SQL Server 索引基础知识

--------- siupan 整理

目录

1 记录数据的基本格式

由于需要给同事培训数据库的索引知识，就收集整理了这个系列的博客。发表在这里，也是对索引知识的一个总结回顾吧。通过总结，我发现自己以前很多很模糊的概念都清晰了很多。

不论是缓存的数据信息，还是物理保存的信息，他们的基本单位都是数据页。所以理解数据页是最最基础的知识点，本篇博客就介绍跟索引有关的数据页的一些基础知识。

1.1 数据页的基础知识

SQL Server 中数据存储的基本单位是页（Page）。数据库中的数据文件（.mdf 或 .ndf）分配的磁盘空间可以从逻辑上划分成页（从 0 到 n 连续编号）。磁盘 I/O 操作在页级执行。也就是说，SQL Server 每次读取或写入数据的最少数据单位是数据页。

注意：日志文件不是用这种方式存储的，而是一系列日志记录。

数据库被分成逻辑页面（每个页面 8KB），并且在每个文件中，所有页面都被连续地从 0 到 x 编号，其中 x 是由文件的大小决定的。我们可以通过指定一个数据库 ID、一个文件 ID、一个页码来引用任何一个数据页。当我们使用 ALTER DATABASE 命令来扩大一个文件时，新的空间会被加到文件的末尾。也就是说，我们所扩大文件的新空间第一个数据页的页码是 x+1。当我们使用 DBCC SHRINKDATABASE 或 DBCC SHRINKFILE 命令来收缩一个数据库时，将会从数据库中页码最高的页面（文件末尾）开始移除页面，并向页码较低的页面移动。这保证了一个文件中的页码总是连续的。

在 SQL Server 中，页的大小为 8 KB。这意味着 SQL Server 数据库中每 MB 有 128 页。依次类推。根据数据库的文件大小，我们可以算出数据库有多少数据页。

1.2 SQL Server 2005 有以下几种页类型：

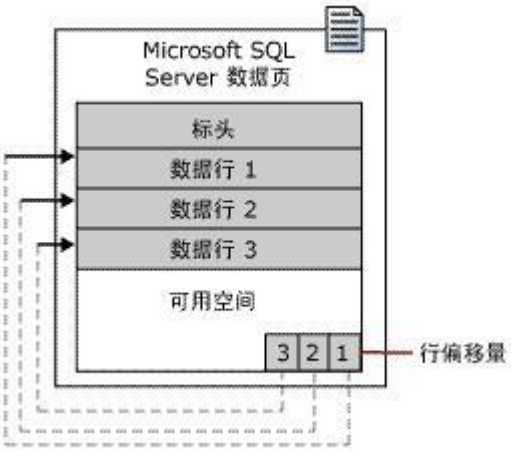
页类型	内容
Data	当 text in row 设置为 ON 时，包含除 text、ntext、image、nvarchar(max)、varchar(max)、varbinary(max) 和 xml 数据之外的所有数据的数据行。
Index	索引条目。
Text/Image	大型对象数据类型： text、 ntext、image、nvarchar(max)、varchar(max)、varbinary(max) 和 xml 数据。 数据行超过 8 KB 时为可变长度数据类型列： varchar、nvarchar、varbinary 和 sql_variant
Global Allocation Map、Shared	有关区是否分配的信息。

Global Allocation Map	
Page Free Space	有关页分配和页的可用空间的信息。
Index Allocation Map	有关每个分配单元中表或索引所使用的区的信息。
Bulk Changed Map	有关每个分配单元中自最后一条 BACKUP LOG 语句之后的大容量操作所修改的区的信息。
Differential Changed Map	有关每个分配单元中自最后一条 BACKUP DATABASE 语句之后更改的区的信息。

1.3 数据页（ Data 类型页）的结构示意图：

每页的开头是 96 字节的标头，用于存储有关页的系统信息。此信息包括页码、页类型、页的可用空间以及拥有该页的对象的分配单元 ID。

在数据页上，数据行紧接着标头按顺序放置。页的末尾是行偏移表，对于页中的每一行，每个行偏移表都包含一个条目。每个条目记录对应行的第一个字节与页首的距离。行偏移表中的条目的顺序与页中行的顺序相反。



有关数据页的更多知识，可以通过下面这篇文章获得更详细的了解：

估计在堆中存储数据所需的空间量
http://technet.microsoft.com/zh-cn/library/ms189124.aspx

另外也可以看我收集的资料：怎样查看表的数据页的结构
http://blog.joycode.com/ghj/articles/113108.aspx

1.4 对大型行的支持

在 SQL Server 2005 中，行不能跨页，但是行的部分可以移出行所在的页，因此行实际可能非常大。
（比如：一行多列时，这一行的部分列在数据页 A，部分列在数据页 B）

页的单个行中的最大数据量和开销是 8,060 字节 (8 KB)。但是，这不包括用 Text/Image 页类型存储的数据。
在 SQL Server 2005 中，包含 **varchar**、**nvarchar**、**varbinary** 或 **sql_variant** 列的表不受此限制的约束。
当表中的所有固定列和可变列的行的总大小超过限制的 8,060 字节时，SQL Server 将从最大长度的列开始动态将一个或多个可变长度列移动到 ROW_OVERFLOW_DATA 分配单元中的页。
每当插入或更新操作将行的总大小增大到超过限制的 8,060 字节时，将会执行此操作。
将列移动到 ROW_OVERFLOW_DATA 分配单元中的页后，将在 IN_ROW_DATA 分配单元中的原始页上维护 24 字节的指针。
如果后续操作减小了行的大小，SQL Server 会动态将列移回到原始数据页。

1.5 SQL Server 的数据页缓存

SQL Server 数据库的主要用途是存储和检索数据，因此密集型磁盘 I/O 是数据库引擎的一大特点。此外，完成磁盘 I/O 操作要消耗许多资源并且耗时较长，所以 SQL Server 侧重于提高 I/O 效率。缓冲区管理是实现高效 I/O 操作的关键环节。SQL Server 2005 的缓冲区管理组件由下列两种机制组成：用于访问及更新数据库页的缓冲区管理器和用于减少数据库文件 I/O 的缓冲区高速缓存（又称为"缓冲池"）。

1.6 缓冲区管理的工作原理

一个缓冲区就是一个 8KB 大小的内存页，其大小与一个数据页或索引页相当。因此，缓冲区高速缓存被划分为多个 8KB 页。缓冲区管理器负责将数据页或索引页从数据库磁盘文件读入缓冲区高速缓存中，并将修改后的页写回磁盘。页一直保留在缓冲区高速缓存中，直到已有一段时间未对其进行引用或者缓冲区管理器需要缓冲区读取更多数据。数据只有在被修改后才重新写入磁盘。在将缓冲区高速缓存中的数据写回磁盘之前，可对其进行多次修改。

1.7 实验

下面做一个简单的实验来看你是否已经掌握的上面的知识点：

准备测试环境

在一个 SQL 2005 数据库中，执行下面脚本。

简单来说，就是创建了 2 个表，注意这两个表，一个是存储的 nchar(2019) 的字段，一个是存储的 nchar(2020) 的字段。 我们将来看这两个表在同样数据下，存储所花费的空间大小。由于缓存和物理存储的基本单位都是数据页，这个表物理存储的大小跟全部缓存的大小会是一样的。

然后我们每个表填充 20 个数据。

复制

```
-- 创建 2 个测试表
CREATE TABLE [dbo].[Table_2019]([Data] [nchar](2019) NOT NULL)
CREATE TABLE [dbo].[Table_2020]([Data] [nchar](2020) NOT NULL)
go
-- 填充数据
declare @i int
set @i = 0
while(@i < 20)
```

```
begin
    insert Table_2019(Data) values('')
    insert Table_2020(Data) values('')
    select @i = @i + 1
end
go
```

这里我们用 nchar 数据类型，是因为：
当指定了 NOT NULL 子句时， nchar 数据类型是一种长度固定的数据类型。
如果插入值的长度比 nchar NOT NULL 列的长度小，将在值的右边填补空格直到达到列的长度。
例如，如果某列定义为 nchar(10)，而要存储的数据是“music”，则 SQL Server 将数据存储为“music_____”，这里“_”表示空格。
http://technet.microsoft.com/zh-cn/library/ms175055.aspx
这样我们填充测试数据的脚本就非常简单。
而且计算数据行所占的空间也非常简单。
另外，我们建立的这两个表都没有索引，所以他们都是堆，有关估计在堆中存储数据所需的空间量请参看以下文章：
http://technet.microsoft.com/zh-cn/library/ms189124.aspx

完成准备工作后，我们来查看这两个所占空间的大小。在 SQL Server Management Studio 中，我们选择测试数据库，然后在右键菜单中依次选择
Reports --> Standard Reports --> Disk Usage by Top Tables 或者 Disk Usage by Table 就可以看到下面统计数据。

Disk Usage by Top Tables: [ghj_Demo]

on GHJ1976-PC\SQLEXPRESS at 2007/12/27 9:21:33

This report provides detailed data on the utilization of disk space by top 1000 tables within the Database.

Table Name	# Records	Reserved (KB)	Data (KB)	Indexes (KB)	Unused (KB)
dbo.Table_2020	20	200	160	8	32
dbo.Table_2019	20	136	80	8	48

这两个表同样 20 条记录。Table_2020 表数据占了 160kb ，即 20 个数据页。Table_2019 表数据占了 80 kb，即 10 个数据页。

为何会这样呢？
Table_2020 表的 1 个数据页只能放下 1 个数据行。
Table_2019 表的 1 个数据页只能放下 2 个数据行。
这两个表的字段长度只差 2 个字节，但是物理存储却是一倍的差距。

参考资料：

SQL Server 数据库中存储引擎深入探讨
http://tech.ccidnet.com/art/1106/20070320/1040665_3.html

《 Microsoft SQL Server 2005 技术内幕：存储引擎》 这本书电子版的一部分
http://book.csdn.net/bookfiles/504/10050417350.shtml

MSDN 中关于"页和区"的描述
http://technet.microsoft.com/zh-cn/library/ms190969.aspx

聚集索引结构
http://technet.microsoft.com/zh-cn/library/ms177443.aspx

行溢出数据超过 8 KB
http://technet.microsoft.com/zh-cn/library/ms186981.aspx

缓冲区管理
http://technet.microsoft.com/zh-cn/library/aa337525.aspx

估计堆的大小
http://technet.microsoft.com/zh-cn/library/ms189124.aspx

nchar 和 nvarchar (Transact-SQL)
http://technet.microsoft.com/zh-cn/library/ms186939.aspx

Teched 2007 上 吴家震 主讲的"微软 SQL 服务器 Always-On Tech-nologies: 高级索引策略" 录像下载地址:
http://msevents.microsoft.com/CUI/EventDetail.aspx?EventID=1032364059&Culture=zh-CN
注意, 这个页面标示的是 "SharePoint 2007 网站性能调优",但是其实是高级索引策略,微软弄错文件了,害得我一个个下下来看,哪个是需要的录像.

