the syntax illustrated by that of most database documentation:

```
-- 0.9 behavior
x = :x_1 COLLATE en_EN
```

The potentially backwards incompatible change arises if the `collate()` operator is being applied to the right-hand column, as follows:

```
print column('x') == literal('somevalue').collate("en_EN")
```

In 0.8, this produces:

```
x = :param_1 COLLATE en_EN
```

However in 0.9, will now produce the more accurate, but probably not what you want, form of:

```
x = (:param_1 COLLATE en_EN)
```

The `ColumnOperators.collate()` operator now works more appropriately within an `ORDER BY` expression as well, as a specific precedence has been given to the `ASC` and `DESC` operators which will again ensure no parentheses are generated:

```
>>> # 0.8
>>> print column('x').collate('en_EN').desc()
(x COLLATE en_EN) DESC

>>> # 0.9
```

```
>>> print column('x').collate('en_EN').desc()
x COLLATE en_EN DESC
```

#2879

Postgresql CREATE TYPE <x> AS ENUM now applies quoting to values

The *postgresql.ENUM* type will now apply escaping to single quote signs within the enumerated values:

```
>>> from sqlalchemy.dialects import postgresql
>>> type = postgresql.ENUM('one', 'two', "three's", name="myenum")
>>> from sqlalchemy.dialects.postgresql import base
>>> print base.CreateEnumType(type).compile(dialect=postgresql.dialect())
CREATE TYPE myenum AS ENUM ('one','two','three''s')
```

Existing workarounds which already escape single quote signs will need to be modified, else they will now double-escape.

#2878

New Features

Event Removal API

Events established using *event.listen()* or *event.listens_for()* can now be removed using the new *event.remove()* function. The `target`, `identifier` and `fn` arguments sent to *event.remove()* need to match exactly those which were sent for listening, and the event will be removed from all locations in which it had been established:

```
@event.listens_for(MyClass, "before_insert", propagate=True)
def my_before_insert(mapper, connection, target):
    """listen for before_insert"""
    # ...

event.remove(MyClass, "before_insert", my_before_insert)
```

In the example above, the `propagate=True` flag is set. This means `my_before_insert()` is established as a listener for `MyClass` as well as all subclasses of `MyClass`. The system tracks everywhere that the `my_before_insert()` listener function had been placed as a result of this call and removes it as a result of calling *event.remove()*.

The removal system uses a registry to associate arguments passed to *event.listen()* with collections of event listeners, which are in many cases wrapped versions of the original user-supplied function. This registry makes heavy use of weak references in order to allow all the contained contents, such as listener targets, to be garbage collected when they go out of scope.

#2268

New Query Options API; `load_only()` option

The system of loader options such as *orm.joinedload()*, *orm.subqueryload()*, *orm.lazyload()*, *orm.defer()*, etc. all build upon a new system known as *Load*. *Load* provides a "method chained" (a.k.a.

generative) approach to loader options, so that instead of joining together long paths using dots or multiple attribute names, an explicit loader style is given for each path.

While the new way is slightly more verbose, it is simpler to understand in that there is no ambiguity in what options are being applied to which paths; it simplifies the method signatures of the options and provides greater flexibility particularly for column-based options. The old systems are to remain functional indefinitely as well and all styles can be mixed.

Old Way

To set a certain style of loading along every link in a multi-element path, the `_all()` option has to be used:

```
query(User).options(joinedload_all("orders.items.keywords"))
```

New Way

Loader options are now chainable, so the same `joinedload(x)` method is applied equally to each link, without the need to keep straight between *joinedload()* and *joinedload_all()*:

```
query(User).options(joinedload("orders").joinedload("items").joinedload("keywords"))
```

Old Way

Setting an option on path that is based on a subclass requires that all links in the path be spelled out as class bound attributes, since the *PropComparator.of_type()* method needs to be called:

```
session.query(Company).\
    options(
        subqueryload_all(
            Company.employees.of_type(Engineer),
            Engineer.machines
        )
    )
```

New Way

Only those elements in the path that actually need *PropComparator.of_type()* need to be set as a class-bound attribute, string-based names can be resumed afterwards:

```
session.query(Company).\
    options(
        subqueryload(Company.employees.of_type(Engineer)).
        subqueryload("machines")
        )
    )
```

Old Way

Setting the loader option on the last link in a long path uses a syntax that looks a lot like it should be setting the option for all links in the path, causing confusion:

```
query(User).options(subqueryload("orders.items.keywords"))
```

New Way

A path can now be spelled out using *defaultload()* for entries in the path where the existing loader style should be unchanged. More verbose but the intent is clearer:

```
query(User).options(defaultload("orders").defaultload("items").subqueryload("keywords"))
```

The dotted style can still be taken advantage of, particularly in the case of skipping over several path elements:

```
query(User).options(defaultload("orders.items").subqueryload("keywords"))
```

Old Way

The *defer()* option on a path needed to be spelled out with the full path for each column:

```
query(User).options(defer("orders.description"), defer("orders.isopen"))
```

New Way

A single *Load* object that arrives at the target path can have *Load.defer()* called upon it repeatedly:

```
query(User).options(defaultload("orders").defer("description").defer("isopen"))
```

The Load Class The *Load* class can be used directly to provide a "bound" target, especially when multiple parent entities are present:

```
from sqlalchemy.orm import Load

query(User, Address).options(Load(Address).joinedload("entries"))
```

Load Only A new option *load_only()* achieves a "defer everything but" style of load, loading only the given columns and deferring the rest:

```
from sqlalchemy.orm import load_only

query(User).options(load_only("name", "fullname"))

# specify explicit parent entity
query(User, Address).options(Load(User).load_only("name", "fullname"))

# specify path
query(User).options(joinedload(User.addresses).load_only("email_address"))
```

Class-specific Wildcards Using *Load*, a wildcard may be used to set the loading for all relationships (or perhaps columns) on a given entity, without affecting any others:

```
# lazyload all User relationships
query(User).options(Load(User).lazyload("*"))

# undefer all User columns
query(User).options(Load(User).undefer("*"))

# lazyload all Address relationships
query(User).options(defaultload(User.addresses).lazyload("*"))

# undefer all Address columns
query(User).options(defaultload(User.addresses).undefer("*"))
```

#1418

New `text()` Capabilities

The *text()* construct gains new methods:

- *TextClause.bindparams()* allows bound parameter types and values to be set flexibly:

```
# setup values
stmt = text("SELECT id, name FROM user "
      "WHERE name=:name AND timestamp=:timestamp").\
      bindparams(name="ed", timestamp=datetime(2012, 11, 10, 15, 12, 35))

# setup types and/or values
stmt = text("SELECT id, name FROM user "
      "WHERE name=:name AND timestamp=:timestamp").\
      bindparams(
          bindparam("name", value="ed"),
          bindparam("timestamp", type_=DateTime()
      ).bindparam(timestamp=datetime(2012, 11, 10, 15, 12, 35))
```

- *TextClause.columns()* supersedes the `typemap` option of *text()*, returning a new construct *TextAsFrom*:

```
# turn a text() into an alias(), with a .c. collection:
stmt = text("SELECT id, name FROM user").columns(id=Integer, name=String)
stmt = stmt.alias()

stmt = select([addresses]).select_from(
              addresses.join(stmt), addresses.c.user_id == stmt.c.id)


# or into a cte():
stmt = text("SELECT id, name FROM user").columns(id=Integer, name=String)
stmt = stmt.cte("x")

stmt = select([addresses]).select_from(
              addresses.join(stmt), addresses.c.user_id == stmt.c.id)
```

#2877

INSERT from SELECT

After literally years of pointless procrastination this relatively minor syntactical feature has been added, and is also backported to 0.8.3, so technically isn't "new" in 0.9. A *select()* construct or other compatible construct can be passed to the new method *Insert.from_select()* where it will be used to render an `INSERT .. SELECT` construct:

```
>>> from sqlalchemy.sql import table, column
>>> t1 = table('t1', column('a'), column('b'))
>>> t2 = table('t2', column('x'), column('y'))
>>> print(t1.insert().from_select(['a', 'b'], t2.select().where(t2.c.y == 5)))
INSERT INTO t1 (a, b) SELECT t2.x, t2.y
FROM t2
WHERE t2.y = :y_1
```

The construct is smart enough to also accommodate ORM objects such as classes and *Query* objects:

```
s = Session()
q = s.query(User.id, User.name).filter_by(name='ed')
ins = insert(Address).from_select((Address.id, Address.email_address), q)
```

rendering:

```
INSERT INTO addresses (id, email_address)
SELECT users.id AS users_id, users.name AS users_name
FROM users WHERE users.name = :name_1
```

#722

New FOR UPDATE support on `select()`, `Query()`

An attempt is made to simplify the specification of the `FOR UPDATE` clause on `SELECT` statements made within Core and ORM, and support is added for the `FOR UPDATE OF` SQL supported by Postgresql and Oracle.

Using the core *GenerativeSelect.with_for_update()*, options like `FOR SHARE` and `NOWAIT` can be specified individually, rather than linking to arbitrary string codes:

```
stmt = select([table]).with_for_update(read=True, nowait=True, of=table)
```

On Posgtresql the above statement might render like:

```
SELECT table.a, table.b FROM table FOR SHARE OF table NOWAIT
```

The *Query* object gains a similar method *Query.with_for_update()* which behaves in the same way. This method supersedes the existing *Query.with_lockmode()* method, which translated `FOR UPDATE` clauses using a different system. At the moment, the "lockmode" string argument is still accepted by the *Session.refresh()* method.

Floating Point String-Conversion Precision Configurable for Native Floating Point Types

The conversion which SQLAlchemy does whenever a DBAPI returns a Python floating point type which is to be converted into a Python `Decimal()` necessarily involves an intermediary step which converts the floating point value to a string. The scale used for this string conversion was previously hardcoded to 10, and is now configurable. The setting is available on both the *Numeric* as well as the *Float* type, as well as all SQL- and dialect-specific descendant types, using the parameter `decimal_return_scale`. If the type supports a `.scale` parameter, as is the case with *Numeric* and some float types such as *mysql.DOUBLE*, the value of `.scale` is used as the default for `.decimal_return_scale` if it is not otherwise specified. If both `.scale` and `.decimal_return_scale` are absent, then the default of 10 takes place. E.g.:

```
from sqlalchemy.dialects.mysql import DOUBLE
import decimal

data = Table('data', metadata,
    Column('double_value',
                mysql.DOUBLE(decimal_return_scale=12, asdecimal=True))
)

conn.execute(
    data.insert(),
    double_value=45.768392065789,
```

```
)
result = conn.scalar(select([data.c.double_value]))

# previously, this would typically be Decimal("45.7683920658"),
# e.g. trimmed to 10 decimal places

# now we get 12, as requested, as MySQL can support this
# much precision for DOUBLE
assert result == decimal.Decimal("45.768392065789")
```

#2867

Column Bundles for ORM queries

The `Bundle` allows for querying of sets of columns, which are then grouped into one name under the tuple returned by the query. The initial purposes of `Bundle` are 1. to allow "composite" ORM columns to be returned as a single value in a column-based result set, rather than expanding them out into individual columns and 2. to allow the creation of custom result-set constructs within the ORM, using ad-hoc columns and return types, without involving the more heavyweight mechanics of mapped classes.

See also:

Composite attributes are now returned as their object form when queried on a per-attribute basis

Column Bundles

#2824

Server Side Version Counting

The versioning feature of the ORM (now also documented at *Configuring a Version Counter*) can now make use of server-side version counting schemes, such as those produced by triggers or database system columns, as well as conditional programmatic schemes outside of the version_id_counter function itself. By providing the value `False` to the `version_id_generator` parameter, the ORM will use the already-set version identifier, or alternatively fetch the version identifier from each row at the same time the INSERT or UPDATE is emitted. When using a server-generated version identifier, it is strongly recommended that this feature be used only on a backend with strong RETURNING support (Postgresql, SQL Server; Oracle also supports RETURNING but the cx_oracle driver has only limited support), else the additional SELECT statements will add significant performance overhead. The example provided at *Server Side Version Counters* illustrates the usage of the Postgresql `xmin` system column in order to integrate it with the ORM's versioning feature.

See also:

Server Side Version Counters

#2793

`include_backrefs=False` option for `@validates`

The `validates()` function now accepts an option `include_backrefs=True`, which will bypass firing the validator for the case where the event initiated from a backref:

```
from sqlalchemy import Column, Integer, ForeignKey
from sqlalchemy.orm import relationship, validates
from sqlalchemy.ext.declarative import declarative_base
```

```
Base = declarative_base()

class A(Base):
    __tablename__ = 'a'

    id = Column(Integer, primary_key=True)
    bs = relationship("B", backref="a")

    @validates("bs")
    def validate_bs(self, key, item):
        print("A.bs validator")
        return item

class B(Base):
    __tablename__ = 'b'

    id = Column(Integer, primary_key=True)
    a_id = Column(Integer, ForeignKey('a.id'))

    @validates("a", include_backrefs=False)
    def validate_a(self, key, item):
        print("B.a validator")
        return item

a1 = A()
a1.bs.append(B())  # prints only "A.bs validator"
```

#1535

Postgresql JSON Type

The Postgresql dialect now features a *postgresql.JSON* type to complement the *postgresql.HSTORE* type.

See also:

postgresql.JSON

#2581

Automap Extension

A new extension is added in **0.9.1** known as *sqlalchemy.ext.automap*. This is an **experimental** extension which expands upon the functionality of Declarative as well as the *DeferredReflection* class. Essentially, the extension provides a base class *AutomapBase* which automatically generates mapped classes and relationships between them based on given table metadata.

The *MetaData* in use normally might be produced via reflection, but there is no requirement that reflection is used. The most basic usage illustrates how *sqlalchemy.ext.automap* is able to deliver mapped classes, including relationships, based on a reflected schema:

```
from sqlalchemy.ext.automap import automap_base
from sqlalchemy.orm import Session
from sqlalchemy import create_engine

Base = automap_base()
```

```
# engine, suppose it has two tables 'user' and 'address' set up
engine = create_engine("sqlite:///mydatabase.db")

# reflect the tables
Base.prepare(engine, reflect=True)

# mapped classes are now created with names matching that of the table
# name.
User = Base.classes.user
Address = Base.classes.address

session = Session(engine)

# rudimentary relationships are produced
session.add(Address(email_address="foo@bar.com", user=User(name="foo")))
session.commit()

# collection-based relationships are by default named "<classname>_collection"
print (u1.address_collection)
```

Beyond that, the `AutomapBase` class is a declarative base, and supports all the features that declarative does. The "automapping" feature can be used with an existing, explicitly declared schema to generate relationships and missing classes only. Naming schemes and relationship-production routines can be dropped in using callable functions.

It is hoped that the `AutomapBase` system provides a quick and modernized solution to the problem that the very famous SQLSoup also tries to solve, that of generating a quick and rudimentary object model from an existing database on the fly. By addressing the issue strictly at the mapper configuration level, and integrating fully with existing Declarative class techniques, `AutomapBase` seeks to provide a well-integrated approach to the issue of expediently auto-generating ad-hoc mappings.

See also:

Automap

Behavioral Improvements

Improvements that should produce no compatibility issues except in exceedingly rare and unusual hypothetical cases, but are good to be aware of in case there are unexpected issues.

Many JOIN and LEFT OUTER JOIN expressions will no longer be wrapped in (SELECT * FROM ..) AS ANON_1

For many years, the SQLAlchemy ORM has been held back from being able to nest a JOIN inside the right side of an existing JOIN (typically a LEFT OUTER JOIN, as INNER JOINs could always be flattened):

```
SELECT a.*, b.*, c.* FROM a LEFT OUTER JOIN (b JOIN c ON b.id = c.id) ON a.id
```

This was due to the fact that SQLite, even today, cannot parse a statement of the above format:

```
SQLite version 3.7.15.2 2013-01-09 11:53:05
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> create table a(id integer);
sqlite> create table b(id integer);
sqlite> create table c(id integer);
sqlite> select a.id, b.id, c.id from a left outer join (b join c on b.id=c.id) on b.id=a.id;
Error: no such column: b.id
```

Right-outer-joins are of course another way to work around right-side parenthesization; this would be significantly complicated and visually unpleasant to implement, but fortunately SQLite doesn't support RIGHT OUTER JOIN either :):

```
sqlite> select a.id, b.id, c.id from b join c on b.id=c.id
   ...> right outer join a on b.id=a.id;
Error: RIGHT and FULL OUTER JOINs are not currently supported
```

Back in 2005, it wasn't clear if other databases had trouble with this form, but today it seems clear every database tested except SQLite now supports it (Oracle 8, a very old database, doesn't support the JOIN keyword at all, but SQLAlchemy has always had a simple rewriting scheme in place for Oracle's syntax). To make matters worse, SQLAlchemy's usual workaround of applying a SELECT often degrades performance on platforms like Postgresql and MySQL:

```
SELECT a.*, anon_1.* FROM a LEFT OUTER JOIN (
                SELECT b.id AS b_id, c.id AS c_id
                FROM b JOIN c ON b.id = c.id
            ) AS anon_1 ON a.id=anon_1.b_id
```

A JOIN like the above form is commonplace when working with joined-table inheritance structures; any time *Query.join()* is used to join from some parent to a joined-table subclass, or when *joinedload()* is used similarly, SQLAlchemy's ORM would always make sure a nested JOIN was never rendered, lest the query wouldn't be able to run on SQLite. Even though the Core has always supported a JOIN of the more compact form, the ORM had to avoid it.

An additional issue would arise when producing joins across many-to-many relationships where special criteria is present in the ON clause. Consider an eager load join like the following:

```
session.query(Order).outerjoin(Order.items)
```

Assuming a many-to-many from `Order` to `Item` which actually refers to a subclass like `Subitem`, the SQL for the above would look like:

```
SELECT order.id, order.name
FROM order LEFT OUTER JOIN order_item ON order.id = order_item.order_id
LEFT OUTER JOIN item ON order_item.item_id = item.id AND item.type = 'subitem'
```

What's wrong with the above query? Basically, that it will load many `order` / `order_item` rows where the criteria of `item.type == 'subitem'` is not true.

As of SQLAlchemy 0.9, an entirely new approach has been taken. The ORM no longer worries about nesting JOINs in the right side of an enclosing JOIN, and it now will render these as often as possible while still returning the correct results. When the SQL statement is passed to be compiled, the **dialect compiler** will **rewrite the join** to suit the target backend, if that backend is known to not support a right-nested JOIN (which currently is only SQLite - if other backends have this issue please let us know!).

So a regular `query(Parent).join(Subclass)` will now usually produce a simpler expression:

```
SELECT parent.id AS parent_id
FROM parent JOIN (
        base_table JOIN subclass_table
        ON base_table.id = subclass_table.id) ON parent.id = base_table.parent_id
```

Joined eager loads like `query(Parent).options(joinedload(Parent.subclasses))` will alias the individual tables instead of wrapping in an `ANON_1`:

```
SELECT parent.*, base_table_1.*, subclass_table_1.* FROM parent
    LEFT OUTER JOIN (
        base_table AS base_table_1 JOIN subclass_table AS subclass_table_1
        ON base_table_1.id = subclass_table_1.id)
        ON parent.id = base_table_1.parent_id
```

Many-to-many joins and eagerloads will right nest the "secondary" and "right" tables:

```
SELECT order.id, order.name
FROM order LEFT OUTER JOIN
(order_item JOIN item ON order_item.item_id = item.id AND item.type = 'subitem')
ON order_item.order_id = order.id
```

All of these joins, when rendered with a *Select* statement that specifically specifies `use_labels=True`, which is true for all the queries the ORM emits, are candidates for "join rewriting", which is the process of rewriting all those right-nested joins into nested SELECT statements, while maintaining the identical labeling used by the *Select*. So SQLite, the one database that won't support this very common SQL syntax even in 2013, shoulders the extra complexity itself, with the above queries rewritten as:

```
-- sqlite only!
SELECT parent.id AS parent_id
    FROM parent JOIN (
        SELECT base_table.id AS base_table_id,
                base_table.parent_id AS base_table_parent_id,
                subclass_table.id AS subclass_table_id
        FROM base_table JOIN subclass_table ON base_table.id = subclass_table.id
    ) AS anon_1 ON parent.id = anon_1.base_table_parent_id

-- sqlite only!
SELECT parent.id AS parent_id, anon_1.subclass_table_1_id AS subclass_table_1_id,
        anon_1.base_table_1_id AS base_table_1_id,
        anon_1.base_table_1_parent_id AS base_table_1_parent_id
FROM parent LEFT OUTER JOIN (
    SELECT base_table_1.id AS base_table_1_id,
        base_table_1.parent_id AS base_table_1_parent_id,
        subclass_table_1.id AS subclass_table_1_id
    FROM base_table AS base_table_1
    JOIN subclass_table AS subclass_table_1 ON base_table_1.id = subclass_table_1.id
) AS anon_1 ON parent.id = anon_1.base_table_1_parent_id

-- sqlite only!
SELECT "order".id AS order_id
FROM "order" LEFT OUTER JOIN (
        SELECT order_item_1.order_id AS order_item_1_order_id,
            order_item_1.item_id AS order_item_1_item_id,
            item.id AS item_id, item.type AS item_type
FROM order_item AS order_item_1
    JOIN item ON item.id = order_item_1.item_id AND item.type IN (?)
) AS anon_1 ON "order".id = anon_1.order_item_1_order_id
```

The *Join.alias()*, *aliased()* and *with_polymorphic()* functions now support a new argument, `flat=True`, which is used to construct aliases of joined-table entities without embedding into a SELECT. This flag is not on by default, to help with backwards compatibility - but now a "polymorhpic" selectable can be joined as a target without any subqueries generated:

```
employee_alias = with_polymorphic(Person, [Engineer, Manager], flat=True)

session.query(Company).join(
                    Company.employees.of_type(employee_alias)
                ).filter(
                    or_(
                        Engineer.primary_language == 'python',
                        Manager.manager_name == 'dilbert'
                    )
                )
```

Generates (everywhere except SQLite):

```
SELECT companies.company_id AS companies_company_id, companies.name AS companies_name
FROM companies JOIN (
    people AS people_1
    LEFT OUTER JOIN engineers AS engineers_1 ON people_1.person_id = engineers_1.person_id
    LEFT OUTER JOIN managers AS managers_1 ON people_1.person_id = managers_1.person_id
) ON companies.company_id = people_1.company_id
WHERE engineers.primary_language = %(primary_language_1)s
    OR managers.manager_name = %(manager_name_1)s
```

#2369 #2587

Right-nested inner joins available in joined eager loads

As of version 0.9.4, the above mentioned right-nested joining can be enabled in the case of a joined eager load where an "outer" join is linked to an "inner" on the right side.

Normally, a joined eager load chain like the following:

```
query(User).options(joinedload("orders", innerjoin=False).joinedload("items", innerjoin=True))
```

Would not produce an inner join; because of the LEFT OUTER JOIN from user->order, joined eager loading could not use an INNER join from order->items without changing the user rows that are returned, and would instead ignore the "chained" `innerjoin=True` directive. How 0.9.0 should have delivered this would be that instead of:

```
FROM users LEFT OUTER JOIN orders ON <onclause> LEFT OUTER JOIN items ON <onclause>
```

the new "right-nested joins are OK" logic would kick in, and we'd get:

```
FROM users LEFT OUTER JOIN (orders JOIN items ON <onclause>) ON <onclause>
```

Since we missed the boat on that, to avoid further regressions we've added the above functionality by specifying the string `"nested"` to *`joinedload.innerjoin`*:

```
query(User).options(joinedload("orders", innerjoin=False).joinedload("items", innerjoin="nested"))
```

This feature is new in 0.9.4.

#2976

ORM can efficiently fetch just-generated INSERT/UPDATE defaults using RETURNING

The *Mapper* has long supported an undocumented flag known as `eager_defaults=True`. The effect of this flag is that when an INSERT or UPDATE proceeds, and the row is known to have server-generated default values, a SELECT would immediately follow it in order to "eagerly" load those new values. Normally, the server-generated columns are marked as "expired" on the object, so that no overhead is incurred unless the application actually accesses these columns soon after the flush. The `eager_defaults` flag was therefore not of much use as it could only decrease performance, and was present only to support exotic event schemes where users needed default values to be available immediately within the flush process.

In 0.9, as a result of the version id enhancements, `eager_defaults` can now emit a RETURNING clause for these values, so on a backend with strong RETURNING support in particular Postgresql, the ORM can fetch newly generated default and SQL expression values inline with the INSERT or UPDATE. `eager_defaults`, when enabled, makes use of RETURNING automatically when the target backend and *Table* supports "implicit returning".

Subquery Eager Loading will apply DISTINCT to the innermost SELECT for some queries

In an effort to reduce the number of duplicate rows that can be generated by subquery eager loading when a many-to-one relationship is involved, a DISTINCT keyword will be applied to the innermost SELECT when the join is targeting columns that do not comprise the primary key, as in when loading along a many to one.

That is, when subquery loading on a many-to-one from A->B:

```
SELECT b.id AS b_id, b.name AS b_name, anon_1.b_id AS a_b_id
FROM (SELECT DISTINCT a_b_id FROM a) AS anon_1
JOIN b ON b.id = anon_1.a_b_id
```

Since `a.b_id` is a non-distinct foreign key, DISTINCT is applied so that redundant `a.b_id` are eliminated. The behavior can be turned on or off unconditionally for a particular *relationship()* using the flag `distinct_target_key`, setting the value to `True` for unconditionally on, `False` for unconditionally off, and `None` for the feature to take effect when the target SELECT is against columns that do not comprise a full primary key. In 0.9, `None` is the default.

The option is also backported to 0.8 where the `distinct_target_key` option defaults to `False`.

While the feature here is designed to help performance by eliminating duplicate rows, the `DISTINCT` keyword in SQL itself can have a negative performance impact. If columns in the SELECT are not indexed, `DISTINCT` will likely perform an `ORDER BY` on the rowset which can be expensive. By keeping the feature limited just to foreign keys which are hopefully indexed in any case, it's expected that the new defaults are reasonable.

The feature also does not eliminate every possible dupe-row scenario; if a many-to-one is present elsewhere in the chain of joins, dupe rows may still be present.

#2836

Backref handlers can now propagate more than one level deep

The mechanism by which attribute events pass along their "initiator", that is the object associated with the start of the event, has been changed; instead of a `AttributeImpl` being passed, a new object *attributes.Event* is passed instead; this object refers to the `AttributeImpl` as well as to an "operation token", representing if the operation is an append, remove, or replace operation.

The attribute event system no longer looks at this "initiator" object in order to halt a recursive series of attribute events. Instead, the system of preventing endless recursion due to mutually-dependent backref handlers has been moved to the ORM backref event handlers specifically, which now take over the role of ensuring that a chain of mutually-dependent events (such as append to collection A.bs, set many-to-one attribute B.a in response) doesn't go into an

endless recursion stream. The rationale here is that the backref system, given more detail and control over event propagation, can finally allow operations more than one level deep to occur; the typical scenario is when a collection append results in a many-to-one replacement operation, which in turn should cause the item to be removed from a previous collection:

```
class Parent(Base):
    __tablename__ = 'parent'

    id = Column(Integer, primary_key=True)
    children = relationship("Child", backref="parent")

class Child(Base):
    __tablename__ = 'child'

    id = Column(Integer, primary_key=True)
    parent_id = Column(ForeignKey('parent.id'))

p1 = Parent()
p2 = Parent()
c1 = Child()

p1.children.append(c1)

assert c1.parent is p1  # backref event establishes c1.parent as p1

p2.children.append(c1)

assert c1.parent is p2  # backref event establishes c1.parent as p2
assert c1 not in p1.children  # second backref event removes c1 from p1.children
```

Above, prior to this change, the `c1` object would still have been present in `p1.children`, even though it is also present in `p2.children` at the same time; the backref handlers would have stopped at replacing `c1.parent` with `p2` instead of `p1`. In 0.9, using the more detailed *Event* object as well as letting the backref handlers make more detailed decisions about these objects, the propagation can continue onto removing `c1` from `p1.children` while maintaining a check against the propagation from going into an endless recursive loop.

End-user code which a. makes use of the *AttributeEvents.set()*, *AttributeEvents.append()*, or *AttributeEvents.remove()* events, and b. initiates further attribute modification operations as a result of these events may need to be modified to prevent recursive loops, as the attribute system no longer stops a chain of events from propagating endlessly in the absence of the backref event handlers. Additionally, code which depends upon the value of the `initiator` will need to be adjusted to the new API, and furthermore must be ready for the value of `initiator` to change from its original value within a string of backref-initiated events, as the backref handlers may now swap in a new `initiator` value for some operations.

#2789

The typing system now handles the task of rendering "literal bind" values

A new method is added to *TypeEngine* *TypeEngine.literal_processor()* as well as *TypeDecorator.process_literal_param()* for *TypeDecorator* which take on the task of rendering so-called "inline literal paramters" - parameters that normally render as "bound" values, but are instead being rendered inline into the SQL statement due to the compiler configuration. This feature is used when generating DDL for constructs such as *CheckConstraint*, as well as by Alembic when using constructs such as `op.inline_literal()`. Previously, a simple "isinstance" check checked for a few basic types, and the "bind processor" was used unconditionally, leading to such issues as strings being encoded into utf-8 prematurely.

Custom types written with *TypeDecorator* should continue to work in "inline literal" scenarios, as the *TypeDecorator.process_literal_param()* falls back to *TypeDecorator.process_bind_param()* by default, as these methods usually handle a data manipulation, not as much how the data is presented to the database. *TypeDecorator.process_literal_param()* can be specified to specifically produce a string representing how a value should be rendered into an inline DDL statement.

#2838

Schema identifiers now carry along their own quoting information

This change simplifies the Core's usage of so-called "quote" flags, such as the `quote` flag passed to *Table* and *Column*. The flag is now internalized within the string name itself, which is now represented as an instance of *quoted_name*, a string subclass. The *IdentifierPreparer* now relies solely on the quoting preferences reported by the *quoted_name* object rather than checking for any explicit `quote` flags in most cases. The issue resolved here includes that various case-sensitive methods such as *Engine.has_table()* as well as similar methods within dialects now function with explicitly quoted names, without the need to complicate or introduce backwards-incompatible changes to those APIs (many of which are 3rd party) with the details of quoting flags - in particular, a wider range of identifiers now function correctly with the so-called "uppercase" backends like Oracle, Firebird, and DB2 (backends that store and report upon table and column names using all uppercase for case insensitive names).

The *quoted_name* object is used internally as needed; however if other keywords require fixed quoting preferences, the class is available publically.

#2812

Improved rendering of Boolean constants, NULL constants, conjunctions

New capabilities have been added to the *true()* and *false()* constants, in particular in conjunction with *and_()* and *or_()* functions as well as the behavior of the WHERE/HAVING clauses in conjunction with these types, boolean types overall, and the *null()* constant.

Starting with a table such as this:

```
from sqlalchemy import Table, Boolean, Integer, Column, MetaData

t1 = Table('t', MetaData(), Column('x', Boolean()), Column('y', Integer))
```

A select construct will now render the boolean column as a binary expression on backends that don't feature `true`/`false` constant beahvior:

```
>>> from sqlalchemy import select, and_, false, true
>>> from sqlalchemy.dialects import mysql, postgresql

>>> print select([t1]).where(t1.c.x).compile(dialect=mysql.dialect())
SELECT t.x, t.y  FROM t WHERE t.x = 1
```

The *and_()* and *or_()* constructs will now exhibit quasi "short circuit" behavior, that is truncating a rendered expression, when a *true()* or *false()* constant is present:

```
>>> print select([t1]).where(and_(t1.c.y > 5, false())).compile(
...     dialect=postgresql.dialect())
SELECT t.x, t.y FROM t WHERE false
```

true() can be used as the base to build up an expression:

```
>>> expr = true()
>>> expr = expr & (t1.c.y > 5)
>>> print select([t1]).where(expr)
SELECT t.x, t.y FROM t WHERE t.y > :y_1
```

The boolean constants `true()` and `false()` themselves render as `0 = 1` and `1 = 1` for a backend with no boolean constants:

```
>>> print select([t1]).where(and_(t1.c.y > 5, false())).compile(
...     dialect=mysql.dialect())
SELECT t.x, t.y FROM t WHERE 0 = 1
```

Interpretation of `None`, while not particularly valid SQL, is at least now consistent:

```
>>> print select([t1.c.x]).where(None)
SELECT t.x FROM t WHERE NULL

>>> print select([t1.c.x]).where(None).where(None)
SELECT t.x FROM t WHERE NULL AND NULL

>>> print select([t1.c.x]).where(and_(None, None))
SELECT t.x FROM t WHERE NULL AND NULL
```

#2804

Label constructs can now render as their name alone in an ORDER BY

For the case where a `Label` is used in both the columns clause as well as the ORDER BY clause of a SELECT, the label will render as just its name in the ORDER BY clause, assuming the underlying dialect reports support of this feature.

E.g. an example like:

```
from sqlalchemy.sql import table, column, select, func

t = table('t', column('c1'), column('c2'))
expr = (func.foo(t.c.c1) + t.c.c2).label("expr")

stmt = select([expr]).order_by(expr)

print stmt
```

Prior to 0.9 would render as:

```
SELECT foo(t.c1) + t.c2 AS expr
FROM t ORDER BY foo(t.c1) + t.c2
```

And now renders as:

```
SELECT foo(t.c1) + t.c2 AS expr
FROM t ORDER BY expr
```

The ORDER BY only renders the label if the label isn't further embedded into an expression within the ORDER BY, other than a simple `ASC` or `DESC`.

The above format works on all databases tested, but might have compatibility issues with older database versions (MySQL 4? Oracle 8? etc.). Based on user reports we can add rules that will disable the feature based on database version detection.

#1068

RowProxy now has tuple-sorting behavior

The *RowProxy* object acts much like a tuple, but up until now would not sort as a tuple if a list of them were sorted using `sorted()`. The `__eq__()` method now compares both sides as a tuple and also an `__lt__()` method has been added:

```
users.insert().execute(
        dict(user_id=1, user_name='foo'),
        dict(user_id=2, user_name='bar'),
        dict(user_id=3, user_name='def'),
    )

rows = users.select().order_by(users.c.user_name).execute().fetchall()

eq_(rows, [(2, 'bar'), (3, 'def'), (1, 'foo')])

eq_(sorted(rows), [(1, 'foo'), (2, 'bar'), (3, 'def')])
```

#2848

A bindparam() construct with no type gets upgraded via copy when a type is available

The logic which "upgrades" a *bindparam()* construct to take on the type of the enclosing expression has been improved in two ways. First, the *bindparam()* object is **copied** before the new type is assigned, so that the given *bindparam()* is not mutated in place. Secondly, this same operation occurs when an *Insert* or *Update* construct is compiled, regarding the "values" that were set in the statement via the *ValuesBase.values()* method.

If given an untyped *bindparam()*:

```
bp = bindparam("some_col")
```

If we use this parameter as follows:

```
expr = mytable.c.col == bp
```

The type for `bp` remains as `NullType`, however if `mytable.c.col` is of type `String`, then `expr.right`, that is the right side of the binary expression, will take on the `String` type. Previously, `bp` itself would have been changed in place to have `String` as its type.

Similarly, this operation occurs in an *Insert* or *Update*:

```
stmt = mytable.update().values(col=bp)
```

Above, `bp` remains unchanged, but the `String` type will be used when the statement is executed, which we can see by examining the `binds` dictionary:

```
>>> compiled = stmt.compile()
>>> compiled.binds['some_col'].type
String
```

The feature allows custom types to take their expected effect within INSERT/UPDATE statements without needing to explicitly specify those types within every `bindparam()` expression.

The potentially backwards-compatible changes involve two unlikely scenarios. Since the bound parameter is **cloned**, users should not be relying upon making in-place changes to a `bindparam()` construct once created. Additionally, code which uses `bindparam()` within an `Insert` or `Update` statement which is relying on the fact that the `bindparam()` is not typed according to the column being assigned towards will no longer function in that way.

#2850

Columns can reliably get their type from a column referred to via ForeignKey

There's a long standing behavior which says that a `Column` can be declared without a type, as long as that `Column` is referred to by a `ForeignKeyConstraint`, and the type from the referenced column will be copied into this one. The problem has been that this feature never worked very well and wasn't maintained. The core issue was that the `ForeignKey` object doesn't know what target `Column` it refers to until it is asked, typically the first time the foreign key is used to construct a `Join`. So until that time, the parent `Column` would not have a type, or more specifically, it would have a default type of `NullType`.

While it's taken a long time, the work to reorganize the initialization of `ForeignKey` objects has been completed such that this feature can finally work acceptably. At the core of the change is that the `ForeignKey.column` attribute no longer lazily initializes the location of the target `Column`; the issue with this system was that the owning `Column` would be stuck with `NullType` as its type until the `ForeignKey` happened to be used.

In the new version, the `ForeignKey` coordinates with the eventual `Column` it will refer to using internal attachment events, so that the moment the referencing `Column` is associated with the `MetaData`, all `ForeignKey` objects that refer to it will be sent a message that they need to initialize their parent column. This system is more complicated but works more solidly; as a bonus, there are now tests in place for a wide variety of `Column` / `ForeignKey` configuration scenarios and error messages have been improved to be very specific to no less than seven different error conditions.

Scenarios which now work correctly include:

1. The type on a `Column` is immediately present as soon as the target `Column` becomes associated with the same `MetaData`; this works no matter which side is configured first:

```
>>> from sqlalchemy import Table, MetaData, Column, Integer, ForeignKey
>>> metadata = MetaData()
>>> t2 = Table('t2', metadata, Column('t1id', ForeignKey('t1.id')))
>>> t2.c.t1id.type
NullType()
>>> t1 = Table('t1', metadata, Column('id', Integer, primary_key=True))
>>> t2.c.t1id.type
Integer()
```

2. The system now works with `ForeignKeyConstraint` as well:

```
>>> from sqlalchemy import Table, MetaData, Column, Integer, ForeignKeyConstraint
>>> metadata = MetaData()
>>> t2 = Table('t2', metadata,
...     Column('t1a'), Column('t1b'),
...     ForeignKeyConstraint(['t1a', 't1b'], ['t1.a', 't1.b']))
>>> t2.c.t1a.type
NullType()
>>> t2.c.t1b.type
NullType()
>>> t1 = Table('t1', metadata,
```

```
...     Column('a', Integer, primary_key=True),
...     Column('b', Integer, primary_key=True))
>>> t2.c.t1a.type
Integer()
>>> t2.c.t1b.type
Integer()
```

3. It even works for "multiple hops" - that is, a *ForeignKey* that refers to a *Column* that refers to another *Column*:

```
>>> from sqlalchemy import Table, MetaData, Column, Integer, ForeignKey
>>> metadata = MetaData()
>>> t2 = Table('t2', metadata, Column('t1id', ForeignKey('t1.id')))
>>> t3 = Table('t3', metadata, Column('t2t1id', ForeignKey('t2.t1id')))
>>> t2.c.t1id.type
NullType()
>>> t3.c.t2t1id.type
NullType()
>>> t1 = Table('t1', metadata, Column('id', Integer, primary_key=True))
>>> t2.c.t1id.type
Integer()
>>> t3.c.t2t1id.type
Integer()
```

#1765

Dialect Changes

Firebird `fdb` is now the default Firebird dialect.

The `fdb` dialect is now used if an engine is created without a dialect specifier, i.e. `firebird://`. `fdb` is a `kinterbasdb` compatible DBAPI which per the Firebird project is now their official Python driver.

#2504

Firebird `fdb` and `kinterbasdb` set `retaining=False` by default

Both the `fdb` and `kinterbasdb` DBAPIs support a flag `retaining=True` which can be passed to the `commit()` and `rollback()` methods of its connection. The documented rationale for this flag is so that the DBAPI can re-use internal transaction state for subsequent transactions, for the purposes of improving performance. However, newer documentation refers to analyses of Firebird's "garbage collection" which expresses that this flag can have a negative effect on the database's ability to process cleanup tasks, and has been reported as *lowering* performance as a result.

It's not clear how this flag is actually usable given this information, and as it appears to be only a performance enhancing feature, it now defaults to `False`. The value can be controlled by passing the flag `retaining=True` to the *create_engine()* call. This is a new flag which is added as of 0.8.2, so applications on 0.8.2 can begin setting this to `True` or `False` as desired.

See also:

sqlalchemy.dialects.firebird.fdb

sqlalchemy.dialects.firebird.kinterbasdb

http://pythonhosted.org/fdb/usage-guide.html#retaining-transactions - information on the "retaining" flag.

#2763

6.3.3 What's New in SQLAlchemy 0.8?

About this Document

This document describes changes between SQLAlchemy version 0.7, undergoing maintenance releases as of October, 2012, and SQLAlchemy version 0.8, which is expected for release in early 2013.

Document date: October 25, 2012 Updated: March 9, 2013

Introduction

This guide introduces what's new in SQLAlchemy version 0.8, and also documents changes which affect users migrating their applications from the 0.7 series of SQLAlchemy to 0.8.

SQLAlchemy releases are closing in on 1.0, and each new version since 0.5 features fewer major usage changes. Most applications that are settled into modern 0.7 patterns should be movable to 0.8 with no changes. Applications that use 0.6 and even 0.5 patterns should be directly migratable to 0.8 as well, though larger applications may want to test with each interim version.

Platform Support

Targeting Python 2.5 and Up Now

SQLAlchemy 0.8 will target Python 2.5 and forward; compatibility for Python 2.4 is being dropped.

The internals will be able to make usage of Python ternaries (that is, `x if y else z`) which will improve things versus the usage of `y and x or z`, which naturally has been the source of some bugs, as well as context managers (that is, `with:`) and perhaps in some cases `try:/except:/else:` blocks which will help with code readability.

SQLAlchemy will eventually drop 2.5 support as well - when 2.6 is reached as the baseline, SQLAlchemy will move to use 2.6/3.3 in-place compatibility, removing the usage of the `2to3` tool and maintaining a source base that works with Python 2 and 3 at the same time.

New ORM Features

Rewritten `relationship()` mechanics

0.8 features a much improved and capable system regarding how *`relationship()`* determines how to join between two entities. The new system includes these features:

- The `primaryjoin` argument is **no longer needed** when constructing a *`relationship()`* against a class that has multiple foreign key paths to the target. Only the `foreign_keys` argument is needed to specify those columns which should be included:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id_one = Column(Integer, ForeignKey('child.id'))
    child_id_two = Column(Integer, ForeignKey('child.id'))
```

```
    child_one = relationship("Child", foreign_keys=child_id_one)
    child_two = relationship("Child", foreign_keys=child_id_two)

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
```

- relationships against self-referential, composite foreign keys where **a column points to itself** are now supported. The canonical case is as follows:

```
class Folder(Base):
    __tablename__ = 'folder'
    __table_args__ = (
      ForeignKeyConstraint(
          ['account_id', 'parent_id'],
          ['folder.account_id', 'folder.folder_id']),
    )

    account_id = Column(Integer, primary_key=True)
    folder_id = Column(Integer, primary_key=True)
    parent_id = Column(Integer)
    name = Column(String)

    parent_folder = relationship("Folder",
                        backref="child_folders",
                        remote_side=[account_id, folder_id]
                  )
```

Above, the `Folder` refers to its parent `Folder` joining from `account_id` to itself, and `parent_id` to `folder_id`. When SQLAlchemy constructs an auto- join, no longer can it assume all columns on the "remote" side are aliased, and all columns on the "local" side are not - the `account_id` column is **on both sides**. So the internal relationship mechanics were totally rewritten to support an entirely different system whereby two copies of `account_id` are generated, each containing different *annotations* to determine their role within the statement. Note the join condition within a basic eager load:

```
SELECT
    folder.account_id AS folder_account_id,
    folder.folder_id AS folder_folder_id,
    folder.parent_id AS folder_parent_id,
    folder.name AS folder_name,
    folder_1.account_id AS folder_1_account_id,
    folder_1.folder_id AS folder_1_folder_id,
    folder_1.parent_id AS folder_1_parent_id,
    folder_1.name AS folder_1_name
FROM folder
    LEFT OUTER JOIN folder AS folder_1
    ON
        folder_1.account_id = folder.account_id
        AND folder.folder_id = folder_1.parent_id

WHERE folder.folder_id = ? AND folder.account_id = ?
```

- Previously difficult custom join conditions, like those involving functions and/or CASTing of types, will now function as expected in most cases:

```
class HostEntry(Base):
    __tablename__ = 'host_entry'

    id = Column(Integer, primary_key=True)
    ip_address = Column(INET)
    content = Column(String(50))

    # relationship() using explicit foreign_keys, remote_side
    parent_host = relationship("HostEntry",
                        primaryjoin=ip_address == cast(content, INET),
                        foreign_keys=content,
                        remote_side=ip_address
                    )
```

The new *relationship()* mechanics make use of a SQLAlchemy concept known as annotations. These annotations are also available to application code explicitly via the *foreign()* and *remote()* functions, either as a means to improve readability for advanced configurations or to directly inject an exact configuration, bypassing the usual join-inspection heuristics:

```
from sqlalchemy.orm import foreign, remote

class HostEntry(Base):
    __tablename__ = 'host_entry'

    id = Column(Integer, primary_key=True)
    ip_address = Column(INET)
    content = Column(String(50))

    # relationship() using explicit foreign() and remote() annotations
    # in lieu of separate arguments
    parent_host = relationship("HostEntry",
                        primaryjoin=remote(ip_address) == \
                                cast(foreign(content), INET),
                    )
```

See also:

Configuring how Relationship Joins - a newly revised section on *relationship()* detailing the latest techniques for customizing related attributes and collection access.

#1401 #610

New Class/Object Inspection System

Lots of SQLAlchemy users are writing systems that require the ability to inspect the attributes of a mapped class, including being able to get at the primary key columns, object relationships, plain attributes, and so forth, typically for the purpose of building data-marshalling systems, like JSON/XML conversion schemes and of course form libraries galore.

Originally, the *Table* and *Column* model were the original inspection points, which have a well-documented system. While SQLAlchemy ORM models are also fully introspectable, this has never been a fully stable and supported feature, and users tended to not have a clear idea how to get at this information.

0.8 now provides a consistent, stable and fully documented API for this purpose, including an inspection system which works on mapped classes, instances, attributes, and other Core and ORM constructs. The entrypoint to this system is the core-level *inspect()* function. In most cases, the object being inspected is one already part of SQLAlchemy's

system, such as *Mapper*, *InstanceState*, *Inspector*. In some cases, new objects have been added with the job of providing the inspection API in certain contexts, such as *AliasedInsp* and *AttributeState*.

A walkthrough of some key capabilities follows:

```
>>> class User(Base):
...     __tablename__ = 'user'
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     name_syn = synonym(name)
...     addresses = relationship("Address")
...

