LINQ over DataSet

LINQ Preview, May 2006

Table of Contents

1. Introduction

One of the key elements of the ADO.NET programming model is the ability to explicitly cache data in a disconnected and backend agnostic manner using the DataSet. A DataSet represents a set of tables and relationships, along with the appropriate metadata to describe the structure and constraints of the data contained in it. ADO.NET includes various classes that make it easy to load data from a database into a DataSet, and push changes made within a DataSet back into the database.

One interesting aspect of DataSet is that it allows applications to bring a subset of the information contained in a database into the application space and then manipulate it in-memory while retaining its relational shape. This enables many scenarios that require flexibility in how data is represented and handled. In particular, generic reporting, analysis, and intelligence applications support this method of manipulation.

In order to query the data within a DataSet, the DataSet API includes methods, such as DataTable.Select(), for searching for data in certain, somewhat predefined ways. However, there hasn't been a general mechanism available for rich query over DataSet objects that provided the typical expressiveness required in many data-centric applications.

LINQ provides a unique opportunity to introduce rich query capabilities on top of DataSet and to do it in a way that integrates with the environment.

2. What's in the Preview

This LINQ Preview now includes support for LINQ over the DataSet. This version supports the standard query operators, plus a few DataSet-specificoperators. Since LINQ over DataSet fully implements the operator patterns expected by LINQ-enabled languages, the query comprehensions syntax in C# can also be used against DataTable objects.

In the Preview, DataSet exposes DataTable objects as enumerations of DataRow objects. The standard query operators' implementation actually executes the queries on enumerations of DataRow objects. This results in a very straightforward implementation and one that is highly consistent with the standard query operators. The implication is that certain classes of queries do not get optimized like a typical database would do; for example, even if there is an index over the columns involved in join condition, the query would not use it for execution.

Additionally, this LINQ Preview includes a modified version of the typed-DataSet generator, which enables the compiler to validate field names, data types, etc. while generating objects from DataRows.

Finally, the "101 LINQ samples" applied to DataSet have been included (there are actually more then 120 samples of LINQ queries against the DataSet). To see these samples, start the sample application to see how to formulate various queries against DataSet.

3. Using LINQ over DataSets

3.1 Creating a Visual Studio project that supports LINQ over DataSets

For this preview, to use LINQ queries on top of the DataSet, you will need to create one of the special LINQ projects provided in the preview. The pre-existing Visual Studio projects will not include LINQ query capabilities for DataSet.

With the LINQ Preview projects, Visual Studio will use the new C# compiler that supports LINQ, and will also add a reference to System.Data.Extensions.dll (where LINQ over DataSet support is implemented); in these project types Visual Studio will also use an updated typed-DataSet generator that generates LINQ-compatible DataSets. As noted above, LINQ support for DataSet is implemented entirely in System.Data.Extensions.dll and doesn't require a new version of System.Data.dll.

3.2 Loading data into DataSets

Before using LINQ queries on your DataSets, you'll first need to populate them with data. From the LINQ over DataSet perspective, how data is loaded into the DataSet is not important. Two common methods are to use the DataAdapter class to retrieve data from a database or take advantage of DLinq to query the database and then load the resulting data into a DataSet.