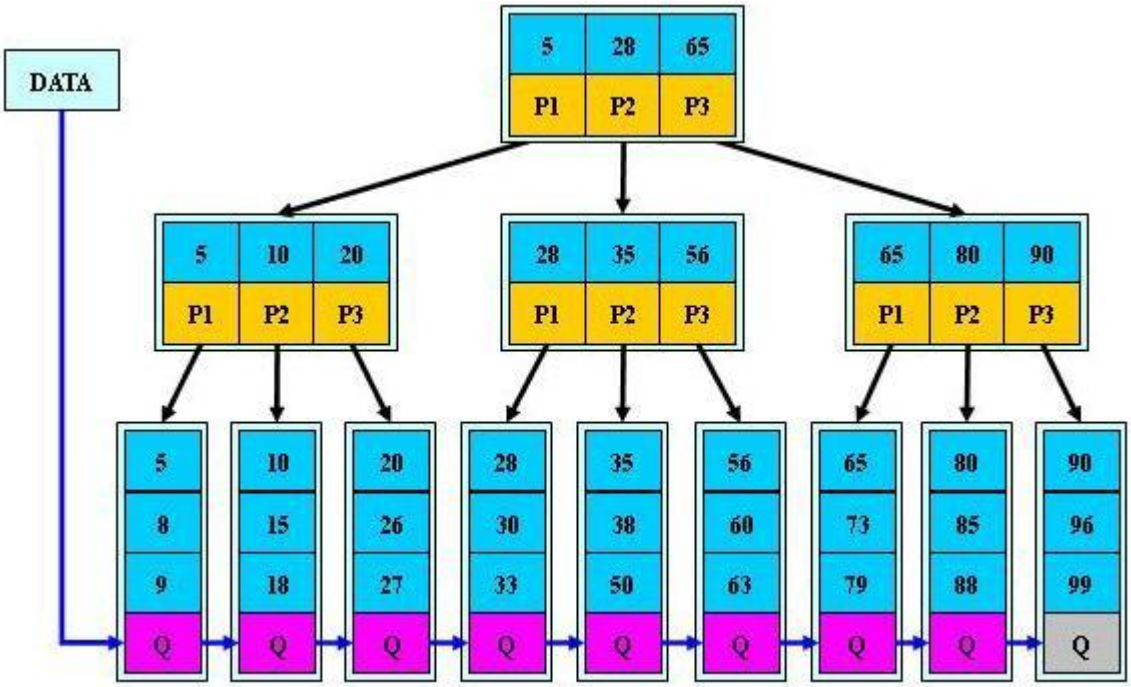
2 聚集索引，非聚集索引

由于需要给同事培训数据库的索引知识，就收集整理了这个系列的博客。发表在这里，也是对索引知识的一个总结回顾吧。通过总结，我发现自己以前很多很模糊的概念都清晰了很多。

不论是 聚集索引，还是非聚集索引，都是用 B+树来实现的。我们在了解这两种索引之前，需要先了解 B+树。如果你对 B 树不了解的话，建议参看以下几篇文章：

BTree,B-Tree,B+Tree,B*Tree 都是什么
http://blog.csdn.net/manesking/archive/2007/02/09/1505979.aspx

2.1 B+ 树的结构图:



B+ 树的特点:

· 所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的；

· 不可能在非叶子结点命中；

· 非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层；

B+ 树中增加一个数据，或者删除一个数据，需要分多种情况处理，比较复杂，这里就不详述这个内容了。

2.2 聚集索引（ Clustered Index）

· 聚集索引的叶节点就是实际的数据页

· 在数据页中数据按照索引顺序存储

· 行的物理位置和行在索引中的位置是相同的

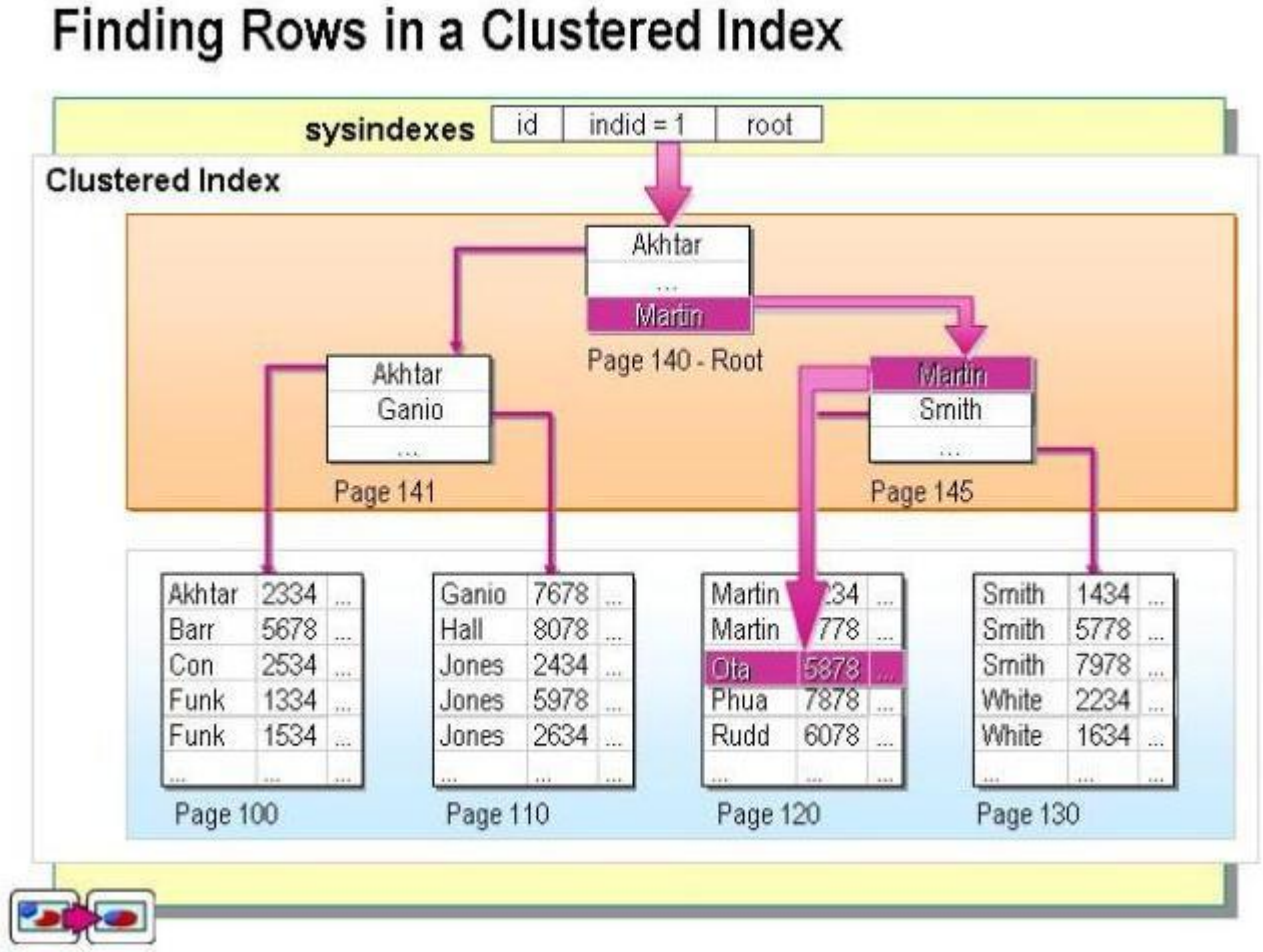
· 每个表只能有一个聚集索引

· 聚集索引的平均大小大约为表大小的 5%左右

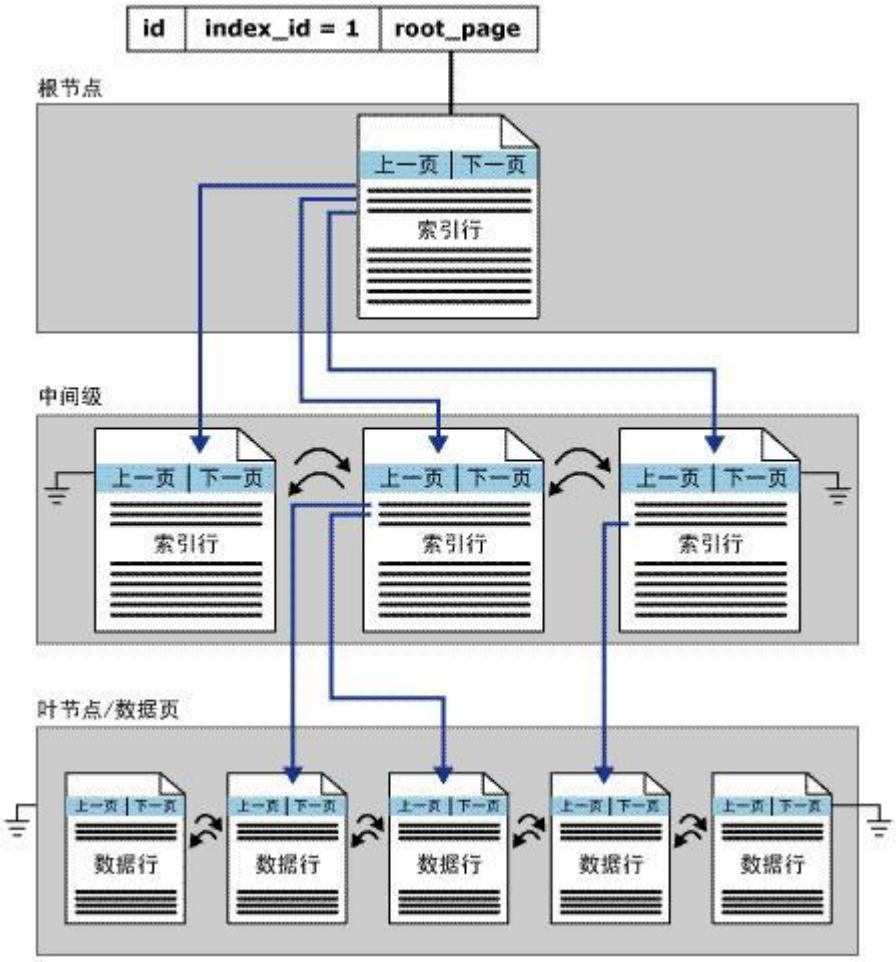
下面是两副简单描述聚集索引的示意图：

在聚集索引中执行下面语句的的过程：

select * from table where firstName = 'Ota'