>>> # universal entry point is inspect()
>>> b = inspect(User)

>>> # b in this case is the Mapper
>>> b
<Mapper at 0x101521950; User>

>>> # Column namespace
>>> b.columns.id
Column('id', Integer(), table=<user>, primary_key=True, nullable=False)

>>> # mapper's perspective of the primary key
>>> b.primary_key
(Column('id', Integer(), table=<user>, primary_key=True, nullable=False),)

>>> # MapperProperties available from .attrs
>>> b.attrs.keys()
['name_syn', 'addresses', 'id', 'name']

>>> # .column_attrs, .relationships, etc. filter this collection
>>> b.column_attrs.keys()
['id', 'name']

>>> list(b.relationships)
[<sqlalchemy.orm.properties.RelationshipProperty object at 0x1015212d0>]

>>> # they are also namespaces
>>> b.column_attrs.id
<sqlalchemy.orm.properties.ColumnProperty object at 0x101525090>

>>> b.relationships.addresses
<sqlalchemy.orm.properties.RelationshipProperty object at 0x1015212d0>

>>> # point inspect() at a mapped, class level attribute,
>>> # returns the attribute itself
>>> b = inspect(User.addresses)
>>> b
<sqlalchemy.orm.attributes.InstrumentedAttribute object at 0x101521fd0>

>>> # From here we can get the mapper:
>>> b.mapper
<Mapper at 0x101525810; Address>

>>> # the parent inspector, in this case a mapper
>>> b.parent
<Mapper at 0x101521950; User>
```

```
>>> # an expression
>>> print b.expression
"user".id = address.user_id

>>> # inspect works on instances
>>> u1 = User(id=3, name='x')
>>> b = inspect(u1)

>>> # it returns the InstanceState
>>> b
<sqlalchemy.orm.state.InstanceState object at 0x10152bed0>

>>> # similar attrs accessor refers to the
>>> b.attrs.keys()
['id', 'name_syn', 'addresses', 'name']

>>> # attribute interface - from attrs, you get a state object
>>> b.attrs.id
<sqlalchemy.orm.state.AttributeState object at 0x10152bf90>

>>> # this object can give you, current value...
>>> b.attrs.id.value
3

>>> # ... current history
>>> b.attrs.id.history
History(added=[3], unchanged=(), deleted=())

>>> # InstanceState can also provide session state information
>>> # lets assume the object is persistent
>>> s = Session()
>>> s.add(u1)
>>> s.commit()

>>> # now we can get primary key identity, always
>>> # works in query.get()
>>> b.identity
(3,)

>>> # the mapper level key
>>> b.identity_key
(<class '__main__.User'>, (3,))

>>> # state within the session
>>> b.persistent, b.transient, b.deleted, b.detached
(True, False, False, False)

>>> # owning session
>>> b.session
<sqlalchemy.orm.session.Session object at 0x101701150>
```

See also:

Runtime Inspection API

#2208

New with_polymorphic() feature, can be used anywhere

The *Query.with_polymorphic()* method allows the user to specify which tables should be present when querying against a joined-table entity. Unfortunately the method is awkward and only applies to the first entity in the list, and otherwise has awkward behaviors both in usage as well as within the internals. A new enhancement to the *aliased()* construct has been added called *with_polymorphic()* which allows any entity to be "aliased" into a "polymorphic" version of itself, freely usable anywhere:

```
from sqlalchemy.orm import with_polymorphic
palias = with_polymorphic(Person, [Engineer, Manager])
session.query(Company).\
            join(palias, Company.employees).\
            filter(or_(Engineer.language=='java', Manager.hair=='pointy'))
```

See also:

Basic Control of Which Tables are Queried - newly updated documentation for polymorphic loading control.

#2333

of_type() works with alias(), with_polymorphic(), any(), has(), joinedload(), subqueryload(), contains_eager()

The *PropComparator.of_type()* method is used to specify a specific subtype to use when constructing SQL expressions along a *relationship()* that has a polymorphic mapping as its target. This method can now be used to target *any number* of target subtypes, by combining it with the new *with_polymorphic()* function:

```
# use eager loading in conjunction with with_polymorphic targets
Job_P = with_polymorphic(Job, [SubJob, ExtraJob], aliased=True)
q = s.query(DataContainer).\
            join(DataContainer.jobs.of_type(Job_P)).\
                options(contains_eager(DataContainer.jobs.of_type(Job_P)))
```

The method now works equally well in most places a regular relationship attribute is accepted, including with loader functions like *joinedload()*, *subqueryload()*, *contains_eager()*, and comparison methods like *PropComparator.any()* and *PropComparator.has()*:

```
# use eager loading in conjunction with with_polymorphic targets
Job_P = with_polymorphic(Job, [SubJob, ExtraJob], aliased=True)
q = s.query(DataContainer).\
            join(DataContainer.jobs.of_type(Job_P)).\
                options(contains_eager(DataContainer.jobs.of_type(Job_P)))

# pass subclasses to eager loads (implicitly applies with_polymorphic)
q = s.query(ParentThing).\
                options(
                    joinedload_all(
                        ParentThing.container,
                        DataContainer.jobs.of_type(SubJob)
                ))

# control self-referential aliasing with any()/has()
Job_A = aliased(Job)
q = s.query(Job).join(DataContainer.jobs).\
                filter(
                    DataContainer.jobs.of_type(Job_A).\
                        any(and_(Job_A.id < Job.id, Job_A.type=='fred')
```

```
                        )
                    )
```

See also:

Creating Joins to Specific Subtypes

#2438 #1106

Events Can Be Applied to Unmapped Superclasses

Mapper and instance events can now be associated with an unmapped superclass, where those events will be propagated to subclasses as those subclasses are mapped. The `propagate=True` flag should be used. This feature allows events to be associated with a declarative base class:

```
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

@event.listens_for("load", Base, propagate=True)
def on_load(target, context):
    print "New instance loaded:", target

# on_load() will be applied to SomeClass
class SomeClass(Base):
    __tablename__ = 'sometable'

    # ...
```

#2585

Declarative Distinguishes Between Modules/Packages

A key feature of Declarative is the ability to refer to other mapped classes using their string name. The registry of class names is now sensitive to the owning module and package of a given class. The classes can be referred to via dotted name in expressions:

```
class Snack(Base):
    # ...

    peanuts = relationship("nuts.Peanut",
            primaryjoin="nuts.Peanut.snack_id == Snack.id")
```

The resolution allows that any full or partial disambiguating package name can be used. If the path to a particular class is still ambiguous, an error is raised.

#2338

New DeferredReflection Feature in Declarative

The "deferred reflection" example has been moved to a supported feature within Declarative. This feature allows the construction of declarative mapped classes with only placeholder `Table` metadata, until a `prepare()` step is called, given an `Engine` with which to reflect fully all tables and establish actual mappings. The system supports overriding

of columns, single and joined inheritance, as well as distinct bases-per-engine. A full declarative configuration can now be created against an existing table that is assembled upon engine creation time in one step:

```
class ReflectedOne(DeferredReflection, Base):
    __abstract__ = True

class ReflectedTwo(DeferredReflection, Base):
    __abstract__ = True

class MyClass(ReflectedOne):
    __tablename__ = 'mytable'

class MyOtherClass(ReflectedOne):
    __tablename__ = 'myothertable'

class YetAnotherClass(ReflectedTwo):
    __tablename__ = 'yetanothertable'

ReflectedOne.prepare(engine_one)
ReflectedTwo.prepare(engine_two)
```

See also:

DeferredReflection

#2485

ORM Classes Now Accepted by Core Constructs

While the SQL expressions used with *Query.filter()*, such as `User.id == 5`, have always been compatible for use with core constructs such as *select()*, the mapped class itself would not be recognized when passed to *select()*, *Select.select_from()*, or *Select.correlate()*. A new SQL registration system allows a mapped class to be accepted as a FROM clause within the core:

```
from sqlalchemy import select

stmt = select([User]).where(User.id == 5)
```

Above, the mapped `User` class will expand into the *Table* to which `User` is mapped.

#2245

Query.update() supports UPDATE..FROM

The new UPDATE..FROM mechanics work in query.update(). Below, we emit an UPDATE against `SomeEntity`, adding a FROM clause (or equivalent, depending on backend) against `SomeOtherEntity`:

```
query(SomeEntity).\
    filter(SomeEntity.id==SomeOtherEntity.id).\
    filter(SomeOtherEntity.foo=='bar').\
    update({"data":"x"})
```

In particular, updates to joined-inheritance entities are supported, provided the target of the UPDATE is local to the table being filtered on, or if the parent and child tables are mixed, they are joined explicitly in the query. Below, given `Engineer` as a joined subclass of `Person`:

```
query(Engineer).\
        filter(Person.id==Engineer.id).\
        filter(Person.name=='dilbert').\
        update({"engineer_data":"java"})
```

would produce:

```
UPDATE engineer SET engineer_data='java' FROM person
WHERE person.id=engineer.id AND person.name='dilbert'
```

#2365

rollback() will only roll back "dirty" objects from a begin_nested()

A behavioral change that should improve efficiency for those users using SAVEPOINT via `Session.begin_nested()` - upon `rollback()`, only those objects that were made dirty since the last flush will be expired, the rest of the `Session` remains intact. This because a ROLLBACK to a SAVEPOINT does not terminate the containing transaction's isolation, so no expiry is needed except for those changes that were not flushed in the current transaction.

#2452

Caching Example now uses dogpile.cache

The caching example now uses dogpile.cache. Dogpile.cache is a rewrite of the caching portion of Beaker, featuring vastly simpler and faster operation, as well as support for distributed locking.

Note that the SQLAlchemy APIs used by the Dogpile example as well as the previous Beaker example have changed slightly, in particular this change is needed as illustrated in the Beaker example:

```
--- examples/beaker_caching/caching_query.py
+++ examples/beaker_caching/caching_query.py
@@ -222,7 +222,8 @@

         """
         if query._current_path:
-            mapper, key = query._current_path[-2:]
+            mapper, prop = query._current_path[-2:]
+            key = prop.key

             for cls in mapper.class_.__mro__:
                 if (cls, key) in self._relationship_options:
```

See also:

`dogpile_caching`

#2589

New Core Features

Fully extensible, type-level operator support in Core

The Core has to date never had any system of adding support for new SQL operators to Column and other expression constructs, other than the `ColumnOperators.op()` method which is "just enough" to make things work. There has also never been any system in place for Core which allows the behavior of existing operators to be overridden. Up until now, the only way operators could be flexibly redefined was in the ORM layer, using `column_property()` given a `comparator_factory` argument. Third party libraries like GeoAlchemy therefore were forced to be ORM-centric and rely upon an array of hacks to apply new opertions as well as to get them to propagate correctly.

The new operator system in Core adds the one hook that's been missing all along, which is to associate new and overridden operators with *types*. Since after all, it's not really a column, CAST operator, or SQL function that really drives what kinds of operations are present, it's the *type* of the expression. The implementation details are minimal - only a few extra methods are added to the core `ColumnElement` type so that it consults its `TypeEngine` object for an optional set of operators. New or revised operations can be associated with any type, either via subclassing of an existing type, by using `TypeDecorator`, or "globally across-the-board" by attaching a new `TypeEngine.Comparator` object to an existing type class.

For example, to add logarithm support to `Numeric` types:

```
from sqlalchemy.types import Numeric
from sqlalchemy.sql import func

class CustomNumeric(Numeric):
    class comparator_factory(Numeric.Comparator):
        def log(self, other):
            return func.log(self.expr, other)
```

The new type is usable like any other type:

```
data = Table('data', metadata,
          Column('id', Integer, primary_key=True),
          Column('x', CustomNumeric(10, 5)),
          Column('y', CustomNumeric(10, 5))
     )

stmt = select([data.c.x.log(data.c.y)]).where(data.c.x.log(2) < value)
print conn.execute(stmt).fetchall()
```

New features which have come from this immediately include support for Postgresql's HSTORE type, as well as new operations associated with Postgresql's ARRAY type. It also paves the way for existing types to acquire lots more operators that are specific to those types, such as more string, integer and date operators.

See also:

Redefining and Creating New Operators

HSTORE

#2547

Multiple-VALUES support for Insert

The `Insert.values()` method now supports a list of dictionaries, which will render a multi-VALUES statement such as `VALUES (<row1>), (<row2>), ...`. This is only relevant to backends which support this syntax,

including Postgresql, SQLite, and MySQL. It is not the same thing as the usual `executemany()` style of INSERT which remains unchanged:

```
users.insert().values([
                    {"name": "some name"},
                    {"name": "some other name"},
                    {"name": "yet another name"},
                ])
```

See also:

Insert.values()

#2623

Type Expressions

SQL expressions can now be associated with types. Historically, *TypeEngine* has always allowed Python-side functions which receive both bound parameters as well as result row values, passing them through a Python side conversion function on the way to/back from the database. The new feature allows similar functionality, except on the database side:

```
from sqlalchemy.types import String
from sqlalchemy import func, Table, Column, MetaData

class LowerString(String):
    def bind_expression(self, bindvalue):
        return func.lower(bindvalue)

    def column_expression(self, col):
        return func.lower(col)

metadata = MetaData()
test_table = Table(
        'test_table',
        metadata,
        Column('data', LowerString)
)
```

Above, the `LowerString` type defines a SQL expression that will be emitted whenever the `test_table.c.data` column is rendered in the columns clause of a SELECT statement:

```
>>> print select([test_table]).where(test_table.c.data == 'HI')
SELECT lower(test_table.data) AS data
FROM test_table
WHERE test_table.data = lower(:data_1)
```

This feature is also used heavily by the new release of GeoAlchemy, to embed PostGIS expressions inline in SQL based on type rules.

See also:

Applying SQL-level Bind/Result Processing

#1534

Core Inspection System

The `inspect()` function introduced in *New Class/Object Inspection System* also applies to the core. Applied to an `Engine` it produces an `Inspector` object:

```
from sqlalchemy import inspect
from sqlalchemy import create_engine

engine = create_engine("postgresql://scott:tiger@localhost/test")
insp = inspect(engine)
print insp.get_table_names()
```

It can also be applied to any `ClauseElement`, which returns the `ClauseElement` itself, such as `Table`, `Column`, `Select`, etc. This allows it to work fluently between Core and ORM constructs.

New Method `Select.correlate_except()`

`select()` now has a method `Select.correlate_except()` which specifies "correlate on all FROM clauses except those specified". It can be used for mapping scenarios where a related subquery should correlate normally, except against a particular target selectable:

```
class SnortEvent(Base):
    __tablename__ = "event"

    id = Column(Integer, primary_key=True)
    signature = Column(Integer, ForeignKey("signature.id"))

    signatures = relationship("Signature", lazy=False)

class Signature(Base):
    __tablename__ = "signature"

    id = Column(Integer, primary_key=True)

    sig_count = column_property(
                    select([func.count('*')]).\
                        where(SnortEvent.signature == id).
                        correlate_except(SnortEvent)
                )
```

See also:

`Select.correlate_except()`

Postgresql HSTORE type

Support for Postgresql's `HSTORE` type is now available as `postgresql.HSTORE`. This type makes great usage of the new operator system to provide a full range of operators for HSTORE types, including index access, concatenation, and containment methods such as `has_key()`, `has_any()`, and `matrix()`:

```
from sqlalchemy.dialects.postgresql import HSTORE

data = Table('data_table', metadata,
        Column('id', Integer, primary_key=True),
```

```
        Column('hstore_data', HSTORE)
    )

engine.execute(
    select([data.c.hstore_data['some_key']])
).scalar()

engine.execute(
    select([data.c.hstore_data.matrix()])
).scalar()
```

See also:

postgresql.HSTORE

postgresql.hstore

#2606

Enhanced Postgresql ARRAY type

The *postgresql.ARRAY* type will accept an optional "dimension" argument, pinning it to a fixed number of dimensions and greatly improving efficiency when retrieving results:

```
# old way, still works since PG supports N-dimensions per row:
Column("my_array", postgresql.ARRAY(Integer))

# new way, will render ARRAY with correct number of [] in DDL,
# will process binds and results more efficiently as we don't need
# to guess how many levels deep to go
Column("my_array", postgresql.ARRAY(Integer, dimensions=2))
```

The type also introduces new operators, using the new type-specific operator framework. New operations include indexed access:

```
result = conn.execute(
    select([mytable.c.arraycol[2]])
)
```

slice access in SELECT:

```
result = conn.execute(
    select([mytable.c.arraycol[2:4]])
)
```

slice updates in UPDATE:

```
conn.execute(
    mytable.update().values({mytable.c.arraycol[2:3]: [7, 8]})
)
```

freestanding array literals:

```
>>> from sqlalchemy.dialects import postgresql
>>> conn.scalar(
...     select([
...         postgresql.array([1, 2]) + postgresql.array([3, 4, 5])
```

```
...     ])
...   )
[1, 2, 3, 4, 5]
```

array concatenation, where below, the right side `[4, 5, 6]` is coerced into an array literal:

```
select([mytable.c.arraycol + [4, 5, 6]])
```

See also:

postgresql.ARRAY

postgresql.array

#2441

New, configurable DATE, TIME types for SQLite

SQLite has no built-in DATE, TIME, or DATETIME types, and instead provides some support for storage of date and time values either as strings or integers. The date and time types for SQLite are enhanced in 0.8 to be much more configurable as to the specific format, including that the "microseconds" portion is optional, as well as pretty much everything else.

```
Column('sometimestamp', sqlite.DATETIME(truncate_microseconds=True))
Column('sometimestamp', sqlite.DATETIME(
                     storage_format=(
                                 "%(year)04d%(month)02d%(day)02d"
                                 "%(hour)02d%(minute)02d%(second)02d%(microsecond)06d"
                     ),
                     regexp="(\d{4})(\d{2})(\d{2})(\d{2})(\d{2})(\d{2})(\d{6})"
                     )
             )
Column('somedate', sqlite.DATE(
                     storage_format="%(month)02d/%(day)02d/%(year)04d",
                     regexp="(?P<month>\d+)/(?P<day>\d+)/(?P<year>\d+)",
                 )
             )
```

Huge thanks to Nate Dub for the sprinting on this at Pycon 2012.

See also:

sqlite.DATETIME

sqlite.DATE

sqlite.TIME

#2363

"COLLATE" supported across all dialects; in particular MySQL, Postgresql, SQLite

The "collate" keyword, long accepted by the MySQL dialect, is now established on all *String* types and will render on any backend, including when features such as *MetaData.create_all()* and *cast()* is used:

```
>>> stmt = select([cast(sometable.c.somechar, String(20, collation='utf8'))])
>>> print stmt
SELECT CAST(sometable.somechar AS VARCHAR(20) COLLATE "utf8") AS anon_1
FROM sometable
```

See also:

String

#2276

"Prefixes" now supported for `update()`, `delete()`

Geared towards MySQL, a "prefix" can be rendered within any of these constructs. E.g.:

```
stmt = table.delete().prefix_with("LOW_PRIORITY", dialect="mysql")


stmt = table.update().prefix_with("LOW_PRIORITY", dialect="mysql")
```

The method is new in addition to those which already existed on *insert()*, *select()* and *Query*.

See also:

Update.prefix_with()

Delete.prefix_with()

Insert.prefix_with()

Select.prefix_with()

Query.prefix_with()

#2431

Behavioral Changes

The consideration of a "pending" object as an "orphan" has been made more aggressive

This is a late add to the 0.8 series, however it is hoped that the new behavior is generally more consistent and intuitive in a wider variety of situations. The ORM has since at least version 0.4 included behavior such that an object that's "pending", meaning that it's associated with a *Session* but hasn't been inserted into the database yet, is automatically expunged from the *Session* when it becomes an "orphan", which means it has been de-associated with a parent object that refers to it with `delete-orphan` cascade on the configured *relationship()*. This behavior is intended to approximately mirror the behavior of a persistent (that is, already inserted) object, where the ORM will emit a DELETE for such objects that become orphans based on the interception of detachment events.

The behavioral change comes into play for objects that are referred to by multiple kinds of parents that each specify `delete-orphan`; the typical example is an *association object* that bridges two other kinds of objects in a many-to-many pattern. Previously, the behavior was such that the pending object would be expunged only when de-associated with *all* of its parents. With the behavioral change, the pending object is expunged as soon as it is de-associated from *any* of the parents that it was previously associated with. This behavior is intended to more closely match that of persistent objects, which are deleted as soon as they are de-associated from any parent.

The rationale for the older behavior dates back at least to version 0.4, and was basically a defensive decision to try to alleviate confusion when an object was still being constructed for INSERT. But the reality is that the object is re-associated with the *Session* as soon as it is attached to any new parent in any case.

It's still possible to flush an object that is not associated with all of its required parents, if the object was either not associated with those parents in the first place, or if it was expunged, but then re-associated with a *Session* via a subsequent attachment event but still not fully associated. In this situation, it is expected that the database would emit an integrity error, as there are likely NOT NULL foreign key columns that are unpopulated. The ORM makes the decision to let these INSERT attempts occur, based on the judgment that an object that is only partially associated with its required parents but has been actively associated with some of them, is more often than not a user error, rather than an intentional omission which should be silently skipped - silently skipping the INSERT here would make user errors of this nature very hard to debug.

The old behavior, for applications that might have been relying upon it, can be re-enabled for any *Mapper* by specifying the flag `legacy_is_orphan` as a mapper option.

The new behavior allows the following test case to work:

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship, backref
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    name = Column(String(64))

class UserKeyword(Base):
    __tablename__ = 'user_keyword'
    user_id = Column(Integer, ForeignKey('user.id'), primary_key=True)
    keyword_id = Column(Integer, ForeignKey('keyword.id'), primary_key=True)

    user = relationship(User,
                backref=backref("user_keywords",
                                cascade="all, delete-orphan")
            )

    keyword = relationship("Keyword",
                backref=backref("user_keywords",
                                cascade="all, delete-orphan")
            )

    # uncomment this to enable the old behavior
    # __mapper_args__ = {"legacy_is_orphan": True}

class Keyword(Base):
    __tablename__ = 'keyword'
    id = Column(Integer, primary_key=True)
    keyword = Column('keyword', String(64))

from sqlalchemy import create_engine
from sqlalchemy.orm import Session

# note we're using Postgresql to ensure that referential integrity
# is enforced, for demonstration purposes.
e = create_engine("postgresql://scott:tiger@localhost/test", echo=True)

Base.metadata.drop_all(e)
Base.metadata.create_all(e)
```

```
session = Session(e)

u1 = User(name="u1")
k1 = Keyword(keyword="k1")

session.add_all([u1, k1])

uk1 = UserKeyword(keyword=k1, user=u1)

# previously, if session.flush() were called here,
# this operation would succeed, but if session.flush()
# were not called here, the operation fails with an
# integrity error.
# session.flush()
del u1.user_keywords[0]

session.commit()
```

#2655

The after_attach event fires after the item is associated with the Session instead of before; before_attach added

Event handlers which use after_attach can now assume the given instance is associated with the given session:

```
@event.listens_for(Session, "after_attach")
def after_attach(session, instance):
    assert instance in session
```

Some use cases require that it work this way. However, other use cases require that the item is *not* yet part of the session, such as when a query, intended to load some state required for an instance, emits autoflush first and would otherwise prematurely flush the target object. Those use cases should use the new "before_attach" event:

```
@event.listens_for(Session, "before_attach")
def before_attach(session, instance):
    instance.some_necessary_attribute = session.query(Widget).\
                                        filter_by(instance.widget_name).\
                                        first()
```

#2464

Query now auto-correlates like a select() does

Previously it was necessary to call `Query.correlate()` in order to have a column- or WHERE-subquery correlate to the parent:

```
subq = session.query(Entity.value).\
                filter(Entity.id==Parent.entity_id).\
                correlate(Parent).\
                as_scalar()
session.query(Parent).filter(subq=="some value")
```

This was the opposite behavior of a plain `select()` construct which would assume auto-correlation by default. The above statement in 0.8 will correlate automatically:

```
subq = session.query(Entity.value).\
                filter(Entity.id==Parent.entity_id).\
                as_scalar()
session.query(Parent).filter(subq=="some value")
```

like in `select()`, correlation can be disabled by calling `query.correlate(None)` or manually set by passing an entity, `query.correlate(someentity)`.

#2179

Correlation is now always context-specific

To allow a wider variety of correlation scenarios, the behavior of *Select.correlate()* and *Query.correlate()* has changed slightly such that the SELECT statement will omit the "correlated" target from the FROM clause only if the statement is actually used in that context. Additionally, it's no longer possible for a SELECT statement that's placed as a FROM in an enclosing SELECT statement to "correlate" (i.e. omit) a FROM clause.

This change only makes things better as far as rendering SQL, in that it's no longer possible to render illegal SQL where there are insufficient FROM objects relative to what's being selected:

```
from sqlalchemy.sql import table, column, select

t1 = table('t1', column('x'))
t2 = table('t2', column('y'))
s = select([t1, t2]).correlate(t1)

print(s)
```

Prior to this change, the above would return:

```
SELECT t1.x, t2.y FROM t2
```

which is invalid SQL as "t1" is not referred to in any FROM clause.

Now, in the absence of an enclosing SELECT, it returns:

```
SELECT t1.x, t2.y FROM t1, t2
```

Within a SELECT, the correlation takes effect as expected:

```
s2 = select([t1, t2]).where(t1.c.x == t2.c.y).where(t1.c.x == s)

print (s2)

SELECT t1.x, t2.y FROM t1, t2
WHERE t1.x = t2.y AND t1.x =
    (SELECT t1.x, t2.y FROM t2)
```

This change is not expected to impact any existing applications, as the correlation behavior remains identical for properly constructed expressions. Only an application that relies, most likely within a testing scenario, on the invalid string output of a correlated SELECT used in a non-correlating context would see any change.

#2668

create_all() and drop_all() will now honor an empty list as such

The methods *MetaData.create_all()* and *MetaData.drop_all()* will now accept a list of *Table* objects that is empty, and will not emit any CREATE or DROP statements. Previously, an empty list was interepreted the same as passing `None` for a collection, and CREATE/DROP would be emitted for all items unconditionally.

This is a bug fix but some applications may have been relying upon the previous behavior.

#2664

Repaired the Event Targeting of `InstrumentationEvents`

The *InstrumentationEvents* series of event targets have documented that the events will only be fired off according to the actual class passed as a target. Through 0.7, this wasn't the case, and any event listener applied to *InstrumentationEvents* would be invoked for all classes mapped. In 0.8, additional logic has been added so that the events will only invoke for those classes sent in. The `propagate` flag here is set to `True` by default as class instrumentation events are typically used to intercept classes that aren't yet created.

#2590

No more magic coercion of "=" to IN when comparing to subquery in MS-SQL

We found a very old behavior in the MSSQL dialect which would attempt to rescue users from themselves when doing something like this:

```
scalar_subq = select([someothertable.c.id]).where(someothertable.c.data=='foo')
select([sometable]).where(sometable.c.id==scalar_subq)
```

SQL Server doesn't allow an equality comparison to a scalar SELECT, that is, "x = (SELECT something)". The MSSQL dialect would convert this to an IN. The same thing would happen however upon a comparison like "(SELECT something) = x", and overall this level of guessing is outside of SQLAlchemy's usual scope so the behavior is removed.

#2277

Fixed the behavior of `Session.is_modified()`

The *Session.is_modified()* method accepts an argument `passive` which basically should not be necessary, the argument in all cases should be the value `True` - when left at its default of `False` it would have the effect of hitting the database, and often triggering autoflush which would itself change the results. In 0.8 the `passive` argument will have no effect, and unloaded attributes will never be checked for history since by definition there can be no pending state change on an unloaded attribute.

See also:

Session.is_modified()

#2320

`Column.key` is honored in the `Select.c` attribute of `select()` with `Select.apply_labels()`

Users of the expression system know that *Select.apply_labels()* prepends the table name to each column name, affecting the names that are available from *Select.c*:

```
s = select([table1]).apply_labels()
s.c.table1_col1
s.c.table1_col2
```

Before 0.8, if the *Column* had a different `Column.key`, this key would be ignored, inconsistently versus when *Select.apply_labels()* were not used:

```
# before 0.8
table1 = Table('t1', metadata,
    Column('col1', Integer, key='column_one')
)
s = select([table1])
s.c.column_one # would be accessible like this
s.c.col1 # would raise AttributeError

s = select([table1]).apply_labels()
s.c.table1_column_one # would raise AttributeError
s.c.table1_col1 # would be accessible like this
```

In 0.8, `Column.key` is honored in both cases:

```
# with 0.8
table1 = Table('t1', metadata,
    Column('col1', Integer, key='column_one')
)
s = select([table1])
s.c.column_one # works
s.c.col1 # AttributeError

s = select([table1]).apply_labels()
s.c.table1_column_one # works
s.c.table1_col1 # AttributeError
```

All other behavior regarding "name" and "key" are the same, including that the rendered SQL will still use the form `<tablename>_<colname>` - the emphasis here was on preventing the `Column.key` contents from being rendered into the `SELECT` statement so that there are no issues with special/ non-ascii characters used in the `Column.key`.

#2397

single_parent warning is now an error

A *relationship()* that is many-to-one or many-to-many and specifies "cascade='all, delete-orphan"', which is an awkward but nonetheless supported use case (with restrictions) will now raise an error if the relationship does not specify the `single_parent=True` option. Previously it would only emit a warning, but a failure would follow almost immediately within the attribute system in any case.

#2405

Adding the `inspector` argument to the `column_reflect` event

0.7 added a new event called `column_reflect`, provided so that the reflection of columns could be augmented as each one were reflected. We got this event slightly wrong in that the event gave no way to get at the current `Inspector` and `Connection` being used for the reflection, in the case that additional information from the database is needed. As this is a new event not widely used yet, we'll be adding the `inspector` argument into it directly:

```
@event.listens_for(Table, "column_reflect")
def listen_for_col(inspector, table, column_info):
    # ...
```

#2418

Disabling auto-detect of collations, casing for MySQL

The MySQL dialect does two calls, one very expensive, to load all possible collations from the database as well as information on casing, the first time an `Engine` connects. Neither of these collections are used for any SQLAlchemy functions, so these calls will be changed to no longer be emitted automatically. Applications that might have relied on these collections being present on `engine.dialect` will need to call upon `_detect_collations()` and `_detect_casing()` directly.

#2404

"Unconsumed column names" warning becomes an exception

Referring to a non-existent column in an `insert()` or `update()` construct will raise an error instead of a warning:

```
t1 = table('t1', column('x'))
t1.insert().values(x=5, z=5) # raises "Unconsumed column names: z"
```

#2415

Inspector.get_primary_keys() is deprecated, use Inspector.get_pk_constraint

These two methods on `Inspector` were redundant, where `get_primary_keys()` would return the same information as `get_pk_constraint()` minus the name of the constraint:

```
>>> insp.get_primary_keys()
["a", "b"]

>>> insp.get_pk_constraint()
{"name":"pk_constraint", "constrained_columns":["a", "b"]}
```

#2422

Case-insensitive result row names will be disabled in most cases

A very old behavior, the column names in `RowProxy` were always compared case-insensitively:

```
>>> row = result.fetchone()
>>> row['foo'] == row['FOO'] == row['Foo']
True
```

This was for the benefit of a few dialects which in the early days needed this, like Oracle and Firebird, but in modern usage we have more accurate ways of dealing with the case-insensitive behavior of these two platforms.

Going forward, this behavior will be available only optionally, by passing the flag `'case_sensitive=False'` to `'create_engine()'`, but otherwise column names requested from the row must match as far as casing.

#2423

`InstrumentationManager` and alternate class instrumentation is now an extension

The `sqlalchemy.orm.interfaces.InstrumentationManager` class is moved to `sqlalchemy.ext.instrumentation.InstrumentationManager`. The "alternate instrumentation" system was built for the benefit of a very small number of installations that needed to work with existing or unusual class instrumentation systems, and generally is very seldom used. The complexity of this system has been exported to an `ext.` module. It remains unused until once imported, typically when a third party library imports `InstrumentationManager`, at which point it is injected back into `sqlalchemy.orm` by replacing the default `InstrumentationFactory` with `ExtendedInstrumentationRegistry`.

Removed

SQLSoup

SQLSoup is a handy package that presents an alternative interface on top of the SQLAlchemy ORM. SQLSoup is now moved into its own project and documented/released separately; see https://bitbucket.org/zzzeek/sqlsoup.

SQLSoup is a very simple tool that could also benefit from contributors who are interested in its style of usage.

#2262

MutableType

The older "mutable" system within the SQLAlchemy ORM has been removed. This refers to the `MutableType` interface which was applied to types such as `PickleType` and conditionally to `TypeDecorator`, and since very early SQLAlchemy versions has provided a way for the ORM to detect changes in so-called "mutable" data structures such as JSON structures and pickled objects. However, the implementation was never reasonable and forced a very inefficient mode of usage on the unit-of-work which caused an expensive scan of all objects to take place during flush. In 0.7, the sqlalchemy.ext.mutable extension was introduced so that user-defined datatypes can appropriately send events to the unit of work as changes occur.

Today, usage of `MutableType` is expected to be low, as warnings have been in place for some years now regarding its inefficiency.

#2442

sqlalchemy.exceptions (has been sqlalchemy.exc for years)

We had left in an alias `sqlalchemy.exceptions` to attempt to make it slightly easier for some very old libraries that hadn't yet been upgraded to use `sqlalchemy.exc`. Some users are still being confused by it however so in 0.8 we're taking it out entirely to eliminate any of that confusion.

#2433

6.3.4 What's New in SQLAlchemy 0.7?

About this Document

This document describes changes between SQLAlchemy version 0.6, last released May 5, 2012, and SQLAlchemy version 0.7, undergoing maintenance releases as of October, 2012.

Document date: July 27, 2011

Introduction

This guide introduces what's new in SQLAlchemy version 0.7, and also documents changes which affect users migrating their applications from the 0.6 series of SQLAlchemy to 0.7.

To as great a degree as possible, changes are made in such a way as to not break compatibility with applications built for 0.6. The changes that are necessarily not backwards compatible are very few, and all but one, the change to mutable attribute defaults, should affect an exceedingly small portion of applications - many of the changes regard non-public APIs and undocumented hacks some users may have been attempting to use.

A second, even smaller class of non-backwards-compatible changes is also documented. This class of change regards those features and behaviors that have been deprecated at least since version 0.5 and have been raising warnings since their deprecation. These changes would only affect applications that are still using 0.4- or early 0.5-style APIs. As the project matures, we have fewer and fewer of these kinds of changes with 0.x level releases, which is a product of our API having ever fewer features that are less than ideal for the use cases they were meant to solve.

An array of existing functionalities have been superseded in SQLAlchemy 0.7. There's not much difference between the terms "superseded" and "deprecated", except that the former has a much weaker suggestion of the old feature would ever be removed. In 0.7, features like `synonym` and `comparable_property`, as well as all the `Extension` and other event classes, have been superseded. But these "superseded" features have been re-implemented such that their implementations live mostly outside of core ORM code, so their continued "hanging around" doesn't impact SQLAlchemy's ability to further streamline and refine its internals, and we expect them to remain within the API for the foreseeable future.

New Features

New Event System

SQLAlchemy started early with the `MapperExtension` class, which provided hooks into the persistence cycle of mappers. As SQLAlchemy quickly became more componentized, pushing mappers into a more focused configurational role, many more "extension", "listener", and "proxy" classes popped up to solve various activity-interception use cases in an ad-hoc fashion. Part of this was driven by the divergence of activities; `ConnectionProxy` objects wanted to provide a system of rewriting statements and parameters; `AttributeExtension` provided a system of replacing incoming values, and `DDL` objects had events that could be switched off of dialect-sensitive callables.

0.7 re-implements virtually all of these plugin points with a new, unified approach, which retains all the functionalities of the different systems, provides more flexibility and less boilerplate, performs better, and eliminates the need to learn radically different APIs for each event subsystem. The pre-existing classes `MapperExtension`, `SessionExtension`, `AttributeExtension`, `ConnectionProxy`, `PoolListener` as well as the `DDLElement.execute_at` method are deprecated and now implemented in terms of the new system - these APIs remain fully functional and are expected to remain in place for the foreseeable future.

The new approach uses named events and user-defined callables to associate activities with events. The API's look and feel was driven by such diverse sources as JQuery, Blinker, and Hibernate, and was also modified further on several occasions during conferences with dozens of users on Twitter, which appears to have a much higher response rate than the mailing list for such questions.

It also features an open-ended system of target specification that allows events to be associated with API classes, such as for all `Session` or `Engine` objects, with specific instances of API classes, such as for a specific `Pool` or `Mapper`, as well as for related objects like a user- defined class that's mapped, or something as specific as a certain attribute on instances of a particular subclass of a mapped parent class. Individual listener subsystems can apply wrappers to incoming user- defined listener functions which modify how they are called - an mapper event can receive either the instance of the object being operated upon, or its underlying `InstanceState` object. An attribute event can opt whether or not to have the responsibility of returning a new value.

Several systems now build upon the new event API, including the new "mutable attributes" API as well as composite attributes. The greater emphasis on events has also led to the introduction of a handful of new events, including attribute expiration and refresh operations, pickle loads/dumps operations, completed mapper construction operations.

See also:

Events

#1902

Hybrid Attributes, implements/supersedes synonym(), comparable_property()

The "derived attributes" example has now been turned into an official extension. The typical use case for `synonym()` is to provide descriptor access to a mapped column; the use case for `comparable_property()` is to be able to return a `PropComparator` from any descriptor. In practice, the approach of "derived" is easier to use, more extensible, is implemented in a few dozen lines of pure Python with almost no imports, and doesn't require the ORM core to even be aware of it. The feature is now known as the "Hybrid Attributes" extension.

`synonym()` and `comparable_property()` are still part of the ORM, though their implementations have been moved outwards, building on an approach that is similar to that of the hybrid extension, so that the core ORM mapper/query/property modules aren't really aware of them otherwise.

See also:

Hybrid Attributes

#1903

Speed Enhancements

As is customary with all major SQLA releases, a wide pass through the internals to reduce overhead and callcounts has been made which further reduces the work needed in common scenarios. Highlights of this release include:

- The flush process will now bundle INSERT statements into batches fed to `cursor.executemany()`, for rows where the primary key is already present. In particular this usually applies to the "child" table on a joined table inheritance configuration, meaning the number of calls to `cursor.execute` for a large bulk insert of joined- table objects can be cut in half, allowing native DBAPI optimizations to take place for those statements passed to `cursor.executemany()` (such as re-using a prepared statement).
- The codepath invoked when accessing a many-to-one reference to a related object that's already loaded has been greatly simplified. The identity map is checked directly without the need to generate a new `Query` object first, which is expensive in the context of thousands of in-memory many-to-ones being accessed. The usage of constructed-per-call "loader" objects is also no longer used for the majority of lazy attribute loads.
- The rewrite of composites allows a shorter codepath when mapper internals access mapped attributes within a flush.
- New inlined attribute access functions replace the previous usage of "history" when the "save-update" and other cascade operations need to cascade among the full scope of datamembers associated with an attribute. This reduces the overhead of generating a new `History` object for this speed-critical operation.
- The internals of the `ExecutionContext`, the object corresponding to a statement execution, have been inlined and simplified.
- The `bind_processor()` and `result_processor()` callables generated by types for each statement execution are now cached (carefully, so as to avoid memory leaks for ad-hoc types and dialects) for the lifespan of that type, further reducing per-statement call overhead.

- The collection of "bind processors" for a particular `Compiled` instance of a statement is also cached on the `Compiled` object, taking further advantage of the "compiled cache" used by the flush process to re-use the same compiled form of INSERT, UPDATE, DELETE statements.

A demonstration of callcount reduction including a sample benchmark script is at http://techspot.zzzeek.org/2010/12/12/a-tale-of-three- profiles/

Composites Rewritten

The "composite" feature has been rewritten, like `synonym()` and `comparable_property()`, to use a lighter weight implementation based on descriptors and events, rather than building into the ORM internals. This allowed the removal of some latency from the mapper/unit of work internals, and simplifies the workings of composite. The composite attribute now no longer conceals the underlying columns it builds upon, which now remain as regular attributes. Composites can also act as a proxy for `relationship()` as well as `Column()` attributes.

The major backwards-incompatible change of composites is that they no longer use the `mutable=True` system to detect in-place mutations. Please use the Mutation Tracking extension to establish in-place change events to existing composite usage.

See also:

Composite Column Types

Mutation Tracking

#2008 #2024

More succinct form of query.join(target, onclause)

The default method of issuing `query.join()` to a target with an explicit onclause is now:

```
query.join(SomeClass, SomeClass.id==ParentClass.some_id)
```

In 0.6, this usage was considered to be an error, because `join()` accepts multiple arguments corresponding to multiple JOIN clauses - the two-argument form needed to be in a tuple to disambiguate between single-argument and two-argument join targets. In the middle of 0.6 we added detection and an error message for this specific calling style, since it was so common. In 0.7, since we are detecting the exact pattern anyway, and since having to type out a tuple for no reason is extremely annoying, the non- tuple method now becomes the "normal" way to do it. The "multiple JOIN" use case is exceedingly rare compared to the single join case, and multiple joins these days are more clearly represented by multiple calls to `join()`.

The tuple form will remain for backwards compatibility.

Note that all the other forms of `query.join()` remain unchanged:

```
query.join(MyClass.somerelation)
query.join("somerelation")
query.join(MyTarget)
# ... etc
```

Querying with Joins

#1923

Mutation event extension, supersedes "mutable=True"

A new extension, *Mutation Tracking*, provides a mechanism by which user-defined datatypes can provide change events back to the owning parent or parents. The extension includes an approach for scalar database values, such as those managed by `PickleType`, `postgresql.ARRAY`, or other custom `MutableType` classes, as well as an approach for ORM "composites", those configured using `composite()`.

See also:

Mutation Tracking

NULLS FIRST / NULLS LAST operators

These are implemented as an extension to the `asc()` and `desc()` operators, called `nullsfirst()` and `nullslast()`.

See also:

`nullsfirst()`

`nullslast()`

#723

select.distinct(), query.distinct() accepts *args for Postgresql DISTINCT ON

This was already available by passing a list of expressions to the `distinct` keyword argument of `select()`, the `distinct()` method of `select()` and `Query` now accept positional arguments which are rendered as DISTINCT ON when a Postgresql backend is used.

distinct()

Query.distinct()

#1069

`Index()` can be placed inline inside of `Table`, `__table_args__`

The Index() construct can be created inline with a Table definition, using strings as column names, as an alternative to the creation of the index outside of the Table. That is:

```
Table('mytable', metadata,
        Column('id',Integer, primary_key=True),
        Column('name', String(50), nullable=False),
        Index('idx_name', 'name')
)
```

The primary rationale here is for the benefit of declarative `__table_args__`, particularly when used with mixins:

```
class HasNameMixin(object):
    name = Column('name', String(50), nullable=False)
    @declared_attr
    def __table_args__(cls):
        return (Index('name'), {})

class User(HasNameMixin, Base):
```

```
    __tablename__ = 'user'
    id = Column('id', Integer, primary_key=True)
```

Indexes

Window Function SQL Construct

A "window function" provides to a statement information about the result set as it's produced. This allows criteria against various things like "row number", "rank" and so forth. They are known to be supported at least by Postgresql, SQL Server and Oracle, possibly others.

The best introduction to window functions is on Postgresql's site, where window functions have been supported since version 8.4:

http://www.postgresql.org/docs/9.0/static/tutorial- window.html

SQLAlchemy provides a simple construct typically invoked via an existing function clause, using the `over()` method, which accepts `order_by` and `partition_by` keyword arguments. Below we replicate the first example in PG's tutorial:

```
from sqlalchemy.sql import table, column, select, func

empsalary = table('empsalary',
                column('depname'),
                column('empno'),
                column('salary'))

s = select([
        empsalary,
        func.avg(empsalary.c.salary).
              over(partition_by=empsalary.c.depname).
              label('avg')
    ])

print s
```

SQL:

```
SELECT empsalary.depname, empsalary.empno, empsalary.salary,
avg(empsalary.salary) OVER (PARTITION BY empsalary.depname) AS avg
FROM empsalary
```

sqlalchemy.sql.expression.over

#1844

execution_options() on Connection accepts "isolation_level" argument

This sets the transaction isolation level for a single `Connection`, until that `Connection` is closed and its underlying DBAPI resource returned to the connection pool, upon which the isolation level is reset back to the default. The default isolation level is set using the `isolation_level` argument to `create_engine()`.