For example, to load data into the DataSet using DLinq from the "AdventureWorks" sample database included with SQL Server 2005, you could use the following code snippet. This particular example obtains the data from the SalesOrderHeader and SalesOrderDetail tables:

```
// Adventureworks inherits from DataContext and was generated using SqlMetal
static void FillOrders(Adventureworks adventureWorks, DataSet ds) {


     // Now execute a query for the SalesOrderHeader
     //information and specify only from the year 2002
    var salesOrderHeader = from salesHeader in

                           adventureWorks.Sales.SalesOrderHeaders

                           where salesHeader.OrderDate.Year == 2002

                           select new {ID=salesHeader.SalesOrderID,

                               salesHeader.OrderDate,

                               IsOnline=salesHeader.OnlineOrderFlag,

                               OrderNumber=salesHeader.SalesOrderNumber};


    // Now execute a query for the SalesOrderDetails, joined with the
    // SalesOrderHeaders and specify only from the year 2002
    var salesHistory = from salesDetail in

                       adventureWorks.Sales.SalesOrderDetails

                       join salesHeader in
```

```
                              adventureWorks.Sales.SalesOrderHeaders
                              on salesDetail.SalesOrderID equals
                                   salesHeader.SalesOrderID
                              where salesHeader.OrderDate.Year == 2002
                              select new{ID=salesDetail.SalesOrderID,
                                  DetailId=salesDetail.SalesOrderDetailID,
                                  Qty=salesDetail.OrderQty,
                                  salesDetail.ProductID,
                                  salesDetail.UnitPrice};

        // And finally add the results of these queries to a DataSet using the
        // custom ToDataTable() operator
        ds.Tables.Add(salesOrderHeader.ToDataTable());
        ds.Tables.Add(salesHistory.ToDataTable());
    }
```

And to load data into the DataSet using a DataAdapter from the same database, you could use the following code snippet.

```
    void FillOrders(DataSet ds) {
        // Create a new adapter and give it a query to fetch all the order headers
        // and lines for a given year. Point connection information to the
        // configuration setting "AdventureWorks".
        SqlDataAdapter da = new SqlDataAdapter(
            "SELECT SalesOrderID, OrderDate, OnlineOrderFlag, SalesOrderNumber " +
            "FROM Sales.SalesOrderHeader " +
            "WHERE DATEPART(YEAR, OrderDate) = @year; " +

            "SELECT d.SalesOrderID, d.SalesOrderDetailID, d.OrderQty, " +
            "       d.ProductID, d.UnitPrice " +
            "FROM Sales.SalesOrderDetail d " +
            "INNER JOIN Sales.SalesOrderHeader h " +
            "ON d.SalesOrderID = h.SalesOrderID " +
            "WHERE DATEPART(YEAR, OrderDate) = @year",
            connectionString);

        da.SelectCommand.Parameters.AddWithValue("@year", 2002);
        da.TableMappings.Add("Table", "SalesOrderHeader");
        da.TableMappings.Add("Table1", "SalesOrderDetail");
        da.Fill(ds);
    }
```

For typed DataSets that have associated TableAdapters, you can also use the TableAdapter methods to load the data.

3.3 Querying DataSets

Once data has been loaded into the DataSet you can begin querying it. Formulating queries over DataSet with LINQ is just like using LINQ against any other data-source. When using LINQ queries over Dataset, the only particular aspect to keep in mind is that instead of querying an enumeration of a given custom type, you are querying an enumeration of DataRow objects, so you have all the members of DataRow available for query expressions. And as such, these query expressions can be as rich and complex as any that you currently use.

For example, continuing the example from the previous section, if the ID and date of the orders that were submitted online was required:

```
DataSet ds = new DataSet();
FillOrders(ds);


DataTable orders = ds.Tables["SalesOrderHeader"];


var ordersQuery = orders.ToQueryable();


var query = from o in ordersQuery
                where o.Field<bool>("OnlineOrderFlag") == true
                select new { SalesOrderID = o.Field<int>("SalesOrderID"),
                OrderDate = o.Field<DateTime>("OrderDate") };
```

LINQ and Query Comprehensions work on sources that are IEnumerable<T> or IQueryable<T>. As DataTable does not implement either interface, you must call ToQueryable() in order to use the DataTable as a source in a LINQ Queries.

Note the "Field" method, which accesses columns of the DataTable in a way that aligns with the LINQ query syntax. You could use the pre-existing column accessor in DataRow (e.g. o["OrderDate"]), but doing so would require you to cast the return object to the appropriate type and if the column is nullable you'd need to check if the value is null via the IsNull method on DataRow. The "Field" accessor is a generic method, which removes the need for the cast (the generic type is indicated instead) and supports nullable types (for example, you could use Field<int?>("age") if the "age" field could be null). For more details regarding the Field<T> and SetField<T> methods see Section 3.7.