一个比较抽象点的聚集索引图示：



2.3 非聚集索引 （ Unclustered Index）

· 非聚集索引的页，不是数据，而是指向数据页的页。

· 若未指定索引类型，则默认为非聚集索引

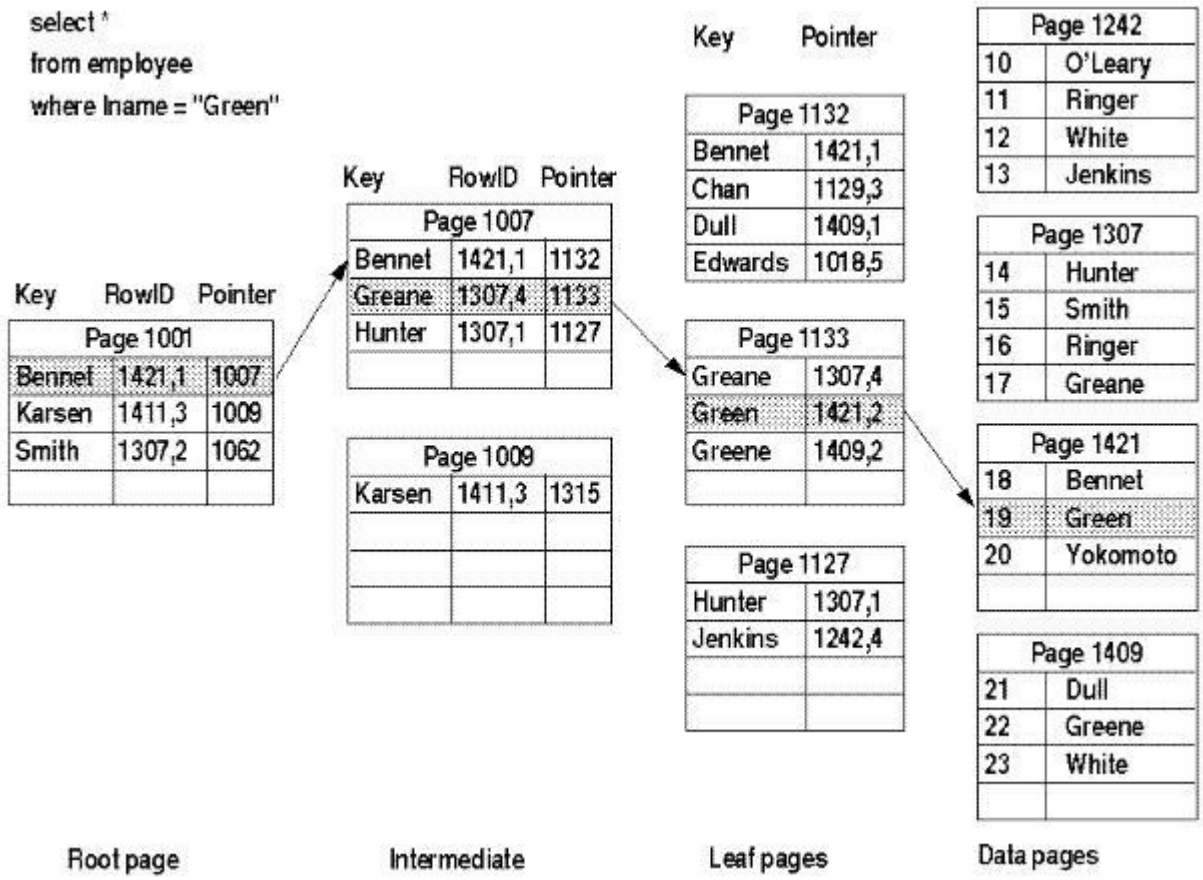
· 叶节点页的次序和表的物理存储次序不同

· 每个表最多可以有 249 个非聚集索引

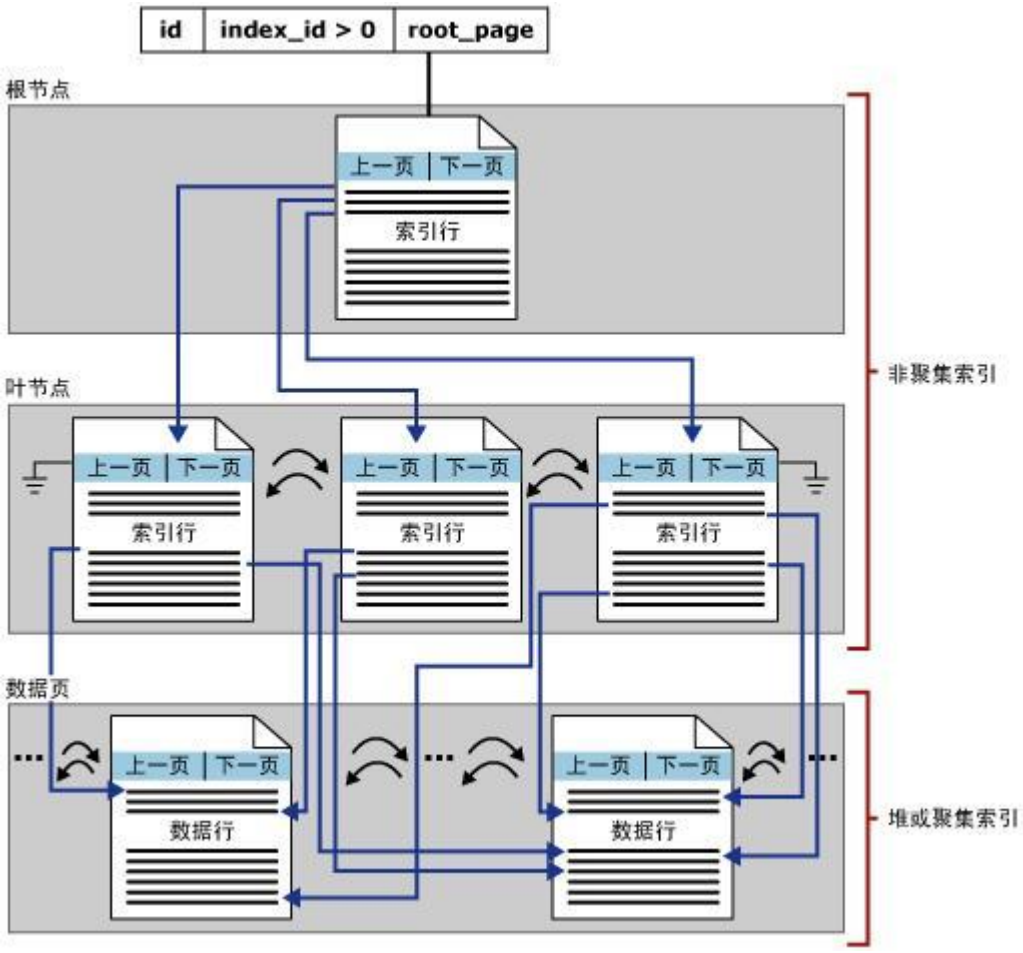
· 在非聚集索引创建之前创建聚集索引（否则会引发索引重建）

在非聚集索引中执行下面语句的的过程：

select * from employee where lname = 'Green'

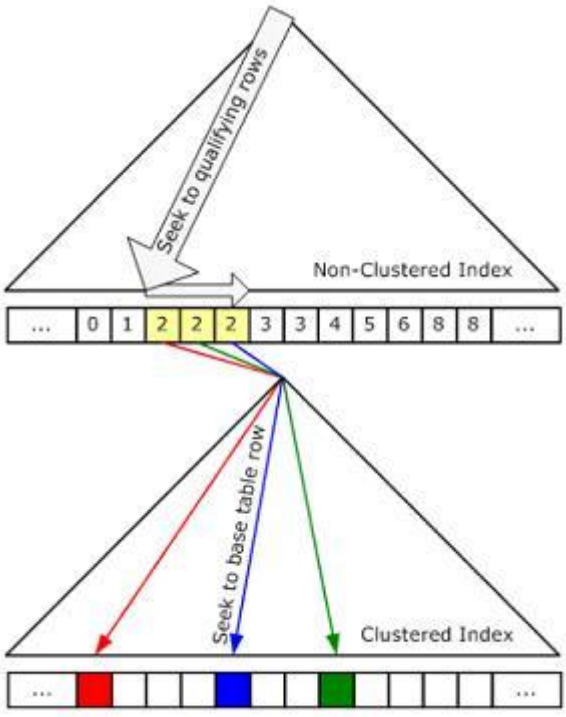


一个比较抽象点的非聚集索引图示：



2.4 什么是 Bookmark Lookup

虽然 SQL 2005 中已经不在提 Bookmark Lookup 了(换汤不换药)，但是我们的很多搜索都是用的这样的搜索过程，如下：先在非聚集中找，然后再在聚集索引中找。



在 http://www.sqlskills.com/ 提供的一个例子中，就给我们演示了 Bookmark Lookup 比 Table Scan 慢的情况，例子的脚本如下：

