对于SqlServer的优化来说，可能优化查询是很常见的事情。关于数据库的优化，本身也是一个涉及面比较的广的话题， 首先，打开【SQL Server Management Studio】，输入一个查询语句看看SqlServer是如何显示查询计划的吧。

select v.OrderID, v.CustomerID, v.CustomerName, v.OrderDate, v.SumMoney, v.Finished

from OrdersView as v

where v.OrderDate >= '2010-12-1' and v.OrderDate < '2011-12-1';

其中，OrdersView是一个视图，其定义如下：

SELECT dbo.Orders.OrderID, dbo.Orders.CustomerID, dbo.Orders.OrderDate,

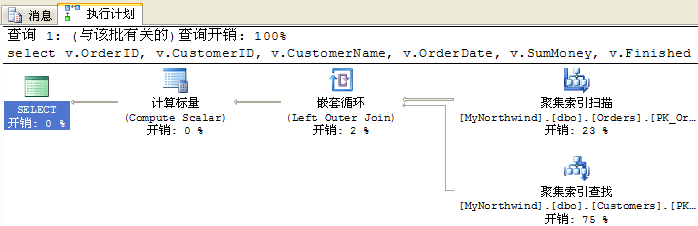
dbo.Orders.SumMoney, dbo.Orders.Finished,

ISNULL(dbo.Customers.CustomerName, N'') AS CustomerName

FROM dbo.Orders LEFT OUTER JOIN

dbo.Customers ON dbo.Orders.CustomerID = dbo.Customers.CustomerID

对于前一句查询，SqlServer给出的查询计划如下（点击工具栏上的【显示估计的执行计划】按钮）：



从这个图，我们至少可以得到3个有用的信息：  
1. 哪些执行步骤花费的成本比较高。显然，最右边的二个步骤的成本是比较高的。  
2. 哪些执行步骤产生的数据量比较多。对于每个步骤所产生的数据量， SqlServer的执行计划是用【线条粗细】来表示的，因此也很容易地从分辨出来。  
3. 每一步执行了什么样的动作。

对于一个比较慢的查询来说，我们通常首先要知道哪些步骤的成本比较高，进而，可以尝试一些改进的方法。 一般来说，如果您不能通过：提高硬件性能或者调整OS,SqlServer的设置之类的方式来解决问题，那么剩下的可选方法通常也只有以下这些了：  
1. 为【scan】这类操作增加相应字段的索引。  
2. 有时重建索引或许也是有效的，具体情形请参考后文。  
3. 调整语句结构，引导SqlServer采用其它的查询方案去执行。  
4. 调整表结构（分表或者分区）。

下面再来说说一些很重要的理论知识，这些内容对于执行计划的理解是很有帮助的。

**Sql Server 查找记录的方法**

说到这里，不得不说SqlServer的索引了。SqlServer有二种索引：聚集索引和非聚集索引。二者的差别在于：【聚集索引】直接决定了记录的存放位置， 或者说：根据聚集索引可以直接获取到记录。【非聚集索引】保存了二个信息：1.相应索引字段的值，2.记录对应聚集索引的位置（如果表没有聚集索引则保存记录指针）。 因此，如果能通过【聚集索引】来查找记录，显然也是最快的。

Sql Server 会有以下方法来查找您需要的数据记录：  
1. 【Table Scan】：遍历整个表，查找所匹配的记录行。这个操作将会一行一行的检查，当然，效率也是最差的。  
2. 【Index Scan】：根据索引，从表中过滤出来一部分记录，再查找所匹配的记录行，显示比第一种方式的查找范围要小，因此比【Table Scan】要快。  
3. 【Index Seek】：根据索引，定位（获取）记录的存放位置，然后取得记录，因此，比起前二种方式会更快。  
4. 【Clustered Index Scan】：和【Table Scan】一样。注意：不要以为这里有个Index，就认为不一样了。 其实它的意思是说：按聚集索引来逐行扫描每一行记录，因为记录就是按聚集索引来顺序存放的。 而【Table Scan】只是说：要扫描的表没有聚集索引而已，因此这二个操作本质上也是一样的。  
5. 【Clustered Index Seek】：直接根据聚集索引获取记录，最快！