Transaction isolation support is currently only supported by the Postgresql and SQLite backends.

execution_options()

#2001

`TypeDecorator` works with integer primary key columns

A `TypeDecorator` which extends the behavior of `Integer` can be used with a primary key column. The "autoincrement" feature of `Column` will now recognize that the underlying database column is still an integer so that lastrowid mechanisms continue to function. The `TypeDecorator` itself will have its result value processor applied to newly generated primary keys, including those received by the DBAPI `cursor.lastrowid` accessor.

#2005 #2006

`TypeDecorator` is present in the "sqlalchemy" import space

No longer need to import this from `sqlalchemy.types`, it's now mirrored in `sqlalchemy`.

New Dialects

Dialects have been added:

- a MySQLdb driver for the Drizzle database:

  Drizzle
- support for the pymysql DBAPI:

  pymsql Notes
- psycopg2 now works with Python 3

Behavioral Changes (Backwards Compatible)

C Extensions Build by Default

This is as of 0.7b4. The exts will build if cPython 2.xx is detected. If the build fails, such as on a windows install, that condition is caught and the non-C install proceeds. The C exts won't build if Python 3 or Pypy is used.

Query.count() simplified, should work virtually always

The very old guesswork which occurred within `Query.count()` has been modernized to use `.from_self()`. That is, `query.count()` is now equivalent to:

```
query.from_self(func.count(literal_column('1'))).scalar()
```

Previously, internal logic attempted to rewrite the columns clause of the query itself, and upon detection of a "subquery" condition, such as a column-based query that might have aggregates in it, or a query with DISTINCT, would go through a convoluted process of rewriting the columns clause. This logic failed in complex conditions, particularly those involving joined table inheritance, and was long obsolete by the more comprehensive `.from_self()` call.

The SQL emitted by `query.count()` is now always of the form:

```
SELECT count(1) AS count_1 FROM (
    SELECT user.id AS user_id, user.name AS user_name from user
) AS anon_1
```

that is, the original query is preserved entirely inside of a subquery, with no more guessing as to how count should be applied.

#2093

To emit a non-subquery form of count() MySQL users have already reported that the MyISAM engine not surprisingly falls over completely with this simple change. Note that for a simple `count()` that optimizes for DBs that can't handle simple subqueries, `func.count()` should be used:

```
from sqlalchemy import func
session.query(func.count(MyClass.id)).scalar()
```

or for `count(*)`:

```
from sqlalchemy import func, literal_column
session.query(func.count(literal_column('*'))).select_from(MyClass).scalar()
```

LIMIT/OFFSET clauses now use bind parameters

The LIMIT and OFFSET clauses, or their backend equivalents (i.e. TOP, ROW NUMBER OVER, etc.), use bind parameters for the actual values, for all backends which support it (most except for Sybase). This allows better query optimizer performance as the textual string for multiple statements with differing LIMIT/OFFSET are now identical.

#805

Logging enhancements

Vinay Sajip has provided a patch to our logging system such that the "hex string" embedded in logging statements for engines and pools is no longer needed to allow the `echo` flag to work correctly. A new system that uses filtered logging objects allows us to maintain our current behavior of `echo` being local to individual engines without the need for additional identifying strings local to those engines.

#1926

Simplified polymorphic_on assignment

The population of the `polymorphic_on` column-mapped attribute, when used in an inheritance scenario, now occurs when the object is constructed, i.e. its `__init__` method is called, using the init event. The attribute then behaves the same as any other column-mapped attribute. Previously, special logic would fire off during flush to populate this column, which prevented any user code from modifying its behavior. The new approach improves upon this in three ways: 1. the polymorphic identity is now present on the object as soon as its constructed; 2. the polymorphic identity can be changed by user code without any difference in behavior from any other column-mapped attribute; 3. the internals of the mapper during flush are simplified and no longer need to make special checks for this column.

#1895

contains_eager() chains across multiple paths (i.e. "all()")

The `'contains_eager()``` modifier now will chain itself for a longer path without the need to emit individual ```'contains_eager()` calls. Instead of:

```
session.query(A).options(contains_eager(A.b), contains_eager(A.b, B.c))
```

you can say:

```
session.query(A).options(contains_eager(A.b, B.c))
```

#2032

Flushing of orphans that have no parent is allowed

We've had a long standing behavior that checks for a so- called "orphan" during flush, that is, an object which is associated with a `relationship()` that specifies "delete- orphan" cascade, has been newly added to the session for an INSERT, and no parent relationship has been established. This check was added years ago to accommodate some test cases which tested the orphan behavior for consistency. In modern SQLA, this check is no longer needed on the Python side. The equivalent behavior of the "orphan check" is accomplished by making the foreign key reference to the object's parent row NOT NULL, where the database does its job of establishing data consistency in the same way SQLA allows most other operations to do. If the object's parent foreign key is nullable, then the row can be inserted. The "orphan" behavior runs when the object was persisted with a particular parent, and is then disassociated with that parent, leading to a DELETE statement emitted for it.

#1912

Warnings generated when collection members, scalar referents not part of the flush

Warnings are now emitted when related objects referenced via a loaded `relationship()` on a parent object marked as "dirty" are not present in the current `Session`.

The `save-update` cascade takes effect when objects are added to the `Session`, or when objects are first associated with a parent, so that an object and everything related to it are usually all present in the same `Session`. However, if `save-update` cascade is disabled for a particular `relationship()`, then this behavior does not occur, and the flush process does not try to correct for it, instead staying consistent to the configured cascade behavior. Previously, when such objects were detected during the flush, they were silently skipped. The new behavior is that a warning is emitted, for the purposes of alerting to a situation that more often than not is the source of unexpected behavior.

#1973

Setup no longer installs a Nose plugin

Since we moved to nose we've used a plugin that installs via setuptools, so that the `nosetests` script would automatically run SQLA's plugin code, necessary for our tests to have a full environment. In the middle of 0.6, we realized that the import pattern here meant that Nose's "coverage" plugin would break, since "coverage" requires that it be started before any modules to be covered are imported; so in the middle of 0.6 we made the situation worse by adding a separate `sqlalchemy-nose` package to the build to overcome this.

In 0.7 we've done away with trying to get `nosetests` to work automatically, since the SQLAlchemy module would produce a large number of nose configuration options for all usages of `nosetests`, not just the SQLAlchemy unit tests themselves, and the additional `sqlalchemy-nose` install was an even worse idea, producing an extra package in Python environments. The `sqla_nose.py` script in 0.7 is now the only way to run the tests with nose.

#1949

Non-`Table`-derived constructs can be mapped

A construct that isn't against any `Table` at all, like a function, can be mapped.

```
from sqlalchemy import select, func
from sqlalchemy.orm import mapper

class Subset(object):
    pass
selectable = select(["x", "y", "z"]).select_from(func.some_db_function()).alias()
mapper(Subset, selectable, primary_key=[selectable.c.x])
```

#1876

aliased() accepts `FromClause` elements

This is a convenience helper such that in the case a plain `FromClause`, such as a `select`, `Table` or `join` is passed to the `orm.aliased()` construct, it passes through to the `.alias()` method of that from construct rather than constructing an ORM level `AliasedClass`.

#2018

Session.connection(), Session.execute() accept 'bind'

This is to allow execute/connection operations to participate in the open transaction of an engine explicitly. It also allows custom subclasses of `Session` that implement their own `get_bind()` method and arguments to use those custom arguments with both the `execute()` and `connection()` methods equally.

Session.connection Session.execute

#1996

Standalone bind parameters in columns clause auto-labeled.

Bind parameters present in the "columns clause" of a select are now auto-labeled like other "anonymous" clauses, which among other things allows their "type" to be meaningful when the row is fetched, as in result row processors.

SQLite - relative file paths are normalized through os.path.abspath()

This so that a script that changes the current directory will continue to target the same location as subsequent SQLite connections are established.

#2036

MS-SQL - `String`/`Unicode`/`VARCHAR`/`NVARCHAR`/`VARBINARY` emit "max" for no length

On the MS-SQL backend, the String/Unicode types, and their counterparts VARCHAR/ NVARCHAR, as well as VARBINARY (#1833) emit "max" as the length when no length is specified. This makes it more compatible with Postgresql's VARCHAR type which is similarly unbounded when no length specified. SQL Server defaults the length on these types to '1' when no length is specified.

Behavioral Changes (Backwards Incompatible)

Note again, aside from the default mutability change, most of these changes are *extremely minor* and will not affect most users.

`PickleType` and ARRAY mutability turned off by default

This change refers to the default behavior of the ORM when mapping columns that have either the `PickleType` or `postgresql.ARRAY` datatypes. The `mutable` flag is now set to `False` by default. If an existing application uses these types and depends upon detection of in-place mutations, the type object must be constructed with `mutable=True` to restore the 0.6 behavior:

```
Table('mytable', metadata,
    # ....

    Column('pickled_data', PickleType(mutable=True))
)
```

The `mutable=True` flag is being phased out, in favor of the new Mutation Tracking extension. This extension provides a mechanism by which user-defined datatypes can provide change events back to the owning parent or parents.

The previous approach of using `mutable=True` does not provide for change events - instead, the ORM must scan through all mutable values present in a session and compare them against their original value for changes every time `flush()` is called, which is a very time consuming event. This is a holdover from the very early days of SQLAlchemy when `flush()` was not automatic and the history tracking system was not nearly as sophisticated as it is now.

Existing applications which use `PickleType`, `postgresql.ARRAY` or other `MutableType` subclasses, and require in-place mutation detection, should migrate to the new mutation tracking system, as `mutable=True` is likely to be deprecated in the future.

#1980

Mutability detection of `composite()` requires the Mutation Tracking Extension

So-called "composite" mapped attributes, those configured using the technique described at Composite Column Types, have been re-implemented such that the ORM internals are no longer aware of them (leading to shorter and more efficient codepaths in critical sections). While composite types are generally intended to be treated as immutable value objects, this was never enforced. For applications that use composites with mutability, the Mutation Tracking extension offers a base class which establishes a mechanism for user-defined composite types to send change event messages back to the owning parent or parents of each object.

Applications which use composite types and rely upon in- place mutation detection of these objects should either migrate to the "mutation tracking" extension, or change the usage of the composite types such that in-place changes are no longer needed (i.e., treat them as immutable value objects).

SQLite - the SQLite dialect now uses `NullPool` for file-based databases

This change is **99.999% backwards compatible**, unless you are using temporary tables across connection pool connections.

A file-based SQLite connection is blazingly fast, and using `NullPool` means that each call to `Engine.connect` creates a new pysqlite connection.

Previously, the `SingletonThreadPool` was used, which meant that all connections to a certain engine in a thread would be the same connection. It's intended that the new approach is more intuitive, particularly when multiple connections are used.

`SingletonThreadPool` is still the default engine when a `:memory:` database is used.

Note that this change **breaks temporary tables used across Session commits**, due to the way SQLite handles temp tables. See the note at http://www.sqlalchemy.org/docs/dialects/sqlite.html#using- temporary-tables-with-sqlite if temporary tables beyond the scope of one pool connection are desired.

#1921

`Session.merge()` checks version ids for versioned mappers

Session.merge() will check the version id of the incoming state against that of the database, assuming the mapping uses version ids and incoming state has a version_id assigned, and raise StaleDataError if they don't match. This is the correct behavior, in that if incoming state contains a stale version id, it should be assumed the state is stale.

If merging data into a versioned state, the version id attribute can be left undefined, and no version check will take place.

This check was confirmed by examining what Hibernate does - both the `merge()` and the versioning features were originally adapted from Hibernate.

#2027

Tuple label names in Query Improved

This improvement is potentially slightly backwards incompatible for an application that relied upon the old behavior.

Given two mapped classes `Foo` and `Bar` each with a column `spam`:

```
qa = session.query(Foo.spam)
qb = session.query(Bar.spam)

qu = qa.union(qb)
```

The name given to the single column yielded by `qu` will be `spam`. Previously it would be something like `foo_spam` due to the way the `union` would combine things, which is inconsistent with the name `spam` in the case of a non-unioned query.

#1942

Mapped column attributes reference the most specific column first

This is a change to the behavior involved when a mapped column attribute references multiple columns, specifically when dealing with an attribute on a joined-table subclass that has the same name as that of an attribute on the superclass.

Using declarative, the scenario is this:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)

class Child(Parent):
```

```
   __tablename__ = 'child'
    id = Column(Integer, ForeignKey('parent.id'), primary_key=True)
```

Above, the attribute `Child.id` refers to both the `child.id` column as well as `parent.id` - this due to the name of the attribute. If it were named differently on the class, such as `Child.child_id`, it then maps distinctly to `child.id`, with `Child.id` being the same attribute as `Parent.id`.

When the `id` attribute is made to reference both `parent.id` and `child.id`, it stores them in an ordered list. An expression such as `Child.id` then refers to just *one* of those columns when rendered. Up until 0.6, this column would be `parent.id`. In 0.7, it is the less surprising `child.id`.

The legacy of this behavior deals with behaviors and restrictions of the ORM that don't really apply anymore; all that was needed was to reverse the order.

A primary advantage of this approach is that it's now easier to construct `primaryjoin` expressions that refer to the local column:

```
class Child(Parent):
   __tablename__ = 'child'
    id = Column(Integer, ForeignKey('parent.id'), primary_key=True)
    some_related = relationship("SomeRelated",
                    primaryjoin="Child.id==SomeRelated.child_id")

class SomeRelated(Base):
   __tablename__ = 'some_related'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
```

Prior to 0.7 the `Child.id` expression would reference `Parent.id`, and it would be necessary to map `child.id` to a distinct attribute.

It also means that a query like this one changes its behavior:

```
session.query(Parent).filter(Child.id > 7)
```

In 0.6, this would render:

```
SELECT parent.id AS parent_id
FROM parent
WHERE parent.id > :id_1
```

in 0.7, you get:

```
SELECT parent.id AS parent_id
FROM parent, child
WHERE child.id > :id_1
```

which you'll note is a cartesian product - this behavior is now equivalent to that of any other attribute that is local to `Child`. The `with_polymorphic()` method, or a similar strategy of explicitly joining the underlying `Table` objects, is used to render a query against all `Parent` objects with criteria against `Child`, in the same manner as that of 0.5 and 0.6:

```
print s.query(Parent).with_polymorphic([Child]).filter(Child.id > 7)
```

Which on both 0.6 and 0.7 renders:

```
SELECT parent.id AS parent_id, child.id AS child_id
FROM parent LEFT OUTER JOIN child ON parent.id = child.id
WHERE child.id > :id_1
```

Another effect of this change is that a joined-inheritance load across two tables will populate from the child table's value, not that of the parent table. An unusual case is that a query against "Parent" using `with_polymorphic="*"` issues a query against "parent", with a LEFT OUTER JOIN to "child". The row is located in "Parent", sees the polymorphic identity corresponds to "Child", but suppose the actual row in "child" has been *deleted*. Due to this corruption, the row comes in with all the columns corresponding to "child" set to NULL - this is now the value that gets populated, not the one in the parent table.

#1892

Mapping to joins with two or more same-named columns requires explicit declaration

This is somewhat related to the previous change in #1892. When mapping to a join, same-named columns must be explicitly linked to mapped attributes, i.e. as described in Mapping a Class Against Multiple Tables.

Given two tables `foo` and `bar`, each with a primary key column `id`, the following now produces an error:

```
foobar = foo.join(bar, foo.c.id==bar.c.foo_id)
mapper(FooBar, foobar)
```

This because the `mapper()` refuses to guess what column is the primary representation of `FooBar.id` - is it `foo.c.id` or is it `bar.c.id` ? The attribute must be explicit:

```
foobar = foo.join(bar, foo.c.id==bar.c.foo_id)
mapper(FooBar, foobar, properties={
    'id':[foo.c.id, bar.c.id]
})
```

#1896

Mapper requires that polymorphic_on column be present in the mapped selectable

This is a warning in 0.6, now an error in 0.7. The column given for `polymorphic_on` must be in the mapped selectable. This to prevent some occasional user errors such as:

```
mapper(SomeClass, sometable, polymorphic_on=some_lookup_table.c.id)
```

where above the polymorphic_on needs to be on a `sometable` column, in this case perhaps `sometable.c.some_lookup_id`. There are also some "polymorphic union" scenarios where similar mistakes sometimes occur.

Such a configuration error has always been "wrong", and the above mapping doesn't work as specified - the column would be ignored. It is however potentially backwards incompatible in the rare case that an application has been unknowingly relying upon this behavior.

#1875

`DDL()` constructs now escape percent signs

Previously, percent signs in `DDL()` strings would have to be escaped, i.e. `%%` depending on DBAPI, for those DBAPIs that accept `pyformat` or `format` binds (i.e. psycopg2, mysql-python), which was inconsistent versus `text()` constructs which did this automatically. The same escaping now occurs for `DDL()` as for `text()`.

#1897

`Table.c` / `MetaData.tables` refined a bit, don't allow direct mutation

Another area where some users were tinkering around in such a way that doesn't actually work as expected, but still left an exceedingly small chance that some application was relying upon this behavior, the construct returned by the `.c` attribute on `Table` and the `.tables` attribute on `MetaData` is explicitly non-mutable. The "mutable" version of the construct is now private. Adding columns to `.c` involves using the `append_column()` method of `Table`, which ensures things are associated with the parent `Table` in the appropriate way; similarly, `MetaData.tables` has a contract with the `Table` objects stored in this dictionary, as well as a little bit of new bookkeeping in that a `set()` of all schema names is tracked, which is satisfied only by using the public `Table` constructor as well as `Table.tometadata()`.

It is of course possible that the `ColumnCollection` and `dict` collections consulted by these attributes could someday implement events on all of their mutational methods such that the appropriate bookkeeping occurred upon direct mutation of the collections, but until someone has the motivation to implement all that along with dozens of new unit tests, narrowing the paths to mutation of these collections will ensure no application is attempting to rely upon usages that are currently not supported.

#1893 #1917

server_default consistently returns None for all inserted_primary_key values

Established consistency when server_default is present on an Integer PK column. SQLA doesn't pre-fetch these, nor do they come back in cursor.lastrowid (DBAPI). Ensured all backends consistently return None in result.inserted_primary_key for these - some backends may have returned a value previously. Using a server_default on a primary key column is extremely unusual. If a special function or SQL expression is used to generate primary key defaults, this should be established as a Python-side "default" instead of server_default.

Regarding reflection for this case, reflection of an int PK col with a server_default sets the "autoincrement" flag to False, except in the case of a PG SERIAL col where we detected a sequence default.

#2020 #2021

The `sqlalchemy.exceptions` alias in sys.modules is removed

For a few years we've added the string `sqlalchemy.exceptions` to `sys.modules`, so that a statement like "`import sqlalchemy.exceptions`" would work. The name of the core exceptions module has been `exc` for a long time now, so the recommended import for this module is:

```
from sqlalchemy import exc
```

The `exceptions` name is still present in "`sqlalchemy`" for applications which might have said `from sqlalchemy import exceptions`, but they should also start using the `exc` name.

Query Timing Recipe Changes

While not part of SQLAlchemy itself, it's worth mentioning that the rework of the `ConnectionProxy` into the new event system means it is no longer appropriate for the "Timing all Queries" recipe. Please adjust query-timers to use the `before_cursor_execute()` and `after_cursor_execute()` events, demonstrated in the updated recipe UsageRecipes/Profiling.

Deprecated API

Default constructor on types will not accept arguments

Simple types like `Integer`, `Date` etc. in the core types module don't accept arguments. The default constructor that accepts/ignores a catchall `\*args, \**kwargs` is restored as of 0.7b4/0.7.0, but emits a deprecation warning.

If arguments are being used with a core type like `Integer`, it may be that you intended to use a dialect specific type, such as `sqlalchemy.dialects.mysql.INTEGER` which does accept a "display_width" argument for example.

compile_mappers() renamed configure_mappers(), simplified configuration internals

This system slowly morphed from something small, implemented local to an individual mapper, and poorly named into something that's more of a global "registry-" level function and poorly named, so we've fixed both by moving the implementation out of `Mapper` altogether and renaming it to `configure_mappers()`. It is of course normally not needed for an application to call `configure_mappers()` as this process occurs on an as-needed basis, as soon as the mappings are needed via attribute or query access.

#1966

Core listener/proxy superseded by event listeners

`PoolListener`, `ConnectionProxy`, `DDLElement.execute_at` are superseded by `event.listen()`, using the `PoolEvents`, `EngineEvents`, `DDLEvents` dispatch targets, respectively.

ORM extensions superseded by event listeners

`MapperExtension`, `AttributeExtension`, `SessionExtension` are superseded by `event.listen()`, using the `MapperEvents`/`InstanceEvents`, `AttributeEvents`, `SessionEvents`, dispatch targets, respectively.

Sending a string to 'distinct' in select() for MySQL should be done via prefixes

This obscure feature allows this pattern with the MySQL backend:

```
select([mytable], distinct='ALL', prefixes=['HIGH_PRIORITY'])
```

The `prefixes` keyword or `prefix_with()` method should be used for non-standard or unusual prefixes:

```
select([mytable]).prefix_with('HIGH_PRIORITY', 'ALL')
```

`useexisting` superseded by `extend_existing` and `keep_existing`

The `useexisting` flag on Table has been superseded by a new pair of flags `keep_existing` and `extend_existing`. `extend_existing` is equivalent to `useexisting` - the existing Table is returned, and additional constructor elements are added. With `keep_existing`, the existing Table is returned, but additional constructor elements are not added - these elements are only applied when the Table is newly created.

Backwards Incompatible API Changes

Callables passed to `bindparam()` don't get evaluated - affects the Beaker example

#1950

Note this affects the Beaker caching example, where the workings of the `_params_from_query()` function needed a slight adjustment. If you're using code from the Beaker example, this change should be applied.

types.type_map is now private, types._type_map

We noticed some users tapping into this dictionary inside of `sqlalchemy.types` as a shortcut to associating Python types with SQL types. We can't guarantee the contents or format of this dictionary, and additionally the business of associating Python types in a one-to-one fashion has some grey areas that should are best decided by individual applications, so we've underscored this attribute.

#1870

Renamed the `alias` keyword arg of standalone `alias()` function to `name`

This so that the keyword argument `name` matches that of the `alias()` methods on all `FromClause` objects as well as the `name` argument on `Query.subquery()`.

Only code that uses the standalone `alias()` function, and not the method bound functions, and passes the alias name using the explicit keyword name `alias`, and not positionally, would need modification here.

Non-public `Pool` methods underscored

All methods of `Pool` and subclasses which are not intended for public use have been renamed with underscores. That they were not named this way previously was a bug.

Pooling methods now underscored or removed:

`Pool.create_connection()` -> `Pool._create_connection()`

`Pool.do_get()` -> `Pool._do_get()`

`Pool.do_return_conn()` -> `Pool._do_return_conn()`

`Pool.do_return_invalid()` -> removed, was not used

`Pool.return_conn()` -> `Pool._return_conn()`

`Pool.get()` -> `Pool._get()`, public API is `Pool.connect()`

`SingletonThreadPool.cleanup()` -> `_cleanup()`

`SingletonThreadPool.dispose_local()` -> removed, use `conn.invalidate()`

#1982

Previously Deprecated, Now Removed

Query.join(), Query.outerjoin(), eagerload(), eagerload_all(), others no longer allow lists of attributes as arguments

Passing a list of attributes or attribute names to `Query.join`, `eagerload()`, and similar has been deprecated since 0.5:

```
# old way, deprecated since 0.5
session.query(Houses).join([Houses.rooms, Room.closets])
session.query(Houses).options(eagerload_all([Houses.rooms, Room.closets]))
```

These methods all accept *args as of the 0.5 series:

```
# current way, in place since 0.5
session.query(Houses).join(Houses.rooms, Room.closets)
session.query(Houses).options(eagerload_all(Houses.rooms, Room.closets))
```

`ScopedSession.mapper` is removed

This feature provided a mapper extension which linked class- based functionality with a particular `ScopedSession`, in particular providing the behavior such that new object instances would be automatically associated with that session. The feature was overused by tutorials and frameworks which led to great user confusion due to its implicit behavior, and was deprecated in 0.5.5. Techniques for replicating its functionality are at [wiki:UsageRecipes/SessionAwareMapper]

6.3.5 What's New in SQLAlchemy 0.6?

About this Document

This document describes changes between SQLAlchemy version 0.5, last released January 16, 2010, and SQLAlchemy version 0.6, last released May 5, 2012.

Document date: June 6, 2010

This guide documents API changes which affect users migrating their applications from the 0.5 series of SQLAlchemy to 0.6. Note that SQLAlchemy 0.6 removes some behaviors which were deprecated throughout the span of the 0.5 series, and also deprecates more behaviors specific to 0.5.

Platform Support

- cPython versions 2.4 and upwards throughout the 2.xx series
- Jython 2.5.1 - using the zxJDBC DBAPI included with Jython.
- cPython 3.x - see [source:sqlalchemy/trunk/README.py3k] for information on how to build for python3.

New Dialect System

Dialect modules are now broken up into distinct subcomponents, within the scope of a single database backend. Dialect implementations are now in the `sqlalchemy.dialects` package. The `sqlalchemy.databases` package still exists as a placeholder to provide some level of backwards compatibility for simple imports.

For each supported database, a sub-package exists within `sqlalchemy.dialects` where several files are contained. Each package contains a module called `base.py` which defines the specific SQL dialect used by that database. It also contains one or more "driver" modules, each one corresponding to a specific DBAPI - these files are named corresponding to the DBAPI itself, such as `pysqlite`, `cx_oracle`, or `pyodbc`. The classes used by SQLAlchemy dialects are first declared in the `base.py` module, defining all behavioral characteristics defined by the database. These include capability mappings, such as "supports sequences", "supports returning", etc., type definitions, and SQL compilation rules. Each "driver" module in turn provides subclasses of those classes as needed which override the default behavior to accommodate the additional features, behaviors, and quirks of that DBAPI. For DBAPIs that support multiple backends (pyodbc, zxJDBC, mxODBC), the dialect module will use mixins from the `sqlalchemy.connectors` package, which provide functionality common to that DBAPI across all backends, most typically dealing with connect arguments. This means that connecting using pyodbc, zxJDBC or mxODBC (when implemented) is extremely consistent across supported backends.

The URL format used by `create_engine()` has been enhanced to handle any number of DBAPIs for a particular backend, using a scheme that is inspired by that of JDBC. The previous format still works, and will select a "default" DBAPI implementation, such as the Postgresql URL below that will use psycopg2:

```
create_engine('postgresql://scott:tiger@localhost/test')
```

However to specify a specific DBAPI backend such as pg8000, add it to the "protocol" section of the URL using a plus sign "+":

```
create_engine('postgresql+pg8000://scott:tiger@localhost/test')
```

Important Dialect Links:

- Documentation on connect arguments: http://www.sqlalchemy.org/docs/06/dbengine.html#create- engine-url-arguments.
- Reference documentation for individual dialects: http://ww w.sqlalchemy.org/docs/06/reference/dialects/index.html
- The tips and tricks at DatabaseNotes.

Other notes regarding dialects:

- the type system has been changed dramatically in SQLAlchemy 0.6. This has an impact on all dialects regarding naming conventions, behaviors, and implementations. See the section on "Types" below.
- the `ResultProxy` object now offers a 2x speed improvement in some cases thanks to some refactorings.
- the `RowProxy`, i.e. individual result row object, is now directly pickleable.
- the setuptools entrypoint used to locate external dialects is now called `sqlalchemy.dialects`. An external dialect written against 0.4 or 0.5 will need to be modified to work with 0.6 in any case so this change does not add any additional difficulties.
- dialects now receive an initialize() event on initial connection to determine connection properties.
- Functions and operators generated by the compiler now use (almost) regular dispatch functions of the form "visit_<opname>" and "visit_<funcname>_fn" to provide customed processing. This replaces the need to copy the "functions" and "operators" dictionaries in compiler subclasses with straightforward visitor methods, and also allows compiler subclasses complete control over rendering, as the full _Function or _BinaryExpression object is passed in.

Dialect Imports

The import structure of dialects has changed. Each dialect now exports its base "dialect" class as well as the full set of SQL types supported on that dialect via `sqlalchemy.dialects.<name>`. For example, to import a set of PG types:

```
from sqlalchemy.dialects.postgresql import INTEGER, BIGINT, SMALLINT,\
                                           VARCHAR, MACADDR, DATE, BYTEA
```

Above, `INTEGER` is actually the plain `INTEGER` type from `sqlalchemy.types`, but the PG dialect makes it available in the same way as those types which are specific to PG, such as `BYTEA` and `MACADDR`.

Expression Language Changes

An Important Expression Language Gotcha

There's one quite significant behavioral change to the expression language which may affect some applications. The boolean value of Python boolean expressions, i.e. `==`, `!=`, and similar, now evaluates accurately with regards to the two clause objects being compared.

As we know, comparing a `ClauseElement` to any other object returns another `ClauseElement`:

```
>>> from sqlalchemy.sql import column
>>> column('foo') == 5
<sqlalchemy.sql.expression._BinaryExpression object at 0x1252490>
```

This so that Python expressions produce SQL expressions when converted to strings:

```
>>> str(column('foo') == 5)
'foo = :foo_1'
```

But what happens if we say this?

```
>>> if column('foo') == 5:
...     print "yes"
...
```

In previous versions of SQLAlchemy, the returned `_BinaryExpression` was a plain Python object which evaluated to `True`. Now it evaluates to whether or not the actual `ClauseElement` should have the same hash value as to that being compared. Meaning:

```
>>> bool(column('foo') == 5)
False
>>> bool(column('foo') == column('foo'))
False
>>> c = column('foo')
>>> bool(c == c)
True
>>>
```

That means code such as the following:

```
if expression:
    print "the expression is:", expression
```

Would not evaluate if `expression` was a binary clause. Since the above pattern should never be used, the base `ClauseElement` now raises an exception if called in a boolean context:

```
>>> bool(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ...
    raise TypeError("Boolean value of this clause is not defined")
TypeError: Boolean value of this clause is not defined
```

Code that wants to check for the presence of a `ClauseElement` expression should instead say:

```
if expression is not None:
    print "the expression is:", expression
```

Keep in mind, **this applies to Table and Column objects too**.

The rationale for the change is twofold:

- Comparisons of the form `if c1 == c2:  <do something>` can actually be written now
- Support for correct hashing of `ClauseElement` objects now works on alternate platforms, namely Jython. Up until this point SQLAlchemy relied heavily on the specific behavior of cPython in this regard (and still had occasional problems with it).

Stricter "executemany" Behavior

An "executemany" in SQLAlchemy corresponds to a call to `execute()`, passing along a collection of bind parameter sets:

```
connection.execute(table.insert(), {'data':'row1'}, {'data':'row2'}, {'data':'row3'})
```

When the `Connection` object sends off the given `insert()` construct for compilation, it passes to the compiler the keynames present in the first set of binds passed along to determine the construction of the statement's VALUES clause. Users familiar with this construct will know that additional keys present in the remaining dictionaries don't have any impact. What's different now is that all subsequent dictionaries need to include at least *every* key that is present in the first dictionary. This means that a call like this no longer works:

```
connection.execute(table.insert(),
                        {'timestamp':today, 'data':'row1'},
                        {'timestamp':today, 'data':'row2'},
                        {'data':'row3'})
```

Because the third row does not specify the 'timestamp' column. Previous versions of SQLAlchemy would simply insert NULL for these missing columns. However, if the `timestamp` column in the above example contained a Python-side default value or function, it would *not* be used. This because the "executemany" operation is optimized for maximum performance across huge numbers of parameter sets, and does not attempt to evaluate Python-side defaults for those missing keys. Because defaults are often implemented either as SQL expressions which are embedded inline with the INSERT statement, or are server side expressions which again are triggered based on the structure of the INSERT string, which by definition cannot fire off conditionally based on each parameter set, it would be inconsistent for Python side defaults to behave differently vs. SQL/server side defaults. (SQL expression based defaults are embedded inline as of the 0.5 series, again to minimize the impact of huge numbers of parameter sets).

SQLAlchemy 0.6 therefore establishes predictable consistency by forbidding any subsequent parameter sets from leaving any fields blank. That way, there's no more silent failure of Python side default values and functions, which additionally are allowed to remain consistent in their behavior versus SQL and server side defaults.

UNION and other "compound" constructs parenthesize consistently

A rule that was designed to help SQLite has been removed, that of the first compound element within another compound (such as, a `union()` inside of an `except_()`) wouldn't be parenthesized. This is inconsistent and produces the wrong results on Postgresql, which has precedence rules regarding INTERSECTION, and its generally a surprise. When using complex composites with SQLite, you now need to turn the first element into a subquery (which is also compatible on PG). A new example is in the SQL expression tutorial at the end of [http://www.sqlalchemy.org/docs/06/sqlexpression.html #unions-and-other-set-operations]. See #1665 and r6690 for more background.

C Extensions for Result Fetching

The `ResultProxy` and related elements, including most common "row processing" functions such as unicode conversion, numerical/boolean conversions and date parsing, have been re-implemented as optional C extensions for the purposes of performance. This represents the beginning of SQLAlchemy's path to the "dark side" where we hope to continue improving performance by reimplementing critical sections in C. The extensions can be built by specifying `--with-cextensions`, i.e. `python setup.py --with- cextensions install`.

The extensions have the most dramatic impact on result fetching using direct `ResultProxy` access, i.e. that which is returned by `engine.execute()`, `connection.execute()`, or `session.execute()`. Within results returned by an ORM `Query` object, result fetching is not as high a percentage of overhead, so ORM performance improves more modestly, and mostly in the realm of fetching large result sets. The performance improvements highly depend on the dbapi in use and on the syntax used to access the columns of each row (eg `row['name']` is much faster than `row.name`). The current extensions have no impact on the speed of inserts/updates/deletes, nor do they improve the latency of SQL execution, that is, an application that spends most of its time executing many statements with very small result sets will not see much improvement.

Performance has been improved in 0.6 versus 0.5 regardless of the extensions. A quick overview of what connecting and fetching 50,000 rows looks like with SQLite, using mostly direct SQLite access, a `ResultProxy`, and a simple mapped ORM object:

```
sqlite select/native: 0.260s

0.6 / C extension

sqlalchemy.sql select: 0.360s
sqlalchemy.orm fetch: 2.500s

0.6 / Pure Python

sqlalchemy.sql select: 0.600s
sqlalchemy.orm fetch: 3.000s

0.5 / Pure Python

sqlalchemy.sql select: 0.790s
sqlalchemy.orm fetch: 4.030s
```

Above, the ORM fetches the rows 33% faster than 0.5 due to in-python performance enhancements. With the C extensions we get another 20%. However, `ResultProxy` fetches improve by 67% with the C extension versus not. Other tests report as much as a 200% speed improvement for some scenarios, such as those where lots of string conversions are occurring.

New Schema Capabilities

The `sqlalchemy.schema` package has received some long- needed attention. The most visible change is the newly expanded DDL system. In SQLAlchemy, it was possible since version 0.5 to create custom DDL strings and associate them with tables or metadata objects:

```
from sqlalchemy.schema import DDL

DDL('CREATE TRIGGER users_trigger ...').execute_at('after-create', metadata)
```

Now the full suite of DDL constructs are available under the same system, including those for CREATE TABLE, ADD CONSTRAINT, etc.:

```
from sqlalchemy.schema import Constraint, AddConstraint

AddContraint(CheckConstraint("value > 5")).execute_at('after-create', mytable)
```

Additionally, all the DDL objects are now regular `ClauseElement` objects just like any other SQLAlchemy expression object:

```
from sqlalchemy.schema import CreateTable

create = CreateTable(mytable)

# dumps the CREATE TABLE as a string
print create

# executes the CREATE TABLE statement
engine.execute(create)
```

and using the `sqlalchemy.ext.compiler` extension you can make your own:

```
from sqlalchemy.schema import DDLElement
from sqlalchemy.ext.compiler import compiles

class AlterColumn(DDLElement):

    def __init__(self, column, cmd):
        self.column = column
        self.cmd = cmd

@compiles(AlterColumn)
def visit_alter_column(element, compiler, **kw):
    return "ALTER TABLE %s ALTER COLUMN %s %s ..." % (
        element.column.table.name,
        element.column.name,
        element.cmd
    )

engine.execute(AlterColumn(table.c.mycolumn, "SET DEFAULT 'test'"))
```

Deprecated/Removed Schema Elements

The schema package has also been greatly streamlined. Many options and methods which were deprecated throughout 0.5 have been removed. Other little known accessors and methods have also been removed.

- the "owner" keyword argument is removed from `Table`. Use "schema" to represent any namespaces to be prepended to the table name.
- deprecated `MetaData.connect()` and `ThreadLocalMetaData.connect()` have been removed - send the "bind" attribute to bind a metadata.
- deprecated metadata.table_iterator() method removed (use sorted_tables)
- the "metadata" argument is removed from `DefaultGenerator` and subclasses, but remains locally present on `Sequence`, which is a standalone construct in DDL.
- deprecated `PassiveDefault` - use `DefaultClause`.
- Removed public mutability from `Index` and `Constraint` objects:
  - `ForeignKeyConstraint.append_element()`
  - `Index.append_column()`
  - `UniqueConstraint.append_column()`
  - `PrimaryKeyConstraint.add()`
  - `PrimaryKeyConstraint.remove()`

These should be constructed declaratively (i.e. in one construction).

- Other removed things:
  - `Table.key` (no idea what this was for)
  - `Column.bind` (get via column.table.bind)
  - `Column.metadata` (get via column.table.metadata)
  - `Column.sequence` (use column.default)

Other Behavioral Changes

- `UniqueConstraint`, `Index`, `PrimaryKeyConstraint` all accept lists of column names or column objects as arguments.
- The `use_alter` flag on `ForeignKey` is now a shortcut option for operations that can be hand-constructed using the `DDL()` event system. A side effect of this refactor is that `ForeignKeyConstraint` objects with `use_alter=True` will *not* be emitted on SQLite, which does not support ALTER for foreign keys. This has no effect on SQLite's behavior since SQLite does not actually honor FOREIGN KEY constraints.
- `Table.primary_key` is not assignable - use `table.append_constraint(PrimaryKeyConstraint(...))`
- A `Column` definition with a `ForeignKey` and no type, e.g. `Column(name, ForeignKey(sometable.c.somecol))` used to get the type of the referenced column. Now support for that automatic type inference is partial and may not work in all cases.

Logging opened up

At the expense of a few extra method calls here and there, you can set log levels for INFO and DEBUG after an engine, pool, or mapper has been created, and logging will commence. The `isEnabledFor(INFO)` method is now called per-`Connection` and `isEnabledFor(DEBUG)` per-`ResultProxy` if already enabled on the parent connection. Pool logging sends to `log.info()` and `log.debug()` with no check - note that pool checkout/checkin is typically once per transaction.

Reflection/Inspector API

The reflection system, which allows reflection of table columns via `Table('sometable', metadata, autoload=True)` has been opened up into its own fine-grained API, which allows direct inspection of database elements such as tables, columns, constraints, indexes, and more. This API expresses return values as simple lists of strings, dictionaries, and `TypeEngine` objects. The internals of `autoload=True` now build upon this system such that the translation of raw database information into `sqlalchemy.schema` constructs is centralized and the contract of individual dialects greatly simplified, vastly reducing bugs and inconsistencies across different backends.

To use an inspector:

```
from sqlalchemy.engine.reflection import Inspector
insp = Inspector.from_engine(my_engine)

print insp.get_schema_names()
```

the `from_engine()` method will in some cases provide a backend-specific inspector with additional capabilities, such as that of Postgresql which provides a `get_table_oid()` method:

```
my_engine = create_engine('postgresql://...')
pg_insp = Inspector.from_engine(my_engine)

print pg_insp.get_table_oid('my_table')
```

RETURNING Support

The `insert()`, `update()` and `delete()` constructs now support a `returning()` method, which corresponds to the SQL RETURNING clause as supported by Postgresql, Oracle, MS-SQL, and Firebird. It is not supported for any other backend at this time.

Given a list of column expressions in the same manner as that of a `select()` construct, the values of these columns will be returned as a regular result set:

```
result = connection.execute(
            table.insert().values(data='some data').returning(table.c.id, table.c.timestamp)
        )
row = result.first()
print "ID:", row['id'], "Timestamp:", row['timestamp']
```

The implementation of RETURNING across the four supported backends varies wildly, in the case of Oracle requiring an intricate usage of OUT parameters which are re-routed into a "mock" result set, and in the case of MS-SQL using an awkward SQL syntax. The usage of RETURNING is subject to limitations:

- it does not work for any "executemany()" style of execution. This is a limitation of all supported DBAPIs.
- Some backends, such as Oracle, only support RETURNING that returns a single row - this includes UPDATE and DELETE statements, meaning the update() or delete() construct must match only a single row, or an error is raised (by Oracle, not SQLAlchemy).

RETURNING is also used automatically by SQLAlchemy, when available and when not otherwise specified by an explicit `returning()` call, to fetch the value of newly generated primary key values for single-row INSERT statements. This means there's no more "SELECT nextval(sequence)" pre- execution for insert statements where the primary key value is required. Truth be told, implicit RETURNING feature does incur more method overhead than the old "select nextval()" system, which used a quick and dirty cursor.execute() to get at the sequence value, and in the case of Oracle requires additional binding of out parameters. So if method/protocol overhead is proving to be more expensive than additional database round trips, the feature can be disabled by specifying `implicit_returning=False` to `create_engine()`.

Type System Changes

New Archicture

The type system has been completely reworked behind the scenes to provide two goals:

- Separate the handling of bind parameters and result row values, typically a DBAPI requirement, from the SQL specification of the type itself, which is a database requirement. This is consistent with the overall dialect refactor that separates database SQL behavior from DBAPI.
- Establish a clear and consistent contract for generating DDL from a `TypeEngine` object and for constructing `TypeEngine` objects based on column reflection.

Highlights of these changes include:

- The construction of types within dialects has been totally overhauled. Dialects now define publically available types as UPPERCASE names exclusively, and internal implementation types using underscore identifiers (i.e. are private). The system by which types are expressed in SQL and DDL has been moved to the compiler system. This has the effect that there are much fewer type objects within most dialects. A detailed document on this architecture for dialect authors is in [source:/lib/sqlalc hemy/dialects/type_migration_guidelines.txt].
- Reflection of types now returns the exact UPPERCASE type within types.py, or the UPPERCASE type within the dialect itself if the type is not a standard SQL type. This means reflection now returns more accurate information about reflected types.
- User defined types that subclass `TypeEngine` and wish to provide `get_col_spec()` should now subclass `UserDefinedType`.
- The `result_processor()` method on all type classes now accepts an additional argument `coltype`. This is the DBAPI type object attached to cursor.description, and should be used when applicable to make better decisions on what kind of result-processing callable should be returned. Ideally result processor functions would never need to use `isinstance()`, which is an expensive call at this level.

Native Unicode Mode

As more DBAPIs support returning Python unicode objects directly, the base dialect now performs a check upon the first connection which establishes whether or not the DBAPI returns a Python unicode object for a basic select of a VARCHAR value. If so, the `String` type and all subclasses (i.e. `Text`, `Unicode`, etc.) will skip the "unicode" check/conversion step when result rows are received. This offers a dramatic performance increase for large result sets. The "unicode mode" currently is known to work with:

- sqlite3 / pysqlite
- psycopg2 - SQLA 0.6 now uses the "UNICODE" type extension by default on each psycopg2 connection object
- pg8000
- cx_oracle (we use an output processor - nice feature !)

Other types may choose to disable unicode processing as needed, such as the `NVARCHAR` type when used with MS-SQL.

In particular, if porting an application based on a DBAPI that formerly returned non-unicode strings, the "native unicode" mode has a plainly different default behavior - columns that are declared as `String` or `VARCHAR` now return unicode by default whereas they would return strings before. This can break code which expects non-unicode strings. The psycopg2 "native unicode" mode can be disabled by passing `use_native_unicode=False` to `create_engine()`.

A more general solution for string columns that explicitly do not want a unicode object is to use a `TypeDecorator` that converts unicode back to utf-8, or whatever is desired:

```
class UTF8Encoded(TypeDecorator):
    """Unicode type which coerces to utf-8."""

    impl = sa.VARCHAR

    def process_result_value(self, value, dialect):
        if isinstance(value, unicode):
            value = value.encode('utf-8')
        return value
```

Note that the `assert_unicode` flag is now deprecated. SQLAlchemy allows the DBAPI and backend database in use to handle Unicode parameters when available, and does not add operational overhead by checking the incoming type; modern systems like sqlite and Postgresql will raise an encoding error on their end if invalid data is passed. In those cases where SQLAlchemy does need to coerce a bind parameter from Python Unicode to an encoded string, or when the Unicode type is used explicitly, a warning is raised if the object is a bytestring. This warning can be suppressed or converted to an exception using the Python warnings filter documented at: http://docs.python.org/library/warnings.html

Generic Enum Type

We now have an `Enum` in the `types` module. This is a string type that is given a collection of "labels" which constrain the possible values given to those labels. By default, this type generates a `VARCHAR` using the size of the largest label, and applies a CHECK constraint to the table within the CREATE TABLE statement. When using MySQL, the type by default uses MySQL's ENUM type, and when using Postgresql the type will generate a user defined type using `CREATE TYPE <mytype> AS ENUM`. In order to create the type using Postgresql, the `name` parameter must be specified to the constructor. The type also accepts a `native_enum=False` option which will issue the VARCHAR/CHECK strategy for all databases. Note that Postgresql ENUM types currently don't work with pg8000 or zxjdbc.

Reflection Returns Dialect-Specific Types

Reflection now returns the most specific type possible from the database. That is, if you create a table using `String`, then reflect it back, the reflected column will likely be `VARCHAR`. For dialects that support a more specific form of the type, that's what you'll get. So a `Text` type would come back as `oracle.CLOB` on Oracle, a `LargeBinary` might be an `mysql.MEDIUMBLOB` etc. The obvious advantage here is that reflection preserves as much information possible from what the database had to say.

Some applications that deal heavily in table metadata may wish to compare types across reflected tables and/or non-reflected tables. There's a semi-private accessor available on `TypeEngine` called `_type_affinity` and an associated comparison helper `_compare_type_affinity`. This accessor returns the "generic" `types` class which the type corresponds to:

```
>>> String(50)._compare_type_affinity(postgresql.VARCHAR(50))
True
>>> Integer()._compare_type_affinity(mysql.REAL)
False
```

Miscellaneous API Changes

The usual "generic" types are still the general system in use, i.e. `String`, `Float`, `DateTime`. There's a few changes there:

- Types no longer make any guesses as to default parameters. In particular, `Numeric`, `Float`, as well as subclasses NUMERIC, FLOAT, DECIMAL don't generate any length or scale unless specified. This also continues to include the controversial `String` and `VARCHAR` types (although MySQL dialect will pre-emptively raise when asked to render VARCHAR with no length). No defaults are assumed, and if they are used in a CREATE TABLE statement, an error will be raised if the underlying database does not allow non-lengthed versions of these types.
- the `Binary` type has been renamed to `LargeBinary`, for BLOB/BYTEA/similar types. For `BINARY` and `VARBINARY`, those are present directly as `types.BINARY`, `types.VARBINARY`, as well as in the MySQL and MS-SQL dialects.
- `PickleType` now uses == for comparison of values when mutable=True, unless the "comparator" argument with a comparison function is specified to the type. If you are pickling a custom object you should implement an `__eq__()` method so that value-based comparisons are accurate.
- The default "precision" and "scale" arguments of Numeric and Float have been removed and now default to None. NUMERIC and FLOAT will be rendered with no numeric arguments by default unless these values are provided.
- DATE, TIME and DATETIME types on SQLite can now take optional "storage_format" and "regexp" argument. "storage_format" can be used to store those types using a custom string format. "regexp" allows to use a custom regular expression to match string values from the database.
- `__legacy_microseconds__` on SQLite `Time` and `DateTime` types is not supported anymore. You should use the new "storage_format" argument instead.
- `DateTime` types on SQLite now use by a default a stricter regular expression to match strings from the database. Use the new "regexp" argument if you are using data stored in a legacy format.

ORM Changes

Upgrading an ORM application from 0.5 to 0.6 should require little to no changes, as the ORM's behavior remains almost identical. There are some default argument and name changes, and some loading behaviors have been improved.

New Unit of Work

The internals for the unit of work, primarily `topological.py` and `unitofwork.py`, have been completely rewritten and are vastly simplified. This should have no impact on usage, as all existing behavior during flush has been maintained exactly (or at least, as far as it is exercised by our testsuite and the handful of production environments which have tested it heavily). The performance of flush() now uses 20-30% fewer method calls and should also use less memory. The intent and flow of the source code should now be reasonably easy to follow, and the architecture of the flush is fairly open-ended at this point, creating room for potential new areas of sophistication. The flush process no longer has any reliance on recursion so flush plans of arbitrary size and complexity can be flushed. Additionally, the mapper's "save" process, which issues INSERT and UPDATE statements, now caches the "compiled" form of the two statements so that callcounts are further dramatically reduced with very large flushes.

Any changes in behavior observed with flush versus earlier versions of 0.6 or 0.5 should be reported to us ASAP - we'll make sure no functionality is lost.

Changes to `query.update()` and `query.delete()`

- the 'expire' option on query.update() has been renamed to 'fetch', thus matching that of query.delete()
- `query.update()` and `query.delete()` both default to 'evaluate' for the synchronize strategy.

- the 'synchronize' strategy for update() and delete() raises an error on failure. There is no implicit fallback onto "fetch". Failure of evaluation is based on the structure of criteria, so success/failure is deterministic based on code structure.

`relation()` is officially named `relationship()`

This to solve the long running issue that "relation" means a "table or derived table" in relational algebra terms. The `relation()` name, which is less typing, will hang around for the foreseeable future so this change should be entirely painless.

Subquery eager loading

A new kind of eager loading is added called "subquery" loading. This is a load that emits a second SQL query immediately after the first which loads full collections for all the parents in the first query, joining upwards to the parent using INNER JOIN. Subquery loading is used simlarly to the current joined-eager loading, using the `'subqueryload()`` and ``subqueryload_all()`` options as well as the ``lazy='subquery'`` setting on ``relationship()`. The subquery load is usually much more efficient for loading many larger collections as it uses INNER JOIN unconditionally and also doesn't re-load parent rows.

`'eagerload()``, ``eagerload_all()`` is now ``joinedload()``, ``joinedload_all()`

To make room for the new subquery load feature, the existing `'eagerload()``/``eagerload_all()`` options are now superseded by ``joinedload()`` and ``joinedload_all()``. The old names will hang around for the foreseeable future just like ``relation()`.

`'lazy=False|None|True|'dynamic'`` now accepts ``lazy='noload'|'joined'|'subquery'|'select'|'dynamic'`

Continuing on the theme of loader strategies opened up, the standard keywords for the `'lazy`` option on ``relationship()`` are now ``select`` for lazy loading (via a SELECT issued on attribute access), ``joined`` for joined-eager loading, ``subquery`` for subquery-eager loading, ``noload`` for no loading should occur, and ``dynamic`` for a "dynamic" relationship. The old ``True``, ``False``, ``None`` arguments are still accepted with the identical behavior as before.

innerjoin=True on relation, joinedload

Joined-eagerly loaded scalars and collections can now be instructed to use INNER JOIN instead of OUTER JOIN. On Postgresql this is observed to provide a 300-600% speedup on some queries. Set this flag for any many-to-one which is on a NOT NULLable foreign key, and similarly for any collection where related items are guaranteed to exist.

At mapper level:

```
mapper(Child, child)
mapper(Parent, parent, properties={
    'child':relationship(Child, lazy='joined', innerjoin=True)
})
```

At query time level:

```
session.query(Parent).options(joinedload(Parent.child, innerjoin=True)).all()
```

The `innerjoin=True` flag at the `relationship()` level will also take effect for any `joinedload()` option which does not override the value.

Many-to-one Enhancements

- many-to-one relations now fire off a lazyload in fewer cases, including in most cases will not fetch the "old" value when a new one is replaced.
- many-to-one relation to a joined-table subclass now uses get() for a simple load (known as the "use_get" condition), i.e. `Related`->''Sub(Base)'', without the need to redefine the primaryjoin condition in terms of the base table. [ticket:1186]
- specifying a foreign key with a declarative column, i.e. `ForeignKey(MyRelatedClass.id)` doesn't break the "use_get" condition from taking place [ticket:1492]
- relationship(), joinedload(), and joinedload_all() now feature an option called "innerjoin". Specify `True` or `False` to control whether an eager join is constructed as an INNER or OUTER join. Default is `False` as always. The mapper options will override whichever setting is specified on relationship(). Should generally be set for many-to-one, not nullable foreign key relations to allow improved join performance. [ticket:1544]
- the behavior of joined eager loading such that the main query is wrapped in a subquery when LIMIT/OFFSET are present now makes an exception for the case when all eager loads are many-to-one joins. In those cases, the eager joins are against the parent table directly along with the limit/offset without the extra overhead of a subquery, since a many-to-one join does not add rows to the result.

  For example, in 0.5 this query:

```
session.query(Address).options(eagerload(Address.user)).limit(10)
```

  would produce SQL like:

```
SELECT * FROM
  (SELECT * FROM addresses LIMIT 10) AS anon_1
  LEFT OUTER JOIN users AS users_1 ON users_1.id = anon_1.addresses_user_id
```

  This because the presence of any eager loaders suggests that some or all of them may relate to multi-row collections, which would necessitate wrapping any kind of rowcount-sensitive modifiers like LIMIT inside of a subquery.

  In 0.6, that logic is more sensitive and can detect if all eager loaders represent many-to-ones, in which case the eager joins don't affect the rowcount:

```
SELECT * FROM addresses LEFT OUTER JOIN users AS users_1 ON users_1.id = addresses.user_id LIMIT
```

Mutable Primary Keys with Joined Table Inheritance

A joined table inheritance config where the child table has a PK that foreign keys to the parent PK can now be updated on a CASCADE-capable database like Postgresql. `mapper()` now has an option `passive_updates=True` which indicates this foreign key is updated automatically. If on a non-cascading database like SQLite or MySQL/MyISAM, set this flag to `False`. A future feature enhancement will try to get this flag to be auto-configuring based on dialect/table style in use.

Beaker Caching

A promising new example of Beaker integration is in `examples/beaker_caching`. This is a straightforward recipe which applies a Beaker cache within the result- generation engine of `Query`. Cache parameters are provided via `query.options()`, and allows full control over the contents of the cache. SQLAlchemy 0.6 includes improvements to the `Session.merge()` method to support this and similar recipes, as well as to provide significantly improved performance in most scenarios.

Other Changes

- the "row tuple" object returned by `Query` when multiple column/entities are selected is now picklable as well as higher performing.
- `query.join()` has been reworked to provide more consistent behavior and more flexibility (includes [ticket:1537])
- `query.select_from()` accepts multiple clauses to produce multiple comma separated entries within the FROM clause. Useful when selecting from multiple-homed join() clauses.
- the "dont_load=True" flag on `Session.merge()` is deprecated and is now "load=False".
- added "make_transient()" helper function which transforms a persistent/ detached instance into a transient one (i.e. deletes the instance_key and removes from any session.) [ticket:1052]
- the allow_null_pks flag on mapper() is deprecated and has been renamed to allow_partial_pks. It is turned "on" by default. This means that a row which has a non-null value for any of its primary key columns will be considered an identity. The need for this scenario typically only occurs when mapping to an outer join. When set to False, a PK that has NULLs in it will not be considered a primary key - in particular this means a result row will come back as None (or not be filled into a collection), and new in 0.6 also indicates that session.merge() won't issue a round trip to the database for such a PK value. [ticket:1680]
- the mechanics of "backref" have been fully merged into the finer grained "back_populates" system, and take place entirely within the `_generate_backref()` method of `RelationProperty`. This makes the initialization procedure of `RelationProperty` simpler and allows easier propagation of settings (such as from subclasses of `RelationProperty`) into the reverse reference. The internal `BackRef()` is gone and `backref()` returns a plain tuple that is understood by `RelationProperty`.
- the keys attribute of `ResultProxy` is now a method, so references to it (`result.keys`) must be changed to method invocations (`result.keys()`)
- `ResultProxy.last_inserted_ids` is now deprecated, use `ResultProxy.inserted_primary_key` instead.

Deprecated/Removed ORM Elements

Most elements that were deprecated throughout 0.5 and raised deprecation warnings have been removed (with a few exceptions). All elements that were marked "pending deprecation" are now deprecated and will raise a warning upon use.

- 'transactional' flag on sessionmaker() and others is removed. Use 'autocommit=True' to indicate 'transactional=False'.
- 'polymorphic_fetch' argument on mapper() is removed. Loading can be controlled using the 'with_polymorphic' option.
- 'select_table' argument on mapper() is removed. Use 'with_polymorphic=("*", <some selectable>)' for this functionality.

- 'proxy' argument on synonym() is removed. This flag did nothing throughout 0.5, as the "proxy generation" behavior is now automatic.
- Passing a single list of elements to joinedload(), joinedload_all(), contains_eager(), lazyload(), defer(), and undefer() instead of multiple positional *args is deprecated.
- Passing a single list of elements to query.order_by(), query.group_by(), query.join(), or query.outerjoin() instead of multiple positional *args is deprecated.
- `query.iterate_instances()` is removed. Use `query.instances()`.
- `Query.query_from_parent()` is removed. Use the sqlalchemy.orm.with_parent() function to produce a "parent" clause, or alternatively `query.with_parent()`.
- `query._from_self()` is removed, use `query.from_self()` instead.
- the "comparator" argument to composite() is removed. Use "comparator_factory".
- `RelationProperty._get_join()` is removed.
- the 'echo_uow' flag on Session is removed. Use logging on the "sqlalchemy.orm.unitofwork" name.
- `session.clear()` is removed. use `session.expunge_all()`.
- `session.save()`, `session.update()`, `session.save_or_update()` are removed. Use `session.add()` and `session.add_all()`.
- the "objects" flag on session.flush() remains deprecated.
- the "dont_load=True" flag on session.merge() is deprecated in favor of "load=False".
- `ScopedSession.mapper` remains deprecated. See the usage recipe at http://www.sqlalchemy.org/trac/wiki/Usag eRecipes/SessionAwareMapper
- passing an `InstanceState` (internal SQLAlchemy state object) to `attributes.init_collection()` or `attributes.get_history()` is deprecated. These functions are public API and normally expect a regular mapped object instance.
- the 'engine' parameter to `declarative_base()` is removed. Use the 'bind' keyword argument.

Extensions

SQLSoup

SQLSoup has been modernized and updated to reflect common 0.5/0.6 capabilities, including well defined session integration. Please read the new docs at [http://www.sqlalc hemy.org/docs/06/reference/ext/sqlsoup.html].

Declarative

The `DeclarativeMeta` (default metaclass for `declarative_base`) previously allowed subclasses to modify `dict_` to add class attributes (e.g. columns). This no longer works, the `DeclarativeMeta` constructor now ignores `dict_`. Instead, the class attributes should be assigned directly, e.g. `cls.id=Column(...)`, or the MixIn class approach should be used instead of the metaclass approach.

6.3.6 What's new in SQLAlchemy 0.5?

About this Document

This document describes changes between SQLAlchemy version 0.4, last released October 12, 2008, and SQLAlchemy version 0.5, last released January 16, 2010.

Document date: August 4, 2009

This guide documents API changes which affect users migrating their applications from the 0.4 series of SQLAlchemy to 0.5. It's also recommended for those working from Essential SQLAlchemy, which only covers 0.4 and seems to even have some old 0.3isms in it. Note that SQLAlchemy 0.5 removes many behaviors which were deprecated throughout the span of the 0.4 series, and also deprecates more behaviors specific to 0.4.

Major Documentation Changes

Some sections of the documentation have been completely rewritten and can serve as an introduction to new ORM features. The `Query` and `Session` objects in particular have some distinct differences in API and behavior which fundamentally change many of the basic ways things are done, particularly with regards to constructing highly customized ORM queries and dealing with stale session state, commits and rollbacks.

- ORM Tutorial
- Session Documentation

Deprecations Source

Another source of information is documented within a series of unit tests illustrating up to date usages of some common `Query` patterns; this file can be viewed at [source:sqlalchemy/trunk/test/orm/test_deprecations.py].

Requirements Changes

- Python 2.4 or higher is required. The SQLAlchemy 0.4 line is the last version with Python 2.3 support.

Object Relational Mapping

- **Column level expressions within Query.** - as detailed in the tutorial, `Query` has the capability to create specific SELECT statements, not just those against full rows:

```
session.query(User.name, func.count(Address.id).label("numaddresses")).join(Address).group_by(Us
```

  The tuples returned by any multi-column/entity query are *named*‘ tuples:

```
for row in session.query(User.name, func.count(Address.id).label('numaddresses')).join(Address).
   print "name", row.name, "number", row.numaddresses
```

  `Query` has a `statement` accessor, as well as a `subquery()` method which allow `Query` to be used to create more complex combinations:

```
subq = session.query(Keyword.id.label('keyword_id')).filter(Keyword.name.in_(['beans', 'carrots'
recipes = session.query(Recipe).filter(exists().
   where(Recipe.id==recipe_keywords.c.recipe_id).
   where(recipe_keywords.c.keyword_id==subq.c.keyword_id)
)
```

- **Explicit ORM aliases are recommended for aliased joins** - The `aliased()` function produces an "alias" of a class, which allows fine-grained control of aliases in conjunction with ORM queries. While a table-level alias (i.e. `table.alias()`) is still usable, an ORM level alias retains the semantics of the ORM mapped object which is significant for inheritance mappings, options, and other scenarios. E.g.:

```
Friend = aliased(Person)
session.query(Person, Friend).join((Friend, Person.friends)).all()
```

- **query.join() greatly enhanced.** - You can now specify the target and ON clause for a join in multiple ways. A target class alone can be provided where SQLA will attempt to form a join to it via foreign key in the same way as `table.join(someothertable)`. A target and an explicit ON condition can be provided, where the ON condition can be a `relation()` name, an actual class descriptor, or a SQL expression. Or the old way of just a `relation()` name or class descriptor works too. See the ORM tutorial which has several examples.
- **Declarative is recommended for applications which don't require (and don't prefer) abstraction between tables and mappers** - The [/docs/05/reference/ext/declarative.html Declarative] module, which is used to combine the expression of `Table`, `mapper()`, and user defined class objects together, is highly recommended as it simplifies application configuration, ensures the "one mapper per class" pattern, and allows the full range of configuration available to distinct `mapper()` calls. Separate `mapper()` and `Table` usage is now referred to as "classical SQLAlchemy usage" and of course is freely mixable with declarative.
- **The .c. attribute has been removed** from classes (i.e. `MyClass.c.somecolumn`). As is the case in 0.4, class- level properties are usable as query elements, i.e. `Class.c.propname` is now superseded by `Class.propname`, and the `c` attribute continues to remain on `Table` objects where they indicate the namespace of `Column` objects present on the table.

  To get at the Table for a mapped class (if you didn't keep it around already):

```
table = class_mapper(someclass).mapped_table
```

  Iterate through columns:

```
for col in table.c:
    print col
```

  Work with a specific column:

```
table.c.somecolumn
```

  The class-bound descriptors support the full set of Column operators as well as the documented relation-oriented operators like `has()`, `any()`, `contains()`, etc.

  The reason for the hard removal of `.c.` is that in 0.5, class-bound descriptors carry potentially different meaning, as well as information regarding class mappings, versus plain `Column` objects - and there are use cases where you'd specifically want to use one or the other. Generally, using class-bound descriptors invokes a set of mapping/polymorphic aware translations, and using table- bound columns does not. In 0.4, these translations were applied across the board to all expressions, but 0.5 differentiates completely between columns and mapped descriptors, only applying translations to the latter. So in many cases, particularly when dealing with joined table inheritance configurations as well as when using `query(<columns>)`, `Class.propname` and `table.c.colname` are not interchangeable.

  For example, `session.query(users.c.id, users.c.name)` is different versus `session.query(User.id, User.name)`; in the latter case, the `Query` is aware of the mapper in use and further mapper-specific operations like `query.join(<propname>)`, `query.with_parent()` etc. may be used, but in the former case cannot. Additionally, in polymorphic inheritance scenarios, the class-bound descriptors refer to the columns present in the polymorphic selectable in use, not necessarily the table column

which directly corresponds to the descriptor. For example, a set of classes related by joined-table inheritance to the `person` table along the `person_id` column of each table will all have their `Class.person_id` attribute mapped to the `person_id` column in `person`, and not their subclass table. Version 0.4 would map this behavior onto table-bound `Column` objects automatically. In 0.5, this automatic conversion has been removed, so that you in fact *can* use table-bound columns as a means to override the translations which occur with polymorphic querying; this allows `Query` to be able to create optimized selects among joined-table or concrete-table inheritance setups, as well as portable subqueries, etc.

- **Session Now Synchronizes Automatically with Transactions.** Session now synchronizes against the transaction automatically by default, including autoflush and autoexpire. A transaction is present at all times unless disabled using the `autocommit` option. When all three flags are set to their default, the Session recovers gracefully after rollbacks and it's very difficult to get stale data into the session. See the new Session documentation for details.
- **Implicit Order By Is Removed**. This will impact ORM users who rely upon SA's "implicit ordering" behavior, which states that all Query objects which don't have an `order_by()` will ORDER BY the "id" or "oid" column of the primary mapped table, and all lazy/eagerly loaded collections apply a similar ordering. In 0.5, automatic ordering must be explicitly configured on `mapper()` and `relation()` objects (if desired), or otherwise when using `Query`.

  To convert an 0.4 mapping to 0.5, such that its ordering behavior will be extremely similar to 0.4 or previous, use the `order_by` setting on `mapper()` and `relation()`:

```
mapper(User, users, properties={
    'addresses':relation(Address, order_by=addresses.c.id)
}, order_by=users.c.id)
```

  To set ordering on a backref, use the `backref()` function:

```
'keywords':relation(Keyword, secondary=item_keywords,
      order_by=keywords.c.name, backref=backref('items', order_by=items.c.id))
```

  Using declarative ? To help with the new `order_by` requirement, `order_by` and friends can now be set using strings which are evaluated in Python later on (this works **only** with declarative, not plain mappers):

```
class MyClass(MyDeclarativeBase):
    ...
    'addresses':relation("Address", order_by="Address.id")
```

  It's generally a good idea to set `order_by` on `relation()s` which load list-based collections of items, since that ordering cannot otherwise be affected. Other than that, the best practice is to use `Query.order_by()` to control ordering of the primary entities being loaded.

- **Session is now autoflush=True/autoexpire=True/autocommit=False.** - To set it up, just call `sessionmaker()` with no arguments. The name `transactional=True` is now `autocommit=False`. Flushes occur upon each query issued (disable with `autoflush=False`), within each `commit()` (as always), and before each `begin_nested()` (so rolling back to the SAVEPOINT is meaningful). All objects are expired after each `commit()` and after each `rollback()`. After rollback, pending objects are expunged, deleted objects move back to persistent. These defaults work together very nicely and there's really no more need for old techniques like `clear()` (which is renamed to `expunge_all()` as well).

  P.S.: sessions are now reusable after a `rollback()`. Scalar and collection attribute changes, adds and deletes are all rolled back.

- **session.add() replaces session.save(), session.update(), session.save_or_update().** - the `session.add(someitem)` and `session.add_all([list of items])` methods replace `save()`, `update()`, and `save_or_update()`. Those methods will remain deprecated throughout 0.5.

- **backref configuration made less verbose.** - The `backref()` function now uses the `primaryjoin` and `secondaryjoin` arguments of the forwards-facing `relation()` when they are not explicitly stated. It's no longer necessary to specify `primaryjoin`/`secondaryjoin` in both directions separately.
- **Simplified polymorphic options.** - The ORM's "polymorphic load" behavior has been simplified. In 0.4, mapper() had an argument called `polymorphic_fetch` which could be configured as `select` or `deferred`. This option is removed; the mapper will now just defer any columns which were not present in the SELECT statement. The actual SELECT statement used is controlled by the `with_polymorphic` mapper argument (which is also in 0.4 and replaces `select_table`), as well as the `with_polymorphic()` method on `Query` (also in 0.4).

  An improvement to the deferred loading of inheriting classes is that the mapper now produces the "optimized" version of the SELECT statement in all cases; that is, if class B inherits from A, and several attributes only present on class B have been expired, the refresh operation will only include B's table in the SELECT statement and will not JOIN to A.
- The `execute()` method on `Session` converts plain strings into `text()` constructs, so that bind parameters may all be specified as ":bindname" without needing to call `text()` explicitly. If "raw" SQL is desired here, use `session.connection().execute("raw text")`.
- `session.Query().iterate_instances()` has been renamed to just `instances()`. The old `instances()` method returning a list instead of an iterator no longer exists. If you were relying on that behavior, you should use `list(your_query.instances())`.

Extending the ORM

In 0.5 we're moving forward with more ways to modify and extend the ORM. Heres a summary:

- **MapperExtension.** - This is the classic extension class, which remains. Methods which should rarely be needed are `create_instance()` and `populate_instance()`. To control the initialization of an object when it's loaded from the database, use the `reconstruct_instance()` method, or more easily the `@reconstructor` decorator described in the documentation.
- **SessionExtension.** - This is an easy to use extension class for session events. In particular, it provides `before_flush()`, `after_flush()` and `after_flush_postexec()` methods. This usage is recommended over `MapperExtension.before_XXX` in many cases since within `before_flush()` you can modify the flush plan of the session freely, something which cannot be done from within `MapperExtension`.
- **AttributeExtension.** - This class is now part of the public API, and allows the interception of userland events on attributes, including attribute set and delete operations, and collection appends and removes. It also allows the value to be set or appended to be modified. The `@validates` decorator, described in the documentation, provides a quick way to mark any mapped attributes as being "validated" by a particular class method.
- **Attribute Instrumentation Customization.** - An API is provided for ambitious efforts to entirely replace SQLAlchemy's attribute instrumentation, or just to augment it in some cases. This API was produced for the purposes of the Trellis toolkit, but is available as a public API. Some examples are provided in the distribution in the `/examples/custom_attributes` directory.

Schema/Types

- **String with no length no longer generates TEXT, it generates VARCHAR** - The `String` type no longer magically converts into a `Text` type when specified with no length. This only has an effect when CREATE TABLE is issued, as it will issue `VARCHAR` with no length parameter, which is not valid on many (but not all) databases. To create a TEXT (or CLOB, i.e. unbounded string) column, use the `Text` type.
- **PickleType() with mutable=True requires an __eq__() method** - The `PickleType` type needs to compare values when mutable=True. The method of comparing `pickle.dumps()` is inefficient and unreliable. If an

incoming object does not implement `__eq__()` and is also not `None`, the `dumps()` comparison is used but a warning is raised. For types which implement `__eq__()` which includes all dictionaries, lists, etc., comparison will use == and is now reliable by default.

- **convert_bind_param() and convert_result_value() methods of TypeEngine/TypeDecorator are removed.** - The O'Reilly book unfortunately documented these methods even though they were deprecated post 0.3. For a user-defined type which subclasses `TypeEngine`, the `bind_processor()` and `result_processor()` methods should be used for bind/result processing. Any user defined type, whether extending `TypeEngine` or `TypeDecorator`, which uses the old 0.3 style can be easily adapted to the new style using the following adapter:

```
class AdaptOldConvertMethods(object):
    """A mixin which adapts 0.3-style convert_bind_param and
    convert_result_value methods

    """
    def bind_processor(self, dialect):
        def convert(value):
            return self.convert_bind_param(value, dialect)
        return convert

    def result_processor(self, dialect):
        def convert(value):
            return self.convert_result_value(value, dialect)
        return convert

    def convert_result_value(self, value, dialect):
        return value

    def convert_bind_param(self, value, dialect):
        return value
```

To use the above mixin:

```
class MyType(AdaptOldConvertMethods, TypeEngine):
   # ...
```

- The `quote` flag on `Column` and `Table` as well as the `quote_schema` flag on `Table` now control quoting both positively and negatively. The default is `None`, meaning let regular quoting rules take effect. When `True`, quoting is forced on. When `False`, quoting is forced off.
- Column `DEFAULT` value DDL can now be more conveniently specified with `Column(..., server_default='val')`, deprecating `Column(..., PassiveDefault('val'))`. `default=` is now exclusively for Python-initiated default values, and can coexist with server_default. A new `server_default=FetchedValue()` replaces the `PassiveDefault('')` idiom for marking columns as subject to influence from external triggers and has no DDL side effects.
- SQLite's `DateTime`, `Time` and `Date` types now **only accept datetime objects, not strings** as bind parameter input. If you'd like to create your own "hybrid" type which accepts strings and returns results as date objects (from whatever format you'd like), create a `TypeDecorator` that builds on `String`. If you only want string-based dates, just use `String`.
- Additionally, the `DateTime` and `Time` types, when used with SQLite, now represent the "microseconds" field of the Python `datetime.datetime` object in the same manner as `str(datetime)` - as fractional seconds, not a count of microseconds. That is:

```
dt = datetime.datetime(2008, 6, 27, 12, 0, 0, 125)  # 125 usec

# old way
'2008-06-27 12:00:00.125'

# new way
'2008-06-27 12:00:00.000125'
```

So if an existing SQLite file-based database intends to be used across 0.4 and 0.5, you either have to upgrade the datetime columns to store the new format (NOTE: please test this, I'm pretty sure its correct):

```
UPDATE mytable SET somedatecol =
  substr(somedatecol, 0, 19) || '.' || substr((substr(somedatecol, 21, -1) / 1000000), 3, -1);
```

or, enable "legacy" mode as follows:

```
from sqlalchemy.databases.sqlite import DateTimeMixin
DateTimeMixin.__legacy_microseconds__ = True
```

Connection Pool no longer threadlocal by default

0.4 has an unfortunate default setting of "pool_threadlocal=True", leading to surprise behavior when, for example, using multiple Sessions within a single thread. This flag is now off in 0.5. To re-enable 0.4's behavior, specify `pool_threadlocal=True` to `create_engine()`, or alternatively use the "threadlocal" strategy via `strategy="threadlocal"`.

*args Accepted, *args No Longer Accepted

The policy with `method(\*args)` vs. `method([args])` is, if the method accepts a variable-length set of items which represent a fixed structure, it takes `\*args`. If the method accepts a variable-length set of items that are data-driven, it takes `[args]`.

- The various Query.options() functions `eagerload()`, `eagerload_all()`, `lazyload()`, `contains_eager()`, `defer()`, `undefer()` all accept variable-length `\*keys` as their argument now, which allows a path to be formulated using descriptors, ie.:

```
query.options(eagerload_all(User.orders, Order.items, Item.keywords))
```

  A single array argument is still accepted for backwards compatibility.

- Similarly, the `Query.join()` and `Query.outerjoin()` methods accept a variable length *args, with a single array accepted for backwards compatibility:

```
query.join('orders', 'items')
query.join(User.orders, Order.items)
```

- the `in_()` method on columns and similar only accepts a list argument now. It no longer accepts `\*args`.

Removed

- **entity_name** - This feature was always problematic and rarely used. 0.5's more deeply fleshed out use cases revealed further issues with `entity_name` which led to its removal. If different mappings are required for a single class, break the class into separate subclasses and map them separately. An example of this is at [wiki:UsageRecipes/EntityName]. More information regarding rationale is described at http://groups.google.com/group/sqlalchemy/browse_thread/thread/9e23a0641a88b96d? hl=en .
- **get()/load() cleanup**

  The `load()` method has been removed. Its functionality was kind of arbitrary and basically copied from Hibernate, where it's also not a particularly meaningful method.

  To get equivalent functionality:

```
x = session.query(SomeClass).populate_existing().get(7)
```

  `Session.get(cls, id)` and `Session.load(cls, id)` have been removed. `Session.get()` is redundant vs. `session.query(cls).get(id)`.

  `MapperExtension.get()` is also removed (as is `MapperExtension.load()`). To override the functionality of `Query.get()`, use a subclass:

```
class MyQuery(Query):
    def get(self, ident):
        # ...

session = sessionmaker(query_cls=MyQuery)()

ad1 = session.query(Address).get(1)
```

- `sqlalchemy.orm.relation()`

  The following deprecated keyword arguments have been removed:

  foreignkey, association, private, attributeext, is_backref

  In particular, `attributeext` is replaced with `extension` - the `AttributeExtension` class is now in the public API.

- `session.Query()`

  The following deprecated functions have been removed:

  list, scalar, count_by, select_whereclause, get_by, select_by, join_by, selectfirst, selectone, select, execute, select_statement, select_text, join_to, join_via, selectfirst_by, selectone_by, apply_max, apply_min, apply_avg, apply_sum

  Additionally, the `id` keyword argument to `join()`, `outerjoin()`, `add_entity()` and `add_column()` has been removed. To target table aliases in `Query` to result columns, use the `aliased` construct:

```
from sqlalchemy.orm import aliased
address_alias = aliased(Address)
print session.query(User, address_alias).join((address_alias, User.addresses)).all()
```

- `sqlalchemy.orm.Mapper`
  - instances()

  - get_session() - this method was not very noticeable, but had the effect of associating lazy loads with a particular session even if the parent object was entirely detached, when an extension such as `scoped_session()` or the old `SessionContextExt` was used. It's possible that some applications which relied upon this behavior will no longer work as expected; but the better programming practice here is to always ensure objects are present within sessions if database access from their attributes are required.

- `mapper(MyClass, mytable)`

  Mapped classes no are longer instrumented with a "c" class attribute; e.g. `MyClass.c`

- `sqlalchemy.orm.collections`

  The _prepare_instrumentation alias for prepare_instrumentation has been removed.

- `sqlalchemy.orm`

  Removed the `EXT_PASS` alias of `EXT_CONTINUE`.

- `sqlalchemy.engine`

  The alias from `DefaultDialect.preexecute_sequences` to `.preexecute_pk_sequences` has been removed.

  The deprecated engine_descriptors() function has been removed.

- `sqlalchemy.ext.activemapper`

  Module removed.

- `sqlalchemy.ext.assignmapper`

  Module removed.

- `sqlalchemy.ext.associationproxy`

  Pass-through of keyword args on the proxy's `.append(item, \**kw)` has been removed and is now simply `.append(item)`

- `sqlalchemy.ext.selectresults`, `sqlalchemy.mods.selectresults`

  Modules removed.

- `sqlalchemy.ext.declarative`

  `declared_synonym()` removed.

- `sqlalchemy.ext.sessioncontext`

  Module removed.

- `sqlalchemy.log`

  The `SADeprecationWarning` alias to `sqlalchemy.exc.SADeprecationWarning` has been removed.

- `sqlalchemy.exc`

  `exc.AssertionError` has been removed and usage replaced by the Python built-in of the same name.

- `sqlalchemy.databases.mysql`

  The deprecated `get_version_info` dialect method has been removed.

Renamed or Moved

- `sqlalchemy.exceptions` is now `sqlalchemy.exc`

  The module may still be imported under the old name until 0.6.

- `FlushError`, `ConcurrentModificationError`, `UnmappedColumnError` -> sqlalchemy.orm.exc

  These exceptions moved to the orm package. Importing 'sqlalchemy.orm' will install aliases in sqlalchemy.exc for compatibility until 0.6.

- `sqlalchemy.logging` -> `sqlalchemy.log`

  This internal module was renamed. No longer needs to be special cased when packaging SA with py2app and similar tools that scan imports.

- `session.Query().iterate_instances()` -> `session.Query().instances()`.

Deprecated

- `Session.save()`, `Session.update()`, `Session.save_or_update()`

  All three replaced by `Session.add()`

- `sqlalchemy.PassiveDefault`

  Use `Column(server_default=...)` Translates to sqlalchemy.DefaultClause() under the hood.

- `session.Query().iterate_instances()`. It has been renamed to `instances()`.

6.3.7 What's new in SQLAlchemy 0.4?

About this Document

This document describes changes between SQLAlchemy version 0.3, last released October 14, 2007, and SQLAlchemy version 0.4, last released October 12, 2008.

Document date: March 21, 2008

First Things First

If you're using any ORM features, make sure you import from `sqlalchemy.orm`:

```
from sqlalchemy import *
from sqlalchemy.orm import *
```

Secondly, anywhere you used to say `engine=`, `connectable=`, `bind_to=`, `something.engine`, `metadata.connect()`, use `bind`:

```
myengine = create_engine('sqlite://')

meta = MetaData(myengine)

meta2 = MetaData()
meta2.bind = myengine

session = create_session(bind=myengine)
```

```
statement = select([table], bind=myengine)
```

Got those ? Good! You're now (95%) 0.4 compatible. If you're using 0.3.10, you can make these changes immediately; they'll work there too.

Module Imports

In 0.3, "`from sqlachemy import *`" would import all of sqlachemy's sub-modules into your namespace. Version 0.4 no longer imports sub-modules into the namespace. This may mean you need to add extra imports into your code.

In 0.3, this code worked:

```
from sqlalchemy import *

class UTCDateTime(types.TypeDecorator):
    pass
```

In 0.4, one must do:

```
from sqlalchemy import *
from sqlalchemy import types

class UTCDateTime(types.TypeDecorator):
    pass
```

Object Relational Mapping

Querying

New Query API Query is standardized on the generative interface (old interface is still there, just deprecated). While most of the generative interface is available in 0.3, the 0.4 Query has the inner guts to match the generative outside, and has a lot more tricks. All result narrowing is via `filter()` and `filter_by()`, limiting/offset is either through array slices or `limit()`/`offset()`, joining is via `join()` and `outerjoin()` (or more manually, through `select_from()` as well as manually-formed criteria).

To avoid deprecation warnings, you must make some changes to your 03 code

User.query.get_by(**kwargs)

```
User.query.filter_by(**kwargs).first()
```

User.query.select_by(**kwargs)

```
User.query.filter_by(**kwargs).all()
```

User.query.select()

```
User.query.filter(xxx).all()
```

New Property-Based Expression Constructs By far the most palpable difference within the ORM is that you can now construct your query criterion using class-based attributes directly. The ".c." prefix is no longer needed when working with mapped classes:

```
session.query(User).filter(and_(User.name == 'fred', User.id > 17))
```

While simple column-based comparisons are no big deal, the class attributes have some new "higher level" constructs available, including what was previously only available in `filter_by()`:

```
# comparison of scalar relations to an instance
filter(Address.user == user)

# return all users who contain a particular address
filter(User.addresses.contains(address))

# return all users who *dont* contain the address
filter(~User.address.contains(address))

# return all users who contain a particular address with
# the email_address like '%foo%'
filter(User.addresses.any(Address.email_address.like('%foo%')))

# same, email address equals 'foo@bar.com'.  can fall back to keyword
# args for simple comparisons
filter(User.addresses.any(email_address = 'foo@bar.com'))

# return all Addresses whose user attribute has the username 'ed'
filter(Address.user.has(name='ed'))

# return all Addresses whose user attribute has the username 'ed'
# and an id > 5 (mixing clauses with kwargs)
filter(Address.user.has(User.id > 5, name='ed'))
```

The `Column` collection remains available on mapped classes in the `.c` attribute. Note that property-based expressions are only available with mapped properties of mapped classes. `.c` is still used to access columns in regular tables and selectable objects produced from SQL Expressions.

Automatic Join Aliasing We've had join() and outerjoin() for a while now:

```
session.query(Order).join('items')...
```

Now you can alias them:

```
session.query(Order).join('items', aliased=True).
   filter(Item.name='item 1').join('items', aliased=True).filter(Item.name=='item 3')
```

The above will create two joins from orders->items using aliases. the `filter()` call subsequent to each will adjust its table criterion to that of the alias. To get at the `Item` objects, use `add_entity()` and target each join with an `id`:

```
session.query(Order).join('items', id='j1', aliased=True).
filter(Item.name == 'item 1').join('items', aliased=True, id='j2').
filter(Item.name == 'item 3').add_entity(Item, id='j1').add_entity(Item, id='j2')
```

Returns tuples in the form: `(Order, Item, Item)`.

Self-referential Queries So query.join() can make aliases now. What does that give us ? Self-referential queries ! Joins can be done without any `Alias` objects:

```
# standard self-referential TreeNode mapper with backref
mapper(TreeNode, tree_nodes, properties={
    'children':relation(TreeNode, backref=backref('parent', remote_side=tree_nodes.id))
})

# query for node with child containing "bar" two levels deep
session.query(TreeNode).join(["children", "children"], aliased=True).filter_by(name='bar')
```

To add criterion for each table along the way in an aliased join, you can use `from_joinpoint` to keep joining against the same line of aliases:

```
# search for the treenode along the path "n1/n12/n122"

# first find a Node with name="n122"
q = sess.query(Node).filter_by(name='n122')

# then join to parent with "n12"
q = q.join('parent', aliased=True).filter_by(name='n12')

# join again to the next parent with 'n1'.  use 'from_joinpoint'
# so we join from the previous point, instead of joining off the
# root table
q = q.join('parent', aliased=True, from_joinpoint=True).filter_by(name='n1')

node = q.first()
```

`query.populate_existing()` The eager version of `query.load()` (or `session.refresh()`). Every instance loaded from the query, including all eagerly loaded items, get refreshed immediately if already present in the session:

```
session.query(Blah).populate_existing().all()
```

Relations

SQL Clauses Embedded in Updates/Inserts For inline execution of SQL clauses, embedded right in the UPDATE or INSERT, during a `flush()`:

```
myobject.foo = mytable.c.value + 1

user.pwhash = func.md5(password)

order.hash = text("select hash from hashing_table")
```

The column-attribute is set up with a deferred loader after the operation, so that it issues the SQL to load the new value when you next access.

Self-referential and Cyclical Eager Loading Since our alias-fu has improved, `relation()` can join along the same table *any number of times*; you tell it how deep you want to go. Lets show the self-referential `TreeNode` more clearly:

```
nodes = Table('nodes', metadata,
     Column('id', Integer, primary_key=True),
     Column('parent_id', Integer, ForeignKey('nodes.id')),
     Column('name', String(30)))

class TreeNode(object):
    pass

mapper(TreeNode, nodes, properties={
    'children':relation(TreeNode, lazy=False, join_depth=3)
})
```

So what happens when we say:

```
create_session().query(TreeNode).all()
```

? A join along aliases, three levels deep off the parent:

```
SELECT
nodes_3.id AS nodes_3_id, nodes_3.parent_id AS nodes_3_parent_id, nodes_3.name AS nodes_3_name,
nodes_2.id AS nodes_2_id, nodes_2.parent_id AS nodes_2_parent_id, nodes_2.name AS nodes_2_name,
nodes_1.id AS nodes_1_id, nodes_1.parent_id AS nodes_1_parent_id, nodes_1.name AS nodes_1_name,
nodes.id AS nodes_id, nodes.parent_id AS nodes_parent_id, nodes.name AS nodes_name
FROM nodes LEFT OUTER JOIN nodes AS nodes_1 ON nodes.id = nodes_1.parent_id
LEFT OUTER JOIN nodes AS nodes_2 ON nodes_1.id = nodes_2.parent_id
LEFT OUTER JOIN nodes AS nodes_3 ON nodes_2.id = nodes_3.parent_id
ORDER BY nodes.oid, nodes_1.oid, nodes_2.oid, nodes_3.oid
```

Notice the nice clean alias names too. The joining doesn't care if it's against the same immediate table or some other object which then cycles back to the beginning. Any kind of chain of eager loads can cycle back onto itself when `join_depth` is specified. When not present, eager loading automatically stops when it hits a cycle.

Composite Types This is one from the Hibernate camp. Composite Types let you define a custom datatype that is composed of more than one column (or one column, if you wanted). Lets define a new type, `Point`. Stores an x/y coordinate:

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __composite_values__(self):
        return self.x, self.y
    def __eq__(self, other):
        return other.x == self.x and other.y == self.y
    def __ne__(self, other):
        return not self.__eq__(other)
```

The way the `Point` object is defined is specific to a custom type; constructor takes a list of arguments, and the `__composite_values__()` method produces a sequence of those arguments. The order will match up to our mapper, as we'll see in a moment.

Let's create a table of vertices storing two points per row:

```
vertices = Table('vertices', metadata,
    Column('id', Integer, primary_key=True),
    Column('x1', Integer),
```

```
    Column('y1', Integer),
    Column('x2', Integer),
    Column('y2', Integer),
    )
```

Then, map it ! We'll create a `Vertex` object which stores two `Point` objects:

```
class Vertex(object):
    def __init__(self, start, end):
        self.start = start
        self.end = end

mapper(Vertex, vertices, properties={
    'start':composite(Point, vertices.c.x1, vertices.c.y1),
    'end':composite(Point, vertices.c.x2, vertices.c.y2)
})
```

Once you've set up your composite type, it's usable just like any other type:

```
v = Vertex(Point(3, 4), Point(26,15))
session.save(v)
session.flush()

# works in queries too
q = session.query(Vertex).filter(Vertex.start == Point(3, 4))
```

If you'd like to define the way the mapped attributes generate SQL clauses when used in expressions, create your own `sqlalchemy.orm.PropComparator` subclass, defining any of the common operators (like `__eq__()`, `__le__()`, etc.), and send it in to `composite()`. Composite types work as primary keys too, and are usable in `query.get()`:

```
# a Document class which uses a composite Version
# object as primary key
document = query.get(Version(1, 'a'))
```

`dynamic_loader()` relations A `relation()` that returns a live `Query` object for all read operations. Write operations are limited to just `append()` and `remove()`, changes to the collection are not visible until the session is flushed. This feature is particularly handy with an "autoflushing" session which will flush before each query.

```
mapper(Foo, foo_table, properties={
    'bars':dynamic_loader(Bar, backref='foo', <other relation() opts>)
})

session = create_session(autoflush=True)
foo = session.query(Foo).first()

foo.bars.append(Bar(name='lala'))

for bar in foo.bars.filter(Bar.name=='lala'):
    print bar

session.commit()
```

New Options: `undefer_group()`, `eagerload_all()` A couple of query options which are handy. `undefer_group()` marks a whole group of "deferred" columns as undeferred:

```
mapper(Class, table, properties={
    'foo' : deferred(table.c.foo, group='group1'),
    'bar' : deferred(table.c.bar, group='group1'),
    'bat' : deferred(table.c.bat, group='group1'),
)

session.query(Class).options(undefer_group('group1')).filter(...).all()
```

and `eagerload_all()` sets a chain of attributes to be eager in one pass:

```
mapper(Foo, foo_table, properties={
   'bar':relation(Bar)
})
mapper(Bar, bar_table, properties={
   'bat':relation(Bat)
})
mapper(Bat, bat_table)

# eager load bar and bat
session.query(Foo).options(eagerload_all('bar.bat')).filter(...).all()
```

New Collection API Collections are no longer proxied by an {{{InstrumentedList}}} proxy, and access to members, methods and attributes is direct. Decorators now intercept objects entering and leaving the collection, and it is now possible to easily write a custom collection class that manages its own membership. Flexible decorators also replace the named method interface of custom collections in 0.3, allowing any class to be easily adapted to use as a collection container.