复制

```
USE CREDIT
go

-- These samples use the Credit database. You can download and restore the
-- credit database from here:
-- http://www.sqlskills.com/resources/conferences/CreditBackup80.zip

-- NOTE: This is a SQL Server 2000 backup and MANY examples will work on
-- SQL Server 2000 in addition to SQL Server 2005.
------------------------------------------------------------------------------
-- (1) Create two tables which are copies of charge:
------------------------------------------------------------------------------

-- Create the HEAP
SELECT * INTO ChargeHeap FROM Charge
go

-- Create the CL Table
SELECT * INTO ChargeCL FROM Charge
go

CREATE CLUSTERED INDEX ChargeCL_CLInd ON ChargeCL (member_no, charge_no)
go

------------------------------------------------------------------------------
-- (2) Add the same non-clustered indexes to BOTH of these tables:
------------------------------------------------------------------------------

-- Create the NC index on the HEAP
CREATE INDEX ChargeHeap_NCInd ON ChargeHeap (Charge_no)
go

-- Create the NC index on the CL Table
CREATE INDEX ChargeCL_NCInd ON ChargeCL (Charge_no)
go

------------------------------------------------------------------------------
-- (3) Begin to query these tables and see what kind of access and I/O returns
------------------------------------------------------------------------------

-- Get ready for a bit of analysis:
SET STATISTICS IO ON
-- Turn Graphical Showplan ON (Ctrl+K)
```

```
-- First, a point query (also, see how a bookmark lookup looks in 2005)
SELECT * FROM ChargeHeap WHERE Charge_no = 12345
go

SELECT * FROM ChargeCL WHERE Charge_no = 12345
go

-- What if our query is less selective?
-- 1000 is .0625% of our data... (1,600,000 million rows)
SELECT * FROM ChargeHeap WHERE Charge_no < 1000
go

SELECT * FROM ChargeCL WHERE Charge_no < 1000
go

-- What if our query is less selective?
-- 16000 is 1% of our data... (1,600,000 million rows)
SELECT * FROM ChargeHeap WHERE Charge_no < 16000
go

SELECT * FROM ChargeCL WHERE Charge_no < 16000
go

------------------------------------------------------------------------------
-- (4) What's the EXACT percentage where the bookmark lookup isn't worth it?
------------------------------------------------------------------------------

-- What happens here: Table Scan or Bookmark lookup?
SELECT * FROM ChargeHeap WHERE Charge_no < 4000
go

SELECT * FROM ChargeCL WHERE Charge_no < 4000
go

-- What happens here: Table Scan or Bookmark lookup?
SELECT * FROM ChargeHeap WHERE Charge_no < 3000
go

SELECT * FROM ChargeCL WHERE Charge_no < 3000
go

-- And - you can narrow it down by trying the middle ground:
-- What happens here: Table Scan or Bookmark lookup?
SELECT * FROM ChargeHeap WHERE Charge_no < 3500
go
```

```
SELECT * FROM ChargeCL WHERE Charge_no < 3500
go

-- And again:
SELECT * FROM ChargeHeap WHERE Charge_no < 3250
go

SELECT * FROM ChargeCL WHERE Charge_no < 3250
go

-- And again:
SELECT * FROM ChargeHeap WHERE Charge_no < 3375
go

SELECT * FROM ChargeCL WHERE Charge_no < 3375
go

-- Don't worry, I won't make you go through it all :)



-- For the Heap Table (in THIS case), the cutoff is: 0.21%
SELECT * FROM ChargeHeap  WHERE Charge_no < 3383
go
SELECT * FROM ChargeHeap WHERE Charge_no < 3384
go


-- For the Clustered Table (in THIS case), the cut-off is: 0.21%
SELECT * FROM ChargeCL WHERE Charge_no < 3438

SELECT * FROM ChargeCL WHERE Charge_no < 3439
go
```

这个例子也就是 吴家震 在 Teched 2007 上的那个演示例子。

小结：

这篇博客只是简单的用几个图表来介绍索引的实现方法： B+数， 聚集索引，非聚集索引，Bookmark Lookup 的信息而已。

参考资料：

表组织和索引组织
http://technet.microsoft.com/zh-cn/library/ms189051.aspx

How Indexes Work
http://manuals.sybase.com/onlinebooks/group-asarc/asg1200e/aseperf/@Generic__BookTextView/3358

Bookmark Lookup
http://blogs.msdn.com/craigfr/archive/2006/06/30/652639.aspx

Logical and Physical Operators Reference
http://msdn2.microsoft.com/en-us/library/ms191158.aspx

3 测试中一些常看的指标和清除缓存的方法

之前的两篇博客中有 2 个例子，来演示要讲述的内容。其中提到了部分查看数据库状态的方法，那里并不是很全面，这篇博客罗列几个我们在后面系列博客中会用到查看这些状态，数据的地方。以及测试中清除缓存的方法。
前面两篇博客的链接地址如下：

SQL Server 索引基础知识(1)--- 记录数据的基本格式

SQL Server 索引基础知识(2)----聚集索引，非聚集索引

3.1 如何获得索引的一些信息

比如：查看索引的深度 SQL 脚本如下：

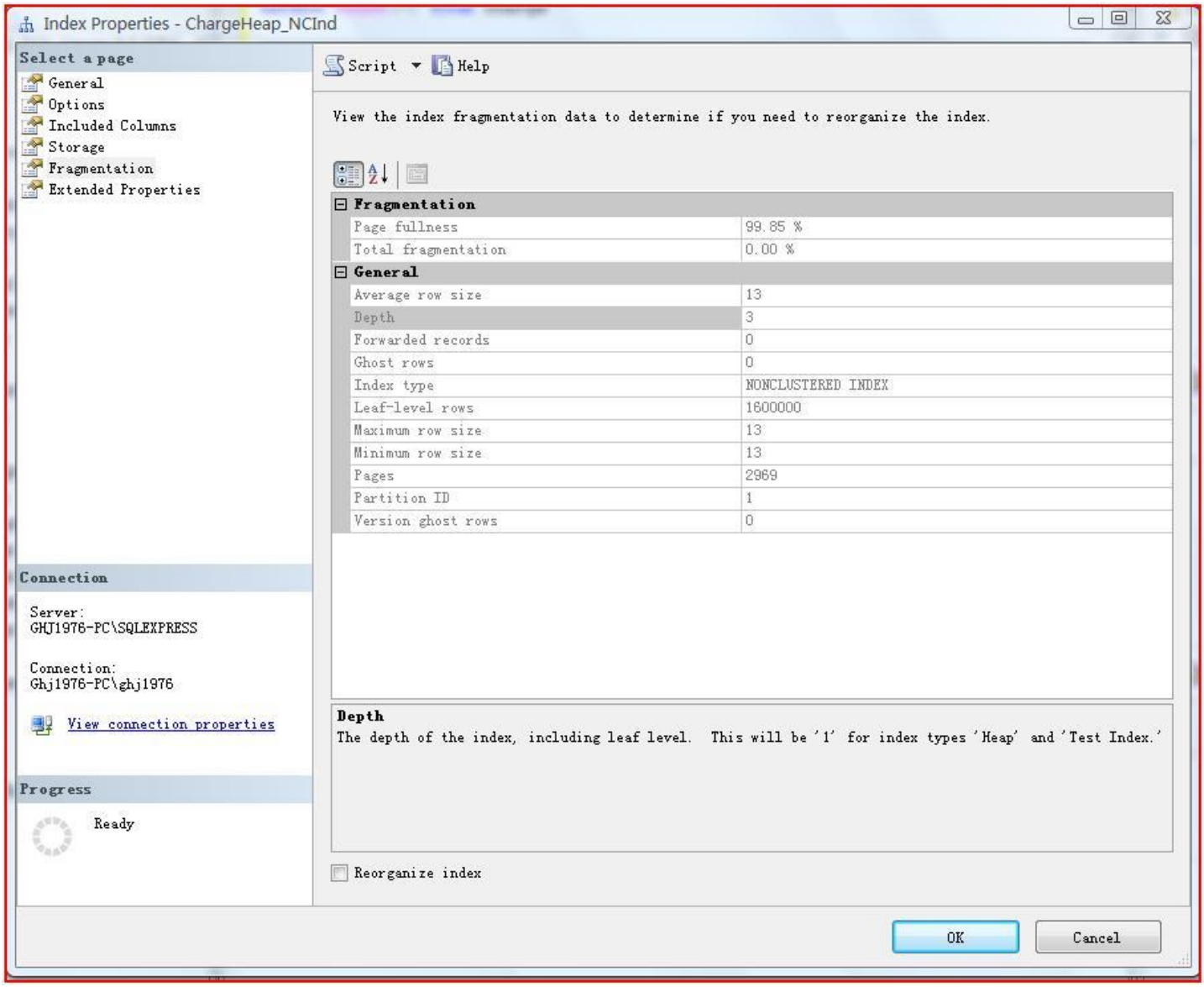
```
select INDEXPROPERTY (OBJECT_ID('ChargeHeap'),'ChargeHeap_NCInd','IndexDepth')
```

其中的 'ChargeHeap' 为我们要查看索引所在的表名， 'ChargeHeap_NCInd'为所要查看的索引名， 'IndexDepth' 为所要查看的索引属性。

更多属性请参看下面页面的参数说明：
http://technet.microsoft.com/zh-cn/library/ms187729.aspx

或者我们在 SQL Server Management Studio 中选中我们要查看的索引，然后在右键菜单中查看索引的属性。其中 Fragmentation 标签页会有很多我们对
这个索引感兴趣的内容，比如下图：

我们可以在这里看到索引的深度，子节点数，数据页数等等信息。这些信息对我们分析查询语句的性能非常有帮助。



3.2 如何查看磁盘 I/O 操作信息

SET STATISTICS IO ON 命令是一个 使 SQL Server 显示有关由 Transact-SQL 语句生成的磁盘活动量的信息。

我们在分析索引性能的时候，会非常有用。

启用了这个属性后，我们在执行 SQL 语句后，会收到类似如下的信息，这有利于我们分析 SQL 的性能：

(3999 row(s) affected)
表 'ChargeCL'。扫描计数 1，逻辑读取 9547 次，物理读取 0 次，预读 0 次，lob 逻辑读取 0 次，lob 物理读取 0 次，lob 预读 0 次。

其中的 lob 逻辑读取、lob 物理读取、lob 预读 这三个指标是 读取 text、ntext、image 或大值类型 (varchar(max)、nvarchar(max)、varbinary(max)) 时的指标。
而 逻辑读取、物理读取、预读 是对普通数据页的读取。

3.3 使用 SQL Server Management Studio Standard Reports

我们在 SQL Server Management Studio 中，选择数据库服务器，或者具体数据库，或者 Security -- Logins 时，或者 Management 时，Notification Services 或者 SQL Server Agent 对象时候，都会看到 SQL Server 替我们提供的一些现成报表，这些报表的数据，有利于我们分析数据库的状态。

比如在 SQL Server 索引基础知识(1)--- 记录数据的基本格式中，我们就使用数据表占用空间的报表

具体报表可以参考以下链接：

SQL Server Management Studio Standard Reports - Overview
http://blogs.msdn.com/buckwoody/archive/2007/10/09/sql-server-management-studio-standard-reports-overview.aspx

3.4 测试中，释放缓存的一些方法

尤其查询语句性能测试时，数据是否被缓存，这是测试中一个重要点。下面几个命令帮助我们清除缓存。方便测试。

清除缓存有关的命令：
SQL 2000 里面除了 dbcc unpintable 好像就没有了 而且这个操作也不会立即释放表内存 Buffer
(DBCC UNPINTABLE does not cause the table to be immediately flushed from the data cache. It specifies that all of the pages for the table in the buffer cache can be flushed if space is needed to read in a new page from disk.)