When writing this query, for this preview only certain method and properties will be available via IntelliSense (e.g. Field will not show up in IntelliSense but is available for use). There are a number of these methods and properties that will not appear in IntelliSense, beyond just the Field method. These methods and properties are still available for use. In future preview releases, support for IntelliSense will improve.

The query above only referred to a single table. It is also possible to do cross-table queries. The following example does a traditional join to obtain the order lines for the online orders for August:

```
DataTable orders     = ds.Tables["SalesOrderHeader"];

DataTable orderLines = ds.Tables["SalesOrderDetail"];


var ordersQuery = orders.ToQueryable();

var orderLinesQuery = orderLines.ToQueryable();


var query = from o in ordersQuery
         join ol in orderLinesQuery
         on o.Field<int>("SalesOrderID") equals ol.Field<int>("SalesOrderID")
         where o.Field<bool>("OnlineOrderFlag") == true &&
         o.Field<DateTime>("OrderDate").Month == 8
         select new { SalesOrderID = o.Field<int>("SalesOrderID"),
                    SalesOrderDetailID = ol.Field<int>("SalesOrderDetailID"),
                    OrderDate = o.Field<DateTime>("OrderDate"),
                    ProductID = ol.Field<int>("ProductID") };
```

If you have a relationship set up in DataSet between tables, you can also use that to walk through relationships without having to explicitly state a join condition; for example, if there was a relationship between orders and order lines:

```
ds.Relations.Add("OrderLines",

    orders.Columns["SalesOrderID"], orderLines.Columns["SalesOrderID"]);
```

then it would be possible to write the query as follows (this is equivalent to the join query above):

```
var query = from o in ordersQuery
            where o.Field<bool>("OnlineOrderFlag") == true &&
                  o.Field<DateTime>("OrderDate").Month == 8
            from ol in o.GetChildRows("OrderLInes")
            select new {   SalesOrderID = o.Field<int>("SalesOrderID"),
                           SalesOrderDetailID =
                           ol.Field<int>("SalesOrderDetailID"),
                           OrderDate = o.Field<DateTime>("OrderDate"),
                           ProductID = ol.Field<int>("ProductID") };
```

Alternatively, it is possible to also use the sequence operator syntax, instead of query comprehensions. For example, here is the same query as above, re-written to use sequence operators:

```
var query = ordersQuery.Where(o => o.Field<bool>("OnlineOrderFlag") == true &&
            o.Field<DateTime>("OrderDate").Month == 8)
            .SelectMany(o => o.GetChildRows("OrderLines")
                                  .Select(ol =>
            new {
                SalesOrderID = o.Field<int>("SalesOrderID"),
                SalesOrderDetailID = ol.Field<int>("SalesOrderDetailID"),
                OrderDate = o.Field<DateTime>("OrderDate"),
                ProductID = ol.Field<int>("ProductID")
                }
            ));
```

3.4 Querying typed DataSets

LINQ queries over typed-DataSets work exactly like they do over regular DataSets. The difference is that since type information is present, uses of the generic "Field" accessor is not required and use of property names is available, greatly simplifying query formulation and enhancing readability. For example, in the previous section the query was:

```
DataTable orders     = ds.Tables["SalesOrderHeader"];

DataTable orderLines = ds.Tables["SalesOrderDetail"];


var query = from o in ordersQuery
            join ol in orderLinesQuery
            on o.Field<int>("SalesOrderID") equals ol.Field<int>("SalesOrderID")
            where o.Field<bool>("OnlineOrderFlag") == true &&
            o.Field<DateTime>("OrderDate").Month == 8
            select new { SalesOrderID = o.Field<int>("SalesOrderID"),
                         SalesOrderDetailID = ol.Field<int>("SalesOrderDetailID"),
                         OrderDate = o.Field<DateTime>("OrderDate"),
                         ProductID = ol.Field<int>("ProductID") };
```

Using a typed DataSet with Orders and OrderLines tables, the query would become:

```
 var query = from o in ds.Orders
            join ol in orderLinesQuery
            on o.SalesOrderID equals ol.SalesOrderID
            where o.OnlineOrderFlag == true &&
            o.OrderDate.Month == 8
            select new { o.SalesOrderID,
                         ol.SalesOrderDetailID,
                         o.OrderDate,
                           ol.ProductID };
```

By using typed-DataSets, the query becomes much easier to formulate and read.