Dictionary-based collections are now much easier to use and fully `dict`-like. Changing `__iter__` is no longer needed for `dict``s, and new built-in ``dict` types cover many needs:

```
# use a dictionary relation keyed by a column
relation(Item, collection_class=column_mapped_collection(items.c.keyword))
# or named attribute
relation(Item, collection_class=attribute_mapped_collection('keyword'))
# or any function you like
relation(Item, collection_class=mapped_collection(lambda entity: entity.a + entity.b))
```

Existing 0.3 `dict`-like and freeform object derived collection classes will need to be updated for the new API. In most cases this is simply a matter of adding a couple decorators to the class definition.

Mapped Relations from External Tables/Subqueries This feature quietly appeared in 0.3 but has been improved in 0.4 thanks to better ability to convert subqueries against a table into subqueries against an alias of that table; this is key for eager loading, aliased joins in queries, etc. It reduces the need to create mappers against select statements when you just need to add some extra columns or subqueries:

```
mapper(User, users, properties={
       'fullname': column_property((users.c.firstname + users.c.lastname).label('fullname')),
       'numposts': column_property(
            select([func.count(1)], users.c.id==posts.c.user_id).correlate(users).label('posts')
       )
    })
```

a typical query looks like:

```
SELECT (SELECT count(1) FROM posts WHERE users.id = posts.user_id) AS count,
users.firstname || users.lastname AS fullname,
users.id AS users_id, users.firstname AS users_firstname, users.lastname AS users_lastname
FROM users ORDER BY users.oid
```

Horizontal Scaling (Sharding) API

[browser:/sqlalchemy/trunk/examples/sharding/attribute_shard .py]

Sessions

New Session Create Paradigm; SessionContext, assignmapper Deprecated That's right, the whole shebang is being replaced with two configurational functions. Using both will produce the most 0.1-ish feel we've had since 0.1 (i.e., the least amount of typing).

Configure your own `Session` class right where you define your `engine` (or anywhere):

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

engine = create_engine('myengine://')
Session = sessionmaker(bind=engine, autoflush=True, transactional=True)

# use the new Session() freely
sess = Session()
sess.save(someobject)
sess.flush()
```

If you need to post-configure your Session, say with an engine, add it later with `configure()`:

```
Session.configure(bind=create_engine(...))
```

All the behaviors of `SessionContext` and the `query` and `__init__` methods of `assignmapper` are moved into the new `scoped_session()` function, which is compatible with both `sessionmaker` as well as `create_session()`:

```
from sqlalchemy.orm import scoped_session, sessionmaker

Session = scoped_session(sessionmaker(autoflush=True, transactional=True))
Session.configure(bind=engine)

u = User(name='wendy')

sess = Session()
sess.save(u)
sess.commit()

# Session constructor is thread-locally scoped.  Everyone gets the same
# Session in the thread when scope="thread".
sess2 = Session()
assert sess is sess2
```

When using a thread-local `Session`, the returned class has all of `Session's` interface implemented as classmethods, and "assignmapper"'s functionality is available using the `mapper` classmethod. Just like the old `objectstore` days....

```
# "assignmapper"-like functionality available via ScopedSession.mapper
Session.mapper(User, users_table)

u = User(name='wendy')

Session.commit()
```

Sessions are again Weak Referencing By Default The weak_identity_map flag is now set to `True` by default on Session. Instances which are externally deferenced and fall out of scope are removed from the session automatically. However, items which have "dirty" changes present will remain strongly referenced until those changes are flushed at which case the object reverts to being weakly referenced (this works for 'mutable' types, like picklable attributes, as well). Setting weak_identity_map to `False` restores the old strong-referencing behavior for those of you using the session like a cache.

Auto-Transactional Sessions As you might have noticed above, we are calling `commit()` on `Session`. The flag `transactional=True` means the `Session` is always in a transaction, `commit()` persists permanently.

Auto-Flushing Sessions Also, `autoflush=True` means the `Session` will `flush()` before each `query` as well as when you call `flush()` or `commit()`. So now this will work:

```
Session = sessionmaker(bind=engine, autoflush=True, transactional=True)

u = User(name='wendy')

sess = Session()
sess.save(u)

# wendy is flushed, comes right back from a query
wendy = sess.query(User).filter_by(name='wendy').one()
```

Transactional methods moved onto sessions `commit()` and `rollback()`, as well as `begin()` are now directly on `Session`. No more need to use `SessionTransaction` for anything (it remains in the background).

```
Session = sessionmaker(autoflush=True, transactional=False)

sess = Session()
sess.begin()

# use the session

sess.commit() # commit transaction
```

Sharing a `Session` with an enclosing engine-level (i.e. non-ORM) transaction is easy:

```
Session = sessionmaker(autoflush=True, transactional=False)

conn = engine.connect()
trans = conn.begin()
```

```
sess = Session(bind=conn)

# ... session is transactional

# commit the outermost transaction
trans.commit()
```

Nested Session Transactions with SAVEPOINT Available at the Engine and ORM level. ORM docs so far:

http://www.sqlalchemy.org/docs/04/session.html#unitofwork_ma naging

Two-Phase Commit Sessions Available at the Engine and ORM level. ORM docs so far:

http://www.sqlalchemy.org/docs/04/session.html#unitofwork_ma naging

Inheritance

Polymorphic Inheritance with No Joins or Unions New docs for inheritance: http://www.sqlalchemy.org/docs/04/mappers.html#advdatamapping_mapper_inheritance_joined

Better Polymorphic Behavior with `get()` All classes within a joined-table inheritance hierarchy get an `_instance_key` using the base class, i.e. `(BaseClass, (1, ), None)`. That way when you call `get()` a `Query` against the base class, it can locate subclass instances in the current identity map without querying the database.

Types

Custom Subclasses of `sqlalchemy.types.TypeDecorator` There is a New API for subclassing a TypeDecorator. Using the 0.3 API causes compilation errors in some cases.

SQL Expressions

All New, Deterministic Label/Alias Generation

All the "anonymous" labels and aliases use a simple <name>_<number> format now. SQL is much easier to read and is compatible with plan optimizer caches. Just check out some of the examples in the tutorials: http://www.sqlalchemy.org/docs/04/ormtutorial.html http://www.sqlalchemy.org/docs/04/sqlexpression.html

Generative select() Constructs

This is definitely the way to go with `select()`. See htt p://www.sqlalchemy.org/docs/04/sqlexpression.html#sql_transf orm .

New Operator System

SQL operators and more or less every SQL keyword there is are now abstracted into the compiler layer. They now act intelligently and are type/backend aware, see: http://www.sq lalchemy.org/docs/04/sqlexpression.html#sql_operators

All `type` Keyword Arguments Renamed to `type_`

Just like it says:

```
b = bindparam('foo', type_=String)
```

in_ Function Changed to Accept Sequence or Selectable

The in_ function now takes a sequence of values or a selectable as its sole argument. The previous API of passing in values as positional arguments still works, but is now deprecated. This means that

```
my_table.select(my_table.c.id.in_(1,2,3)
my_table.select(my_table.c.id.in_(*listOfIds)
```

should be changed to

```
my_table.select(my_table.c.id.in_([1,2,3])
my_table.select(my_table.c.id.in_(listOfIds)
```

Schema and Reflection

`MetaData`, `BoundMetaData`, `DynamicMetaData`...

In the 0.3.x series, `BoundMetaData` and `DynamicMetaData` were deprecated in favor of `MetaData` and `ThreadLocalMetaData`. The older names have been removed in 0.4. Updating is simple:

```
+-------------------------------------+-------------------------+
|If You Had                           | Now Use                 |
+=====================================+=========================+
| ``MetaData``                        | ``MetaData``            |
+-------------------------------------+-------------------------+
| ``BoundMetaData``                   | ``MetaData``            |
+-------------------------------------+-------------------------+
| ``DynamicMetaData`` (with one       | ``MetaData``            |
| engine or threadlocal=False)        |                         |
+-------------------------------------+-------------------------+
| ``DynamicMetaData``                 | ``ThreadLocalMetaData`` |
| (with different engines per thread) |                         |
+-------------------------------------+-------------------------+
```

The seldom-used `name` parameter to `MetaData` types has been removed. The `ThreadLocalMetaData` constructor now takes no arguments. Both types can now be bound to an `Engine` or a single `Connection`.

One Step Multi-Table Reflection

You can now load table definitions and automatically create `Table` objects from an entire database or schema in one pass:

```
>>> metadata = MetaData(myengine, reflect=True)
>>> metadata.tables.keys()
['table_a', 'table_b', 'table_c', '...']
```

`MetaData` also gains a `.reflect()` method enabling finer control over the loading process, including specification of a subset of available tables to load.

SQL Execution

`engine`, `connectable`, and `bind_to` are all now `bind`

`Transactions`, `NestedTransactions` and `TwoPhaseTransactions`

Connection Pool Events

The connection pool now fires events when new DB-API connections are created, checked out and checked back into the pool. You can use these to execute session-scoped SQL setup statements on fresh connections, for example.

Oracle Engine Fixed

In 0.3.11, there were bugs in the Oracle Engine on how Primary Keys are handled. These bugs could cause programs that worked fine with other engines, such as sqlite, to fail when using the Oracle Engine. In 0.4, the Oracle Engine has been reworked, fixing these Primary Key problems.

Out Parameters for Oracle

```
result = engine.execute(text("begin foo(:x, :y, :z); end;", bindparams=[bindparam('x', Numeric), outp
assert result.out_parameters == {'y':10, 'z':75}
```

Connection-bound `MetaData`, `Sessions`

`MetaData` and `Session` can be explicitly bound to a connection:

```
conn = engine.connect()
sess = create_session(bind=conn)
```

Faster, More Foolproof `ResultProxy` Objects

CHAPTER 7

Indices and tables

- glossary
- genindex

e

examples.adjacency_list, 417
examples.association, 418
examples.custom_attributes, 426
examples.dogpile_caching, 427
examples.dynamic_dict, 418
examples.elementtree, 424
examples.generic_associations, 418
examples.graphs, 418
examples.inheritance, 426
examples.join_conditions, 423
examples.large_collection, 419
examples.materialized_paths, 419
examples.nested_sets, 419
examples.performance, 419
examples.postgis, 428
examples.sharding, 427
examples.versioned_history, 424
examples.versioned_rows, 425
examples.vertical, 426

s

sqlalchemy.dialects.firebird.base, 843
sqlalchemy.dialects.firebird.fdb, 844
sqlalchemy.dialects.firebird.kinterbasdb, 845
sqlalchemy.dialects.mssql.adodbapi, 860
sqlalchemy.dialects.mssql.base, 846
sqlalchemy.dialects.mssql.mxodbc, 859
sqlalchemy.dialects.mssql.pymssql, 859
sqlalchemy.dialects.mssql.pyodbc, 857
sqlalchemy.dialects.mssql.zxjdbc, 860
sqlalchemy.dialects.mysql.base, 860
sqlalchemy.dialects.mysql.cymysql, 881
sqlalchemy.dialects.mysql.gaerdbms, 882
sqlalchemy.dialects.mysql.mysqlconnector, 880
sqlalchemy.dialects.mysql.mysqldb, 879
sqlalchemy.dialects.mysql.oursql, 881
sqlalchemy.dialects.mysql.pymysql, 880
sqlalchemy.dialects.mysql.pyodbc, 882
sqlalchemy.dialects.mysql.zxjdbc, 883
sqlalchemy.dialects.oracle.base, 883
sqlalchemy.dialects.oracle.cx_oracle, 890
sqlalchemy.dialects.oracle.zxjdbc, 894
sqlalchemy.dialects.postgresql.base, 894
sqlalchemy.dialects.postgresql.pg8000, 923
sqlalchemy.dialects.postgresql.psycopg2, 919
sqlalchemy.dialects.postgresql.psycopg2cffi, 924
sqlalchemy.dialects.postgresql.pypostgresql, 925
sqlalchemy.dialects.postgresql.zxjdbc, 925
sqlalchemy.dialects.sqlite, 932
sqlalchemy.dialects.sqlite.base, 925
sqlalchemy.dialects.sqlite.pysqlcipher, 938
sqlalchemy.dialects.sqlite.pysqlite, 934
sqlalchemy.dialects.sybase.base, 939
sqlalchemy.dialects.sybase.mxodbc, 940
sqlalchemy.dialects.sybase.pyodbc, 939
sqlalchemy.dialects.sybase.pysybase, 939
sqlalchemy.engine, 748
sqlalchemy.exc, 821
sqlalchemy.ext.associationproxy, 336
sqlalchemy.ext.automap, 347
sqlalchemy.ext.baked, 358
sqlalchemy.ext.compiler, 607
sqlalchemy.ext.declarative, 385
sqlalchemy.ext.horizontal_shard, 403
sqlalchemy.ext.hybrid, 404
sqlalchemy.ext.instrumentation, 415
sqlalchemy.ext.mutable, 392
sqlalchemy.ext.orderinglist, 400
sqlalchemy.ext.serializer, 613
sqlalchemy.inspection, 817
sqlalchemy.interfaces, 818

sqlalchemy.orm, 41
sqlalchemy.orm.exc, 330
sqlalchemy.orm.instrumentation, 289
sqlalchemy.orm.interfaces, 331
sqlalchemy.orm.session, 237
sqlalchemy.pool, 778
sqlalchemy.schema, 614
sqlalchemy.sql.expression, 459
sqlalchemy.sql.functions, 598
sqlalchemy.types, 703

Symbols

_ConnectionFairy (class in sqlalchemy.pool), 787
_ConnectionRecord (class in sqlalchemy.pool), 788
_CreateDropBase (class in sqlalchemy.schema), 700
__add__() (sqlalchemy.sql.operators.ColumnOperators method), 490
__and__() (sqlalchemy.sql.operators.ColumnOperators method), 490
__and__() (sqlalchemy.sql.operators.Operators method), 504
__call__() (sqlalchemy.orm.scoping.scoped_session method), 230
__call__() (sqlalchemy.orm.session.sessionmaker method), 238
__call__() (sqlalchemy.schema.DDLElement method), 697
__delattr__ (sqlalchemy.sql.operators.ColumnOperators attribute), 491
__delete__() (sqlalchemy.orm.attributes.InstrumentedAttribute method), 304
__div__() (sqlalchemy.sql.operators.ColumnOperators method), 491
__eq__() (sqlalchemy.orm.attributes.QueryableAttribute method), 324
__eq__() (sqlalchemy.orm.interfaces.PropComparator method), 308
__eq__() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 293
__eq__() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 314
__eq__() (sqlalchemy.schema.Column method), 619
__eq__() (sqlalchemy.sql.expression.ColumnElement method), 483
__eq__() (sqlalchemy.sql.operators.ColumnOperators method), 491
__format__() (sqlalchemy.sql.operators.ColumnOperators method), 491
__ge__() (sqlalchemy.sql.operators.ColumnOperators method), 491
__get__() (sqlalchemy.orm.attributes.InstrumentedAttribute method), 304
__getattribute__ (sqlalchemy.sql.operators.ColumnOperators attribute), 491
__getitem__() (sqlalchemy.sql.operators.ColumnOperators method), 491
__gt__() (sqlalchemy.sql.operators.ColumnOperators method), 491
__hash__ (sqlalchemy.sql.operators.ColumnOperators attribute), 491
__init__ (sqlalchemy.dialects.mssql.BIT attribute), 853
__init__ (sqlalchemy.dialects.mssql.MONEY attribute), 854
__init__ (sqlalchemy.dialects.mssql.SMALLMONEY attribute), 855
__init__ (sqlalchemy.dialects.mssql.SQL_VARIANT attribute), 855
__init__ (sqlalchemy.dialects.mssql.TINYINT attribute), 856
__init__ (sqlalchemy.dialects.mssql.UNIQUEIDENTIFIER attribute), 856
__init__ (sqlalchemy.dialects.mysql.DATE attribute), 870
__init__ (sqlalchemy.dialects.postgresql.CIDR attribute), 907
__init__ (sqlalchemy.dialects.postgresql.INET attribute), 912
__init__ (sqlalchemy.dialects.postgresql.MACADDR attribute), 915
__init__ (sqlalchemy.dialects.postgresql.OID attribute), 915
__init__ (sqlalchemy.dialects.postgresql.TSVECTOR attribute), 916
__init__ (sqlalchemy.ext.mutable.Mutable attribute), 398
__init__ (sqlalchemy.sql.expression.ColumnElement attribute), 483
__init__ (sqlalchemy.sql.expression.UpdateBase attribute), 591
__init__ (sqlalchemy.sql.operators.ColumnOperators attribute), 491
__init__() (sqlalchemy.dialects.mssql.CHAR method), 853

__init__() (sqlalchemy.dialects.mssql.IMAGE method), 854
__init__() (sqlalchemy.dialects.mssql.NCHAR method), 854
__init__() (sqlalchemy.dialects.mssql.NTEXT method), 854
__init__() (sqlalchemy.dialects.mssql.NVARCHAR method), 855
__init__() (sqlalchemy.dialects.mssql.SMALLDATETIME method), 855
__init__() (sqlalchemy.dialects.mssql.TEXT method), 855
__init__() (sqlalchemy.dialects.mssql.VARCHAR method), 856
__init__() (sqlalchemy.dialects.mysql.BIGINT method), 869
__init__() (sqlalchemy.dialects.mysql.BIT method), 869
__init__() (sqlalchemy.dialects.mysql.BLOB method), 869
__init__() (sqlalchemy.dialects.mysql.BOOLEAN method), 869
__init__() (sqlalchemy.dialects.mysql.CHAR method), 870
__init__() (sqlalchemy.dialects.mysql.DATETIME method), 870
__init__() (sqlalchemy.dialects.mysql.DECIMAL method), 870
__init__() (sqlalchemy.dialects.mysql.DOUBLE method), 871
__init__() (sqlalchemy.dialects.mysql.ENUM method), 871
__init__() (sqlalchemy.dialects.mysql.FLOAT method), 872
__init__() (sqlalchemy.dialects.mysql.INTEGER method), 872
__init__() (sqlalchemy.dialects.mysql.LONGTEXT method), 873
__init__() (sqlalchemy.dialects.mysql.MEDIUMINT method), 873
__init__() (sqlalchemy.dialects.mysql.MEDIUMTEXT method), 874
__init__() (sqlalchemy.dialects.mysql.NCHAR method), 874
__init__() (sqlalchemy.dialects.mysql.NUMERIC method), 874
__init__() (sqlalchemy.dialects.mysql.NVARCHAR method), 875
__init__() (sqlalchemy.dialects.mysql.REAL method), 875
__init__() (sqlalchemy.dialects.mysql.SET method), 875
__init__() (sqlalchemy.dialects.mysql.SMALLINT method), 876
__init__() (sqlalchemy.dialects.mysql.TEXT method), 877
__init__() (sqlalchemy.dialects.mysql.TIME method), 877
__init__() (sqlalchemy.dialects.mysql.TIMESTAMP method), 877
__init__() (sqlalchemy.dialects.mysql.TINYINT method), 878
__init__() (sqlalchemy.dialects.mysql.TINYTEXT method), 878
__init__() (sqlalchemy.dialects.mysql.VARCHAR method), 879
__init__() (sqlalchemy.dialects.oracle.BFILE method), 887
__init__() (sqlalchemy.dialects.oracle.DATE method), 888
__init__() (sqlalchemy.dialects.oracle.INTERVAL method), 888
__init__() (sqlalchemy.dialects.oracle.LONG method), 889
__init__() (sqlalchemy.dialects.oracle.NCLOB method), 888
__init__() (sqlalchemy.dialects.postgresql.ARRAY method), 906
__init__() (sqlalchemy.dialects.postgresql.BYTEA method), 907
__init__() (sqlalchemy.dialects.postgresql.DOUBLE_PRECISION method), 907
__init__() (sqlalchemy.dialects.postgresql.ENUM method), 909
__init__() (sqlalchemy.dialects.postgresql.ExcludeConstraint method), 918
__init__() (sqlalchemy.dialects.postgresql.HSTORE method), 911
__init__() (sqlalchemy.dialects.postgresql.JSON method), 913
__init__() (sqlalchemy.dialects.postgresql.REAL method), 915
__init__() (sqlalchemy.dialects.postgresql.UUID method), 916
__init__() (sqlalchemy.engine.Connection method), 758
__init__() (sqlalchemy.engine.CreateEnginePlugin method), 767
__init__() (sqlalchemy.engine.interfaces.Compiled method), 824
__init__() (sqlalchemy.engine.reflection.Inspector method), 647
__init__() (sqlalchemy.ext.associationproxy.AssociationProxy method), 345
__init__() (sqlalchemy.ext.horizontal_shard.ShardedSession method), 404
__init__() (sqlalchemy.ext.hybrid.hybrid_method method), 414
__init__() (sqlalchemy.ext.hybrid.hybrid_property method), 415

__init__() (sqlalchemy.ext.orderinglist.OrderingList method), 402
__init__() (sqlalchemy.orm.collections.MappedCollection method), 124
__init__() (sqlalchemy.orm.descriptor_props.CompositeProperty method), 299
__init__() (sqlalchemy.orm.descriptor_props.SynonymProperty method), 323
__init__() (sqlalchemy.orm.mapper.Mapper method), 83
__init__() (sqlalchemy.orm.properties.ColumnProperty method), 298
__init__() (sqlalchemy.orm.properties.ComparableProperty method), 299
__init__() (sqlalchemy.orm.properties.RelationshipProperty method), 322
__init__() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 315
__init__() (sqlalchemy.orm.query.Bundle method), 184
__init__() (sqlalchemy.orm.scoping.scoped_session method), 230
__init__() (sqlalchemy.orm.session.Session method), 239
__init__() (sqlalchemy.orm.session.sessionmaker method), 238
__init__() (sqlalchemy.pool.Pool method), 783
__init__() (sqlalchemy.pool.QueuePool method), 785
__init__() (sqlalchemy.pool.SingletonThreadPool method), 786
__init__() (sqlalchemy.schema.CheckConstraint method), 673
__init__() (sqlalchemy.schema.Column method), 619
__init__() (sqlalchemy.schema.ColumnCollectionConstraint method), 671
__init__() (sqlalchemy.schema.Constraint method), 670
__init__() (sqlalchemy.schema.CreateSchema method), 703
__init__() (sqlalchemy.schema.CreateTable method), 700
__init__() (sqlalchemy.schema.DDL method), 699
__init__() (sqlalchemy.schema.DropSchema method), 703
__init__() (sqlalchemy.schema.ForeignKey method), 675
__init__() (sqlalchemy.schema.ForeignKeyConstraint method), 679
__init__() (sqlalchemy.schema.Index method), 690
__init__() (sqlalchemy.schema.MetaData method), 629
__init__() (sqlalchemy.schema.PrimaryKeyConstraint method), 683
__init__() (sqlalchemy.schema.Sequence method), 658
__init__() (sqlalchemy.schema.Table method), 636
__init__() (sqlalchemy.schema.ThreadLocalMetaData method), 644
__init__() (sqlalchemy.schema.UniqueConstraint method), 685
__init__() (sqlalchemy.sql.compiler.DDLCompiler method), 824
__init__() (sqlalchemy.sql.compiler.IdentifierPreparer method), 839
__init__() (sqlalchemy.sql.compiler.SQLCompiler method), 840
__init__() (sqlalchemy.sql.expression.BindParameter method), 478
__init__() (sqlalchemy.sql.expression.Case method), 479
__init__() (sqlalchemy.sql.expression.Cast method), 479
__init__() (sqlalchemy.sql.expression.ColumnClause method), 481
__init__() (sqlalchemy.sql.expression.Delete method), 570
__init__() (sqlalchemy.sql.expression.Extract method), 499
__init__() (sqlalchemy.sql.expression.FunctionFilter method), 499
__init__() (sqlalchemy.sql.expression.Insert method), 575
__init__() (sqlalchemy.sql.expression.Join method), 541
__init__() (sqlalchemy.sql.expression.Label method), 500
__init__() (sqlalchemy.sql.expression.Over method), 500
__init__() (sqlalchemy.sql.expression.Select method), 546
__init__() (sqlalchemy.sql.expression.TableClause method), 563
__init__() (sqlalchemy.sql.expression.Tuple method), 503
__init__() (sqlalchemy.sql.expression.TypeCoerce method), 503
__init__() (sqlalchemy.sql.expression.Update method), 583
__init__() (sqlalchemy.sql.expression.WithinGroup method), 503
__init__() (sqlalchemy.sql.functions.Function method), 599
__init__() (sqlalchemy.sql.functions.FunctionElement method), 599
__init__() (sqlalchemy.types.ARRAY method), 714
__init__() (sqlalchemy.types.Boolean method), 704
__init__() (sqlalchemy.types.DateTime method), 704
__init__() (sqlalchemy.types.Enum method), 705
__init__() (sqlalchemy.types.Float method), 706
__init__() (sqlalchemy.types.Interval method), 706
__init__() (sqlalchemy.types.JSON method), 717
__init__() (sqlalchemy.types.LargeBinary method), 707
__init__() (sqlalchemy.types.Numeric method), 708
__init__() (sqlalchemy.types.PickleType method), 708
__init__() (sqlalchemy.types.String method), 709
__init__() (sqlalchemy.types.TypeDecorator method), 721
__init__() (sqlalchemy.types.Unicode method), 711
__init__() (sqlalchemy.types.UnicodeText method), 711
__init__() (sqlalchemy.util.ScopedRegistry method), 231
__invert__() (sqlalchemy.sql.operators.ColumnOperators method), 491

__invert__() (sqlalchemy.sql.operators.Operators method), 504
__le__ (sqlalchemy.orm.instrumentation.ClassManager attribute), 291
__le__() (sqlalchemy.orm.attributes.QueryableAttribute method), 324
__le__() (sqlalchemy.orm.interfaces.PropComparator method), 308
__le__() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 293
__le__() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 315
__le__() (sqlalchemy.schema.Column method), 622
__le__() (sqlalchemy.sql.expression.ColumnElement method), 483
__le__() (sqlalchemy.sql.operators.ColumnOperators method), 492
__lshift__() (sqlalchemy.sql.operators.ColumnOperators method), 492
__lt__ (sqlalchemy.orm.instrumentation.ClassManager attribute), 291
__lt__() (sqlalchemy.orm.attributes.QueryableAttribute method), 325
__lt__() (sqlalchemy.orm.interfaces.PropComparator method), 308
__lt__() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 293
__lt__() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 315
__lt__() (sqlalchemy.schema.Column method), 623
__lt__() (sqlalchemy.sql.expression.ColumnElement method), 483
__lt__() (sqlalchemy.sql.operators.ColumnOperators method), 492
__mod__() (sqlalchemy.sql.operators.ColumnOperators method), 492
__mul__() (sqlalchemy.sql.operators.ColumnOperators method), 492
__ne__ (sqlalchemy.orm.instrumentation.ClassManager attribute), 291
__ne__() (sqlalchemy.dialects.postgresql.ranges.RangeOperators method), 917
__ne__() (sqlalchemy.orm.attributes.QueryableAttribute method), 325
__ne__() (sqlalchemy.orm.interfaces.PropComparator method), 308
__ne__() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 293
__ne__() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 315
__ne__() (sqlalchemy.schema.Column method), 623
__ne__() (sqlalchemy.sql.expression.ColumnElement method), 483
__ne__() (sqlalchemy.sql.operators.ColumnOperators method), 492
__neg__() (sqlalchemy.sql.operators.ColumnOperators method), 492
__new__() (sqlalchemy.sql.operators.ColumnOperators method), 492
__or__() (sqlalchemy.sql.operators.ColumnOperators method), 492
__or__() (sqlalchemy.sql.operators.Operators method), 504
__pos__() (sqlalchemy.sql.operators.ColumnOperators method), 493
__rdiv__() (sqlalchemy.sql.operators.ColumnOperators method), 493
__reduce__() (sqlalchemy.sql.operators.ColumnOperators method), 493
__reduce_ex__() (sqlalchemy.sql.operators.ColumnOperators method), 493
__repr__ (sqlalchemy.sql.operators.ColumnOperators attribute), 493
__rmod__() (sqlalchemy.sql.operators.ColumnOperators method), 493
__rmul__() (sqlalchemy.sql.operators.ColumnOperators method), 493
__rshift__() (sqlalchemy.sql.operators.ColumnOperators method), 493
__rsub__() (sqlalchemy.sql.operators.ColumnOperators method), 493
__rtruediv__() (sqlalchemy.sql.operators.ColumnOperators method), 493
__set__() (sqlalchemy.orm.attributes.InstrumentedAttribute method), 304
__setattr__ (sqlalchemy.sql.operators.ColumnOperators attribute), 493
__sizeof__() (sqlalchemy.sql.operators.ColumnOperators method), 493
__str__ (sqlalchemy.sql.operators.ColumnOperators attribute), 493
__sub__() (sqlalchemy.sql.operators.ColumnOperators method), 493
__subclasshook__() (sqlalchemy.sql.operators.ColumnOperators method), 494
__truediv__() (sqlalchemy.sql.operators.ColumnOperators method), 494
_asdict() (sqlalchemy.util.KeyedTuple method), 186
_connection_record (sqlalchemy.pool._ConnectionFairy attribute), 787
_cursor_description() (sqlalchemy.engine.ResultProxy method), 773
_declarative_constructor() (in module sqlalchemy.ext.declarative.api), 387
_fields (sqlalchemy.util.KeyedTuple attribute), 186
_get_listen_keys() (sqlalchemy.ext.mutable.Mutable method), 398

_listen_on_attribute() (sqlalchemy.ext.mutable.Mutable method), 398
_parents (sqlalchemy.ext.mutable.Mutable attribute), 398
_parents (sqlalchemy.ext.mutable.MutableBase attribute), 397
_process_row (sqlalchemy.engine.ResultProxy attribute), 773
_soft_close() (sqlalchemy.engine.ResultProxy method), 773

A

AbstractConcreteBase (class in sqlalchemy.ext.declarative), 388
active_history (sqlalchemy.orm.interfaces.AttributeExtension attribute), 335
adapt() (sqlalchemy.types.SchemaType method), 709
adapt() (sqlalchemy.types.TypeDecorator method), 722
adapt() (sqlalchemy.types.TypeEngine method), 734
adapt_to_entity() (sqlalchemy.orm.interfaces.PropComparator method), 308
adapt_to_entity() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 293
adapter (sqlalchemy.orm.attributes.QueryableAttribute attribute), 325
adapter (sqlalchemy.orm.interfaces.PropComparator attribute), 308
adapter (sqlalchemy.orm.properties.ColumnProperty.Comparator attribute), 293
adapter (sqlalchemy.orm.properties.RelationshipProperty.Comparator attribute), 316
add() (sqlalchemy.orm.session.Session method), 240
add() (sqlalchemy.sql.expression.ColumnCollection method), 482
add_all() (sqlalchemy.orm.session.Session method), 241
add_column() (sqlalchemy.orm.query.Query method), 159
add_columns() (sqlalchemy.orm.query.Query method), 159
add_criteria() (sqlalchemy.ext.baked.BakedQuery method), 364
add_entity() (sqlalchemy.orm.query.Query method), 159
add_is_dependent_on() (sqlalchemy.schema.Table method), 636
add_properties() (sqlalchemy.orm.mapper.Mapper method), 83
add_property() (sqlalchemy.orm.mapper.Mapper method), 83
AddConstraint (class in sqlalchemy.schema), 703
adds() (sqlalchemy.orm.collections.collection static method), 121
adjacent_to() (sqlalchemy.dialects.postgresql.ranges.RangeOperatorComparator method), 917
after_attach() (sqlalchemy.orm.events.SessionEvents method), 276
after_attach() (sqlalchemy.orm.interfaces.SessionExtension method), 334
after_begin() (sqlalchemy.orm.events.SessionEvents method), 276
after_begin() (sqlalchemy.orm.interfaces.SessionExtension method), 334
after_bulk_delete() (sqlalchemy.orm.events.SessionEvents method), 277
after_bulk_delete() (sqlalchemy.orm.interfaces.SessionExtension method), 334
after_bulk_update() (sqlalchemy.orm.events.SessionEvents method), 277
after_bulk_update() (sqlalchemy.orm.interfaces.SessionExtension method), 334
after_commit() (sqlalchemy.orm.events.SessionEvents method), 278
after_commit() (sqlalchemy.orm.interfaces.SessionExtension method), 334
after_configured() (sqlalchemy.orm.events.MapperEvents method), 263
after_create() (sqlalchemy.events.DDLEvents method), 809
after_cursor_execute() (sqlalchemy.events.ConnectionEvents method), 795
after_delete() (sqlalchemy.orm.events.MapperEvents method), 264
after_delete() (sqlalchemy.orm.interfaces.MapperExtension method), 332
after_drop() (sqlalchemy.events.DDLEvents method), 810
after_execute() (sqlalchemy.events.ConnectionEvents method), 796
after_flush() (sqlalchemy.orm.events.SessionEvents method), 279
after_flush() (sqlalchemy.orm.interfaces.SessionExtension method), 334
after_flush_postexec() (sqlalchemy.orm.events.SessionEvents method), 279
after_flush_postexec() (sqlalchemy.orm.interfaces.SessionExtension method), 335
after_insert() (sqlalchemy.orm.events.MapperEvents method), 264
after_insert() (sqlalchemy.orm.interfaces.MapperExtension method), 332
after_parent_attach() (sqlalchemy.events.DDLEvents method), 810
after_rollback() (sqlalchemy.orm.events.SessionEvents method), 280
after_rollback() (sqlalchemy.orm.interfaces.SessionExtension method), 335
after_soft_rollback() (sqlalchemy.orm.events.SessionEvents method), 280
after_transaction_create() (sqlalchemy.orm.events.SessionEvents

method), 281
after_transaction_end() (sqlalchemy.orm.events.SessionEvents method), 281
after_update() (sqlalchemy.orm.events.MapperEvents method), 265
after_update() (sqlalchemy.orm.interfaces.MapperExtension method), 332
against() (sqlalchemy.schema.DDLElement method), 697
aggregate_order_by (class in sqlalchemy.dialects.postgresql), 904
Alias (class in sqlalchemy.sql.expression), 512
alias() (in module sqlalchemy.sql.expression), 507
alias() (sqlalchemy.schema.Table method), 636
alias() (sqlalchemy.sql.expression.Alias method), 512
alias() (sqlalchemy.sql.expression.CompoundSelect method), 516
alias() (sqlalchemy.sql.expression.FromClause method), 529
alias() (sqlalchemy.sql.expression.GenerativeSelect method), 531
alias() (sqlalchemy.sql.expression.Join method), 541
alias() (sqlalchemy.sql.expression.Select method), 546
alias() (sqlalchemy.sql.expression.TableClause method), 563
alias() (sqlalchemy.sql.functions.FunctionElement method), 599
aliased() (in module sqlalchemy.orm), 182
AliasedClass (class in sqlalchemy.orm.util), 183
AliasedInsp (class in sqlalchemy.orm.util), 184
All() (in module sqlalchemy.dialects.postgresql), 907
all() (sqlalchemy.ext.baked.Result method), 364
all() (sqlalchemy.orm.query.Query method), 159
all() (sqlalchemy.types.ARRAY.Comparator method), 713
all_() (in module sqlalchemy.sql.expression), 459
all_() (sqlalchemy.orm.attributes.QueryableAttribute method), 325
all_() (sqlalchemy.orm.interfaces.PropComparator method), 308
all_() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 293
all_() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 316
all_() (sqlalchemy.schema.Column method), 623
all_() (sqlalchemy.sql.expression.ColumnElement method), 483
all_() (sqlalchemy.sql.operators.ColumnOperators method), 494
all_orm_descriptors (sqlalchemy.orm.mapper.Mapper attribute), 84
AmbiguousForeignKeysError, 821
and_() (in module sqlalchemy.sql.expression), 459
anon_label (sqlalchemy.schema.Column attribute), 623
anon_label (sqlalchemy.sql.expression.ColumnElement attribute), 483
ansi_bind_rules (sqlalchemy.sql.compiler.SQLCompiler attribute), 840
AnsiFunction (class in sqlalchemy.sql.functions), 598
Any() (in module sqlalchemy.dialects.postgresql), 907
any() (sqlalchemy.ext.associationproxy.AssociationProxy method), 346
any() (sqlalchemy.orm.attributes.QueryableAttribute method), 325
any() (sqlalchemy.orm.interfaces.PropComparator method), 308
any() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 293
any() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 316
any() (sqlalchemy.types.ARRAY.Comparator method), 713
any_() (in module sqlalchemy.sql.expression), 460
any_() (sqlalchemy.orm.attributes.QueryableAttribute method), 325
any_() (sqlalchemy.orm.interfaces.PropComparator method), 308
any_() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 294
any_() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 316
any_() (sqlalchemy.schema.Column method), 623
any_() (sqlalchemy.sql.expression.ColumnElement method), 484
any_() (sqlalchemy.sql.operators.ColumnOperators method), 494
append() (sqlalchemy.ext.orderinglist.OrderingList method), 403
append() (sqlalchemy.orm.events.AttributeEvents method), 259
append() (sqlalchemy.orm.interfaces.AttributeExtension method), 336
append_column() (sqlalchemy.schema.Table method), 637
append_column() (sqlalchemy.sql.expression.Select method), 546
append_constraint() (sqlalchemy.schema.Table method), 637
append_correlation() (sqlalchemy.sql.expression.Select method), 546
append_ddl_listener() (sqlalchemy.schema.MetaData method), 630
append_ddl_listener() (sqlalchemy.schema.Table method), 637
append_from() (sqlalchemy.sql.expression.Select method), 547
append_group_by() (sqlalchemy.sql.expression.CompoundSelect method), 516

append_group_by() (sqlalchemy.sql.expression.GenerativeSelect method), 532
append_group_by() (sqlalchemy.sql.expression.Select method), 547
append_having() (sqlalchemy.sql.expression.Select method), 547
append_order_by() (sqlalchemy.sql.expression.CompoundSelect method), 516
append_order_by() (sqlalchemy.sql.expression.GenerativeSelect method), 532
append_order_by() (sqlalchemy.sql.expression.Select method), 547
append_prefix() (sqlalchemy.sql.expression.Select method), 547
append_whereclause() (sqlalchemy.sql.expression.Select method), 547
appender() (sqlalchemy.orm.collections.collection static method), 121
apply_labels() (sqlalchemy.sql.expression.CompoundSelect method), 516
apply_labels() (sqlalchemy.sql.expression.GenerativeSelect method), 532
apply_labels() (sqlalchemy.sql.expression.Select method), 547
argument_for() (sqlalchemy.schema.CheckConstraint method), 673
argument_for() (sqlalchemy.schema.ColumnCollectionConstraint method), 671
argument_for() (sqlalchemy.schema.ForeignKey method), 676
argument_for() (sqlalchemy.schema.ForeignKeyConstraint method), 680
argument_for() (sqlalchemy.schema.Index method), 690
argument_for() (sqlalchemy.schema.PrimaryKeyConstraint method), 683
argument_for() (sqlalchemy.schema.Table method), 637
argument_for() (sqlalchemy.schema.UniqueConstraint method), 685
argument_for() (sqlalchemy.sql.base.DialectKWArgs class method), 498
argument_for() (sqlalchemy.sql.expression.Delete method), 570
argument_for() (sqlalchemy.sql.expression.Insert method), 575
argument_for() (sqlalchemy.sql.expression.Update method), 583
argument_for() (sqlalchemy.sql.expression.UpdateBase method), 591
ArgumentError, 821
ARRAY (class in sqlalchemy.dialects.postgresql), 905
array (class in sqlalchemy.dialects.postgresql), 905
ARRAY (class in sqlalchemy.types), 712
array() (sqlalchemy.dialects.postgresql.HSTORE.Comparator method), 911
ARRAY.Comparator (class in sqlalchemy.dialects.postgresql), 906
ARRAY.Comparator (class in sqlalchemy.types), 713
array_agg (class in sqlalchemy.sql.functions), 602
array_agg() (in module sqlalchemy.dialects.postgresql), 907
array_for_multi_clause (sqlalchemy.sql.functions.OrderedSetAgg attribute), 602
array_for_multi_clause (sqlalchemy.sql.functions.percentile_cont attribute), 606
array_for_multi_clause (sqlalchemy.sql.functions.percentile_disc attribute), 606
as_declarative() (in module sqlalchemy.ext.declarative), 386
as_mutable() (sqlalchemy.ext.mutable.Mutable class method), 398
as_scalar() (sqlalchemy.orm.query.Query method), 159
as_scalar() (sqlalchemy.sql.expression.CompoundSelect method), 517
as_scalar() (sqlalchemy.sql.expression.GenerativeSelect method), 532
as_scalar() (sqlalchemy.sql.expression.Select method), 547
as_scalar() (sqlalchemy.sql.expression.SelectBase method), 560
asc() (in module sqlalchemy.sql.expression), 460
asc() (sqlalchemy.orm.attributes.QueryableAttribute method), 325
asc() (sqlalchemy.orm.interfaces.PropComparator method), 308
asc() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 294
asc() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 316
asc() (sqlalchemy.schema.Column method), 623
asc() (sqlalchemy.sql.expression.ColumnElement method), 484
asc() (sqlalchemy.sql.operators.ColumnOperators method), 494
AssertionPool (class in sqlalchemy.pool), 787
associate_with() (sqlalchemy.ext.mutable.Mutable class method), 399
associate_with_attribute() (sqlalchemy.ext.mutable.Mutable class method), 399
ASSOCIATION_PROXY (in module sqlalchemy.ext.associationproxy), 347
association_proxy() (in module sqlalchemy.ext.associationproxy), 344
AssociationProxy (class in sqlalchemy.ext.associationproxy), 345
astext (sqlalchemy.dialects.postgresql.JSON.Comparator attribute), 913

attr (sqlalchemy.ext.associationproxy.AssociationProxy attribute), 346
attribute_instrument() (sqlalchemy.orm.events.InstrumentationEvents method), 289
attribute_mapped_collection() (in module sqlalchemy.orm.collections), 118
AttributeEvents (class in sqlalchemy.orm.events), 258
AttributeExtension (class in sqlalchemy.orm.interfaces), 335
AttributeState (class in sqlalchemy.orm.state), 290
attrs (sqlalchemy.orm.mapper.Mapper attribute), 84
attrs (sqlalchemy.orm.state.InstanceState attribute), 302
autocommit() (sqlalchemy.sql.expression.CompoundSelect method), 517
autocommit() (sqlalchemy.sql.expression.GenerativeSelect method), 532
autocommit() (sqlalchemy.sql.expression.Select method), 548
autocommit() (sqlalchemy.sql.expression.SelectBase method), 560
autoflush() (sqlalchemy.orm.query.Query method), 159
automap_base() (in module sqlalchemy.ext.automap), 355
AutomapBase (class in sqlalchemy.ext.automap), 355

B

backref() (in module sqlalchemy.orm), 139
bake_lazy_loaders() (in module sqlalchemy.ext.baked), 365
baked_lazyload() (in module sqlalchemy.ext.baked), 365
baked_lazyload() (sqlalchemy.orm.strategy_options.Load method), 187
baked_lazyload_all() (in module sqlalchemy.ext.baked), 365
BakedQuery (class in sqlalchemy.ext.baked), 364
bakery() (in module sqlalchemy.ext.baked), 364
bakery() (sqlalchemy.ext.baked.BakedQuery class method), 364
base_columns (sqlalchemy.sql.expression.ColumnElement attribute), 484
base_mapper (sqlalchemy.orm.mapper.Mapper attribute), 84
before_attach() (sqlalchemy.orm.events.SessionEvents method), 282
before_commit() (sqlalchemy.orm.events.SessionEvents method), 282
before_commit() (sqlalchemy.orm.interfaces.SessionExtension method), 335
before_compile() (sqlalchemy.orm.events.QueryEvents method), 288
before_configured() (sqlalchemy.orm.events.MapperEvents method), 266
before_create() (sqlalchemy.events.DDLEvents method), 810
before_cursor_execute() (sqlalchemy.events.ConnectionEvents method), 796
before_delete() (sqlalchemy.orm.events.MapperEvents method), 267
before_delete() (sqlalchemy.orm.interfaces.MapperExtension method), 332
before_drop() (sqlalchemy.events.DDLEvents method), 811
before_execute() (sqlalchemy.events.ConnectionEvents method), 797
before_flush() (sqlalchemy.orm.events.SessionEvents method), 282
before_flush() (sqlalchemy.orm.interfaces.SessionExtension method), 335
before_insert() (sqlalchemy.orm.events.MapperEvents method), 267
before_insert() (sqlalchemy.orm.interfaces.MapperExtension method), 332
before_parent_attach() (sqlalchemy.events.DDLEvents method), 811
before_update() (sqlalchemy.orm.events.MapperEvents method), 268
before_update() (sqlalchemy.orm.interfaces.MapperExtension method), 333
begin() (sqlalchemy.engine.Connection method), 758
begin() (sqlalchemy.engine.Engine method), 767
begin() (sqlalchemy.events.ConnectionEvents method), 798
begin() (sqlalchemy.interfaces.ConnectionProxy method), 819
begin() (sqlalchemy.orm.session.Session method), 241
begin_nested() (sqlalchemy.engine.Connection method), 759
begin_nested() (sqlalchemy.orm.session.Session method), 241
begin_twophase() (sqlalchemy.engine.Connection method), 759
begin_twophase() (sqlalchemy.events.ConnectionEvents method), 798
begin_twophase() (sqlalchemy.interfaces.ConnectionProxy method), 819
between() (in module sqlalchemy.sql.expression), 461
between() (sqlalchemy.orm.attributes.QueryableAttribute method), 325
between() (sqlalchemy.orm.interfaces.PropComparator method), 309
between() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 294
between() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 316
between() (sqlalchemy.schema.Column method), 623
between() (sqlalchemy.sql.expression.ColumnElement method), 484

between() (sqlalchemy.sql.operators.ColumnOperators method), 494
BFILE (class in sqlalchemy.dialects.oracle), 887
BIGINT (class in sqlalchemy.dialects.mysql), 868
BIGINT (class in sqlalchemy.types), 714
BigInteger (class in sqlalchemy.types), 704
BINARY (class in sqlalchemy.dialects.mysql), 869
BINARY (class in sqlalchemy.types), 715
BinaryExpression (class in sqlalchemy.sql.expression), 477
bind (sqlalchemy.schema.DDLElement attribute), 697
bind (sqlalchemy.schema.Index attribute), 691
bind (sqlalchemy.schema.MetaData attribute), 630
bind (sqlalchemy.schema.Table attribute), 638
bind (sqlalchemy.schema.ThreadLocalMetaData attribute), 644
bind (sqlalchemy.sql.expression.ColumnElement attribute), 484
bind (sqlalchemy.sql.expression.Delete attribute), 570
bind (sqlalchemy.sql.expression.Executable attribute), 528
bind (sqlalchemy.sql.expression.GenerativeSelect attribute), 532
bind (sqlalchemy.sql.expression.Insert attribute), 575
bind (sqlalchemy.sql.expression.Update attribute), 584
bind (sqlalchemy.sql.expression.UpdateBase attribute), 591
bind (sqlalchemy.types.SchemaType attribute), 709
bind_expression() (sqlalchemy.types.TypeDecorator method), 722
bind_expression() (sqlalchemy.types.TypeEngine method), 734
bind_mapper() (sqlalchemy.orm.session.Session method), 241
bind_processor() (sqlalchemy.types.TypeDecorator method), 722
bind_processor() (sqlalchemy.types.TypeEngine method), 735
bind_table() (sqlalchemy.orm.session.Session method), 241
bindparam() (in module sqlalchemy.sql.expression), 462
BindParameter (class in sqlalchemy.sql.expression), 478
bindparams() (sqlalchemy.sql.expression.TextClause method), 501
BIT (class in sqlalchemy.dialects.mssql), 853
BIT (class in sqlalchemy.dialects.mysql), 869
BIT (class in sqlalchemy.dialects.postgresql), 907
BLOB (class in sqlalchemy.dialects.mysql), 869
BLOB (class in sqlalchemy.types), 715
BOOLEAN (class in sqlalchemy.dialects.mysql), 869
BOOLEAN (class in sqlalchemy.types), 715
Boolean (class in sqlalchemy.types), 704
bulk_insert_mappings() (sqlalchemy.orm.session.Session method), 241
bulk_replace() (in module sqlalchemy.orm.collections), 125
bulk_save_objects() (sqlalchemy.orm.session.Session method), 242
bulk_update_mappings() (sqlalchemy.orm.session.Session method), 243
Bundle (class in sqlalchemy.orm.query), 184
BYTEA (class in sqlalchemy.dialects.postgresql), 907

C

c (sqlalchemy.orm.mapper.Mapper attribute), 85
c (sqlalchemy.orm.query.Bundle attribute), 185
c (sqlalchemy.schema.Table attribute), 638
c (sqlalchemy.sql.expression.Alias attribute), 512
c (sqlalchemy.sql.expression.CompoundSelect attribute), 517
c (sqlalchemy.sql.expression.CTE attribute), 524
c (sqlalchemy.sql.expression.FromClause attribute), 529
c (sqlalchemy.sql.expression.GenerativeSelect attribute), 532
c (sqlalchemy.sql.expression.Join attribute), 542
c (sqlalchemy.sql.expression.Select attribute), 548
c (sqlalchemy.sql.expression.TableClause attribute), 563
callable_ (sqlalchemy.schema.DDLElement attribute), 697
callables (sqlalchemy.orm.state.InstanceState attribute), 302
cascade (sqlalchemy.orm.interfaces.MapperProperty attribute), 305
cascade (sqlalchemy.orm.properties.RelationshipProperty attribute), 322
cascade_iterator() (sqlalchemy.orm.descriptor_props.SynonymProperty method), 323
cascade_iterator() (sqlalchemy.orm.interfaces.MapperProperty method), 305
cascade_iterator() (sqlalchemy.orm.mapper.Mapper method), 85
cascade_iterator() (sqlalchemy.orm.properties.ColumnProperty method), 298
CascadeOptions (class in sqlalchemy.orm.util), 291
cascading (sqlalchemy.ext.declarative.declared_attr attribute), 387
Case (class in sqlalchemy.sql.expression), 478
case() (in module sqlalchemy.sql.expression), 464
Cast (class in sqlalchemy.sql.expression), 479
cast() (in module sqlalchemy.sql.expression), 466
cast() (sqlalchemy.schema.Column method), 623
cast() (sqlalchemy.sql.expression.ColumnElement method), 484
chained_exception (sqlalchemy.engine.ExceptionContext attribute), 771
changed() (sqlalchemy.ext.mutable.Mutable method), 399

changed() (sqlalchemy.ext.mutable.MutableComposite method), 400
CHAR (class in sqlalchemy.dialects.mssql), 853
CHAR (class in sqlalchemy.dialects.mysql), 870
CHAR (class in sqlalchemy.types), 715
char_length (class in sqlalchemy.sql.functions), 603
check_modified() (sqlalchemy.orm.identity.IdentityMap method), 300
CheckConstraint (class in sqlalchemy.schema), 673
checkin() (sqlalchemy.events.PoolEvents method), 790
checkin() (sqlalchemy.interfaces.PoolListener method), 820
checkout() (sqlalchemy.events.PoolEvents method), 791
checkout() (sqlalchemy.interfaces.PoolListener method), 820
CIDR (class in sqlalchemy.dialects.postgresql), 907
CircularDependencyError, 821
class_ (sqlalchemy.orm.mapper.Mapper attribute), 85
class_attribute (sqlalchemy.orm.descriptor_props.SynonymProperty attribute), 323
class_attribute (sqlalchemy.orm.interfaces.MapperProperty attribute), 305
class_attribute (sqlalchemy.orm.properties.ColumnProperty attribute), 298
class_attribute (sqlalchemy.orm.properties.RelationshipProperty attribute), 322
class_instrument() (sqlalchemy.orm.events.InstrumentationEvents method), 289
class_manager (sqlalchemy.orm.mapper.Mapper attribute), 85
class_mapper() (in module sqlalchemy.orm), 81
class_uninstrument() (sqlalchemy.orm.events.InstrumentationEvents method), 290
classes (sqlalchemy.ext.automap.AutomapBase attribute), 356
ClassManager (class in sqlalchemy.orm.instrumentation), 291
classname_for_table() (in module sqlalchemy.ext.automap), 356
ClauseElement (class in sqlalchemy.sql.expression), 479
ClauseList (class in sqlalchemy.sql.expression), 481
clauses (sqlalchemy.sql.functions.FunctionElement attribute), 599
clear() (sqlalchemy.ext.mutable.MutableDict method), 400
clear() (sqlalchemy.orm.collections.MappedCollection method), 125
clear() (sqlalchemy.orm.instrumentation.ClassManager method), 291
clear() (sqlalchemy.schema.MetaData method), 631
clear() (sqlalchemy.util.ScopedRegistry method), 231
clear_managers() (in module sqlalchemy.pool), 790
clear_mappers() (in module sqlalchemy.orm), 81
CLOB (class in sqlalchemy.types), 715
close() (sqlalchemy.engine.Connection method), 759
close() (sqlalchemy.engine.ResultProxy method), 773
close() (sqlalchemy.engine.Transaction method), 777
close() (sqlalchemy.orm.session.Session method), 243
close_all() (sqlalchemy.orm.session.Session method), 244
close_all() (sqlalchemy.orm.session.sessionmaker method), 238
closed (sqlalchemy.engine.Connection attribute), 759
coalesce (class in sqlalchemy.sql.functions), 603
coerce() (sqlalchemy.ext.mutable.Mutable method), 399
coerce() (sqlalchemy.ext.mutable.MutableBase class method), 397
coerce() (sqlalchemy.ext.mutable.MutableDict class method), 400
coerce_arguments (sqlalchemy.sql.functions.GenericFunction attribute), 602
coerce_compared_value() (sqlalchemy.types.Interval method), 707
coerce_compared_value() (sqlalchemy.types.TypeDecorator method), 722
coerce_compared_value() (sqlalchemy.types.TypeEngine method), 735
coerce_compared_value() (sqlalchemy.types.UserDefinedType method), 734
coerce_to_is_types (sqlalchemy.types.TypeDecorator attribute), 722
collate() (in module sqlalchemy.sql.expression), 468
collate() (sqlalchemy.orm.attributes.QueryableAttribute method), 325
collate() (sqlalchemy.orm.interfaces.PropComparator method), 309
collate() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 294
collate() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 317
collate() (sqlalchemy.schema.Column method), 623
collate() (sqlalchemy.sql.expression.ColumnElement method), 484
collate() (sqlalchemy.sql.operators.ColumnOperators method), 494
collection (class in sqlalchemy.orm.collections), 120, 126
collection_adapter (in module sqlalchemy.orm.collections), 126
CollectionAdapter (class in sqlalchemy.orm.collections), 126
Column (class in sqlalchemy.schema), 619
column (sqlalchemy.schema.ForeignKey attribute), 677
column() (in module sqlalchemy.sql.expression), 467
column() (sqlalchemy.sql.expression.Select method), 548
column_attrs (sqlalchemy.orm.mapper.Mapper attribute), 85

column_descriptions (sqlalchemy.orm.query.Query attribute), 159
column_expression() (sqlalchemy.types.TypeDecorator method), 723
column_expression() (sqlalchemy.types.TypeEngine method), 735
column_keys (sqlalchemy.schema.ForeignKeyConstraint attribute), 680
column_mapped_collection() (in module sqlalchemy.orm.collections), 118
column_property() (in module sqlalchemy.orm), 43
column_reflect() (sqlalchemy.events.DDLEvents method), 812
ColumnClause (class in sqlalchemy.sql.expression), 481
ColumnCollection (class in sqlalchemy.sql.expression), 482
ColumnCollectionConstraint (class in sqlalchemy.schema), 671
ColumnCollectionMixin (class in sqlalchemy.schema), 671
ColumnDefault (class in sqlalchemy.schema), 656
ColumnElement (class in sqlalchemy.sql.expression), 482
ColumnOperators (class in sqlalchemy.sql.operators), 490
ColumnProperty (class in sqlalchemy.orm.properties), 292
ColumnProperty.Comparator (class in sqlalchemy.orm.properties), 292
columns (sqlalchemy.orm.mapper.Mapper attribute), 85
columns (sqlalchemy.orm.query.Bundle attribute), 185
columns (sqlalchemy.schema.ColumnCollectionMixin attribute), 671
columns (sqlalchemy.schema.Table attribute), 638
columns (sqlalchemy.sql.expression.Alias attribute), 512
columns (sqlalchemy.sql.expression.CompoundSelect attribute), 517
columns (sqlalchemy.sql.expression.CTE attribute), 524
columns (sqlalchemy.sql.expression.FromClause attribute), 529
columns (sqlalchemy.sql.expression.GenerativeSelect attribute), 532
columns (sqlalchemy.sql.expression.Join attribute), 542
columns (sqlalchemy.sql.expression.Select attribute), 548
columns (sqlalchemy.sql.expression.TableClause attribute), 563
columns (sqlalchemy.sql.functions.FunctionElement attribute), 599
columns() (sqlalchemy.sql.expression.TextClause method), 502
commit() (sqlalchemy.engine.Transaction method), 777
commit() (sqlalchemy.events.ConnectionEvents method), 799
commit() (sqlalchemy.interfaces.ConnectionProxy method), 819
commit() (sqlalchemy.orm.session.Session method), 244
commit_twophase() (sqlalchemy.events.ConnectionEvents method), 799
commit_twophase() (sqlalchemy.interfaces.ConnectionProxy method), 819
common_parent() (sqlalchemy.orm.mapper.Mapper method), 85
comparable_using() (in module sqlalchemy.ext.declarative), 388
ComparableProperty (class in sqlalchemy.orm.properties), 299
Comparator (class in sqlalchemy.ext.hybrid), 415
comparator (sqlalchemy.sql.expression.ColumnElement attribute), 484
comparator() (sqlalchemy.ext.hybrid.hybrid_property method), 415
comparator_factory (sqlalchemy.dialects.postgresql.HSTORE attribute), 911
comparator_factory (sqlalchemy.dialects.postgresql.JSON attribute), 914
comparator_factory (sqlalchemy.dialects.postgresql.JSONB attribute), 915
comparator_factory (sqlalchemy.types.ARRAY attribute), 714
comparator_factory (sqlalchemy.types.JSON attribute), 717
comparator_factory (sqlalchemy.types.TypeEngine attribute), 735
compare() (sqlalchemy.schema.Column method), 623
compare() (sqlalchemy.schema.Table method), 638
compare() (sqlalchemy.sql.expression.Alias method), 513
compare() (sqlalchemy.sql.expression.BinaryExpression method), 478
compare() (sqlalchemy.sql.expression.BindParameter method), 478
compare() (sqlalchemy.sql.expression.ClauseElement method), 479
compare() (sqlalchemy.sql.expression.ClauseList method), 481
compare() (sqlalchemy.sql.expression.ColumnElement method), 484
compare() (sqlalchemy.sql.expression.CompoundSelect method), 517
compare() (sqlalchemy.sql.expression.CTE method), 525
compare() (sqlalchemy.sql.expression.Delete method), 570
compare() (sqlalchemy.sql.expression.GenerativeSelect method), 533
compare() (sqlalchemy.sql.expression.Insert method), 575
compare() (sqlalchemy.sql.expression.Join method), 543
compare() (sqlalchemy.sql.expression.Select method), 548
compare() (sqlalchemy.sql.expression.TableClause method), 563

compare() (sqlalchemy.sql.expression.UnaryExpression method), 507
compare() (sqlalchemy.sql.expression.Update method), 584
compare() (sqlalchemy.sql.expression.UpdateBase method), 591
compare_against_backend() (sqlalchemy.types.TypeDecorator method), 723
compare_against_backend() (sqlalchemy.types.TypeEngine method), 736
compare_values() (sqlalchemy.types.TypeDecorator method), 723
compare_values() (sqlalchemy.types.TypeEngine method), 736
compile() (sqlalchemy.engine.interfaces.Compiled method), 824
compile() (sqlalchemy.schema.Column method), 624
compile() (sqlalchemy.schema.Table method), 638
compile() (sqlalchemy.sql.compiler.DDLCompiler method), 825
compile() (sqlalchemy.sql.expression.Alias method), 513
compile() (sqlalchemy.sql.expression.ClauseElement method), 479
compile() (sqlalchemy.sql.expression.ColumnElement method), 484
compile() (sqlalchemy.sql.expression.CompoundSelect method), 517
compile() (sqlalchemy.sql.expression.CTE method), 525
compile() (sqlalchemy.sql.expression.Delete method), 571
compile() (sqlalchemy.sql.expression.GenerativeSelect method), 533
compile() (sqlalchemy.sql.expression.Insert method), 576
compile() (sqlalchemy.sql.expression.Join method), 543
compile() (sqlalchemy.sql.expression.Select method), 548
compile() (sqlalchemy.sql.expression.TableClause method), 563
compile() (sqlalchemy.sql.expression.Update method), 584
compile() (sqlalchemy.sql.expression.UpdateBase method), 591
compile() (sqlalchemy.types.TypeDecorator method), 723
compile() (sqlalchemy.types.TypeEngine method), 736
Compiled (class in sqlalchemy.engine.interfaces), 823
CompileError, 821
compiles() (in module sqlalchemy.ext.compiler), 613
composite() (in module sqlalchemy.orm), 55
CompositeProperty (class in sqlalchemy.orm.descriptor_props), 299
CompositeProperty.Comparator (class in sqlalchemy.orm.descriptor_props), 299
composites (sqlalchemy.orm.mapper.Mapper attribute), 85
CompoundSelect (class in sqlalchemy.sql.expression), 516
concat (class in sqlalchemy.sql.functions), 603
concat() (sqlalchemy.orm.attributes.QueryableAttribute method), 326
concat() (sqlalchemy.orm.interfaces.PropComparator method), 309
concat() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 294
concat() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 317
concat() (sqlalchemy.schema.Column method), 625
concat() (sqlalchemy.sql.expression.ColumnElement method), 485
concat() (sqlalchemy.sql.operators.ColumnOperators method), 494
Concatenable (class in sqlalchemy.types), 738
concrete (sqlalchemy.orm.mapper.Mapper attribute), 86
ConcreteBase (class in sqlalchemy.ext.declarative), 389
ConcurrentModificationError (in module sqlalchemy.orm.exc), 330
configure() (sqlalchemy.orm.scoping.scoped_session method), 230
configure() (sqlalchemy.orm.session.sessionmaker method), 238
configure_mappers() (in module sqlalchemy.orm), 81
configured (sqlalchemy.orm.mapper.Mapper attribute), 86
connect() (sqlalchemy.engine.Connectable method), 765
connect() (sqlalchemy.engine.Connection method), 759
connect() (sqlalchemy.engine.Engine method), 768
connect() (sqlalchemy.engine.interfaces.Dialect method), 832
connect() (sqlalchemy.events.PoolEvents method), 791
connect() (sqlalchemy.interfaces.PoolListener method), 820
connect() (sqlalchemy.pool.Pool method), 785
connect() (sqlalchemy.pool.QueuePool method), 786
Connectable (class in sqlalchemy.engine), 765
Connection (class in sqlalchemy.engine), 758
connection (sqlalchemy.engine.Connection attribute), 759
connection (sqlalchemy.engine.ExceptionContext attribute), 771
connection (sqlalchemy.pool._ConnectionFairy attribute), 787
connection (sqlalchemy.pool._ConnectionRecord attribute), 788
connection() (sqlalchemy.orm.session.Session method), 244
ConnectionEvents (class in sqlalchemy.events), 794
ConnectionProxy (class in sqlalchemy.interfaces), 818
Constraint (class in sqlalchemy.schema), 670

construct_arguments (sqlalchemy.engine.default.DefaultDialect attribute), 825
construct_params() (sqlalchemy.engine.interfaces.Compiled method), 824
construct_params() (sqlalchemy.sql.compiler.SQLCompiler method), 840
contained_by() (sqlalchemy.dialects.postgresql.ARRAY.Comparator method), 906
contained_by() (sqlalchemy.dialects.postgresql.HSTORE.Comparator method), 911
contained_by() (sqlalchemy.dialects.postgresql.JSONB.Comparator method), 914
contained_by() (sqlalchemy.dialects.postgresql.ranges.RangeOperators.comparator_factory method), 917
contains() (in module sqlalchemy.event), 817
contains() (sqlalchemy.dialects.postgresql.ARRAY.Comparator method), 906
contains() (sqlalchemy.dialects.postgresql.HSTORE.Comparator method), 911
contains() (sqlalchemy.dialects.postgresql.JSONB.Comparator method), 914
contains() (sqlalchemy.dialects.postgresql.ranges.RangeOperators.comparator_factory method), 917
contains() (sqlalchemy.ext.associationproxy.AssociationProxy method), 346
contains() (sqlalchemy.orm.attributes.QueryableAttribute method), 326
contains() (sqlalchemy.orm.interfaces.PropComparator method), 309
contains() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 294
contains() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 317
contains() (sqlalchemy.schema.Column method), 625
contains() (sqlalchemy.sql.expression.ColumnElement method), 485
contains() (sqlalchemy.sql.operators.ColumnOperators method), 494
contains_alias() (in module sqlalchemy.orm), 154
contains_column() (sqlalchemy.schema.CheckConstraint method), 674
contains_column() (sqlalchemy.schema.ColumnCollectionConstraint method), 672
contains_column() (sqlalchemy.schema.ForeignKeyConstraint method), 680
contains_column() (sqlalchemy.schema.PrimaryKeyConstraint method), 683
contains_column() (sqlalchemy.schema.UniqueConstraint method), 686
contains_eager() (in module sqlalchemy.orm), 154
contains_eager() (sqlalchemy.orm.strategy_options.Load method), 187
contextual_connect() (sqlalchemy.engine.Connectable method), 765
contextual_connect() (sqlalchemy.engine.Connection method), 760
contextual_connect() (sqlalchemy.engine.Engine method), 768
conv() (in module sqlalchemy.schema), 687
converter() (sqlalchemy.orm.collections.collection static method), 122
copy() (sqlalchemy.orm.instrumentation.ClassManager method), 291
copy() (sqlalchemy.schema.Column method), 625
copy() (sqlalchemy.schema.ForeignKey method), 677
copy() (sqlalchemy.types.SchemaType method), 709
copy() (sqlalchemy.types.TypeDecorator method), 724
correlate() (sqlalchemy.orm.query.Query method), 160
correlate() (sqlalchemy.sql.expression.Select method), 549
correlate_except() (sqlalchemy.sql.expression.Select method), 550
correspond_on_equivalents() (sqlalchemy.schema.Table method), 639
correspond_on_equivalents() (sqlalchemy.sql.expression.Alias method), 514
correspond_on_equivalents() (sqlalchemy.sql.expression.CompoundSelect method), 518
correspond_on_equivalents() (sqlalchemy.sql.expression.CTE method), 526
correspond_on_equivalents() (sqlalchemy.sql.expression.FromClause method), 529
correspond_on_equivalents() (sqlalchemy.sql.expression.GenerativeSelect method), 533
correspond_on_equivalents() (sqlalchemy.sql.expression.Join method), 543
correspond_on_equivalents() (sqlalchemy.sql.expression.Select method), 550
correspond_on_equivalents() (sqlalchemy.sql.expression.TableClause method), 564