SQL 2005/2008 让 DBA 能够更自由的对 SQL 所占用的内存空间做处理 如：
CHECKPOINT
将当前数据库的全部脏页写入磁盘。“脏页”是已输入缓存区高速缓存且已修改但尚未写入磁盘的数据页。 CHECKPOINT 可创建一个检查点，在该点保证全部脏页都已写入磁盘，从而在以后的恢复过程中节省时间。

DBCC DROPCLEANBUFFERS
从缓冲池中删除所有清除缓冲区。

DBCC FREEPROCCACHE
从过程缓存中删除所有元素。

DBCC FREESYSTEMCACHE
从所有缓存中释放所有未使用的缓存条目。 SQL Server 2005 数据库引擎会事先在后台清理未使用的缓存条目，以使内存可用于当前条目。但是，可以使用此命令从所有缓存中手动删除未使用的条目。

另外还可以 sp_cursor_list 查看全部游标
DBCC OPENTRAN 查看数据库打开事务状态等

4 主键与聚集索引

有些人可能对主键和聚集索引有所混淆，其实这两个是不同的概念，下面是一个简单的描述。不想看绕口文字者，直接看两者的对比表。尤其是最后一项的比较。

4.1 主键（ PRIMARY KEY ）

来自 MSDN 的描述：

表通常具有包含唯一标识表中每一行的值的一列或一组列。这样的一列或多列称为表的主键 (PK)，用于强制表的实体完整性。在创建或修改表时，您可以通过定义 PRIMARY KEY 约束来创建主键。

一个表只能有一个 PRIMARY KEY 约束，并且 PRIMARY KEY 约束中的列不能接受空值。由于 PRIMARY KEY 约束可保证数据的唯一性，因此经常对标识列定义这种约束。

如果为表指定了 PRIMARY KEY 约束，则 SQL Server 2005 数据库引擎 将通过为主键列创建唯一索引来强制数据的唯一性。当在查询中使用主键时，此索引还可用来对数据进行快速访问。因此，所选的主键必须遵守创建唯一索引的规则。

创建主键时，数据库引擎 会自动创建唯一的索引来强制实施 PRIMARY KEY 约束的唯一性要求。如果表中不存在聚集索引或未显式指定非聚集索引，则将创建唯一的聚集索引以强制实施 PRIMARY KEY 约束。

4.2 聚集索引

聚集索引基于数据行的键值在表内排序和存储这些数据行。每个表只能有一个聚集索引，因为数据行本身只能按一个顺序存储。

每个表几乎都对列定义聚集索引来实现下列功能：

- 可用于经常使用的查询。
- 提供高度唯一性。

4.3 两者的比较

下面是一个简单的比较表

	主键	**聚集索引**
用途	强制表的实体完整性	对数据行的排序，方便查询用
一个表多少个	一个表最多一个主键	一个表最多一个聚集索引
是否允许多个字段来定义	一个主键可以多个字段来定义	一个索引可以多个字段来定义
是否允许	如果要创建的数据列中数据存在 null，无法建立主键。	没有限制建立聚集索引的列一定必须 not null .

null 数据行出现	创建表时指定的 PRIMARY KEY 约束列隐式转换为 NOT NULL。	也就是可以列的数据是 null 参看最后一项比较
是否要求数据必须唯一	要求数据必须唯一	数据即可以唯一，也可以不唯一。看你定义这个索引的 UNIQUE 设置。 （这一点需要看后面的一个比较，虽然你的数据列可能不唯一，但是系统会替你产生一个你看不到的唯一列）
创建的逻辑	数据库在创建主键同时，会自动建立一个唯一索引。 如果这个表之前没有聚集索引，同时建立主键时候没有强制指定使用非聚集索引，则建立主键时候，同时建立一个唯一的聚集索引	如果未使用 UNIQUE 属性创建聚集索引，数据库引擎将向表自动添加一个四字节 uniqueifier 列。 必要时，数据库引擎 将向行自动添加一个 uniqueifier 值，使每个键唯一。此列和列值供内部使用，用户不能查看或访问。

5 理解 newid()和 newsequentialid()

在 SQL Server 2005 中新增了一个函数:newsequentialid()，MSDN 中对这个函数的描述如下：

在指定计算机上创建大于先前通过该函数生成的任何 GUID 的 GUID。
NEWSEQUENTIALID() 不能在查询中引用。
NEWSEQUENTIALID() 只能与 uniqueidentifier 类型表列上的 DEFAULT 约束一起使用。

这个函数的具体用法在下面这篇博客中已经有详细的描述了。

使用 NEWSEQUENTIALID 解决 GUID 聚集索引问题
http://www.cnblogs.com/Mirricle/archive/2007/08/15/856726.html

简单来说， newsequentialid 函数比起 newid 函数最大的好处是：

如果你在一个 UNIQUEIDENTIFIER 字段上建立索引，使用 newid 产生的新的值是不固定的，所以新的值导致索引 B+树的变化是随机的。
而 newsequentialid 产生的新的值是有规律的，则索引 B+树的变化是有规律的。有规律和无规律就会带来性能的改进。

上面是一个粗略的描述，下面是比较详细点的解释：

（我们这里解释的更详细一些，是为了让大家对索引的基础知识了解得更深入些。）

B+ 树不考虑层级变化,增加数据的情况分以下几种情况:

5.1 The insert algorithm for B+ Trees

Leaf Page Full	Index Page FULL	Action
NO	NO	Place the record in sorted position in the appropriate leaf page
YES	NO	1. Split the leaf page 2. Place Middle Key in the index page in sorted order. 3. Left leaf page contains records with keys below the middle key. 4. Right leaf page contains records with keys equal to or greater than the middle key.
YES	YES	1. Split the leaf page. 2. Records with keys < middle key go to the left leaf page. 3. Records with keys >= middle key go to the right leaf page. 4. Split the index page. 5. Keys < middle key go to the left index page. 6. Keys > middle key go to the right index page. 7. The middle key goes to the next (higher level) index. IF the next level index page is full, continue splitting the index pages.

更多 B+ 树的算法请参看后面链接: http://www.sci.unich.it/~acciaro/bpiutrees.pdf

对于数据库的索引来说，上面情况中 ,第三种情况发生的概率很低，更多的是 1,2 这两种情况。

数据库中增加记录时，对索引的 B+树的操作，其实就是对 左右叶子节点，上级节点的操作。

而找到这几个节点后的操作，在实际上，都不是性能消耗最大的地方。性能消耗最大的地方在于搜索找到需要操作的叶子节点。

对于 B+ 树来说， 几层的 B+ 树，找到叶子节点就需要找几个数据页。那为何说 Guid 有规律时速度要比无规律时候快呢？

原因很简单：

1、缓存的命中率问题
(你可以参看我之前写的这篇博客：理解缓存 http://blog.joycode.com/ghj/archive/2007/09/01/107863.aspx)

当每次产生的 Guid 是有规律时，找到需要操作的叶子节点的几个中间节点，可能已经在之前的访问中被缓存了。

这样，系统不需要大量的读入缓存命中率很低的索引数据页，这样可以节省内存，同时提高搜索速度。

2、连续和不连续的磁盘 I/O 操作对性能的影响

我们都知道，现在很多业务逻辑的瓶颈是硬盘的速度。而硬盘速度提升的空间仍然不大。下面对硬盘读写操作的一些法则对我们优化跟硬盘 I/O 有关的方面很有帮助。

请记住下面的经验法则：标准的 Wide Ultra SCSI-3 硬盘每秒钟可为 Windows 和 SQL Server 提供 75 个不连续（随机）的 I/O 操作和 150 个连续的 I/O 操作。这种硬盘的标称传输率在 40 MB/秒左右。请记住更有可能限制数据库服务器的传输率是每秒钟 75/150 I/O，而不是 40 MB/秒。

读/写磁头和相关的磁盘取数臂需要移动才能在 SQL Server 和 Windows 所要求的硬盘盘片的位置上进行查找和操作。如果数据所在的硬盘盘片的位置不连续，硬盘驱动器要花多得多的时间才能将磁盘取数臂和读/写磁头移动到所有需要的硬盘盘片位置。如果所需要的数据全部位于硬盘盘片上的连续物理扇区，情况则相反，磁盘取数臂和读/写磁头只需进行很小的移动就能完成所需磁盘 I/O 操作。连续和不连续的情况下所花的时间有很大的差异，每个不连续的数据查找大约要花 50 毫秒，而连续的数据查找则只需大约 2-3 毫秒。请注意这些值是粗略估计出来的，具体值将取决于不连续的数据在磁盘上分布的疏密、硬盘盘片的旋转速度 (RPM) 以及硬盘的其它物理属性。主要要记住的一点是连续 I/O 有益于 SQL Server 性能。

之前已提到标准的硬盘支持每秒 75 个不连续的 I/O 和每秒 150 个连续的 I/O。还要记住的重要一点是读或写 8KB 的时间与读或写 64 KB 的时间几乎相同。在 8 KB 到 64 KB 范围之内，单个磁盘 I/O 传输操作所花的时间主要是磁盘取数臂和读/写磁头运动的时间。因此，从数学上来讲，当需要传输 64 KB 以上的 SQL 数据时，尽可能地执行 64 KB 磁盘传输是有益的，因为 64 KB 传输基本上与 8 KB 传输一样快，而每次传输的 SQL Server 数据是 8 KB 传输的 8 倍。请记住 Read-Ahead Manager 以 64 KB 字节片（也称为 SQL Server 扩展盘区）执行磁盘操作。Log Manager 也以较大的 I/O 传输量来执行连续写操作。要记住的主要事项是充分利用 Read-Ahead Manager，并将 SQL Server 日志文件与其它非连续存取的文件分开，以有效提高 SQL Server 的性能。

参考资料 ：

NEWSEQUENTIALID()
http://technet.microsoft.com/en-us/library/ms189786.aspx

重新组织和重新生成索引
http://technet.microsoft.com/zh-cn/library/ms189858.aspx

6 索引的代价，使用场景

前几天给同事培训了聚集索引，非聚集索引的知识后，在一个同事新作的项目中，竟然出现了滥用聚集索引的问题。看来没有培训最最基础的索引的意义，代价，使用场景，是一个非常大的失误。这篇博客就是从这个角度来罗列索引的基础知识。

6.1 使用索引的意义

- 索引在数据库中的作用类似于目录在书籍中的作用，用来提高查找信息的速度。
- 使用索引查找数据，无需对整表进行扫描，可以快速找到所需数据。

6.2 使用索引的代价

- 索引需要占用数据表以外的物理存储空间。

- 创建索引和维护索引要花费一定的时间。
- 当对表进行更新操作时，索引需要被重建，这样降低了数据的维护速度。

6.3 创建索引的列

- 主键
- 外键或在表联接操作中经常用到的列
- 在经常查询的字段上最好建立索引

6.4 不创建索引的列

- 很少在查询中被引用
- 包含较少的惟一值
- 定义为 text、ntext 或者 image 数据类型的列

6.5 Heaps 是 staging data 的很好选择，当它没有任何 Index 时

- Excellent for high performance data loading (parallel bulk load and parallel index creation after load)
- Excellent as a partition to a partitioned view or a partitioned table