3.5 Using LINQ over DataSet with TableAdapters

TableAdapters are a combination of a typed DataSet, pre-configured adapters that encapsulate connection information, and predefined queries that are exposed as methods and can either fill an existing typed DataTable or create and return a newly populated one.

In all cases, as TableAdapters operate on typed DataSets support for LINQ queries over them applies as described in section 3.4 above.

3.6 Custom Operators

To enable LINQ queries over DataSet there are a number of custom operators added to enable querying over a set of DataRows. These custom operators and their behaviors are described below:

3.6.1 ToDataTable

The `ToDataTable` operator creates a DataTable from a sequence.

```
public static DataTable ToDataTable (
   this IEnumerable<T> source);
```

The `ToDataTable` operator enumerates the source sequence and returns a DataTable containing the elements of the sequence. To accomplish this, each element within the sequence is iterated, and each property on the element is added as a new DataColumn and the value of each property is inserted. When new properties are found on subsequent elements in the sequence, new columns are added to the DataTable in order to support full data fidelity.

Here is an example that creates a new DataTable from a given source:

```
var ordersQuery = ds.Tables["SalesOrderHeader"].ToQueryable();
var orderLinesQuery = ds.Tables["SalesOrderDetail"].ToQueryable();

var query = from o in ordersQuery
        where o.Field<bool>("OnlineOrderFlag") == true
        select new {SalesOrderID = o.Field<int>("SalesOrderID"),
                    OrderDate = o.Field<DateTime>("OrderDate")};

DataTable newDataTable = query.ToDataTable();
```

3.6.2 LoadSequence

The `LoadSequence` operator adds data into an existing DataTable from a source sequence.

```
public static void LoadSequence(
    this IEnumerable<T> source);


public static void LoadSequence(
    this IEnumerable<T> source, System.Data.LoadOption option);
```

The `LoadSequence` method enumerates the source sequence and adds DataRows to the existing DataTable. To accomplish this, each element within the sequence is iterated, and each property on the element is added a a new DataColumn, the value of each property is inserted into that specific DataColumn, and then the DataRow is added to the existing DataTable. When new properties are found on subsequent elements in the sequence, new columns are added to the DataTable in order to support full data fidelity.

```
var ordersQuery = ds.Tables["SalesOrderHeader"].ToQueryable();
var orderLinesQuery = ds.Tables["SalesOrderDetail"].ToQueryable();

DataTable myDataTable = new DataTable("myTable");
myDataTable.Columns.Add(new DataColumn("SalesOrderID", typeof(int)));
myDataTable.Columns.Add(new DataColumn("OrderDate", typeof(DateTime)));


var query = from o in ordersQuery
             where o.Field<bool>("OnlineOrderFlag") == true
             select new {SalesOrderID = o.Field<int>("SalesOrderID"),
                         OrderDate = o.Field<DateTime>("OrderDate")};

myDataTable.LoadSequence(query);
```

3.6.3 DistinctRows

The `DistinctRows` operator eliminates duplicate elements from a sequence.

```
public static IEnumerable<DataRow> DistinctRows (
    this IEnumerable<DataRow> source);
```

The `DistinctRow` operator allocates and returns an enumerable set of DataRow's that capture the source argument. An `ArgumentNullException` is thrown if the source argument is null.

When the enumerable set of DataRow's returned by `DistinctRows` is enumerated, it enumerates the source sequence, yielding each DataRow that hasn't previously been yielded. DataRows are compared using the number of Columns, the static type for each Column, and then using `IComparable` on the dynamic type or the `Equals` static method in `System.Object`.