corresponding_column() (sqlalchemy.schema.Table method), 639
corresponding_column() (sqlalchemy.sql.expression.Alias method), 514
corresponding_column() (sqlalchemy.sql.expression.CompoundSelect method), 518
corresponding_column() (sqlalchemy.sql.expression.CTE method), 526
corresponding_column() (sqlalchemy.sql.expression.FromClause method), 529

corresponding_column() (sqlalchemy.sql.expression.GenerativeSelect method), 534
corresponding_column() (sqlalchemy.sql.expression.Join method), 544
corresponding_column() (sqlalchemy.sql.expression.Select method), 550
corresponding_column() (sqlalchemy.sql.expression.TableClause method), 564
count (class in sqlalchemy.sql.functions), 603
count() (sqlalchemy.orm.query.Query method), 160
count() (sqlalchemy.schema.Table method), 639
count() (sqlalchemy.sql.expression.Alias method), 514
count() (sqlalchemy.sql.expression.CompoundSelect method), 518
count() (sqlalchemy.sql.expression.CTE method), 526
count() (sqlalchemy.sql.expression.FromClause method), 530
count() (sqlalchemy.sql.expression.GenerativeSelect method), 534
count() (sqlalchemy.sql.expression.Join method), 544
count() (sqlalchemy.sql.expression.Select method), 550
count() (sqlalchemy.sql.expression.TableClause method), 564
count_from_0() (in module sqlalchemy.ext.orderinglist), 402
count_from_1() (in module sqlalchemy.ext.orderinglist), 402
count_from_n_factory() (in module sqlalchemy.ext.orderinglist), 402
create() (sqlalchemy.dialects.postgresql.ENUM method), 909
create() (sqlalchemy.engine.Connectable method), 765
create() (sqlalchemy.schema.Index method), 691
create() (sqlalchemy.schema.Sequence method), 659
create() (sqlalchemy.schema.Table method), 639
create() (sqlalchemy.types.Enum method), 705
create() (sqlalchemy.types.SchemaType method), 709
create_all() (sqlalchemy.schema.MetaData method), 631
create_connect_args() (sqlalchemy.engine.interfaces.Dialect method), 832
create_cursor() (sqlalchemy.engine.interfaces.ExecutionContext method), 838
create_engine() (in module sqlalchemy), 742
create_row_processor() (sqlalchemy.orm.descriptor_props.SynonymProperty method), 323
create_row_processor() (sqlalchemy.orm.interfaces.MapperProperty method), 305
create_row_processor() (sqlalchemy.orm.query.Bundle method), 185
create_xid() (sqlalchemy.engine.default.DefaultDialect method), 826
create_xid() (sqlalchemy.engine.interfaces.Dialect method), 832
CreateColumn (class in sqlalchemy.schema), 701
CreateEnginePlugin (class in sqlalchemy.engine), 766
CreateIndex (class in sqlalchemy.schema), 702
CreateSchema (class in sqlalchemy.schema), 703
CreateSequence (class in sqlalchemy.schema), 702
CreateTable (class in sqlalchemy.schema), 700
CTE (class in sqlalchemy.sql.expression), 524
cte() (sqlalchemy.orm.query.Query method), 160
cte() (sqlalchemy.sql.expression.CompoundSelect method), 518
cte() (sqlalchemy.sql.expression.GenerativeSelect method), 534
cte() (sqlalchemy.sql.expression.Select method), 550
cte() (sqlalchemy.sql.expression.SelectBase method), 560
cume_dist (class in sqlalchemy.sql.functions), 603
current_date (class in sqlalchemy.sql.functions), 603
current_time (class in sqlalchemy.sql.functions), 604
current_timestamp (class in sqlalchemy.sql.functions), 604
current_user (class in sqlalchemy.sql.functions), 604
cursor (sqlalchemy.engine.ExceptionContext attribute), 771
cursor() (sqlalchemy.pool._ConnectionFairy method), 787
cursor_execute() (sqlalchemy.interfaces.ConnectionProxy method), 819
custom_op (class in sqlalchemy.sql.operators), 504

D

DatabaseError, 821
DataError, 821
DATE (class in sqlalchemy.dialects.mysql), 870
DATE (class in sqlalchemy.dialects.oracle), 887
DATE (class in sqlalchemy.dialects.sqlite), 933
DATE (class in sqlalchemy.types), 715
Date (class in sqlalchemy.types), 704
DATERANGE (class in sqlalchemy.dialects.postgresql), 916
DATETIME (class in sqlalchemy.dialects.mysql), 870
DATETIME (class in sqlalchemy.dialects.sqlite), 932
DATETIME (class in sqlalchemy.types), 715
DateTime (class in sqlalchemy.types), 704
DATETIME2 (class in sqlalchemy.dialects.mssql), 854
DATETIMEOFFSET (class in sqlalchemy.dialects.mssql), 854
dbapi_error() (sqlalchemy.events.ConnectionEvents method), 799
dbapi_exception_translation_map (sqlalchemy.engine.default.DefaultDialect attribute), 826
DBAPIError, 821
DDL (class in sqlalchemy.schema), 699
DDLCompiler (class in sqlalchemy.sql.compiler), 824
DDLElement (class in sqlalchemy.schema), 697
DDLEvents (class in sqlalchemy.events), 808

DECIMAL (class in sqlalchemy.dialects.mysql), 870
DECIMAL (class in sqlalchemy.types), 715
declarative_base() (in module sqlalchemy.ext.declarative), 385
declared_attr (class in sqlalchemy.ext.declarative), 386
default_from() (sqlalchemy.sql.compiler.SQLCompiler method), 840
default_isolation_level (sqlalchemy.engine.Connection attribute), 760
default_schema_name (sqlalchemy.engine.reflection.Inspector attribute), 647
DefaultClause (class in sqlalchemy.schema), 657
DefaultDialect (class in sqlalchemy.engine.default), 825
DefaultExecutionContext (class in sqlalchemy.engine.default), 837
DefaultGenerator (class in sqlalchemy.schema), 657
defaultload() (in module sqlalchemy.orm), 155
defaultload() (sqlalchemy.orm.strategy_options.Load method), 187
defer() (in module sqlalchemy.orm), 142
defer() (sqlalchemy.orm.strategy_options.Load method), 187
deferred() (in module sqlalchemy.orm), 142
DeferredReflection (class in sqlalchemy.ext.declarative), 390
define_constraint_remote_table() (sqlalchemy.sql.compiler.DDLCompiler method), 825
defined() (sqlalchemy.dialects.postgresql.HSTORE.Comparator method), 911
del_attribute() (in module sqlalchemy.orm.attributes), 256
Delete (class in sqlalchemy.sql.expression), 570
delete() (in module sqlalchemy.sql.expression), 567
delete() (sqlalchemy.dialects.postgresql.HSTORE.Comparator method), 911
delete() (sqlalchemy.orm.query.Query method), 161
delete() (sqlalchemy.orm.session.Session method), 245
delete() (sqlalchemy.schema.Table method), 639
delete() (sqlalchemy.sql.expression.TableClause method), 564
deleted (sqlalchemy.orm.session.Session attribute), 245
deleted (sqlalchemy.orm.state.InstanceState attribute), 302
deleted_to_detached() (sqlalchemy.orm.events.SessionEvents method), 283
deleted_to_persistent() (sqlalchemy.orm.events.SessionEvents method), 283
deleter() (sqlalchemy.ext.hybrid.hybrid_property method), 415
denormalize_name() (sqlalchemy.engine.default.DefaultDialect method), 826
denormalize_name() (sqlalchemy.engine.interfaces.Dialect method), 832
dense_rank (class in sqlalchemy.sql.functions), 604
deregister() (in module sqlalchemy.ext.compiler), 613
desc() (in module sqlalchemy.sql.expression), 468
desc() (sqlalchemy.orm.attributes.QueryableAttribute method), 326
desc() (sqlalchemy.orm.interfaces.PropComparator method), 309
desc() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 294
desc() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 317
desc() (sqlalchemy.schema.Column method), 625
desc() (sqlalchemy.sql.expression.ColumnElement method), 485
desc() (sqlalchemy.sql.operators.ColumnOperators method), 494
description (sqlalchemy.sql.expression.ColumnElement attribute), 485
description (sqlalchemy.sql.expression.CompoundSelect attribute), 520
description (sqlalchemy.sql.expression.FromClause attribute), 530
description (sqlalchemy.sql.expression.GenerativeSelect attribute), 536
description (sqlalchemy.sql.expression.Select attribute), 552
Deserializer() (in module sqlalchemy.ext.serializer), 614
detach() (sqlalchemy.engine.Connection method), 760
detach() (sqlalchemy.pool._ConnectionFairy method), 787
detached (sqlalchemy.orm.state.InstanceState attribute), 302
detached_to_persistent() (sqlalchemy.orm.events.SessionEvents method), 284
DetachedInstanceError, 330
Dialect (class in sqlalchemy.engine.interfaces), 830
dialect (sqlalchemy.schema.DDLElement attribute), 697
dialect_impl() (sqlalchemy.types.TypeDecorator method), 724
dialect_impl() (sqlalchemy.types.TypeEngine method), 736
dialect_kwargs (sqlalchemy.schema.CheckConstraint attribute), 674
dialect_kwargs (sqlalchemy.schema.ColumnCollectionConstraint attribute), 672
dialect_kwargs (sqlalchemy.schema.ForeignKey attribute), 677
dialect_kwargs (sqlalchemy.schema.ForeignKeyConstraint attribute), 681
dialect_kwargs (sqlalchemy.schema.Index attribute), 691
dialect_kwargs (sqlalchemy.schema.PrimaryKeyConstraint attribute), 684
dialect_kwargs (sqlalchemy.schema.Table attribute), 639
dialect_kwargs (sqlalchemy.schema.UniqueConstraint attribute), 686

dialect_kwargs (sqlalchemy.sql.base.DialectKWArgs attribute), 498
dialect_kwargs (sqlalchemy.sql.expression.Delete attribute), 571
dialect_kwargs (sqlalchemy.sql.expression.Insert attribute), 576
dialect_kwargs (sqlalchemy.sql.expression.Update attribute), 585
dialect_kwargs (sqlalchemy.sql.expression.UpdateBase attribute), 592
dialect_options (sqlalchemy.schema.CheckConstraint attribute), 674
dialect_options (sqlalchemy.schema.ColumnCollectionConstraint attribute), 672
dialect_options (sqlalchemy.schema.ForeignKey attribute), 678
dialect_options (sqlalchemy.schema.ForeignKeyConstraint attribute), 681
dialect_options (sqlalchemy.schema.Index attribute), 691
dialect_options (sqlalchemy.schema.PrimaryKeyConstraint attribute), 684
dialect_options (sqlalchemy.schema.Table attribute), 640
dialect_options (sqlalchemy.schema.UniqueConstraint attribute), 686
dialect_options (sqlalchemy.sql.base.DialectKWArgs attribute), 499
dialect_options (sqlalchemy.sql.expression.Delete attribute), 572
dialect_options (sqlalchemy.sql.expression.Insert attribute), 577
dialect_options (sqlalchemy.sql.expression.Update attribute), 585
dialect_options (sqlalchemy.sql.expression.UpdateBase attribute), 592
DialectEvents (class in sqlalchemy.events), 806
DialectKWArgs (class in sqlalchemy.sql.base), 498
dict (sqlalchemy.orm.state.InstanceState attribute), 302
dict_getter() (sqlalchemy.ext.instrumentation.InstrumentationManager method), 416
dirty (sqlalchemy.orm.session.Session attribute), 245
DisconnectionError, 821
dispose() (sqlalchemy.engine.Engine method), 768
dispose() (sqlalchemy.ext.instrumentation.InstrumentationManager method), 416
dispose() (sqlalchemy.orm.instrumentation.ClassManager method), 291
dispose() (sqlalchemy.pool.Pool method), 785
dispose() (sqlalchemy.schema.ThreadLocalMetaData method), 644
dispose_collection() (sqlalchemy.orm.events.AttributeEvents method), 259
distinct() (in module sqlalchemy.sql.expression), 468
distinct() (sqlalchemy.orm.attributes.QueryableAttribute method), 326
distinct() (sqlalchemy.orm.interfaces.PropComparator method), 309
distinct() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 294
distinct() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 317
distinct() (sqlalchemy.orm.query.Query method), 162
distinct() (sqlalchemy.schema.Column method), 625
distinct() (sqlalchemy.sql.expression.ColumnElement method), 485
distinct() (sqlalchemy.sql.expression.Select method), 552
distinct() (sqlalchemy.sql.operators.ColumnOperators method), 494
do_begin() (sqlalchemy.engine.interfaces.Dialect method), 832
do_begin_twophase() (sqlalchemy.engine.default.DefaultDialect method), 826
do_begin_twophase() (sqlalchemy.engine.interfaces.Dialect method), 832
do_close() (sqlalchemy.engine.interfaces.Dialect method), 832
do_commit() (sqlalchemy.engine.interfaces.Dialect method), 832
do_commit_twophase() (sqlalchemy.engine.default.DefaultDialect method), 826
do_commit_twophase() (sqlalchemy.engine.interfaces.Dialect method), 833
do_connect() (sqlalchemy.events.DialectEvents method), 807
do_execute() (sqlalchemy.engine.interfaces.Dialect method), 833
do_execute() (sqlalchemy.events.DialectEvents method), 807
do_execute_no_params() (sqlalchemy.engine.interfaces.Dialect method), 833
do_execute_no_params() (sqlalchemy.events.DialectEvents method), 808
do_executemany() (sqlalchemy.engine.interfaces.Dialect method), 833
do_executemany() (sqlalchemy.events.DialectEvents method), 808
do_init() (sqlalchemy.orm.descriptor_props.CompositeProperty method), 300
do_init() (sqlalchemy.orm.descriptor_props.SynonymProperty method), 323
do_init() (sqlalchemy.orm.interfaces.MapperProperty method), 305
do_prepare_twophase() (sqlalchemy.engine.default.DefaultDialect method), 826
do_prepare_twophase() (sqlalchemy.engine.interfaces.Dialect method), 833

do_recover_twophase() (sqlalchemy.engine.default.DefaultDialect method), 827
do_recover_twophase() (sqlalchemy.engine.interfaces.Dialect method), 833
do_release_savepoint() (sqlalchemy.engine.interfaces.Dialect method), 833
do_rollback() (sqlalchemy.engine.interfaces.Dialect method), 833
do_rollback_to_savepoint() (sqlalchemy.engine.interfaces.Dialect method), 833
do_rollback_twophase() (sqlalchemy.engine.default.DefaultDialect method), 827
do_rollback_twophase() (sqlalchemy.engine.interfaces.Dialect method), 834
do_savepoint() (sqlalchemy.engine.interfaces.Dialect method), 834
DontWrapMixin (class in sqlalchemy.exc), 821
DOUBLE (class in sqlalchemy.dialects.mysql), 871
DOUBLE_PRECISION (class in sqlalchemy.dialects.oracle), 888
DOUBLE_PRECISION (class in sqlalchemy.dialects.postgresql), 907
driver (sqlalchemy.engine.Engine attribute), 768
drop() (sqlalchemy.dialects.postgresql.ENUM method), 909
drop() (sqlalchemy.engine.Connectable method), 765
drop() (sqlalchemy.schema.Index method), 692
drop() (sqlalchemy.schema.Sequence method), 659
drop() (sqlalchemy.schema.Table method), 640
drop() (sqlalchemy.types.Enum method), 705
drop() (sqlalchemy.types.SchemaType method), 709
drop_all() (sqlalchemy.schema.MetaData method), 631
DropConstraint (class in sqlalchemy.schema), 703
DropIndex (class in sqlalchemy.schema), 703
DropSchema (class in sqlalchemy.schema), 703
DropSequence (class in sqlalchemy.schema), 702
DropTable (class in sqlalchemy.schema), 701
dumps() (in module sqlalchemy.ext.serializer), 614
dynamic_loader() (in module sqlalchemy.orm), 139

E

eagerload() (in module sqlalchemy.orm), 155
eagerload_all() (in module sqlalchemy.orm), 155
effective_value (sqlalchemy.sql.expression.BindParameter attribute), 478
empty() (sqlalchemy.orm.attributes.History method), 257
enable_assertions() (sqlalchemy.orm.query.Query method), 163
enable_eagerloads() (sqlalchemy.orm.query.Query method), 163
enable_relationship_loading() (sqlalchemy.orm.session.Session method), 245
endswith() (sqlalchemy.orm.attributes.QueryableAttribute method), 326
endswith() (sqlalchemy.orm.interfaces.PropComparator method), 309
endswith() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 294
endswith() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 318
endswith() (sqlalchemy.schema.Column method), 625
endswith() (sqlalchemy.sql.expression.ColumnElement method), 485
endswith() (sqlalchemy.sql.operators.ColumnOperators method), 494
Engine (class in sqlalchemy.engine), 767
engine (sqlalchemy.engine.ExceptionContext attribute), 771
engine_connect() (sqlalchemy.events.ConnectionEvents method), 800
engine_created() (sqlalchemy.engine.CreateEnginePlugin method), 767
engine_created() (sqlalchemy.engine.default.DefaultDialect method), 827
engine_created() (sqlalchemy.engine.interfaces.Dialect class method), 834
engine_disposed() (sqlalchemy.events.ConnectionEvents method), 801
engine_from_config() (in module sqlalchemy), 745
entity (sqlalchemy.orm.mapper.Mapper attribute), 86
ENUM (class in sqlalchemy.dialects.mysql), 871
ENUM (class in sqlalchemy.dialects.postgresql), 908
Enum (class in sqlalchemy.types), 704
escape_literal_column() (sqlalchemy.sql.compiler.SQLCompiler method), 840
evaluates_none() (sqlalchemy.types.TypeDecorator method), 724
evaluates_none() (sqlalchemy.types.TypeEngine method), 736
Event (class in sqlalchemy.orm.attributes), 300
Events (class in sqlalchemy.event.base), 790
examples.adjacency_list (module), 417
examples.association (module), 418
examples.custom_attributes (module), 426
examples.dogpile_caching (module), 427
examples.dynamic_dict (module), 418
examples.elementtree (module), 424
examples.generic_associations (module), 418
examples.graphs (module), 418
examples.inheritance (module), 426
examples.join_conditions (module), 423
examples.large_collection (module), 419
examples.materialized_paths (module), 419
examples.nested_sets (module), 419
examples.performance (module), 419
examples.postgis (module), 428

examples.sharding (module), 427
examples.versioned_history (module), 424
examples.versioned_rows (module), 425
examples.vertical (module), 426
except_() (in module sqlalchemy.sql.expression), 507
except_() (sqlalchemy.orm.query.Query method), 163
except_() (sqlalchemy.sql.expression.Select method), 552
except_all() (in module sqlalchemy.sql.expression), 507
except_all() (sqlalchemy.orm.query.Query method), 163
except_all() (sqlalchemy.sql.expression.Select method), 552
exception (sqlalchemy.engine.interfaces.ExecutionContext attribute), 838
ExceptionContext (class in sqlalchemy.engine), 771
ExcludeConstraint (class in sqlalchemy.dialects.postgresql), 918
Executable (class in sqlalchemy.sql.expression), 528
execute() (sqlalchemy.engine.Connectable method), 766
execute() (sqlalchemy.engine.Connection method), 760
execute() (sqlalchemy.engine.Engine method), 768
execute() (sqlalchemy.engine.interfaces.Compiled method), 824
execute() (sqlalchemy.interfaces.ConnectionProxy method), 819
execute() (sqlalchemy.orm.session.Session method), 246
execute() (sqlalchemy.schema.DDLElement method), 697
execute() (sqlalchemy.sql.compiler.DDLCompiler method), 825
execute() (sqlalchemy.sql.expression.CompoundSelect method), 520
execute() (sqlalchemy.sql.expression.Delete method), 572
execute() (sqlalchemy.sql.expression.Executable method), 528
execute() (sqlalchemy.sql.expression.GenerativeSelect method), 536
execute() (sqlalchemy.sql.expression.Insert method), 577
execute() (sqlalchemy.sql.expression.Select method), 552
execute() (sqlalchemy.sql.expression.Update method), 585
execute() (sqlalchemy.sql.expression.UpdateBase method), 593
execute() (sqlalchemy.sql.functions.FunctionElement method), 600
execute_at() (sqlalchemy.schema.DDLElement method), 697
execute_if() (sqlalchemy.schema.DDLElement method), 698
execute_sequence_format (sqlalchemy.engine.default.DefaultDialect attribute), 827
execution_context (sqlalchemy.engine.ExceptionContext attribute), 772
execution_options() (sqlalchemy.engine.Connection method), 761
execution_options() (sqlalchemy.engine.Engine method), 769
execution_options() (sqlalchemy.orm.query.Query method), 163
execution_options() (sqlalchemy.sql.expression.CompoundSelect method), 520
execution_options() (sqlalchemy.sql.expression.Delete method), 572
execution_options() (sqlalchemy.sql.expression.Executable method), 528
execution_options() (sqlalchemy.sql.expression.GenerativeSelect method), 536
execution_options() (sqlalchemy.sql.expression.Insert method), 577
execution_options() (sqlalchemy.sql.expression.Select method), 552
execution_options() (sqlalchemy.sql.expression.Update method), 585
execution_options() (sqlalchemy.sql.expression.UpdateBase method), 593
ExecutionContext (class in sqlalchemy.engine.interfaces), 838
exists() (in module sqlalchemy.sql.expression), 507
exists() (sqlalchemy.orm.query.Query method), 163
exists() (sqlalchemy.schema.Table method), 640
expire() (sqlalchemy.orm.events.InstanceEvents method), 271
expire() (sqlalchemy.orm.session.Session method), 247
expire_all() (sqlalchemy.orm.session.Session method), 248
expired_attributes (sqlalchemy.orm.state.InstanceState attribute), 302
expression (sqlalchemy.orm.properties.ColumnProperty attribute), 299
expression (sqlalchemy.schema.Column attribute), 625
expression (sqlalchemy.sql.expression.ColumnElement attribute), 486
expression() (sqlalchemy.ext.hybrid.hybrid_method method), 414
expression() (sqlalchemy.ext.hybrid.hybrid_property method), 415
expunge() (sqlalchemy.orm.session.Session method), 248
expunge_all() (sqlalchemy.orm.session.Session method), 248
ExtendedInstrumentationRegistry (class in sqlalchemy.ext.instrumentation), 417
extension_type (sqlalchemy.ext.associationproxy.AssociationProxy attribute), 346
extension_type (sqlalchemy.orm.attributes.QueryableAttribute attribute), 326
extension_type (sqlalchemy.orm.base.InspectionAttr attribute), 300

extension_type (sqlalchemy.orm.descriptor_props.SynonymProperty attribute), 324
extension_type (sqlalchemy.orm.properties.ColumnProperty attribute), 299
extension_type (sqlalchemy.orm.properties.RelationshipProperty attribute), 322
Extract (class in sqlalchemy.sql.expression), 499
extract() (in module sqlalchemy.sql.expression), 469

F

false() (in module sqlalchemy.sql.expression), 469
False_ (class in sqlalchemy.sql.elements), 499
fetchall() (sqlalchemy.engine.ResultProxy method), 774
FetchedValue (class in sqlalchemy.schema), 657
fetchmany() (sqlalchemy.engine.ResultProxy method), 774
fetchone() (sqlalchemy.engine.ResultProxy method), 774
filter() (sqlalchemy.orm.query.Query method), 164
filter() (sqlalchemy.sql.expression.FunctionFilter method), 500
filter() (sqlalchemy.sql.functions.FunctionElement method), 600
filter_by() (sqlalchemy.orm.query.Query method), 164
filter_states_for_dep() (sqlalchemy.orm.session.UOWTransaction method), 330
finalize_flush_changes() (sqlalchemy.orm.session.UOWTransaction method), 330
first() (sqlalchemy.engine.ResultProxy method), 774
first() (sqlalchemy.ext.baked.Result method), 364
first() (sqlalchemy.orm.query.Query method), 164
first_connect() (sqlalchemy.events.PoolEvents method), 792
first_connect() (sqlalchemy.interfaces.PoolListener method), 820
first_init() (sqlalchemy.orm.events.InstanceEvents method), 272
flag_modified() (in module sqlalchemy.orm.attributes), 256
FLOAT (class in sqlalchemy.dialects.mysql), 872
FLOAT (class in sqlalchemy.types), 715
Float (class in sqlalchemy.types), 705
flush() (sqlalchemy.orm.session.Session method), 248
FlushError, 330
for_session() (sqlalchemy.ext.baked.BakedQuery method), 364
for_update (sqlalchemy.sql.expression.CompoundSelect attribute), 521
for_update (sqlalchemy.sql.expression.GenerativeSelect attribute), 536
for_update (sqlalchemy.sql.expression.Select attribute), 553
foreign() (in module sqlalchemy.orm), 139
foreign_key_constraints (sqlalchemy.schema.Table attribute), 640
foreign_keys (sqlalchemy.schema.Table attribute), 640
foreign_keys (sqlalchemy.sql.expression.Alias attribute), 514
foreign_keys (sqlalchemy.sql.expression.ColumnElement attribute), 486
foreign_keys (sqlalchemy.sql.expression.CompoundSelect attribute), 521
foreign_keys (sqlalchemy.sql.expression.CTE attribute), 526
foreign_keys (sqlalchemy.sql.expression.FromClause attribute), 530
foreign_keys (sqlalchemy.sql.expression.GenerativeSelect attribute), 536
foreign_keys (sqlalchemy.sql.expression.Join attribute), 544
foreign_keys (sqlalchemy.sql.expression.Select attribute), 553
foreign_keys (sqlalchemy.sql.expression.TableClause attribute), 565
ForeignKey (class in sqlalchemy.schema), 675
ForeignKeyConstraint (class in sqlalchemy.schema), 678
format_column() (sqlalchemy.sql.compiler.IdentifierPreparer method), 839
format_schema() (sqlalchemy.sql.compiler.IdentifierPreparer method), 839
format_table() (sqlalchemy.sql.compiler.IdentifierPreparer method), 840
format_table_seq() (sqlalchemy.sql.compiler.IdentifierPreparer method), 840
from_engine() (sqlalchemy.engine.reflection.Inspector class method), 647
from_select() (sqlalchemy.sql.expression.Insert method), 577
from_self() (sqlalchemy.orm.query.Query method), 165
from_statement() (sqlalchemy.orm.query.Query method), 167
FromClause (class in sqlalchemy.sql.expression), 529
fromkeys() (sqlalchemy.orm.instrumentation.ClassManager method), 291
froms (sqlalchemy.sql.expression.Select attribute), 553
func (in module sqlalchemy.sql.expression), 469
func (sqlalchemy.sql.expression.Over attribute), 500
funcfilter() (in module sqlalchemy.sql.expression), 470
Function (class in sqlalchemy.sql.functions), 598
FunctionElement (class in sqlalchemy.sql.functions), 599
FunctionFilter (class in sqlalchemy.sql.expression), 499

G

generate_relationship() (in module sqlalchemy.ext.automap), 357
GenerativeSelect (class in sqlalchemy.sql.expression), 531
GenericFunction (class in sqlalchemy.sql.functions), 601
get() (sqlalchemy.ext.baked.Result method), 364

get() (sqlalchemy.orm.instrumentation.ClassManager method), 291
get() (sqlalchemy.orm.query.Query method), 167
get_attribute() (in module sqlalchemy.orm.attributes), 256
get_attribute_history() (sqlalchemy.orm.session.UOWTransaction method), 330
get_bind() (sqlalchemy.orm.session.Session method), 249
get_children() (sqlalchemy.schema.CheckConstraint method), 675
get_children() (sqlalchemy.schema.ColumnCollectionConstraint method), 672
get_children() (sqlalchemy.schema.ForeignKey method), 678
get_children() (sqlalchemy.schema.ForeignKeyConstraint method), 681
get_children() (sqlalchemy.schema.Index method), 692
get_children() (sqlalchemy.schema.PrimaryKeyConstraint method), 684
get_children() (sqlalchemy.schema.SchemaItem method), 633
get_children() (sqlalchemy.schema.UniqueConstraint method), 686
get_children() (sqlalchemy.sql.expression.ClauseElement method), 480
get_children() (sqlalchemy.sql.expression.ColumnElement method), 486
get_children() (sqlalchemy.sql.expression.GenerativeSelect method), 536
get_children() (sqlalchemy.sql.expression.Select method), 553
get_children() (sqlalchemy.sql.expression.UpdateBase method), 593
get_children() (sqlalchemy.sql.functions.FunctionElement method), 600
get_columns() (sqlalchemy.engine.default.DefaultDialect method), 827
get_columns() (sqlalchemy.engine.interfaces.Dialect method), 834
get_columns() (sqlalchemy.engine.reflection.Inspector method), 647
get_dbapi_type() (sqlalchemy.types.TypeDecorator method), 724
get_dbapi_type() (sqlalchemy.types.TypeEngine method), 737
get_dialect() (sqlalchemy.engine.url.URL method), 746
get_dialect_cls() (sqlalchemy.engine.default.DefaultDialect method), 828
get_dialect_cls() (sqlalchemy.engine.interfaces.Dialect class method), 834
get_enums() (sqlalchemy.dialects.postgresql.base.PGInspector method), 901
get_foreign_keys() (sqlalchemy.engine.default.DefaultDialect method), 828
get_foreign_keys() (sqlalchemy.engine.interfaces.Dialect method), 835
get_foreign_keys() (sqlalchemy.engine.reflection.Inspector method), 647
get_foreign_table_names() (sqlalchemy.dialects.postgresql.base.PGInspector method), 901
get_history() (in module sqlalchemy.orm.attributes), 256
get_history() (sqlalchemy.orm.descriptor_props.CompositeProperty method), 300
get_indexes() (sqlalchemy.engine.default.DefaultDialect method), 828
get_indexes() (sqlalchemy.engine.interfaces.Dialect method), 835
get_indexes() (sqlalchemy.engine.reflection.Inspector method), 648
get_instance_dict() (sqlalchemy.ext.instrumentation.InstrumentationManager method), 416
get_isolation_level() (sqlalchemy.engine.Connection method), 763
get_isolation_level() (sqlalchemy.engine.default.DefaultDialect method), 828
get_isolation_level() (sqlalchemy.engine.interfaces.Dialect method), 835
get_lastrowid() (sqlalchemy.engine.default.DefaultExecutionContext method), 837
get_pk_constraint() (sqlalchemy.engine.default.DefaultDialect method), 829
get_pk_constraint() (sqlalchemy.engine.interfaces.Dialect method), 835
get_pk_constraint() (sqlalchemy.engine.reflection.Inspector method), 648
get_primary_keys() (sqlalchemy.engine.default.DefaultDialect method), 829
get_primary_keys() (sqlalchemy.engine.interfaces.Dialect method), 835
get_primary_keys() (sqlalchemy.engine.reflection.Inspector method), 648
get_property() (sqlalchemy.orm.mapper.Mapper method), 86
get_property_by_column() (sqlalchemy.orm.mapper.Mapper method), 86
get_referent() (sqlalchemy.schema.ForeignKey method), 678
get_result_processor() (sqlalchemy.engine.default.DefaultExecutionContext method), 838
get_rowcount() (sqlalchemy.engine.interfaces.ExecutionContext method), 839
get_schema_names() (sqlalchemy.engine.reflection.Inspector method), 648
get_select_precolumns() (sqlalchemy.sql.compiler.SQLCompiler method), 841

get_sorted_table_and_fkc_names() (sqlalchemy.engine.reflection.Inspector method), 648
get_table_names() (sqlalchemy.engine.default.DefaultDialect method), 829
get_table_names() (sqlalchemy.engine.interfaces.Dialect method), 836
get_table_names() (sqlalchemy.engine.reflection.Inspector method), 649
get_table_oid() (sqlalchemy.dialects.postgresql.base.PGInspector method), 902
get_table_options() (sqlalchemy.engine.reflection.Inspector method), 649
get_temp_table_names() (sqlalchemy.engine.default.DefaultDialect method), 829
get_temp_table_names() (sqlalchemy.engine.interfaces.Dialect method), 836
get_temp_table_names() (sqlalchemy.engine.reflection.Inspector method), 649
get_temp_view_names() (sqlalchemy.engine.default.DefaultDialect method), 829
get_temp_view_names() (sqlalchemy.engine.interfaces.Dialect method), 836
get_temp_view_names() (sqlalchemy.engine.reflection.Inspector method), 649
get_unique_constraints() (sqlalchemy.engine.default.DefaultDialect method), 829
get_unique_constraints() (sqlalchemy.engine.interfaces.Dialect method), 836
get_unique_constraints() (sqlalchemy.engine.reflection.Inspector method), 650
get_view_definition() (sqlalchemy.engine.default.DefaultDialect method), 829
get_view_definition() (sqlalchemy.engine.interfaces.Dialect method), 836
get_view_definition() (sqlalchemy.engine.reflection.Inspector method), 650
get_view_names() (sqlalchemy.engine.default.DefaultDialect method), 829
get_view_names() (sqlalchemy.engine.interfaces.Dialect method), 836
get_view_names() (sqlalchemy.engine.reflection.Inspector method), 650
group_by() (sqlalchemy.orm.query.Query method), 168
group_by() (sqlalchemy.sql.expression.CompoundSelect method), 521
group_by() (sqlalchemy.sql.expression.GenerativeSelect method), 537
group_by() (sqlalchemy.sql.expression.Select method), 553

H

handle_dbapi_exception() (sqlalchemy.engine.interfaces.ExecutionContext method), 839
handle_error() (sqlalchemy.events.ConnectionEvents method), 801
has() (sqlalchemy.ext.associationproxy.AssociationProxy method), 346
has() (sqlalchemy.orm.attributes.QueryableAttribute method), 326
has() (sqlalchemy.orm.interfaces.PropComparator method), 309
has() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 294
has() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 318
has() (sqlalchemy.util.ScopedRegistry method), 231
has_all() (sqlalchemy.dialects.postgresql.HSTORE.Comparator method), 911
has_all() (sqlalchemy.dialects.postgresql.JSONB.Comparator method), 914
has_any() (sqlalchemy.dialects.postgresql.HSTORE.Comparator method), 911
has_any() (sqlalchemy.dialects.postgresql.JSONB.Comparator method), 914
has_changes() (sqlalchemy.orm.attributes.History method), 257
has_identity (sqlalchemy.orm.state.InstanceState attribute), 302
has_inherited_table() (in module sqlalchemy.ext.declarative), 387
has_key() (sqlalchemy.dialects.postgresql.HSTORE.Comparator method), 911
has_key() (sqlalchemy.dialects.postgresql.JSONB.Comparator method), 915
has_key() (sqlalchemy.engine.RowProxy method), 777
has_key() (sqlalchemy.orm.instrumentation.ClassManager method), 291
has_parent() (sqlalchemy.orm.instrumentation.ClassManager method), 291
has_sequence() (sqlalchemy.engine.default.DefaultDialect method), 829
has_sequence() (sqlalchemy.engine.interfaces.Dialect method), 836
has_table() (sqlalchemy.engine.default.DefaultDialect method), 830
has_table() (sqlalchemy.engine.Engine method), 769
has_table() (sqlalchemy.engine.interfaces.Dialect method), 836
hashable (sqlalchemy.types.TypeEngine attribute), 737
HasPrefixes (class in sqlalchemy.sql.expression), 540
HasSuffixes (class in sqlalchemy.sql.expression), 540
having() (sqlalchemy.orm.query.Query method), 168
having() (sqlalchemy.sql.expression.Select method), 553
History (class in sqlalchemy.orm.attributes), 257
history (sqlalchemy.orm.state.AttributeState attribute), 290

HSTORE (class in sqlalchemy.dialects.postgresql), 909
hstore (class in sqlalchemy.dialects.postgresql), 911
HSTORE.Comparator (class in sqlalchemy.dialects.postgresql), 911
hybrid_method (class in sqlalchemy.ext.hybrid), 414
HYBRID_METHOD (in module sqlalchemy.ext.hybrid), 415
hybrid_property (class in sqlalchemy.ext.hybrid), 414
HYBRID_PROPERTY (in module sqlalchemy.ext.hybrid), 415

I

Identified (class in sqlalchemy.log), 839
identifier (sqlalchemy.sql.functions.AnsiFunction attribute), 598
identifier (sqlalchemy.sql.functions.array_agg attribute), 602
identifier (sqlalchemy.sql.functions.char_length attribute), 603
identifier (sqlalchemy.sql.functions.coalesce attribute), 603
identifier (sqlalchemy.sql.functions.concat attribute), 603
identifier (sqlalchemy.sql.functions.count attribute), 603
identifier (sqlalchemy.sql.functions.cume_dist attribute), 603
identifier (sqlalchemy.sql.functions.current_date attribute), 603
identifier (sqlalchemy.sql.functions.current_time attribute), 604
identifier (sqlalchemy.sql.functions.current_timestamp attribute), 604
identifier (sqlalchemy.sql.functions.current_user attribute), 604
identifier (sqlalchemy.sql.functions.dense_rank attribute), 604
identifier (sqlalchemy.sql.functions.GenericFunction attribute), 602
identifier (sqlalchemy.sql.functions.localtime attribute), 604
identifier (sqlalchemy.sql.functions.localtimestamp attribute), 604
identifier (sqlalchemy.sql.functions.max attribute), 605
identifier (sqlalchemy.sql.functions.min attribute), 605
identifier (sqlalchemy.sql.functions.mode attribute), 605
identifier (sqlalchemy.sql.functions.next_value attribute), 605
identifier (sqlalchemy.sql.functions.now attribute), 605
identifier (sqlalchemy.sql.functions.OrderedSetAgg attribute), 602
identifier (sqlalchemy.sql.functions.percent_rank attribute), 605
identifier (sqlalchemy.sql.functions.percentile_cont attribute), 606
identifier (sqlalchemy.sql.functions.percentile_disc attribute), 606
identifier (sqlalchemy.sql.functions.random attribute), 606
identifier (sqlalchemy.sql.functions.rank attribute), 606
identifier (sqlalchemy.sql.functions.ReturnTypeFromArgs attribute), 602
identifier (sqlalchemy.sql.functions.session_user attribute), 607
identifier (sqlalchemy.sql.functions.sum attribute), 607
identifier (sqlalchemy.sql.functions.sysdate attribute), 607
identifier (sqlalchemy.sql.functions.user attribute), 607
IdentifierError, 822
IdentifierPreparer (class in sqlalchemy.sql.compiler), 839
identity (sqlalchemy.orm.state.InstanceState attribute), 303
identity_key (sqlalchemy.orm.state.InstanceState attribute), 303
identity_key() (in module sqlalchemy.orm.util), 82
identity_key() (sqlalchemy.orm.session.Session method), 249
identity_key() (sqlalchemy.orm.session.sessionmaker method), 239
identity_key_from_instance() (sqlalchemy.orm.mapper.Mapper method), 86
identity_key_from_primary_key() (sqlalchemy.orm.mapper.Mapper method), 86
identity_key_from_row() (sqlalchemy.orm.mapper.Mapper method), 86
identity_map (sqlalchemy.orm.session.Session attribute), 249
IdentityMap (class in sqlalchemy.orm.identity), 300
ilike() (sqlalchemy.orm.attributes.QueryableAttribute method), 326
ilike() (sqlalchemy.orm.interfaces.PropComparator method), 309
ilike() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 295
ilike() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 318
ilike() (sqlalchemy.schema.Column method), 625
ilike() (sqlalchemy.sql.expression.ColumnElement method), 486
ilike() (sqlalchemy.sql.operators.ColumnOperators method), 494
IMAGE (class in sqlalchemy.dialects.mssql), 854
immediateload() (in module sqlalchemy.orm), 155
immediateload() (sqlalchemy.orm.strategy_options.Load method), 187
impl (sqlalchemy.types.Interval attribute), 707

impl (sqlalchemy.types.PickleType attribute), 709
implicit_returning (sqlalchemy.sql.expression.TableClause attribute), 565
in_() (sqlalchemy.orm.attributes.QueryableAttribute method), 327
in_() (sqlalchemy.orm.interfaces.PropComparator method), 310
in_() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 295
in_() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 318
in_() (sqlalchemy.schema.Column method), 626
in_() (sqlalchemy.sql.expression.ColumnElement method), 486
in_() (sqlalchemy.sql.operators.ColumnOperators method), 495
in_transaction() (sqlalchemy.engine.Connection method), 764
Index (class in sqlalchemy.schema), 689
Indexable (class in sqlalchemy.types), 738
INET (class in sqlalchemy.dialects.postgresql), 912
info (MapperProperty attribute), 304
info (sqlalchemy.engine.Connection attribute), 764
info (sqlalchemy.ext.associationproxy.AssociationProxy attribute), 346
info (sqlalchemy.orm.attributes.QueryableAttribute attribute), 327
info (sqlalchemy.orm.base.InspectionAttrInfo attribute), 301
info (sqlalchemy.orm.session.Session attribute), 249
info (sqlalchemy.pool._ConnectionFairy attribute), 788
info (sqlalchemy.pool._ConnectionRecord attribute), 789
info (sqlalchemy.schema.CheckConstraint attribute), 675
info (sqlalchemy.schema.Column attribute), 626
info (sqlalchemy.schema.ColumnCollectionConstraint attribute), 672
info (sqlalchemy.schema.ForeignKey attribute), 678
info (sqlalchemy.schema.ForeignKeyConstraint attribute), 681
info (sqlalchemy.schema.Index attribute), 692
info (sqlalchemy.schema.PrimaryKeyConstraint attribute), 684
info (sqlalchemy.schema.SchemaItem attribute), 633
info (sqlalchemy.schema.Table attribute), 640
info (sqlalchemy.schema.UniqueConstraint attribute), 686
inherits (sqlalchemy.orm.mapper.Mapper attribute), 86
init() (sqlalchemy.orm.descriptor_props.SynonymProperty method), 324
init() (sqlalchemy.orm.events.InstanceEvents method), 272
init() (sqlalchemy.orm.interfaces.MapperProperty method), 305
init() (sqlalchemy.orm.properties.ColumnProperty method), 299
init() (sqlalchemy.orm.properties.RelationshipProperty method), 322
init_collection() (in module sqlalchemy.orm.attributes), 256
init_collection() (sqlalchemy.orm.events.AttributeEvents method), 260
init_failed() (sqlalchemy.orm.interfaces.MapperExtension method), 333
init_failure() (sqlalchemy.orm.events.InstanceEvents method), 272
init_instance() (sqlalchemy.orm.interfaces.MapperExtension method), 333
initialize() (sqlalchemy.engine.interfaces.Dialect method), 836
initialize_instance_dict() (sqlalchemy.ext.instrumentation.InstrumentationM method), 416
inner_columns (sqlalchemy.sql.expression.Select attribute), 553
Insert (class in sqlalchemy.sql.expression), 575
insert() (in module sqlalchemy.sql.expression), 567
insert() (sqlalchemy.ext.orderinglist.OrderingList method), 403
insert() (sqlalchemy.schema.Table method), 640
insert() (sqlalchemy.sql.expression.TableClause method), 565
inserted_primary_key (sqlalchemy.engine.ResultProxy attribute), 774
inspect() (in module sqlalchemy.inspection), 817
InspectionAttr (class in sqlalchemy.orm.base), 300
InspectionAttrInfo (class in sqlalchemy.orm.base), 301
Inspector (class in sqlalchemy.engine.reflection), 646
install_descriptor() (sqlalchemy.ext.instrumentation.InstrumentationManage method), 416
install_member() (sqlalchemy.ext.instrumentation.InstrumentationManager method), 416
install_state() (sqlalchemy.ext.instrumentation.InstrumentationManager method), 416
instance_state() (in module sqlalchemy.orm.attributes), 256
InstanceEvents (class in sqlalchemy.orm.events), 271
instances() (sqlalchemy.orm.query.Query method), 168
InstanceState (class in sqlalchemy.orm.state), 301
instrument_attribute() (sqlalchemy.ext.instrumentation.InstrumentationMan method), 416
instrument_class() (sqlalchemy.orm.events.MapperEvents method), 269
instrument_class() (sqlalchemy.orm.interfaces.MapperExtension method), 333
instrument_class() (sqlalchemy.orm.interfaces.MapperProperty method), 305
instrument_collection_class() (sqlalchemy.ext.instrumentation.InstrumentationManager

method), 416
instrument_declarative() (in module sqlalchemy.ext.declarative), 388
instrumentation_finders (in module sqlalchemy.ext.instrumentation), 417
INSTRUMENTATION_MANAGER (in module sqlalchemy.ext.instrumentation), 416
InstrumentationEvents (class in sqlalchemy.orm.events), 289
InstrumentationFactory (class in sqlalchemy.orm.instrumentation), 416
InstrumentationManager (class in sqlalchemy.ext.instrumentation), 416
InstrumentedAttribute (class in sqlalchemy.orm.attributes), 304
InstrumentedDict (class in sqlalchemy.orm.collections), 126
InstrumentedList (class in sqlalchemy.orm.collections), 126
InstrumentedSet (class in sqlalchemy.orm.collections), 126
INT (in module sqlalchemy.types), 715
INT4RANGE (class in sqlalchemy.dialects.postgresql), 916
INT8RANGE (class in sqlalchemy.dialects.postgresql), 916
INTEGER (class in sqlalchemy.dialects.mysql), 872
INTEGER (class in sqlalchemy.types), 717
Integer (class in sqlalchemy.types), 706
IntegrityError, 822
InterfaceError, 822
InternalError, 822
internally_instrumented() (sqlalchemy.orm.collections.collection static method), 122
intersect() (in module sqlalchemy.sql.expression), 508
intersect() (sqlalchemy.orm.query.Query method), 168
intersect() (sqlalchemy.sql.expression.Select method), 553
intersect_all() (in module sqlalchemy.sql.expression), 508
intersect_all() (sqlalchemy.orm.query.Query method), 168
intersect_all() (sqlalchemy.sql.expression.Select method), 553
INTERVAL (class in sqlalchemy.dialects.oracle), 888
INTERVAL (class in sqlalchemy.dialects.postgresql), 912
Interval (class in sqlalchemy.types), 706
invalidate() (sqlalchemy.engine.Connection method), 764
invalidate() (sqlalchemy.events.PoolEvents method), 792
invalidate() (sqlalchemy.orm.session.Session method), 250
invalidate() (sqlalchemy.pool._ConnectionFairy method), 788
invalidate() (sqlalchemy.pool._ConnectionRecord method), 789
invalidate_pool_on_disconnect (sqlalchemy.engine.ExceptionContext attribute), 772
invalidated (sqlalchemy.engine.Connection attribute), 764
InvalidRequestError, 822
is_() (sqlalchemy.orm.attributes.QueryableAttribute method), 327
is_() (sqlalchemy.orm.interfaces.PropComparator method), 310
is_() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 295
is_() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 318
is_() (sqlalchemy.schema.Column method), 626
is_() (sqlalchemy.sql.expression.ColumnElement method), 486
is_() (sqlalchemy.sql.operators.ColumnOperators method), 495
is_active (sqlalchemy.orm.session.Session attribute), 250
is_aliased_class (sqlalchemy.ext.associationproxy.AssociationProxy attribute), 346
is_aliased_class (sqlalchemy.orm.base.InspectionAttr attribute), 300
is_attribute (sqlalchemy.ext.associationproxy.AssociationProxy attribute), 347
is_attribute (sqlalchemy.orm.base.InspectionAttr attribute), 300
is_bound() (sqlalchemy.schema.MetaData method), 631
is_bound() (sqlalchemy.schema.ThreadLocalMetaData method), 644
is_clause_element (sqlalchemy.ext.associationproxy.AssociationProxy attribute), 347
is_clause_element (sqlalchemy.orm.base.InspectionAttr attribute), 301
is_clause_element (sqlalchemy.sql.expression.ColumnElement attribute), 487
is_deleted() (sqlalchemy.orm.session.UOWTransaction method), 330
is_derived_from() (sqlalchemy.schema.Table method), 641
is_derived_from() (sqlalchemy.sql.expression.FromClause method), 530
is_derived_from() (sqlalchemy.sql.expression.GenerativeSelect method), 537
is_derived_from() (sqlalchemy.sql.expression.TableClause method), 565
is_disconnect (sqlalchemy.engine.ExceptionContext attribute), 772
is_disconnect (sqlalchemy.engine.interfaces.ExecutionContext attribute), 839
is_disconnect() (sqlalchemy.engine.interfaces.Dialect method), 836

is_insert (sqlalchemy.engine.ResultProxy attribute), 775
is_instance (sqlalchemy.ext.associationproxy.AssociationProxy attribute), 347
is_instance (sqlalchemy.orm.base.InspectionAttr attribute), 301
is_instrumented() (in module sqlalchemy.orm.instrumentation), 257
is_mapper (sqlalchemy.ext.associationproxy.AssociationProxy attribute), 347
is_mapper (sqlalchemy.orm.base.InspectionAttr attribute), 301
is_mapper (sqlalchemy.orm.mapper.Mapper attribute), 86
is_modified() (sqlalchemy.orm.session.Session method), 251
is_property (sqlalchemy.ext.associationproxy.AssociationProxy attribute), 347
is_property (sqlalchemy.orm.base.InspectionAttr attribute), 301
is_property (sqlalchemy.orm.interfaces.MapperProperty attribute), 306
is_selectable (sqlalchemy.ext.associationproxy.AssociationProxy attribute), 347
is_selectable (sqlalchemy.orm.base.InspectionAttr attribute), 301
is_selectable (sqlalchemy.sql.expression.ColumnElement attribute), 487
is_valid (sqlalchemy.pool._ConnectionFairy attribute), 788
isa() (sqlalchemy.orm.mapper.Mapper method), 86
isdelete (sqlalchemy.sql.compiler.SQLCompiler attribute), 841
isinsert (sqlalchemy.sql.compiler.SQLCompiler attribute), 841
isnot() (sqlalchemy.orm.attributes.QueryableAttribute method), 327
isnot() (sqlalchemy.orm.interfaces.PropComparator method), 310
isnot() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 295
isnot() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 319
isnot() (sqlalchemy.schema.Column method), 626
isnot() (sqlalchemy.sql.expression.ColumnElement method), 487
isnot() (sqlalchemy.sql.operators.ColumnOperators method), 495
isupdate (sqlalchemy.sql.compiler.SQLCompiler attribute), 841
items() (sqlalchemy.engine.RowProxy method), 777
items() (sqlalchemy.orm.instrumentation.ClassManager method), 291
iterate_properties (sqlalchemy.orm.mapper.Mapper attribute), 87
iterator() (sqlalchemy.orm.collections.collection static method), 122
iteritems() (sqlalchemy.orm.instrumentation.ClassManager method), 291
iterkeys() (sqlalchemy.orm.instrumentation.ClassManager method), 291
itervalues() (sqlalchemy.orm.instrumentation.ClassManager method), 292

J

Join (class in sqlalchemy.sql.expression), 541
join() (in module sqlalchemy.orm), 188
join() (in module sqlalchemy.sql.expression), 508
join() (sqlalchemy.orm.query.Query method), 168
join() (sqlalchemy.schema.Table method), 641
join() (sqlalchemy.sql.expression.Alias method), 514
join() (sqlalchemy.sql.expression.CompoundSelect method), 521
join() (sqlalchemy.sql.expression.CTE method), 526
join() (sqlalchemy.sql.expression.FromClause method), 530
join() (sqlalchemy.sql.expression.GenerativeSelect method), 537
join() (sqlalchemy.sql.expression.Join method), 544
join() (sqlalchemy.sql.expression.Select method), 553
join() (sqlalchemy.sql.expression.TableClause method), 565
joinedload() (in module sqlalchemy.orm), 155
joinedload() (sqlalchemy.orm.strategy_options.Load method), 187
joinedload_all() (in module sqlalchemy.orm), 157
JSON (class in sqlalchemy.dialects.mysql), 872
JSON (class in sqlalchemy.dialects.postgresql), 912
JSON (class in sqlalchemy.types), 715
JSON.Comparator (class in sqlalchemy.dialects.postgresql), 913
JSON.Comparator (class in sqlalchemy.types), 717
JSON.JSONIndexType (class in sqlalchemy.types), 717
JSON.JSONPathType (class in sqlalchemy.types), 717
JSONB (class in sqlalchemy.dialects.postgresql), 914
JSONB.Comparator (class in sqlalchemy.dialects.postgresql), 914

K

key (sqlalchemy.schema.Table attribute), 641
key (sqlalchemy.sql.expression.ColumnElement attribute), 487
KeyedTuple (class in sqlalchemy.util), 185
keys() (sqlalchemy.dialects.postgresql.HSTORE.Comparator method), 911
keys() (sqlalchemy.engine.ResultProxy method), 775
keys() (sqlalchemy.engine.RowProxy method), 777
keys() (sqlalchemy.orm.instrumentation.ClassManager method), 292

keys() (sqlalchemy.util.KeyedTuple method), 186
kwargs (sqlalchemy.schema.CheckConstraint attribute), 675
kwargs (sqlalchemy.schema.ColumnCollectionConstraint attribute), 673
kwargs (sqlalchemy.schema.ForeignKey attribute), 678
kwargs (sqlalchemy.schema.ForeignKeyConstraint attribute), 681
kwargs (sqlalchemy.schema.Index attribute), 692
kwargs (sqlalchemy.schema.PrimaryKeyConstraint attribute), 684
kwargs (sqlalchemy.schema.Table attribute), 641
kwargs (sqlalchemy.schema.UniqueConstraint attribute), 686
kwargs (sqlalchemy.sql.base.DialectKWArgs attribute), 499
kwargs (sqlalchemy.sql.expression.Delete attribute), 572
kwargs (sqlalchemy.sql.expression.Insert attribute), 578
kwargs (sqlalchemy.sql.expression.Update attribute), 586
kwargs (sqlalchemy.sql.expression.UpdateBase attribute), 593

L

Label (class in sqlalchemy.sql.expression), 500
label() (in module sqlalchemy.sql.expression), 471
label() (sqlalchemy.orm.query.Bundle method), 185
label() (sqlalchemy.orm.query.Query method), 172
label() (sqlalchemy.schema.Column method), 626
label() (sqlalchemy.sql.expression.ColumnElement method), 487
label() (sqlalchemy.sql.expression.CompoundSelect method), 522
label() (sqlalchemy.sql.expression.GenerativeSelect method), 537
label() (sqlalchemy.sql.expression.Select method), 554
label() (sqlalchemy.sql.expression.SelectBase method), 562
LargeBinary (class in sqlalchemy.types), 707
last_inserted_params() (sqlalchemy.engine.ResultProxy method), 775
last_updated_params() (sqlalchemy.engine.ResultProxy method), 775
lastrow_has_defaults() (sqlalchemy.engine.interfaces.ExecutionContext method), 839
lastrow_has_defaults() (sqlalchemy.engine.ResultProxy method), 775
lastrowid (sqlalchemy.engine.ResultProxy attribute), 775
lazyload() (in module sqlalchemy.orm), 157
lazyload() (sqlalchemy.orm.strategy_options.Load method), 188
like() (sqlalchemy.orm.attributes.QueryableAttribute method), 328
like() (sqlalchemy.orm.interfaces.PropComparator method), 310
like() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 296
like() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 319
like() (sqlalchemy.schema.Column method), 626
like() (sqlalchemy.sql.expression.ColumnElement method), 487
like() (sqlalchemy.sql.operators.ColumnOperators method), 495
limit() (sqlalchemy.orm.query.Query method), 172
limit() (sqlalchemy.sql.expression.CompoundSelect method), 522
limit() (sqlalchemy.sql.expression.GenerativeSelect method), 537
limit() (sqlalchemy.sql.expression.Select method), 554
link() (sqlalchemy.orm.collections.collection static method), 122
linker() (sqlalchemy.orm.collections.collection static method), 122
listen() (in module sqlalchemy.event), 815
listens_for() (in module sqlalchemy.event), 816
literal() (in module sqlalchemy.sql.expression), 471
literal_column() (in module sqlalchemy.sql.expression), 471
literal_processor() (sqlalchemy.types.TypeDecorator method), 724
literal_processor() (sqlalchemy.types.TypeEngine method), 737
Load (class in sqlalchemy.orm.strategy_options), 186
load() (sqlalchemy.orm.events.InstanceEvents method), 273
load_dialect_impl() (sqlalchemy.types.TypeDecorator method), 725
load_history() (sqlalchemy.orm.state.AttributeState method), 290
load_only() (in module sqlalchemy.orm), 143
load_only() (sqlalchemy.orm.strategy_options.Load method), 188
loaded_as_persistent() (sqlalchemy.orm.events.SessionEvents method), 284
loaded_value (sqlalchemy.orm.state.AttributeState attribute), 290