聚集索引提高性能的方法，在前面几篇博客中分别提到过，下面只是一个简单的大纲，细节请参看前面几篇博客。

6.6 何时创建聚集索引？

Clustered Index 会提高大多数 table 的性能，尤其是当它满足以下条件时：

- 独特 , 狭窄, 静止： 最重要的条件
- 持续增长的, 最好是只向上增加。例如：

o Identity

o Date, identity

o GUID (only when using newsequentialid() function)

6.7 聚集索引唯一性（独特型的问题）

由于聚集索引的 B+树结构的叶子节点必须指向具体数据。如果你要建立聚集索引的列不唯一，并且你指定的创建的聚集索引是非唯一的聚集索引，则会有以下情况：
如果未使用 UNIQUE 属性创建聚集索引，数据库引擎 将向表自动添加一个四字节 uniqueifier 列。必要时，数据库引擎 将向行自动添加一个 uniqueifier 值，使每个键唯一。此列和列值供内部使用，用户不能查看或访问。

参看我的这篇博客：

SQL Server 索引基础知识(4)----主键与聚集索引

6.8 聚集索引持续向上增长的需求

具体来说下面两个问题要求建立聚集索引的列最好是持续向上增长的

1、缓存的命中率问题。（需要从 B+树的结构分析）
2、连续和不连续的磁盘 I/O 操作对性能的影响 。

细节参看我的这篇博客：

SQL Server 索引基础知识(5)----理解 newid()和 newsequentialid()

至于，如果你的数据已经存在重复了，而且是不应该出现的，则可以参看下面这篇 KB ：

如何删除 SQL Server 表中的重复行
http://support.microsoft.com/kb/139444/zh-cn

6.9 非聚集索引提高性能的方法

非聚集索引由于 B+树的节点不是具体数据页，有时候由于这个原因，会导致非聚集索引甚至不如表遍历来的快，参看我在下面这篇博客中给的例子 SQL Server 索引基础知识(2)----聚集索引，非聚集索引。

但是，非聚集索引有个特性，如果你要查询的内容，在非聚集索引中以及被覆盖到了，则不需要继续到聚集索引，或者 RID 中去寻找数据了，这时候就可以很大的提高性能，这就是 覆盖面(Covering) 的问题。

由于聚集索引叶子节点就是具体数据，所以 聚集索引的覆盖率是 100%，

通过提高覆盖面来提高性能的问题也就只有非聚集索引（ Nonclustered Indexes）才存在。

当查询中所有的 columns 都包括在 index 上时，我们说这个 index covers the query. Columns 的顺序在此不重要

(Select 时候的顺序不重要,但是 Index 建立的顺序可得小心了)。

在 SQL Server 2005 中，为了提高这种 Covering 带来的好处，甚至 可以通过将非键列添加到非聚集索引的叶级别来扩展非聚集索引的功能。

比如下面的脚本 ，虽然我们是对 Title, Revision 建立的非聚集索引，但是这个非聚集索引的叶子节点上还包含 FileName 字段的信息。

复制

```
USE AdventureWorks;
GO
CREATE INDEX IX_Document_Title
ON Production.Document (Title, Revision)
INCLUDE (FileName);
```

下面的代码就是测试 Covering 的.

我已经在每个查询使用的方式，逻辑读的个数都标在每个查询前面了。

```
SET STATISTICS IO ON
-- Turn Graphical Showplan ON (Ctrl+K)

USE CREDIT
go
-- 逻辑读取 144 次    Clustered Index Scan
SELECT m.LastName, m.FirstName, m.Phone_No
FROM dbo.Member AS m WITH (INDEX (0))
WHERE m.LastName LIKE '[S-Z]%'
go

--CREATE INDEX MemberLastName ON Member(LastName)
go
-- 逻辑读取 6354 次 BookMark Lookup
SELECT m.LastName, m.FirstName, m.Phone_No
FROM dbo.Member AS m WITH (INDEX (MemberLastName))
WHERE m.LastName LIKE '[S-Z]%'
go

--CREATE INDEX NCLastNameCombo ON Member(LastName, FirstName, Phone_No)
go
-- 逻辑读取 21 次   Index Seek
SELECT m.LastName, m.FirstName, m.Phone_No
FROM dbo.Member AS m
WHERE m.LastName LIKE '[S-Z]%'
go

--CREATE INDEX NCLastNameCombo2 ON Member(FirstName, LastName, Phone_No)
go
-- 逻辑读取 59 次     Index Scan
SELECT m.LastName, m.FirstName, m.Phone_No
FROM dbo.Member AS m WITH (INDEX (NCLastNameCombo2))
WHERE m.LastName LIKE '[S-Z]%'
go

-- If you want to clean up the indexes:
--DROP INDEX Member.MemberLastName
--DROP INDEX Member.NCLastNameCombo
--DROP INDEX Member.NCLastNameCombo2
```

6.10 参考资料

Teched 2007 上 吴家震 主讲的"微软 SQL 服务器 Always-On Tech-nologies: 高级索引策略" 录像下载地址:
http://msevents.microsoft.com/CUI/EventDetail.aspx?EventID=1032364059&Culture=zh-CN
注意, 这个页面标示的是 "SharePoint 2007 网站性能调优",但是其实是高级索引策略,微软弄错文件了,害得我一个个下下来看, 哪个是需要的录像.

7 Indexing for AND

我们通过一个实例来看 有 And 操作符时候的最常见的一种情况。我们有下面一个表，

复制

```
CREATE TABLE [dbo].[member](
        [member_no] [dbo].[numeric_id] IDENTITY(1,1) NOT NULL,
        [lastname] [dbo].[shortstring] NOT NULL,
        [firstname] [dbo].[shortstring] NOT NULL,
        [middleinitial] [dbo].[letter] NULL,
        [street] [dbo].[shortstring] NOT NULL,
        [city] [dbo].[shortstring] NOT NULL,
        [state_prov] [dbo].[statecode] NOT NULL,
        [country] [dbo].[countrycode] NOT NULL,
        [mail_code] [dbo].[mailcode] NOT NULL,
        [phone_no] [dbo].[phonenumber] NULL,
        [photograph] [image] NULL,
        [issue_dt] [datetime] NOT NULL   DEFAULT (getdate()),
        [expr_dt] [datetime] NOT NULL  DEFAULT (dateadd(year,1,getdate())),
        [region_no] [dbo].[numeric_id] NOT NULL,
        [corp_no] [dbo].[numeric_id] NULL,
        [prev_balance] [money] NULL   DEFAULT (0),
        [curr_balance] [money] NULL  DEFAULT (0),
        [member_code] [dbo].[status_code] NOT NULL   DEFAULT ('  ')
)
```

这个表具备下面的四个索引：

索引名	细节	索引的列
member_corporation_link	nonclustered located on PRIMARY	corp_no
member_ident	clustered, unique, primary key located on PRIMARY	member_no
member_region_link	nonclustered located on PRIMARY	region_no
MemberFirstName	nonclustered located on PRIMARY	firstname

当我们执行下面的 SQL 查询时候，

复制

```
SELECT m.Member_No, m.FirstName, m.Region_No
FROM dbo.Member AS m
WHERE m.FirstName LIKE 'K%'
```

```
        AND m.Region_No > 6
        AND m.Member_No < 5000
go
```

SQL Server 会根据索引方式，优化成下面方式来执行。

```
select a.Member_No,a.FirstName,b.Region_No
from
(select m.Member_No, m.FirstName from dbo.Member AS m
where m.FirstName LIKE 'K%' and m.Member_No < 5000) a ,
-- 这个查询可以直接使用 MemberFirstName 非聚集索引，而且这个非聚集索引覆盖了所有查询列
-- 实际执行时，只需要 逻辑读取 3 次

(SELECT m.Member_No, m.Region_No from dbo.Member AS m
where m.Region_No > 6) b

-- 这个查询可以直接使用 member_region_link 非聚集索引，而且这个非聚集索引覆盖了所有查询列
-- 实际执行时，只需要 逻辑读取 10 次

where a.Member_No = b.Member_No
```

不信，你可以看这两个 SQL 的执行计划，以及逻辑读信息，都是一样的。

其实上面的 SQL，如果优化成下面的方式，实际的逻辑读消耗也是一样的。为何 SQL Server 不会优化成下面的方式。是因为 and 操作符优化的另外一个原则。

1/26 的数据和 1/6 的数据找交集的速度要比 1/52 的数据和 1/3 的数据找交集速度要慢。

复制

```
select a.Member_No,a.FirstName,b.Region_No
from
(select m.Member_No, m.FirstName from dbo.Member AS m
where m.FirstName LIKE 'K%'
-- 1/26 数据
) a,

(SELECT m.Member_No, m.Region_No from dbo.Member AS m
where m.Region_No > 6 and m.Member_No < 5000
-- 1/3 * 1/ 2 数据
) b
where a.Member_No = b.Member_No
```

当然，我们要学习 SQL 如何优化的话，就会用到查询语句中的一个功能，指定查询使用哪个索引来进行。

比如下面的查询语句：

复制

```
SELECT m.Member_No, m.FirstName, m.Region_No
FROM dbo.Member AS m WITH (INDEX (0))
WHERE m.FirstName LIKE 'K%'
```

```
        AND m.Region_No > 6
        AND m.Member_No < 5000
go

SELECT m.Member_No, m.FirstName, m.Region_No
FROM dbo.Member AS m WITH (INDEX (1))
WHERE m.FirstName LIKE 'K%'
        AND m.Region_No > 6
        AND m.Member_No < 5000
go
SELECT m.Member_No, m.FirstName, m.Region_No
FROM dbo.Member AS m WITH (INDEX (MemberCovering3))
WHERE m.FirstName LIKE 'K%'
        AND m.Region_No > 6
        AND m.Member_No < 5000
go
SELECT m.Member_No, m.FirstName, m.Region_No
FROM dbo.Member AS m WITH (INDEX (MemberFirstName, member_region_link))
WHERE m.FirstName LIKE 'K%'
        AND m.Region_No > 6
        AND m.Member_No < 5000
go
```

这里 Index 计算符可以是 0 ，1， 指定的一个或者多个索引名字。对于 0 ，1 的意义如下：

如果存在聚集索引，则 INDEX(0) 强制执行聚集索引扫描，INDEX(1) 强制执行聚集索引扫描或查找（使用性能最高的一种）。如果不存在聚集索引，则 INDEX(0) 强制执行表扫描，INDEX(1) 被解释为错误。

7.1 总结知识点：

- 简单来说，我们可以这么理解： SQL Server 对于每一条查询语句。会根据实际索引情况（sysindexes 系统表中存储这些信息），分析每种组合可能的成本。然后选择它认为成本最小的一种。作为它实际执行的计划。
- 成本代价计算的一个主要组成部分是逻辑 I/O 的数量，特别是对于单表的查询。
- AND 操作要满足所有条件，这样，经常会要求对几个数据集作交集。数据集越小，数据集的交集计算越节省成本。