LINQ defines a number of set operators (`Distinct`, `EqualAll`, `Union`, `Intersect`, `Except`) which compares equality of source elements by calling the element's `GetHashCode` and `Equals` methods. In the case of DataRows, this does a reference comparison which is generally not the expected behavior for set operations over tabular data. Hence, DataSet adds a number of custom operators (`DistinctRows`, `EqualAllRows`, `UnionRows`, `IntersectRows`, and `ExceptRows`) that compare row values instead.

3.6.4 EqualAllRows

The `EqualAllRows` operator checks whether two sequences are equal.

```
public static bool EqualAllRows<DataRow>(
    this IEnumerable<DataRow> first,
    IEnumerable<DataRow> second);
```

The `EqualAllRows` operator enumerates the two source sequences in parallel and compares corresponding DataRows by comparing the number of Columns, the static type for each Column, and then using `IComparable` on the dynamic type or the `Equals` static method in `System.Object`. The method returns true if all corresponding elements compare equal and the two sequences are of equal length. Otherwise, the method returns false.

3.6.5 UnionRows

The `UnionRows` operator produces the set union of two sequences of DataRows.

```
public static IEnumerable<T> UnionRows<T>(
    this IEnumerable<T> first,
    IEnumerable<T> second);
```

The `Union` operator allocates and returns an enumerable object that captures the arguments passed to the operator. An `ArgumentNullException` is thrown if any argument is null.

When the object returned by `Union` is enumerated, it enumerates the first and second sequences, in that order, yielding each element that hasn't previously been yielded. DataRows are compared using the number of Columns, the static type for each Column, and then using `IComparable` on the dynamic type or the `Equals` static method in `System.Object`.

3.6.6 IntersectRows

The `IntersectRows` operator produces the set intersection of two sequences of DataRows.

```
public static IEnumerable<DataRow> IntersectRows(
    this IEnumerable<DataRow> first,
    IEnumerable<DataRow> second);
```

The `IntersectRows` operator allocates and returns an enumerable set of DataRows that captures the arguments passed to the operator.

When the object returned by `IntersectRows` is enumerated, it enumerates the first sequence, collecting all distinct DataRows of that sequence. It then enumerates the second sequence, marking those DataRows that occur in both sequences. It finally yields the marked DataRows in the order in which they were collected. DataRows are compared using the number of Columns, the static type for each Column, and then using `IComparable` on the dynamic type or the `Equals` static method in `System.Object`.

3.6.7 ExceptRows

The `ExceptRows` operator produces the set difference between two sequences of DataRows.

```
public static IEnumerable<DataRow> ExceptRows (
    this IEnumerable<DataRow> first,
    IEnumerable<DataRow> second);
```

The `ExceptRows` operator allocates and returns an enumerable set of DataRow's that captures the arguments passed to the operator. An `ArgumentNullException` is thrown if any argument is null.

When the sequence of DataRows returned by `ExceptRows` is enumerated, it enumerates the first sequence, collecting all distinct DataRows of that sequence. It then enumerates the second sequence, removing those DataRows that were also contained in the first sequence. It finally yields the remaining DataRows in the order in which they were collected. DataRows are compared using the number of Columns, the static type for each Column, and then using `IComparable` on the dynamic type or the `Equals` static method in `System.Object`.