loads() (in module sqlalchemy.ext.serializer), 614
local_attr (sqlalchemy.ext.associationproxy.AssociationProxy attribute), 347
local_table (sqlalchemy.orm.mapper.Mapper attribute), 87
localtime (class in sqlalchemy.sql.functions), 604
localtimestamp (class in sqlalchemy.sql.functions), 604
locate_all_froms() (sqlalchemy.sql.expression.Select method), 554
LONG (class in sqlalchemy.dialects.oracle), 889
LONGBLOB (class in sqlalchemy.dialects.mysql), 873
LONGTEXT (class in sqlalchemy.dialects.mysql), 873

M

MACADDR (class in sqlalchemy.dialects.postgresql), 915
make_transient() (in module sqlalchemy.orm.session), 255
make_transient_to_detached() (in module sqlalchemy.orm.session), 255
make_url() (in module sqlalchemy.engine.url), 746
manage() (in module sqlalchemy.pool), 789
manage() (sqlalchemy.ext.instrumentation.InstrumentationManager method), 416
manage() (sqlalchemy.orm.instrumentation.ClassManager method), 292
manager_getter() (sqlalchemy.ext.instrumentation.InstrumentationManager method), 416
MANYTOMANY (in module sqlalchemy.orm.interfaces), 304
MANYTOONE (in module sqlalchemy.orm.interfaces), 304
mapped_collection() (in module sqlalchemy.orm.collections), 118
mapped_table (sqlalchemy.orm.mapper.Mapper attribute), 87
MappedCollection (class in sqlalchemy.orm.collections), 124
Mapper (class in sqlalchemy.orm.mapper), 83
mapper (sqlalchemy.orm.mapper.Mapper attribute), 87
mapper (sqlalchemy.orm.properties.RelationshipProperty attribute), 322
mapper (sqlalchemy.orm.properties.RelationshipProperty.Comparator attribute), 319
mapper (sqlalchemy.orm.state.InstanceState attribute), 303
mapper() (in module sqlalchemy.orm), 75
mapper_configured() (sqlalchemy.orm.events.MapperEvents method), 270
MapperEvents (class in sqlalchemy.orm.events), 262
MapperExtension (class in sqlalchemy.orm.interfaces), 332
MapperProperty (class in sqlalchemy.orm.interfaces), 304
match() (sqlalchemy.orm.attributes.QueryableAttribute method), 328
match() (sqlalchemy.orm.interfaces.PropComparator method), 311
match() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 296
match() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 319
match() (sqlalchemy.schema.Column method), 627
match() (sqlalchemy.sql.expression.ColumnElement method), 487
match() (sqlalchemy.sql.operators.ColumnOperators method), 496
MatchType (class in sqlalchemy.types), 707
matrix() (sqlalchemy.dialects.postgresql.HSTORE.Comparator method), 911
max (class in sqlalchemy.sql.functions), 605
MEDIUMBLOB (class in sqlalchemy.dialects.mysql), 873
MEDIUMINT (class in sqlalchemy.dialects.mysql), 873
MEDIUMTEXT (class in sqlalchemy.dialects.mysql), 873
merge() (sqlalchemy.orm.descriptor_props.SynonymProperty method), 324
merge() (sqlalchemy.orm.interfaces.MapperProperty method), 306
merge() (sqlalchemy.orm.session.Session method), 252
merge_result() (sqlalchemy.orm.query.Query method), 172
MetaData (class in sqlalchemy.schema), 629
min (class in sqlalchemy.sql.functions), 605
mode (class in sqlalchemy.sql.functions), 605
MONEY (class in sqlalchemy.dialects.mssql), 854
MultipleResultsFound, 331
Mutable (class in sqlalchemy.ext.mutable), 398
MutableBase (class in sqlalchemy.ext.mutable), 397
MutableComposite (class in sqlalchemy.ext.mutable), 399
MutableDict (class in sqlalchemy.ext.mutable), 400

N

name (sqlalchemy.engine.Engine attribute), 770
name (sqlalchemy.sql.functions.AnsiFunction attribute), 598
name (sqlalchemy.sql.functions.array_agg attribute), 602
name (sqlalchemy.sql.functions.char_length attribute), 603
name (sqlalchemy.sql.functions.coalesce attribute), 603
name (sqlalchemy.sql.functions.concat attribute), 603
name (sqlalchemy.sql.functions.count attribute), 603
name (sqlalchemy.sql.functions.cume_dist attribute), 603
name (sqlalchemy.sql.functions.current_date attribute), 603
name (sqlalchemy.sql.functions.current_time attribute), 604
name (sqlalchemy.sql.functions.current_timestamp attribute), 604
name (sqlalchemy.sql.functions.current_user attribute), 604
name (sqlalchemy.sql.functions.dense_rank attribute), 604
name (sqlalchemy.sql.functions.GenericFunction attribute), 602
name (sqlalchemy.sql.functions.localtime attribute), 604
name (sqlalchemy.sql.functions.localtimestamp attribute), 604
name (sqlalchemy.sql.functions.max attribute), 605
name (sqlalchemy.sql.functions.min attribute), 605

name (sqlalchemy.sql.functions.mode attribute), 605
name (sqlalchemy.sql.functions.next_value attribute), 605
name (sqlalchemy.sql.functions.now attribute), 605
name (sqlalchemy.sql.functions.OrderedSetAgg attribute), 602
name (sqlalchemy.sql.functions.percent_rank attribute), 606
name (sqlalchemy.sql.functions.percentile_cont attribute), 606
name (sqlalchemy.sql.functions.percentile_disc attribute), 606
name (sqlalchemy.sql.functions.random attribute), 606
name (sqlalchemy.sql.functions.rank attribute), 606
name (sqlalchemy.sql.functions.ReturnTypeFromArgs attribute), 602
name (sqlalchemy.sql.functions.session_user attribute), 607
name (sqlalchemy.sql.functions.sum attribute), 607
name (sqlalchemy.sql.functions.sysdate attribute), 607
name (sqlalchemy.sql.functions.user attribute), 607
name_for_collection_relationship() (in module sqlalchemy.ext.automap), 357
name_for_scalar_relationship() (in module sqlalchemy.ext.automap), 357
NCHAR (class in sqlalchemy.dialects.mssql), 854
NCHAR (class in sqlalchemy.dialects.mysql), 874
NCHAR (class in sqlalchemy.types), 718
NCLOB (class in sqlalchemy.dialects.oracle), 888
NestedTransaction (class in sqlalchemy.engine), 772
new (sqlalchemy.orm.session.Session attribute), 252
next_value (class in sqlalchemy.sql.functions), 605
next_value() (sqlalchemy.schema.Sequence method), 659
no_autoflush (sqlalchemy.orm.session.Session attribute), 252
NO_STATE (in module sqlalchemy.orm.exc), 331
NoForeignKeysError, 822
NoInspectionAvailable, 822
noload() (in module sqlalchemy.orm), 157
noload() (sqlalchemy.orm.strategy_options.Load method), 188
non_added() (sqlalchemy.orm.attributes.History method), 257
non_deleted() (sqlalchemy.orm.attributes.History method), 257
non_primary (sqlalchemy.orm.mapper.Mapper attribute), 87
NoReferencedColumnError, 822
NoReferencedTableError, 822
NoReferenceError, 822
NoResultFound, 331
normalize_name() (sqlalchemy.engine.default.DefaultDialect method), 830
normalize_name() (sqlalchemy.engine.interfaces.Dialect method), 836
NoSuchColumnError, 822
NoSuchModuleError, 822
NoSuchTableError, 822
not_() (in module sqlalchemy.sql.expression), 471
not_extend_left_of() (sqlalchemy.dialects.postgresql.ranges.RangeOperator method), 917
not_extend_right_of() (sqlalchemy.dialects.postgresql.ranges.RangeOperato method), 917
NOT_EXTENSION (in module sqlalchemy.orm.interfaces), 306
notilike() (sqlalchemy.orm.attributes.QueryableAttribute method), 328
notilike() (sqlalchemy.orm.interfaces.PropComparator method), 311
notilike() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 296
notilike() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 319
notilike() (sqlalchemy.schema.Column method), 627
notilike() (sqlalchemy.sql.expression.ColumnElement method), 488
notilike() (sqlalchemy.sql.operators.ColumnOperators method), 496
notin_() (sqlalchemy.orm.attributes.QueryableAttribute method), 328
notin_() (sqlalchemy.orm.interfaces.PropComparator method), 311
notin_() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 296
notin_() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 320
notin_() (sqlalchemy.schema.Column method), 627
notin_() (sqlalchemy.sql.expression.ColumnElement method), 488
notin_() (sqlalchemy.sql.operators.ColumnOperators method), 496
notlike() (sqlalchemy.orm.attributes.QueryableAttribute method), 329
notlike() (sqlalchemy.orm.interfaces.PropComparator method), 311
notlike() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 297
notlike() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 320
notlike() (sqlalchemy.schema.Column method), 627
notlike() (sqlalchemy.sql.expression.ColumnElement method), 488
notlike() (sqlalchemy.sql.operators.ColumnOperators method), 496
NotSupportedError, 822
now (class in sqlalchemy.sql.functions), 605
NTEXT (class in sqlalchemy.dialects.mssql), 854
Null (class in sqlalchemy.sql.elements), 500
NULL (sqlalchemy.types.JSON attribute), 717

null() (in module sqlalchemy.sql.expression), 472
NullPool (class in sqlalchemy.pool), 787
nullsfirst() (in module sqlalchemy.sql.expression), 472
nullsfirst() (sqlalchemy.orm.attributes.QueryableAttribute method), 329
nullsfirst() (sqlalchemy.orm.interfaces.PropComparator method), 311
nullsfirst() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 297
nullsfirst() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 320
nullsfirst() (sqlalchemy.schema.Column method), 628
nullsfirst() (sqlalchemy.sql.expression.ColumnElement method), 488
nullsfirst() (sqlalchemy.sql.operators.ColumnOperators method), 496
nullslast() (in module sqlalchemy.sql.expression), 472
nullslast() (sqlalchemy.orm.attributes.QueryableAttribute method), 329
nullslast() (sqlalchemy.orm.interfaces.PropComparator method), 312
nullslast() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 297
nullslast() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 320
nullslast() (sqlalchemy.schema.Column method), 628
nullslast() (sqlalchemy.sql.expression.ColumnElement method), 488
nullslast() (sqlalchemy.sql.operators.ColumnOperators method), 496
NullType (class in sqlalchemy.types), 738
NUMBER (class in sqlalchemy.dialects.oracle), 889
NUMERIC (class in sqlalchemy.dialects.mysql), 874
NUMERIC (class in sqlalchemy.types), 718
Numeric (class in sqlalchemy.types), 707
NUMRANGE (class in sqlalchemy.dialects.postgresql), 916
NVARCHAR (class in sqlalchemy.dialects.mssql), 854
NVARCHAR (class in sqlalchemy.dialects.mysql), 875
NVARCHAR (class in sqlalchemy.types), 718

O

object (sqlalchemy.orm.state.InstanceState attribute), 303
object_mapper() (in module sqlalchemy.orm), 80
object_session() (in module sqlalchemy.orm.session), 255
object_session() (sqlalchemy.orm.session.Session method), 252
object_session() (sqlalchemy.orm.session.sessionmaker method), 239
object_state() (in module sqlalchemy.orm.util), 256
ObjectDeletedError, 331
ObjectDereferencedError, 331
of_type() (sqlalchemy.orm.interfaces.PropComparator method), 312
of_type() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 297
of_type() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 320
offset() (sqlalchemy.orm.query.Query method), 172
offset() (sqlalchemy.sql.expression.CompoundSelect method), 522
offset() (sqlalchemy.sql.expression.GenerativeSelect method), 538
offset() (sqlalchemy.sql.expression.Select method), 554
OID (class in sqlalchemy.dialects.postgresql), 915
on (sqlalchemy.schema.DDLElement attribute), 699
on_connect() (sqlalchemy.engine.default.DefaultDialect method), 830
one() (sqlalchemy.ext.baked.Result method), 364
one() (sqlalchemy.orm.query.Query method), 172
one_or_none() (sqlalchemy.ext.baked.Result method), 365
one_or_none() (sqlalchemy.orm.query.Query method), 172
ONETOMANY (in module sqlalchemy.orm.interfaces), 306
op() (sqlalchemy.orm.attributes.QueryableAttribute method), 329
op() (sqlalchemy.orm.interfaces.PropComparator method), 312
op() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 297
op() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 320
op() (sqlalchemy.schema.Column method), 628
op() (sqlalchemy.sql.expression.ColumnElement method), 488
op() (sqlalchemy.sql.operators.ColumnOperators method), 496
op() (sqlalchemy.sql.operators.Operators method), 505
operate() (sqlalchemy.orm.interfaces.PropComparator method), 313
operate() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 321
operate() (sqlalchemy.sql.expression.ColumnElement method), 489
operate() (sqlalchemy.sql.operators.ColumnOperators method), 497
operate() (sqlalchemy.sql.operators.Operators method), 505
OperationalError, 823
Operators (class in sqlalchemy.sql.operators), 504
options() (sqlalchemy.orm.query.Query method), 173
or_() (in module sqlalchemy.sql.expression), 473
order_by() (sqlalchemy.orm.query.Query method), 173
order_by() (sqlalchemy.sql.expression.CompoundSelect method), 522

order_by() (sqlalchemy.sql.expression.GenerativeSelect method), 538
order_by() (sqlalchemy.sql.expression.Select method), 555
OrderedSetAgg (class in sqlalchemy.sql.functions), 602
ordering_list() (in module sqlalchemy.ext.orderinglist), 402
OrderingList (class in sqlalchemy.ext.orderinglist), 402
orig (sqlalchemy.exc.StatementError attribute), 823
original_exception (sqlalchemy.engine.ExceptionContext attribute), 772
original_init (sqlalchemy.orm.instrumentation.ClassManager attribute), 292
outerjoin() (in module sqlalchemy.orm), 188
outerjoin() (in module sqlalchemy.sql.expression), 509
outerjoin() (sqlalchemy.orm.query.Query method), 173
outerjoin() (sqlalchemy.schema.Table method), 641
outerjoin() (sqlalchemy.sql.expression.Alias method), 515
outerjoin() (sqlalchemy.sql.expression.CompoundSelect method), 522
outerjoin() (sqlalchemy.sql.expression.CTE method), 526
outerjoin() (sqlalchemy.sql.expression.FromClause method), 530
outerjoin() (sqlalchemy.sql.expression.GenerativeSelect method), 538
outerjoin() (sqlalchemy.sql.expression.Join method), 544
outerjoin() (sqlalchemy.sql.expression.Select method), 555
outerjoin() (sqlalchemy.sql.expression.TableClause method), 565
outparam() (in module sqlalchemy.sql.expression), 473
Over (class in sqlalchemy.sql.expression), 500
over() (in module sqlalchemy.sql.expression), 473
over() (sqlalchemy.sql.expression.FunctionFilter method), 500
over() (sqlalchemy.sql.expression.WithinGroup method), 503
over() (sqlalchemy.sql.functions.FunctionElement method), 600
overlap() (sqlalchemy.dialects.postgresql.ARRAY.Comparator method), 906
overlaps() (sqlalchemy.dialects.postgresql.ranges.RangeOperators.comparator_factory method), 917

P

packagenames (sqlalchemy.sql.functions.FunctionElement attribute), 600
parameters (sqlalchemy.engine.ExceptionContext attribute), 772
params (sqlalchemy.engine.interfaces.Compiled attribute), 824
params (sqlalchemy.exc.StatementError attribute), 823
params (sqlalchemy.sql.compiler.DDLCompiler attribute), 825
params (sqlalchemy.sql.compiler.SQLCompiler attribute), 841
params() (sqlalchemy.ext.baked.Result method), 365
params() (sqlalchemy.orm.query.Query method), 173
params() (sqlalchemy.sql.expression.Alias method), 515
params() (sqlalchemy.sql.expression.ClauseElement method), 480
params() (sqlalchemy.sql.expression.ColumnElement method), 489
params() (sqlalchemy.sql.expression.CompoundSelect method), 523
params() (sqlalchemy.sql.expression.CTE method), 527
params() (sqlalchemy.sql.expression.Delete method), 572
params() (sqlalchemy.sql.expression.GenerativeSelect method), 538
params() (sqlalchemy.sql.expression.Insert method), 578
params() (sqlalchemy.sql.expression.Join method), 545
params() (sqlalchemy.sql.expression.Select method), 555
params() (sqlalchemy.sql.expression.Update method), 586
params() (sqlalchemy.sql.expression.UpdateBase method), 593
parent (sqlalchemy.orm.attributes.QueryableAttribute attribute), 330
PassiveDefault (class in sqlalchemy.schema), 657
pending (sqlalchemy.orm.state.InstanceState attribute), 303
pending_to_persistent() (sqlalchemy.orm.events.SessionEvents method), 285
pending_to_transient() (sqlalchemy.orm.events.SessionEvents method), 285
percent_rank (class in sqlalchemy.sql.functions), 605
percentile_cont (class in sqlalchemy.sql.functions), 606
percentile_disc (class in sqlalchemy.sql.functions), 606
persistent (sqlalchemy.orm.state.InstanceState attribute), 303
persistent_to_deleted() (sqlalchemy.orm.events.SessionEvents method), 286
persistent_to_detached() (sqlalchemy.orm.events.SessionEvents method), 286
persistent_to_transient() (sqlalchemy.orm.events.SessionEvents method), 287
PGInspector (class in sqlalchemy.dialects.postgresql.base), 901
pickle() (sqlalchemy.orm.events.InstanceEvents method), 274
PickleType (class in sqlalchemy.types), 708
polymorphic_identity (sqlalchemy.orm.mapper.Mapper attribute), 87
polymorphic_iterator() (sqlalchemy.orm.mapper.Mapper method), 87

polymorphic_map (sqlalchemy.orm.mapper.Mapper attribute), 87
polymorphic_on (sqlalchemy.orm.mapper.Mapper attribute), 88
polymorphic_union() (in module sqlalchemy.orm.util), 82
Pool (class in sqlalchemy.pool), 783
PoolEvents (class in sqlalchemy.events), 790
PoolListener (class in sqlalchemy.interfaces), 819
pop() (sqlalchemy.ext.mutable.MutableDict method), 400
pop() (sqlalchemy.ext.orderinglist.OrderingList method), 403
pop() (sqlalchemy.orm.collections.MappedCollection method), 125
pop() (sqlalchemy.orm.instrumentation.ClassManager method), 292
popitem() (sqlalchemy.ext.mutable.MutableDict method), 400
popitem() (sqlalchemy.orm.collections.MappedCollection method), 125
popitem() (sqlalchemy.orm.instrumentation.ClassManager method), 292
populate_existing() (sqlalchemy.orm.query.Query method), 173
post_configure_attribute() (sqlalchemy.ext.instrumentation.InstrumentationManager method), 416
post_exec() (sqlalchemy.engine.interfaces.ExecutionContext method), 839
post_instrument_class() (sqlalchemy.orm.descriptor_props.SynonymProperty method), 324
post_instrument_class() (sqlalchemy.orm.interfaces.MapperProperty method), 306
postfetch_cols() (sqlalchemy.engine.ResultProxy method), 775
pre_exec() (sqlalchemy.engine.interfaces.ExecutionContext method), 839
prefetch_cols() (sqlalchemy.engine.ResultProxy method), 775
prefix_with() (sqlalchemy.orm.query.Query method), 173
prefix_with() (sqlalchemy.sql.expression.Delete method), 572
prefix_with() (sqlalchemy.sql.expression.HasPrefixes method), 540
prefix_with() (sqlalchemy.sql.expression.Insert method), 578
prefix_with() (sqlalchemy.sql.expression.Select method), 555
prefix_with() (sqlalchemy.sql.expression.Update method), 586
prefix_with() (sqlalchemy.sql.expression.UpdateBase method), 594
prepare() (sqlalchemy.engine.TwoPhaseTransaction method), 777
prepare() (sqlalchemy.ext.automap.AutomapBase class method), 356
prepare() (sqlalchemy.ext.declarative.DeferredReflection class method), 391
prepare() (sqlalchemy.orm.session.Session method), 253
prepare_instrumentation() (in module sqlalchemy.orm.collections), 126
prepare_twophase() (sqlalchemy.events.ConnectionEvents method), 803
prepare_twophase() (sqlalchemy.interfaces.ConnectionProxy method), 819
preparer (sqlalchemy.engine.default.DefaultDialect attribute), 830
primary_key (sqlalchemy.orm.mapper.Mapper attribute), 88
primary_key (sqlalchemy.schema.Table attribute), 642
primary_key (sqlalchemy.sql.expression.Alias attribute), 515
primary_key (sqlalchemy.sql.expression.ColumnElement attribute), 489
primary_key (sqlalchemy.sql.expression.CompoundSelect attribute), 523
primary_key (sqlalchemy.sql.expression.CTE attribute), 527
primary_key (sqlalchemy.sql.expression.FromClause attribute), 531
primary_key (sqlalchemy.sql.expression.GenerativeSelect attribute), 539
primary_key (sqlalchemy.sql.expression.Join attribute), 545
primary_key (sqlalchemy.sql.expression.Select attribute), 556
primary_key (sqlalchemy.sql.expression.TableClause attribute), 566
primary_key_from_instance() (sqlalchemy.orm.mapper.Mapper method), 88
primary_mapper() (sqlalchemy.orm.mapper.Mapper method), 88
PrimaryKeyConstraint (class in sqlalchemy.schema), 682
process_bind_param() (sqlalchemy.types.TypeDecorator method), 725
process_literal_param() (sqlalchemy.types.TypeDecorator method), 725
process_result_value() (sqlalchemy.types.TypeDecorator method), 725
ProgrammingError, 823
PropComparator (class in sqlalchemy.orm.interfaces), 306
property (sqlalchemy.orm.attributes.QueryableAttribute attribute), 330
proxy_set (sqlalchemy.sql.expression.ColumnElement attribute), 489
prune() (sqlalchemy.orm.session.Session method), 253

Python Enhancement Proposals
PEP 249, 789
python_type (sqlalchemy.types.TypeDecorator attribute), 725
python_type (sqlalchemy.types.TypeEngine attribute), 737

Q

Query (class in sqlalchemy.orm.query), 159
query() (sqlalchemy.orm.session.Session method), 253
query_property() (sqlalchemy.orm.scoping.scoped_session method), 230
QueryableAttribute (class in sqlalchemy.orm.attributes), 324
QueryContext (class in sqlalchemy.orm.query), 324
QueryEvents (class in sqlalchemy.orm.events), 288
QueuePool (class in sqlalchemy.pool), 785
quote (sqlalchemy.schema.CheckConstraint attribute), 675
quote (sqlalchemy.schema.Column attribute), 628
quote (sqlalchemy.schema.ColumnCollectionConstraint attribute), 673
quote (sqlalchemy.schema.ForeignKey attribute), 678
quote (sqlalchemy.schema.ForeignKeyConstraint attribute), 681
quote (sqlalchemy.schema.Index attribute), 692
quote (sqlalchemy.schema.PrimaryKeyConstraint attribute), 684
quote (sqlalchemy.schema.SchemaItem attribute), 633
quote (sqlalchemy.schema.Table attribute), 642
quote (sqlalchemy.schema.UniqueConstraint attribute), 686
quote() (sqlalchemy.sql.compiler.IdentifierPreparer method), 840
quote_identifier() (sqlalchemy.sql.compiler.IdentifierPreparer method), 840
quote_schema (sqlalchemy.schema.Table attribute), 642
quote_schema() (sqlalchemy.sql.compiler.IdentifierPreparer method), 840
quoted_name (class in sqlalchemy.sql.elements), 506

R

random (class in sqlalchemy.sql.functions), 606
RangeOperators (class in sqlalchemy.dialects.postgresql.ranges), 917
RangeOperators.comparator_factory (class in sqlalchemy.dialects.postgresql.ranges), 917
rank (class in sqlalchemy.sql.functions), 606
RAW (class in sqlalchemy.dialects.oracle), 890
raw_connection() (sqlalchemy.engine.Engine method), 770
REAL (class in sqlalchemy.dialects.mssql), 855
REAL (class in sqlalchemy.dialects.mysql), 875
REAL (class in sqlalchemy.dialects.postgresql), 915
REAL (class in sqlalchemy.types), 718
reconstruct_instance() (sqlalchemy.orm.interfaces.MapperExtension method), 333
reconstructor() (in module sqlalchemy.orm), 158
recreate() (sqlalchemy.pool.Pool method), 785
reduce_columns() (sqlalchemy.sql.expression.Select method), 556
references() (sqlalchemy.schema.Column method), 629
references() (sqlalchemy.schema.ForeignKey method), 678
referred_table (sqlalchemy.schema.ForeignKeyConstraint attribute), 681
reflect() (sqlalchemy.schema.MetaData method), 631
reflecttable() (sqlalchemy.engine.interfaces.Dialect method), 837
reflecttable() (sqlalchemy.engine.reflection.Inspector method), 650
refresh() (sqlalchemy.orm.events.InstanceEvents method), 274
refresh() (sqlalchemy.orm.session.Session method), 253
refresh_flush() (sqlalchemy.orm.events.InstanceEvents method), 275
register_function() (in module sqlalchemy.sql.functions), 606
relation() (in module sqlalchemy.orm), 139
relationship() (in module sqlalchemy.orm), 131
RelationshipProperty (class in sqlalchemy.orm.properties), 313
RelationshipProperty.Comparator (class in sqlalchemy.orm.properties), 314
relationships (sqlalchemy.orm.mapper.Mapper attribute), 88
release_savepoint() (sqlalchemy.events.ConnectionEvents method), 803
release_savepoint() (sqlalchemy.interfaces.ConnectionProxy method), 819
remote() (in module sqlalchemy.orm), 139
remote_attr (sqlalchemy.ext.associationproxy.AssociationProxy attribute), 347
remove() (in module sqlalchemy.event), 816
remove() (sqlalchemy.ext.orderinglist.OrderingList method), 403
remove() (sqlalchemy.orm.collections.MappedCollection method), 125
remove() (sqlalchemy.orm.events.AttributeEvents method), 260
remove() (sqlalchemy.orm.interfaces.AttributeExtension method), 336
remove() (sqlalchemy.orm.scoping.scoped_session method), 230
remove() (sqlalchemy.schema.MetaData method), 632
remove_state() (sqlalchemy.ext.instrumentation.InstrumentationManager method), 416

remove_state_actions() (sqlalchemy.orm.session.UOWTransaction method), 330
remover() (sqlalchemy.orm.collections.collection static method), 122
removes() (sqlalchemy.orm.collections.collection static method), 123
removes_return() (sqlalchemy.orm.collections.collection static method), 123
render_literal_value() (sqlalchemy.sql.compiler.SQLCompiler method), 841
render_table_with_column_in_update_from (sqlalchemy.sql.compiler.SQLCompiler attribute), 841
reorder() (sqlalchemy.ext.orderinglist.OrderingList method), 403
replace() (sqlalchemy.sql.expression.ColumnCollection method), 482
replace_selectable() (sqlalchemy.schema.Table method), 642
replace_selectable() (sqlalchemy.sql.expression.Alias method), 515
replace_selectable() (sqlalchemy.sql.expression.CompoundSelect method), 523
replace_selectable() (sqlalchemy.sql.expression.CTE method), 527
replace_selectable() (sqlalchemy.sql.expression.FromClause method), 531
replace_selectable() (sqlalchemy.sql.expression.GenerativeSelect method), 539
replace_selectable() (sqlalchemy.sql.expression.Join method), 545
replace_selectable() (sqlalchemy.sql.expression.Select method), 556
replace_selectable() (sqlalchemy.sql.expression.TableClause method), 566
replaces() (sqlalchemy.orm.collections.collection static method), 123
reset() (sqlalchemy.events.PoolEvents method), 793
reset_isolation_level() (sqlalchemy.engine.interfaces.Dialect method), 837
reset_joinpoint() (sqlalchemy.orm.query.Query method), 174
ResourceClosedError, 823
Result (class in sqlalchemy.ext.baked), 364
result() (sqlalchemy.engine.interfaces.ExecutionContext method), 839
result_processor() (sqlalchemy.types.TypeDecorator method), 726
result_processor() (sqlalchemy.types.TypeEngine method), 737
ResultProxy (class in sqlalchemy.engine), 773
return_defaults() (sqlalchemy.sql.expression.Insert method), 579
return_defaults() (sqlalchemy.sql.expression.Update method), 586
return_defaults() (sqlalchemy.sql.expression.ValuesBase method), 596
returned_defaults (sqlalchemy.engine.ResultProxy attribute), 775
returning (sqlalchemy.sql.compiler.SQLCompiler attribute), 841
returning() (sqlalchemy.sql.expression.Delete method), 573
returning() (sqlalchemy.sql.expression.Insert method), 579
returning() (sqlalchemy.sql.expression.Update method), 587
returning() (sqlalchemy.sql.expression.UpdateBase method), 594
returning_precedes_values (sqlalchemy.sql.compiler.SQLCompiler attribute), 841
returns_rows (sqlalchemy.engine.ResultProxy attribute), 776
ReturnTypeFromArgs (class in sqlalchemy.sql.functions), 602
reverse_operate() (sqlalchemy.orm.interfaces.PropComparator method), 313
reverse_operate() (sqlalchemy.orm.properties.RelationshipProperty.Compar method), 321
reverse_operate() (sqlalchemy.sql.expression.ColumnElement method), 489
reverse_operate() (sqlalchemy.sql.operators.ColumnOperators method), 497
reverse_operate() (sqlalchemy.sql.operators.Operators method), 506
rollback() (sqlalchemy.engine.Transaction method), 777
rollback() (sqlalchemy.events.ConnectionEvents method), 804
rollback() (sqlalchemy.interfaces.ConnectionProxy method), 819
rollback() (sqlalchemy.orm.session.Session method), 253
rollback_savepoint() (sqlalchemy.events.ConnectionEvents method), 804
rollback_savepoint() (sqlalchemy.interfaces.ConnectionProxy method), 819
rollback_twophase() (sqlalchemy.events.ConnectionEvents method), 804
rollback_twophase() (sqlalchemy.interfaces.ConnectionProxy method), 819
rowcount (sqlalchemy.engine.ResultProxy attribute), 776
RowProxy (class in sqlalchemy.engine), 776
run_callable() (sqlalchemy.engine.Connection method), 764
run_callable() (sqlalchemy.engine.Engine method), 770

S

SADeprecationWarning, 823
SAPendingDeprecationWarning, 823
savepoint() (sqlalchemy.events.ConnectionEvents method), 805
savepoint() (sqlalchemy.interfaces.ConnectionProxy method), 819
SAWarning, 823
scalar (sqlalchemy.ext.associationproxy.AssociationProxy attribute), 347
scalar() (sqlalchemy.engine.Connectable method), 766
scalar() (sqlalchemy.engine.Connection method), 764
scalar() (sqlalchemy.engine.interfaces.Compiled method), 824
scalar() (sqlalchemy.engine.ResultProxy method), 776
scalar() (sqlalchemy.orm.query.Query method), 174
scalar() (sqlalchemy.orm.session.Session method), 254
scalar() (sqlalchemy.sql.compiler.DDLCompiler method), 825
scalar() (sqlalchemy.sql.expression.CompoundSelect method), 523
scalar() (sqlalchemy.sql.expression.Delete method), 574
scalar() (sqlalchemy.sql.expression.Executable method), 529
scalar() (sqlalchemy.sql.expression.GenerativeSelect method), 539
scalar() (sqlalchemy.sql.expression.Insert method), 580
scalar() (sqlalchemy.sql.expression.Select method), 556
scalar() (sqlalchemy.sql.expression.Update method), 588
scalar() (sqlalchemy.sql.expression.UpdateBase method), 595
scalar() (sqlalchemy.sql.functions.FunctionElement method), 600
ScalarSelect (class in sqlalchemy.sql.expression), 546
schema (sqlalchemy.sql.expression.FromClause attribute), 531
schema_for_object (sqlalchemy.engine.Connection attribute), 764
schema_for_object (sqlalchemy.engine.Engine attribute), 770
SchemaEventTarget (class in sqlalchemy.events), 812
SchemaItem (class in sqlalchemy.schema), 633
SchemaType (class in sqlalchemy.types), 709
scoped_session (class in sqlalchemy.orm.scoping), 229
ScopedRegistry (class in sqlalchemy.util), 230
Select (class in sqlalchemy.sql.expression), 546
select() (in module sqlalchemy.sql.expression), 509
select() (sqlalchemy.schema.Table method), 642
select() (sqlalchemy.sql.expression.Alias method), 515
select() (sqlalchemy.sql.expression.CompoundSelect method), 523
select() (sqlalchemy.sql.expression.CTE method), 527
select() (sqlalchemy.sql.expression.FromClause method), 531
select() (sqlalchemy.sql.expression.GenerativeSelect method), 539
select() (sqlalchemy.sql.expression.Join method), 545
select() (sqlalchemy.sql.expression.Select method), 556
select() (sqlalchemy.sql.expression.TableClause method), 566
select() (sqlalchemy.sql.functions.FunctionElement method), 600
select_entity_from() (sqlalchemy.orm.query.Query method), 174
select_from() (sqlalchemy.orm.query.Query method), 175
select_from() (sqlalchemy.sql.expression.Select method), 556
Selectable (class in sqlalchemy.sql.expression), 560
selectable (sqlalchemy.orm.mapper.Mapper attribute), 88
selectable (sqlalchemy.orm.query.Query attribute), 176
SelectBase (class in sqlalchemy.sql.expression), 560
self_and_descendants (sqlalchemy.orm.mapper.Mapper attribute), 89
self_group() (sqlalchemy.schema.Table method), 642
self_group() (sqlalchemy.sql.expression.ClauseElement method), 480
self_group() (sqlalchemy.sql.expression.ColumnElement method), 489
self_group() (sqlalchemy.sql.expression.Delete method), 574
self_group() (sqlalchemy.sql.expression.GenerativeSelect method), 539
self_group() (sqlalchemy.sql.expression.Insert method), 580
self_group() (sqlalchemy.sql.expression.Select method), 557
self_group() (sqlalchemy.sql.expression.TableClause method), 566
self_group() (sqlalchemy.sql.expression.Update method), 588
self_group() (sqlalchemy.sql.expression.UpdateBase method), 595
self_group() (sqlalchemy.sql.functions.FunctionElement method), 601
Sequence (class in sqlalchemy.schema), 657
Serializer() (in module sqlalchemy.ext.serializer), 614
Session (class in sqlalchemy.orm.session), 239
session (sqlalchemy.orm.state.InstanceState attribute), 303
session_factory (sqlalchemy.orm.scoping.scoped_session attribute), 230
session_user (class in sqlalchemy.sql.functions), 607
SessionEvents (class in sqlalchemy.orm.events), 276
SessionExtension (class in sqlalchemy.orm.interfaces), 334
sessionmaker (class in sqlalchemy.orm.session), 237
SessionTransaction (class in sqlalchemy.orm.session), 254

SET (class in sqlalchemy.dialects.mysql), 875
set() (sqlalchemy.orm.collections.MappedCollection method), 125
set() (sqlalchemy.orm.events.AttributeEvents method), 261
set() (sqlalchemy.orm.interfaces.AttributeExtension method), 336
set() (sqlalchemy.util.ScopedRegistry method), 231
set_attribute() (in module sqlalchemy.orm.attributes), 257
set_committed_value() (in module sqlalchemy.orm.attributes), 257
set_connection_execution_options() (sqlalchemy.events.ConnectionEvents method), 805
set_engine_execution_options() (sqlalchemy.events.ConnectionEvents method), 806
set_input_sizes() (sqlalchemy.engine.default.DefaultExecutionContext method), 838
set_isolation_level() (sqlalchemy.engine.default.DefaultDialect method), 830
set_isolation_level() (sqlalchemy.engine.interfaces.Dialect method), 837
set_parent() (sqlalchemy.orm.interfaces.MapperProperty method), 306
set_parent() (sqlalchemy.orm.properties.ColumnProperty method), 299
set_parent() (sqlalchemy.orm.properties.RelationshipProperty method), 322
set_shard() (sqlalchemy.ext.horizontal_shard.ShardedQuery method), 404
setdefault() (sqlalchemy.ext.mutable.MutableDict method), 400
setdefault() (sqlalchemy.orm.collections.MappedCollection method), 125
setdefault() (sqlalchemy.orm.instrumentation.ClassManager method), 292
setter() (sqlalchemy.ext.hybrid.hybrid_property method), 415
setup() (sqlalchemy.orm.descriptor_props.SynonymProperty method), 324
setup() (sqlalchemy.orm.interfaces.MapperProperty method), 306
ShardedQuery (class in sqlalchemy.ext.horizontal_shard), 404
ShardedSession (class in sqlalchemy.ext.horizontal_shard), 404
shares_lineage() (sqlalchemy.schema.Column method), 629
shares_lineage() (sqlalchemy.sql.expression.ColumnElement method), 489
should_autocommit_text() (sqlalchemy.engine.interfaces.ExecutionContext method), 839
should_evaluate_none (sqlalchemy.types.TypeEngine attribute), 737
single (sqlalchemy.orm.mapper.Mapper attribute), 89
single_entity (sqlalchemy.orm.query.Bundle attribute), 185
SingletonThreadPool (class in sqlalchemy.pool), 786
slice() (sqlalchemy.dialects.postgresql.HSTORE.Comparator method), 911
slice() (sqlalchemy.orm.query.Query method), 176
SMALLDATETIME (class in sqlalchemy.dialects.mssql), 855
SMALLINT (class in sqlalchemy.dialects.mysql), 876
SMALLINT (class in sqlalchemy.types), 718
SmallInteger (class in sqlalchemy.types), 709
SMALLMONEY (class in sqlalchemy.dialects.mssql), 855
soft_invalidate() (sqlalchemy.events.PoolEvents method), 793
sort_tables() (in module sqlalchemy.schema), 696
sort_tables_and_constraints() (in module sqlalchemy.schema), 696
sorted_tables (sqlalchemy.schema.MetaData attribute), 632
spoil() (sqlalchemy.ext.baked.BakedQuery method), 364
sql_compiler (sqlalchemy.engine.interfaces.Compiled attribute), 824
SQL_VARIANT (class in sqlalchemy.dialects.mssql), 855
sqlalchemy.dialects.firebird.base (module), 843
sqlalchemy.dialects.firebird.fdb (module), 844
sqlalchemy.dialects.firebird.kinterbasdb (module), 845
sqlalchemy.dialects.mssql.adodbapi (module), 860
sqlalchemy.dialects.mssql.base (module), 846
sqlalchemy.dialects.mssql.mxodbc (module), 859
sqlalchemy.dialects.mssql.pymssql (module), 859
sqlalchemy.dialects.mssql.pyodbc (module), 857
sqlalchemy.dialects.mssql.zxjdbc (module), 860
sqlalchemy.dialects.mysql.base (module), 860
sqlalchemy.dialects.mysql.cymysql (module), 881
sqlalchemy.dialects.mysql.gaerdbms (module), 882
sqlalchemy.dialects.mysql.mysqlconnector (module), 880
sqlalchemy.dialects.mysql.mysqldb (module), 879
sqlalchemy.dialects.mysql.oursql (module), 881
sqlalchemy.dialects.mysql.pymysql (module), 880
sqlalchemy.dialects.mysql.pyodbc (module), 882
sqlalchemy.dialects.mysql.zxjdbc (module), 883
sqlalchemy.dialects.oracle.base (module), 883
sqlalchemy.dialects.oracle.cx_oracle (module), 890
sqlalchemy.dialects.oracle.zxjdbc (module), 894
sqlalchemy.dialects.postgresql.base (module), 894
sqlalchemy.dialects.postgresql.pg8000 (module), 923
sqlalchemy.dialects.postgresql.psycopg2 (module), 919
sqlalchemy.dialects.postgresql.psycopg2cffi (module), 924

sqlalchemy.dialects.postgresql.pypostgresql (module), 925
sqlalchemy.dialects.postgresql.zxjdbc (module), 925
sqlalchemy.dialects.sqlite (module), 932
sqlalchemy.dialects.sqlite.base (module), 925
sqlalchemy.dialects.sqlite.pysqlcipher (module), 938
sqlalchemy.dialects.sqlite.pysqlite (module), 934
sqlalchemy.dialects.sybase.base (module), 939
sqlalchemy.dialects.sybase.mxodbc (module), 940
sqlalchemy.dialects.sybase.pyodbc (module), 939
sqlalchemy.dialects.sybase.pysybase (module), 939
sqlalchemy.engine (module), 748
sqlalchemy.exc (module), 821
sqlalchemy.ext.associationproxy (module), 336
sqlalchemy.ext.automap (module), 347
sqlalchemy.ext.baked (module), 358
sqlalchemy.ext.compiler (module), 607
sqlalchemy.ext.declarative (module), 385
sqlalchemy.ext.horizontal_shard (module), 403
sqlalchemy.ext.hybrid (module), 404
sqlalchemy.ext.instrumentation (module), 415
sqlalchemy.ext.mutable (module), 392
sqlalchemy.ext.orderinglist (module), 400
sqlalchemy.ext.serializer (module), 613
sqlalchemy.inspection (module), 817
sqlalchemy.interfaces (module), 818
sqlalchemy.orm (module), 41, 44, 48, 54, 75, 89, 130, 140, 157, 158
sqlalchemy.orm.exc (module), 330
sqlalchemy.orm.instrumentation (module), 289
sqlalchemy.orm.interfaces (module), 331
sqlalchemy.orm.session (module), 189, 237
sqlalchemy.pool (module), 778
sqlalchemy.schema (module), 614, 615, 644, 651, 659, 692
sqlalchemy.sql.expression (module), 459, 507, 567, 598
sqlalchemy.sql.functions (module), 598
sqlalchemy.types (module), 703, 719, 734
sqlalchemy_exception (sqlalchemy.engine.ExceptionContext attribute), 772
SQLAlchemyError, 823
SQLCompiler (class in sqlalchemy.sql.compiler), 840
StaleDataError, 331
startswith() (sqlalchemy.orm.attributes.QueryableAttribute method), 330
startswith() (sqlalchemy.orm.interfaces.PropComparator method), 313
startswith() (sqlalchemy.orm.properties.ColumnProperty.Comparator method), 298
startswith() (sqlalchemy.orm.properties.RelationshipProperty.Comparator method), 321
startswith() (sqlalchemy.schema.Column method), 629
startswith() (sqlalchemy.sql.expression.ColumnElement method), 489
startswith() (sqlalchemy.sql.operators.ColumnOperators method), 498
state_getter() (sqlalchemy.ext.instrumentation.InstrumentationManager method), 417
state_getter() (sqlalchemy.orm.instrumentation.ClassManager method), 292
statement (sqlalchemy.engine.ExceptionContext attribute), 772
statement (sqlalchemy.exc.StatementError attribute), 823
statement (sqlalchemy.orm.query.Query attribute), 176
statement_compiler (sqlalchemy.engine.default.DefaultDialect attribute), 830
StatementError, 823
StaticPool (class in sqlalchemy.pool), 787
strictly_left_of() (sqlalchemy.dialects.postgresql.ranges.RangeOperators.co method), 917
strictly_right_of() (sqlalchemy.dialects.postgresql.ranges.RangeOperators.c method), 917
String (class in sqlalchemy.types), 709
subquery() (in module sqlalchemy.sql.expression), 511
subquery() (sqlalchemy.orm.query.Query method), 176
subqueryload() (in module sqlalchemy.orm), 157
subqueryload() (sqlalchemy.orm.strategy_options.Load method), 188
subqueryload_all() (in module sqlalchemy.orm), 157
suffix_with() (sqlalchemy.orm.query.Query method), 176
suffix_with() (sqlalchemy.sql.expression.CTE method), 528
suffix_with() (sqlalchemy.sql.expression.HasSuffixes method), 540
suffix_with() (sqlalchemy.sql.expression.Select method), 557
sum (class in sqlalchemy.sql.functions), 607
sum() (sqlalchemy.orm.attributes.History method), 257
supports_execution (sqlalchemy.sql.expression.ColumnElement attribute), 490
supports_sane_multi_rowcount() (sqlalchemy.engine.ResultProxy method), 776
supports_sane_rowcount() (sqlalchemy.engine.ResultProxy method), 776
synonym() (in module sqlalchemy.orm), 53
synonym_for() (in module sqlalchemy.ext.declarative), 388
SynonymProperty (class in sqlalchemy.orm.descriptor_props), 323
synonyms (sqlalchemy.orm.mapper.Mapper attribute), 89
sysdate (class in sqlalchemy.sql.functions), 607

T

Table (class in sqlalchemy.schema), 633
table (sqlalchemy.orm.properties.RelationshipProperty attribute), 323

table() (in module sqlalchemy.sql.expression), 511
table_names() (sqlalchemy.engine.Engine method), 770
TableClause (class in sqlalchemy.sql.expression), 562
tables (sqlalchemy.orm.mapper.Mapper attribute), 89
tables (sqlalchemy.schema.MetaData attribute), 632
target (sqlalchemy.schema.DDLElement attribute), 699
target_class (sqlalchemy.ext.associationproxy.AssociationProxy attribute), 347
target_fullname (sqlalchemy.schema.ForeignKey attribute), 678
TEXT (class in sqlalchemy.dialects.mssql), 855
TEXT (class in sqlalchemy.dialects.mysql), 876
TEXT (class in sqlalchemy.types), 718
Text (class in sqlalchemy.types), 710
text() (in module sqlalchemy.sql.expression), 474
TextAsFrom (class in sqlalchemy.sql.expression), 567
TextClause (class in sqlalchemy.sql.expression), 501
thread safety
  Connection, 758
  Transaction, 777
  transactions, 751
ThreadLocalMetaData (class in sqlalchemy.schema), 644
ThreadLocalRegistry (class in sqlalchemy.util), 231
TIME (class in sqlalchemy.dialects.mssql), 856
TIME (class in sqlalchemy.dialects.mysql), 877
TIME (class in sqlalchemy.dialects.sqlite), 933
TIME (class in sqlalchemy.types), 718
Time (class in sqlalchemy.types), 710
TimeoutError, 823
TIMESTAMP (class in sqlalchemy.dialects.mysql), 877
TIMESTAMP (class in sqlalchemy.types), 718
timetuple (sqlalchemy.sql.expression.ColumnElement attribute), 490
timetuple (sqlalchemy.sql.operators.ColumnOperators attribute), 498
TINYBLOB (class in sqlalchemy.dialects.mysql), 878
TINYINT (class in sqlalchemy.dialects.mssql), 856
TINYINT (class in sqlalchemy.dialects.mysql), 878
TINYTEXT (class in sqlalchemy.dialects.mysql), 878
tometadata() (sqlalchemy.schema.Table method), 643
Transaction (class in sqlalchemy.engine), 777
transaction (sqlalchemy.orm.session.Session attribute), 254
transaction() (sqlalchemy.engine.Connection method), 765
transaction() (sqlalchemy.engine.Engine method), 770
transient (sqlalchemy.orm.state.InstanceState attribute), 303
transient_to_pending() (sqlalchemy.orm.events.SessionEvents method), 287
translate_connect_args() (sqlalchemy.engine.url.URL method), 746
true() (in module sqlalchemy.sql.expression), 475
True_ (class in sqlalchemy.sql.elements), 503
TSRANGE (class in sqlalchemy.dialects.postgresql), 916
TSTZRANGE (class in sqlalchemy.dialects.postgresql), 917
TSVECTOR (class in sqlalchemy.dialects.postgresql), 915
Tuple (class in sqlalchemy.sql.expression), 503
tuple_() (in module sqlalchemy.sql.expression), 476
TwoPhaseTransaction (class in sqlalchemy.engine), 777
type (sqlalchemy.dialects.postgresql.hstore attribute), 912
type (sqlalchemy.sql.expression.ColumnElement attribute), 490
type (sqlalchemy.sql.functions.array_agg attribute), 603
type (sqlalchemy.sql.functions.char_length attribute), 603
type (sqlalchemy.sql.functions.concat attribute), 603
type (sqlalchemy.sql.functions.count attribute), 603
type (sqlalchemy.sql.functions.cume_dist attribute), 603
type (sqlalchemy.sql.functions.current_date attribute), 603
type (sqlalchemy.sql.functions.current_time attribute), 604
type (sqlalchemy.sql.functions.current_timestamp attribute), 604
type (sqlalchemy.sql.functions.current_user attribute), 604
type (sqlalchemy.sql.functions.dense_rank attribute), 604
type (sqlalchemy.sql.functions.localtime attribute), 604
type (sqlalchemy.sql.functions.localtimestamp attribute), 604
type (sqlalchemy.sql.functions.next_value attribute), 605
type (sqlalchemy.sql.functions.now attribute), 605
type (sqlalchemy.sql.functions.percent_rank attribute), 606
type (sqlalchemy.sql.functions.rank attribute), 606
type (sqlalchemy.sql.functions.session_user attribute), 607
type (sqlalchemy.sql.functions.sysdate attribute), 607
type (sqlalchemy.sql.functions.user attribute), 607
type_coerce() (in module sqlalchemy.sql.expression), 476
type_descriptor() (sqlalchemy.engine.default.DefaultDialect method), 830
type_descriptor() (sqlalchemy.engine.interfaces.Dialect class method), 837
type_engine() (sqlalchemy.types.TypeDecorator method), 726
TypeCoerce (class in sqlalchemy.sql.expression), 503
TypeDecorator (class in sqlalchemy.types), 720
TypeEngine (class in sqlalchemy.types), 734
TypeEngine.Comparator (class in sqlalchemy.types), 734

U

UnaryExpression (class in sqlalchemy.sql.expression), 506
unbake_lazy_loaders() (in module sqlalchemy.ext.baked), 365

UnboundExecutionError, 823
undefer() (in module sqlalchemy.orm), 143
undefer() (sqlalchemy.orm.strategy_options.Load method), 188
undefer_group() (in module sqlalchemy.orm), 144
undefer_group() (sqlalchemy.orm.strategy_options.Load method), 188
unformat_identifiers() (sqlalchemy.sql.compiler.IdentifierPreparer method), 840
Unicode (class in sqlalchemy.types), 711
UnicodeText (class in sqlalchemy.types), 711
uninstall_descriptor() (sqlalchemy.ext.instrumentation.InstrumentationManager method), 417
uninstall_member() (sqlalchemy.ext.instrumentation.InstrumentationManager method), 417
union() (in module sqlalchemy.sql.expression), 512
union() (sqlalchemy.orm.query.Query method), 176
union() (sqlalchemy.sql.expression.Select method), 557
union_all() (in module sqlalchemy.sql.expression), 512
union_all() (sqlalchemy.orm.query.Query method), 177
union_all() (sqlalchemy.sql.expression.Select method), 557
unique_connection() (sqlalchemy.pool.Pool method), 785
unique_connection() (sqlalchemy.pool.QueuePool method), 786
unique_params() (sqlalchemy.sql.expression.Alias method), 516
unique_params() (sqlalchemy.sql.expression.ClauseElement method), 481
unique_params() (sqlalchemy.sql.expression.ColumnElement method), 490
unique_params() (sqlalchemy.sql.expression.CompoundSelect method), 524
unique_params() (sqlalchemy.sql.expression.CTE method), 528
unique_params() (sqlalchemy.sql.expression.Delete method), 574
unique_params() (sqlalchemy.sql.expression.GenerativeSelect method), 539
unique_params() (sqlalchemy.sql.expression.Insert method), 580
unique_params() (sqlalchemy.sql.expression.Join method), 546
unique_params() (sqlalchemy.sql.expression.Select method), 557
unique_params() (sqlalchemy.sql.expression.Update method), 588
unique_params() (sqlalchemy.sql.expression.UpdateBase method), 595
UniqueConstraint (class in sqlalchemy.schema), 685
UNIQUEIDENTIFIER (class in sqlalchemy.dialects.mssql), 856
unloaded (sqlalchemy.orm.state.InstanceState attribute), 303
UnmappedClassError, 331
UnmappedColumnError, 331
UnmappedError, 331
UnmappedInstanceError, 331
unmodified (sqlalchemy.orm.state.InstanceState attribute), 303
unmodified_intersection() (sqlalchemy.orm.state.InstanceState method), 304
unpickle() (sqlalchemy.orm.events.InstanceEvents method), 275
unregister() (sqlalchemy.orm.instrumentation.ClassManager method), 292
UnsupportedCompilationError, 823
UOWTransaction (class in sqlalchemy.orm.session), 330
Update (class in sqlalchemy.sql.expression), 583
update() (in module sqlalchemy.sql.expression), 568
update() (sqlalchemy.ext.mutable.MutableDict method), 400
update() (sqlalchemy.orm.collections.MappedCollection method), 125
update() (sqlalchemy.orm.instrumentation.ClassManager method), 292
update() (sqlalchemy.orm.query.Query method), 177
update() (sqlalchemy.schema.Table method), 644
update() (sqlalchemy.sql.expression.TableClause method), 567
update_execution_options() (sqlalchemy.engine.Engine method), 771
update_from_clause() (sqlalchemy.sql.compiler.SQLCompiler method), 841
update_limit_clause() (sqlalchemy.sql.compiler.SQLCompiler method), 841
update_tables_clause() (sqlalchemy.sql.compiler.SQLCompiler method), 841
UpdateBase (class in sqlalchemy.sql.expression), 590
URL (class in sqlalchemy.engine.url), 746
user (class in sqlalchemy.sql.functions), 607
UserDefinedType (class in sqlalchemy.types), 733
UUID (class in sqlalchemy.dialects.postgresql), 916

V

validates() (in module sqlalchemy.orm), 49
validators (sqlalchemy.orm.mapper.Mapper attribute), 89
vals() (sqlalchemy.dialects.postgresql.HSTORE.Comparator method), 911
value (sqlalchemy.orm.state.AttributeState attribute), 291
value() (sqlalchemy.orm.query.Query method), 178
values() (sqlalchemy.orm.instrumentation.ClassManager method), 292
values() (sqlalchemy.orm.query.Query method), 178
values() (sqlalchemy.sql.expression.Insert method), 581
values() (sqlalchemy.sql.expression.Update method), 588

values() (sqlalchemy.sql.expression.ValuesBase method), 596
ValuesBase (class in sqlalchemy.sql.expression), 595
VARBINARY (class in sqlalchemy.dialects.mysql), 878
VARBINARY (class in sqlalchemy.types), 718
VARCHAR (class in sqlalchemy.dialects.mssql), 856
VARCHAR (class in sqlalchemy.dialects.mysql), 878
VARCHAR (class in sqlalchemy.types), 718
Variant (class in sqlalchemy.types), 738
viewitems() (sqlalchemy.orm.instrumentation.ClassManager method), 292
viewkeys() (sqlalchemy.orm.instrumentation.ClassManager method), 292
viewvalues() (sqlalchemy.orm.instrumentation.ClassManager method), 292

W

was_deleted (sqlalchemy.orm.state.InstanceState attribute), 304
was_deleted() (in module sqlalchemy.orm.util), 255
where() (sqlalchemy.sql.expression.Delete method), 574
where() (sqlalchemy.sql.expression.ScalarSelect method), 546
where() (sqlalchemy.sql.expression.Select method), 557
where() (sqlalchemy.sql.expression.Update method), 590
whereclause (sqlalchemy.orm.query.Query attribute), 179
with_criteria() (sqlalchemy.ext.baked.BakedQuery method), 364
with_entities() (sqlalchemy.orm.query.Query method), 179
with_for_update() (sqlalchemy.orm.query.Query method), 179
with_for_update() (sqlalchemy.sql.expression.CompoundSelect method), 524
with_for_update() (sqlalchemy.sql.expression.GenerativeSelect method), 539
with_for_update() (sqlalchemy.sql.expression.Select method), 558
with_hint() (sqlalchemy.orm.query.Query method), 179
with_hint() (sqlalchemy.sql.expression.Delete method), 574
with_hint() (sqlalchemy.sql.expression.Insert method), 582
with_hint() (sqlalchemy.sql.expression.Select method), 558
with_hint() (sqlalchemy.sql.expression.Update method), 590
with_hint() (sqlalchemy.sql.expression.UpdateBase method), 595
with_labels() (sqlalchemy.orm.query.Query method), 180
with_lockmode() (sqlalchemy.orm.query.Query method), 180
with_only_columns() (sqlalchemy.sql.expression.Select method), 559
with_parent() (in module sqlalchemy.orm), 188
with_parent() (sqlalchemy.orm.query.Query method), 180
with_polymorphic() (in module sqlalchemy.orm), 61
with_polymorphic() (sqlalchemy.orm.query.Query method), 180
with_polymorphic_mappers (sqlalchemy.orm.mapper.Mapper attribute), 89
with_session() (sqlalchemy.orm.query.Query method), 181
with_statement_hint() (sqlalchemy.orm.query.Query method), 181
with_statement_hint() (sqlalchemy.sql.expression.Select method), 560
with_transformation() (sqlalchemy.orm.query.Query method), 181
with_variant() (sqlalchemy.types.TypeDecorator method), 726
with_variant() (sqlalchemy.types.TypeEngine method), 738
within_group() (in module sqlalchemy.sql.expression), 477
within_group() (sqlalchemy.sql.functions.FunctionElement method), 601
within_group_type() (sqlalchemy.sql.functions.FunctionElement method), 601
within_group_type() (sqlalchemy.sql.functions.OrderedSetAgg method), 602
WithinGroup (class in sqlalchemy.sql.expression), 503

Y

YEAR (class in sqlalchemy.dialects.mysql), 879
yield_per() (sqlalchemy.orm.query.Query method), 181

Z

zero_indexes (sqlalchemy.types.ARRAY attribute), 714