8 数据基本格式补充

我在 SQL Server 索引基础知识系列中，第一篇就讲了记录数据的基本格式。那里主要讲解的是，数据库的最小读存单元：数据页。一个数据页是 8K 大小。

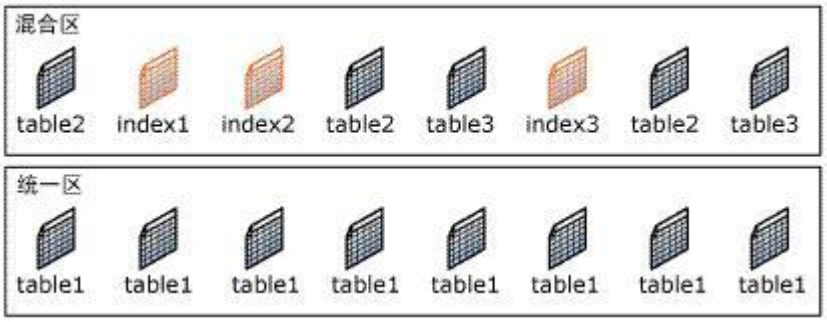
对于数据库来说，它不会每次有一个数据页变化后，就存到硬盘。而是变化达到一定数量级后才会作这个操作。 这时候，数据库并不是以数据页来作为操作单元，而是以 64k 的数据（8 个数据页，一个区）作为操作单元。

区是管理空间的基本单位。一个区是八个物理上连续的页（即 64 KB）。这意味着 SQL Server 数据库中每 MB 有 16 个区。

为了使空间分配更有效， SQL Server 不会将所有区分配给包含少量数据的表。SQL Server 有两种类型的区：

- 统一区，由单个对象所有。区中的所有 8 页只能由所属对象使用。
- 混合区，最多可由八个对象共享。区中八页的每页可由不同的对象所有。

通常从混合区向新表或索引分配页。当表或索引增长到 8 页时，将变成使用统一区进行后续分配。如果对现有表创建索引，并且该表包含的行足以在索引中生成 8 页，则对该索引的所有分配都使用统一区进行。



为何会这样呢？

其实很简单：

读或写 8KB 的时间与读或写 64 KB 的时间几乎相同。
在 8 KB 到 64 KB 范围之内，单个磁盘 I/O 传输操作所花的时间主要是磁盘取数臂和读/写磁头运动的时间。
因此，从数学上来讲，当需要传输 64 KB 以上的 SQL 数据时，
尽可能地执行 64 KB 磁盘传输是有益的，即分成数个 64K 的操作。
因为 64 KB 传输基本上与 8 KB 传输一样快，而每次传输的 SQL Server 数据是 8 KB 传输的 8 倍。

参看：

磁盘 I/O 性能
http://windows.chinaitlab.com/skill/9872.html

参考资料：

MSDN 中关于“页和区”的描述

http://technet.microsoft.com/zh-cn/library/ms190969.aspx

9 Indexing for OR

我们仍然是通过例子来理解 OR 运算符的特征

我们仍然使用 SQL Server 索引基础知识(7)----Indexing for AND 中的 member 表,这时候,这个表的索引如下:

名字	描述	列

member_corporation_link	nonclustered located on PRIMARY	corp_no
member_ident	clustered, unique, primary key located on PRIMARY	member_no
member_region_link	nonclustered located on PRIMARY	region_no
MemberFirstName	nonclustered located on PRIMARY	firstname
MemberLastName	nonclustered located on PRIMARY	lastname

我们执行下面的查询

复制

```
SELECT m.LastName, m.FirstName, m.Region_No
FROM dbo.Member AS m
WHERE m.FirstName = 'Kimberly'
        OR m.LastName = 'Tripp'
go
```

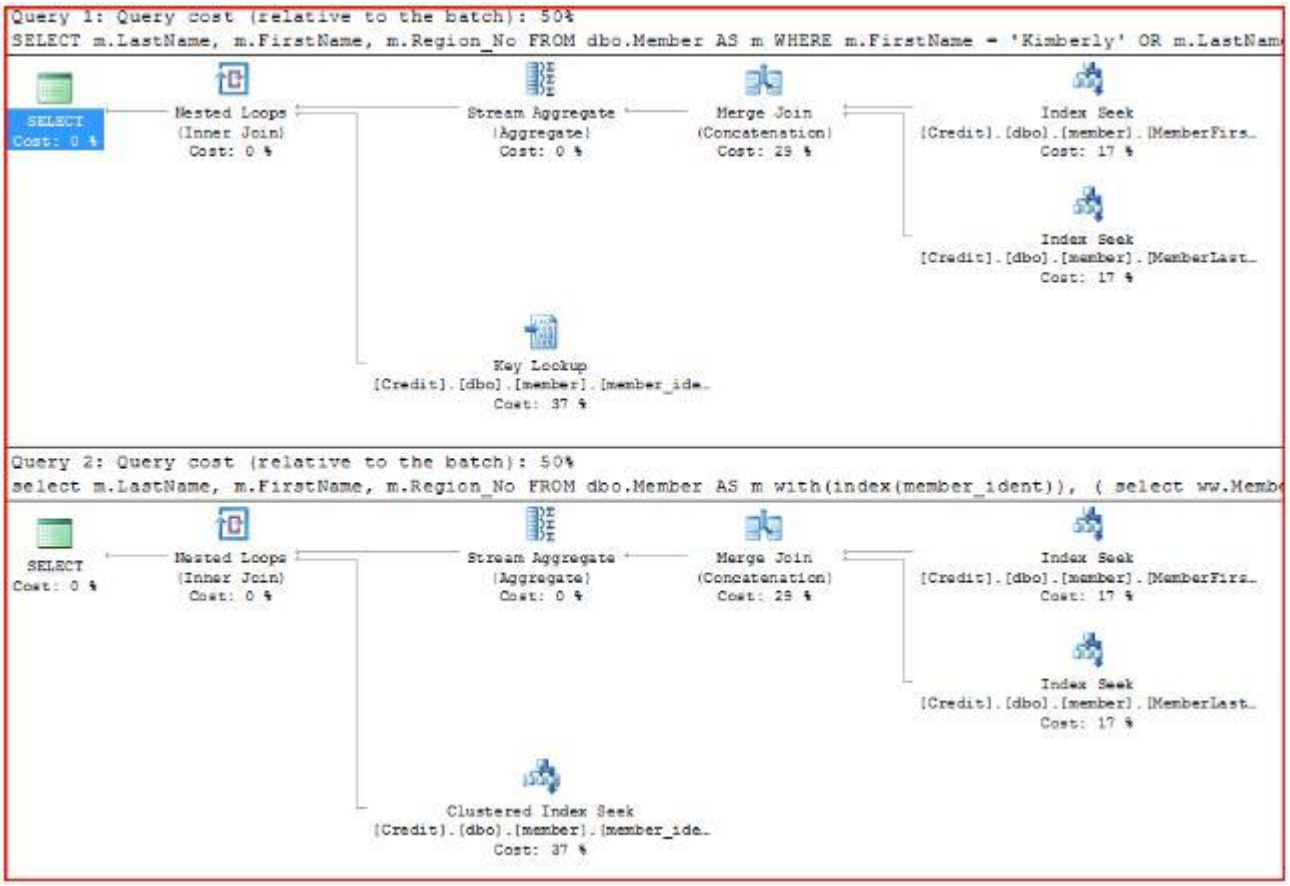
我们用另外一个 SQL 来模拟 SQL 的执行计划，就是下面的语句：
我们会看到上面语句跟下面的语句基本一样，只是一个是 Key Lookup ，一个是 Clustered Index Seek。这里由于是模拟情况，可以认为是一样的。

复制

```
select m.LastName, m.FirstName, m.Region_No
FROM dbo.Member AS m with(index(member_ident)),
(
select ww.Member_No from (
select Member_No from dbo.Member where FirstName = 'Kimberly'
union all
select Member_No from dbo.Member where LastName = 'Tripp'
) ww
group by ww.Member_No
)
n
where m.Member_No = n.Member_No
go
```

这两个查询的扫描计数均是 2，逻辑读取均是 10 次。

他们俩个的执行计划如下图，点击看大图：



知识点小结：

OR 会做什么?

- 将多个结果集集合起来，上图中，使用 Merge Join（Concatenation）把数据汇集起来。
- 保证每一个 row 只出现一回，上图中，使用 Stream Aggregate(Aggregate)进行排重。

上面的例子中，我们用 union 来演示 or 的情况。 OR 和 UNION 相似。不同之处如下：

- OR 根据 row's unique identifier (RID or Clustering Key) 去掉副本
- UNION 根据 SELECT list 去掉副本
- UNION ALL 不去除副本

OR 的一个简单应用就是 In 关键字。

- 如果有 In 搜索关键字的对应索引。则系统会使用这个索引。
- 如果没有，则遍历（表遍历或者索引遍历）是高性能的选择。

10 Join 时的三种算法简介

我们书写查询语句的时候， Join 参数之前可以是下面三个 { LOOP | MERGE | HASH } JOIN 。 如果不使用，则系统自己分析那种方式快，使用那种方式。

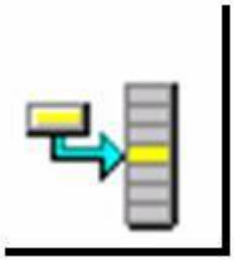
这其实是 SQL Server 联结时候使用的三种算法。尽管每种算法都并不是很复杂，但考虑到性能优化，在产品级的优化器实现时往往使用的是改进过的变种算法。譬如 SQL Server 支持 block nested loops、index nexted loops、sort-merge、hash join 以及 hash team。我们在这里只对上述三种基本算法的原型做一个简单的介绍。