3.7 Field<T> and SetField<T>

When working with DataSet and querying for specific values, there are a number of differences when working with LINQ that require a set of custom methods for accessing data contained within a row. In particular there are two scenarios that required the creation of `Field<T>` and `SetField<T> methods`. These are:

1) The difference between the representation of null values within the DataSet, which uses DbNull.Value, and LINQ, which uses the nullable types support introduced in the .NET Framework 2.0 release. Take the following example where the CarrierTrackingNumber field in the DataRow can be null, you must protect against the case when DbNull.Value will be returned as casting DbNull.Value to string will fail:

```
var query = from o in orderLinesQuery
            where !o.IsNull("CarrierTrackingNumber") &&
            (string)o["CarrierTrackingNumber"] == "AEB6-4356-80"
            select new { SalesOrderID = o.Field<int>("SalesOrderID"),
                             OrderDate = o.Field<DateTime>("OrderDate")
      };
```

Instead, consider the following query which uses the Field method, and guards against these null conditions within it:

```
var query = from o in orderLinesQuery
            where o. Field<string> ("CarrierTrackingNumber")
                                    == "AEB6-4356-80"
            select new { SalesOrderID = o.Field<int>("SalesOrderID"),
                             OrderDate = o.Field<DateTime>("OrderDate")
      };
```

2) When doing a comparison between column values of two DataRows, the comparison is done via the indexer on DataRow. For value types the comparison is done on the boxed value. Consider the following sample where the values for each DataRow are the same, the comparison will return false.

```
var query = from o in ordersQuery
            where o["SalesOrderID"] == o["SalesOrderID"]
            select new { SalesOrderID = o.Field<int>("SalesOrderID"),
                         OrderDate = o.Field<DateTime>("OrderDate") };
```

By introducing the Field method the composition of the query is the above cases is not only less verbose but also far less error prone.

3.7.1 Field<T>

The `Field<T>` method provides a way to get the value of a column within a DataRow without having to worry about the different representations of null values in DataSet and LINQ and also provides a way have comparisons of values correctly evaluated.

```
public static T Field (

    this DataRow first,
    System.Data.DataColumn column,

    System.Data.DataRowVersion version);


public static T Field (

    this DataRow first,
    string columnName,

    System.Data.DataRowVersion version);


public static T Field (

    this DataRow first,
    int ordinal,

    System.Data.DataRowVersion version);


public static T Field (

    this DataRow first,
    System.Data.DataColumn column);


public static T Field (

    this DataRow first,
    string columnName);


public static T Field (

    this DataRow first,
    int ordinal);
```

3.7.2 SetField<T>

The `SetField<T>` method provides a way to set the value of a column within a DataRow without having to worry about the different representations of null values in DataSet and LINQ.

```
public static void SetField (

    this DataRow first,
    System.Data.DataColumn column,
    T value);


public static void SetField (

    this DataRow first,
    string columnName,

    T value);


public static void SetField (

    this DataRow first,
    int ordinal,

    T value);
```

4. Future Directions

4.1 Integration with other LINQ sources

This preview contains one custom operator: *ToDataTable*. This operator causes a given query to execute and pushes the results into a DataTable object. For this preview a very simple approach was utilized: turning each public property and member variable into a column in the DataTable. Also, in this preview, there is no way to "go back" – that is, to push the changes in the resulting DataTable back to the source.

In the future the following enhancements are under consideration:

- Expand *ToDataTable* to understand relationships and to produce multiple tables in a DataSet, so as better to reflect the structure of the input enumeration/query.
- Have a full-circle model where you can obtain data querying a source (e.g. a DLinq data source), load it to a DataTable using *ToDataTable* or *DataTable.LoadSequence*( ), manipulate it in the table and then push it back to the data source as a set of changes to be applied to the underlying store.

4.2 More query processing

In this preview, queries against DataTable execute using a "brute force" approach. For example, if a join is executed against two tables and there is an index on one of them that's on the same column as the join condition, then the LINQ query could use the index to execute the query significantly more efficiently.

The LINQ team is currently evaluating building a basic infrastructure that can look at some of these smarter execution strategies for queries. Using straightforward, well-known techniques for heuristic optimizations, the query processor can achieve significant improvements in performance when querying against a medium and/or large number of rows (e.g. more than 1000 rows).

4.3 Sending feedback to our team

For any feedback regarding this preview post any thoughts, comments, questions, etc. to the LINQ Project forum at http://forums.microsoft.com/MSDN/ShowForum.aspx?ForumID=123&SiteID=1 .