知识点：

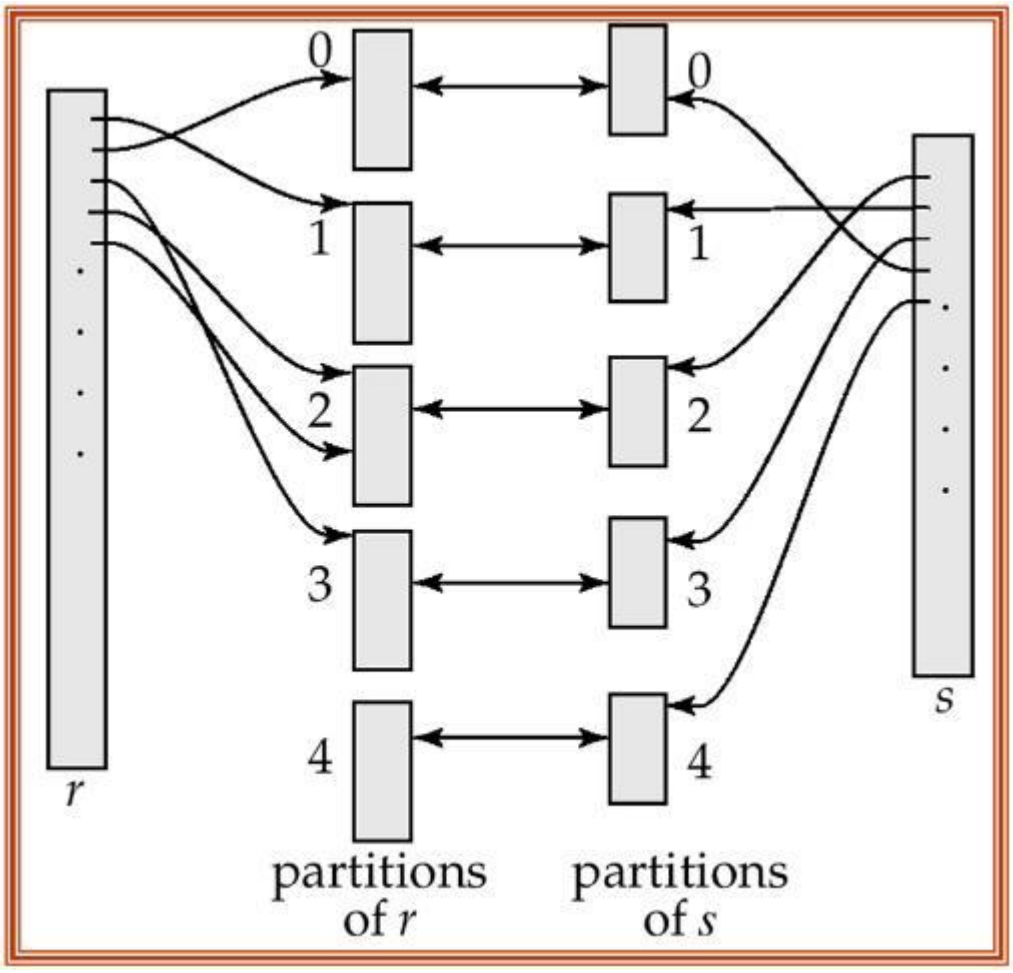
Tables join 总是两个两个进行的。所以下面的算法都是两个表的联结。

10.1 Hash Join （哈希联结）

- 下图是 SQL Server 标示这种联结的图标，从图标我们就可以看到这个查询的逻辑。



- 逻辑步骤，如下图：
  - 以数据少的数据表的 Join 字段建立 Hash 值。
  - 对应的数据表计算 Join 字段的 Hash ，再与前一个数据表做比对。



- 特点：
  - 处理大量、未排序、无索引的数据

- 一般来说，查询优化器会首先考虑 Nested Loop 和 Sort-Merge，但如果两个集合量都不小且没有合适的索引时，才会考虑使用 Hash Join。
- Hash Join 也用于许多集合比较操作，inner join、left/right/full outer join、intersect、difference 等等，当然了，需要保证都是等值联结。
- Hash Join 一个较大限制是它只能应用于等值联结(equality join)，这主要是由于哈希函数及其桶的确定性及无序性所导致的。

10.2 Nested Loop Join （嵌套循环联结）

- 下图是 SQL Server 标示这种联结的图标，从图标我们就可以看到这个查询的逻辑。

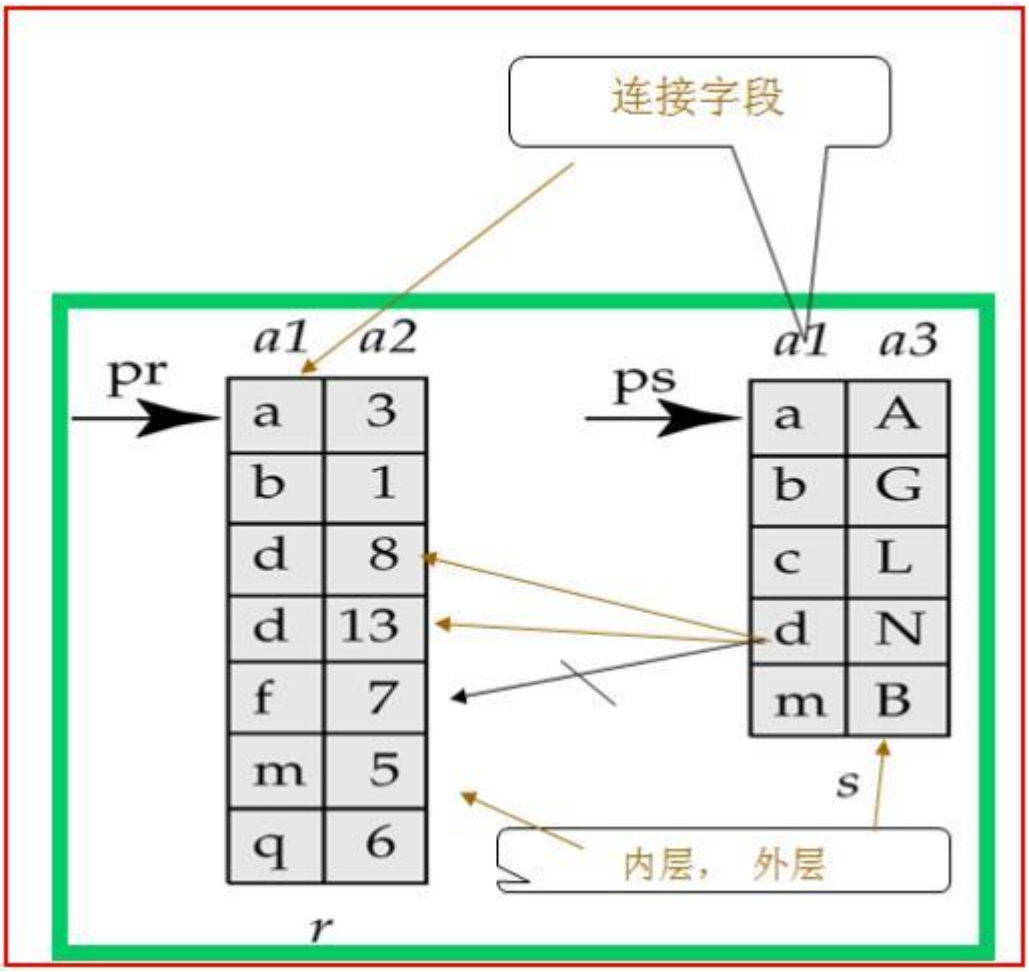
- 逻辑步骤，如下图：
  - 从外层的数据表取出一笔记录
  - 使用这个记录扫描内层的数据表
  - 再回到外层的数据表，重复上述的步骤。

- 特点：
  - 适用于一个集合大而另一个集合小的情况 (将小集合做为外循环)，I/O 性能不错。
  - 当外循环输入相当小而内循环非常大且有索引建立在 JOIN 字段上时，I/O 性能相当不错。
  - 当两个集合中只有一个在 JOIN 字段上建立索引时，一定要将该集合作为内循环。
  - 对于一对一的匹配关系 (两个具有唯一约束字段的联结)，可以在找到匹配元组后跳过该次内循环的剩余部分(类似于编程语言循环语句中的 continue)。
  - 可以不是等值联结

10.3 Merge Join （合并联结）

- 下图是 SQL Server 标示这种联结的图标，从图标我们就可以看到这个查询的逻辑。

- 逻辑步骤。如下图：
  - 使用两个数据表用来 Join 的字段既有的索引
  - 两边的数据表以光标由小到大比较，一边移动到比另一个数据表大时，换移另一个数据表。



- 特点：
  - 可以不是等值联结
  - MERGE JOIN 必须等待两个单独的 SORT JOIN 完成(如果使用索引作为数据源,可以跳过 SORT JOIN 这个步骤), 如果两个表的规模相差很大, 会很影响查询的性能。
  - 当查询的两个表都很大或者都很小的时候， 则应该只用 MERGE JOIN，如果两个表都很小，则表扫描和分类将进行的很快；

10.4 分别使用这三种 Join 的例子：

复制

```
create table t1 ( i int not null )
create table t2 ( i int not null )
go
set showplan_text on
go
--   Hash Match(Inner Join, HASH:([ghj_Demo].[dbo].[t2].[i])=([ghj_Demo].[dbo].[t1].[i]))
select * from t1 join t2 on t1.i = t2.i
go

set showplan_text off
go
alter table t1 add primary key ( i )
alter table t2 add primary key ( i )
go

set showplan_text on
go
--   |--Nested Loops(Inner Join, OUTER REFERENCES:([ghj_Demo].[dbo].[t1].[i]))
select * from t1 join t2 on t1.i = t2.i
go
set showplan_text off
go

set showplan_text on
go
-- Merge Join(Inner Join, MERGE:([ghj_Demo].[dbo].[t1].[i])=([ghj_Demo].[dbo].[t2].[i]),
-- RESIDUAL:([ghj_Demo].[dbo].[t2].[i]=[ghj_Demo].[dbo].[t1].[i]))
select * from t1 join t2 on t1.i = t2.i option(merge join)
go
set showplan_text off
go


drop table t1, t2
```

参考资料：

Example of merge ,hash and nested join
http://forums.microsoft.com/MSDN/ShowPost.aspx?PostID=498748&SiteId=1

11 附录

SQL Server 索引基础知识(1)
http://msdn.microsoft.com/zh-cn/library/dd368022.aspx
SQL Server 索引基础知识(2)
http://msdn.microsoft.com/zh-cn/library/dd368025.aspx
SQL Server 索引基础知识(3)
http://msdn.microsoft.com/zh-cn/library/dd368028.aspx
SQL Server 索引基础知识(4)
http://msdn.microsoft.com/zh-cn/library/dd368030.aspx
SQL Server 索引基础知识(5)
http://msdn.microsoft.com/zh-cn/library/dd368031.aspx
SQL Server 索引基础知识(6)
http://msdn.microsoft.com/zh-cn/library/dd368032.aspx
SQL Server 索引基础知识(7)
http://msdn.microsoft.com/zh-cn/library/dd368033.aspx
SQL Server 索引基础知识(8)
http://msdn.microsoft.com/zh-cn/library/dd370675.aspx
SQL Server 索引基础知识(9)----
http://msdn.microsoft.com/zh-cn/library/dd370676.aspx
SQL Server 索引基础知识(10)----
http://msdn.microsoft.com/zh-cn/library/dd370677.aspx