所以，当发现某个查询比较慢时，可以首先检查哪些操作的成本比较高，再看看那些操作是查找记录时， 是不是【Table Scan】或者【Clustered Index Scan】，如果确实和这二种操作类型有关，则要考虑增加索引来解决了。 不过，增加索引后，也会影响数据表的修改动作，因为修改数据表时，要更新相应字段的索引。所以索引过多，也会影响性能。 还有一种情况是不适合增加索引的：某个字段用0或1表示的状态。例如可能有绝大多数是1，那么此时加索引根本就没有意义。 这时只能考虑为0或者1这二种情况分开来保存了，分表或者分区都是不错的选择。

如果不能通过增加索引和调整表来解决，那么可以试试调整语句结构，引导SqlServer采用其它的查询方案去执行。 这种方法要求：1.对语句所要完成的功能很清楚，2.对要查询的数据表结构很清楚，3.对相关的业务背景知识很清楚。 如果能通过这种方法去解决，当然也是很好的解决方法了。不过，有时SqlServer比较智能，即使你调整语句结构，也不会影响它的执行计划。

如何比较二个同样功能的语句的性能好坏呢，我建议采用二种方法：1. 直接把二个查询语句放在【SQL Server Management Studio】，然后去看它们的【执行计划】，SqlServer会以百分比的方式告诉你二个查询的【查询开销】。 这种方法简单，通常也是可以参考的，不过，有时也会不准，具体原因请接着往下看(可能索引统计信息过旧)。  
2. 根据真实的程序调用，写相应的测试代码去调用：这种方法就麻烦一些，但是它更能代表现实调用情况， 得到的结果也是更具有参考价值的，因此也是值得的。

**Sql Server Join 方式**

在Sql Server中，我们每个join命令，都会在内部执行时，采用三种更具体的方式来运行：

1. 【Nested Loops join】，**如果一个联接输入很小，而另一个联接输入很大而且已在其联接列上创建了索引， 则索引 Nested Loops 连接是最快的联接操作**，因为它们需要的 I/O 和比较都最少。

嵌套循环联接也称为“嵌套迭代”，它将一个联接输入用作外部输入表（显示为图形执行计划中的顶端输入），将另一个联接输入用作内部（底端）输入表。外部循环逐行处理外部输入表。内部循环会针对每个外部行执行，在内部输入表中搜索匹配行。可以用下面的伪码来理解：

foreach(row r1 in outer table)

foreach(row r2 in inner table)

if( r1, r2 符合匹配条件 )

output(r1, r2);

最简单的情况是，搜索时扫描整个表或索引；这称为“单纯嵌套循环联接”。如果搜索时使用索引，则称为“索引嵌套循环联接”。如果将索引生成为查询计划的一部分（并在查询完成后立即将索引破坏），则称为“临时索引嵌套循环联接”。查询优化器考虑了所有这些不同情况。

如果外部输入较小而内部输入较大且预先创建了索引，则嵌套循环联接尤其有效。**在许多小事务中（如那些只影响较小的一组行的事务），索引嵌套循环联接优于合并联接和哈希联接。但在大型查询中，嵌套循环联接通常不是最佳选择。**

2. 【Merge Join】，**如果两个联接输入并不小但已在二者联接列上排序（例如，如果它们是通过扫描已排序的索引获得的），则合并联接是最快的联接操作。如果两个联接输入都很大，而且这两个输入的大小差不多，则预先排序的合并联接提供的性能与哈希联接相近。但是，如果这两个输入的大小相差很大，则哈希联接操作通常快得多。**

合并联接要求两个输入都在合并列上排序，而合并列由联接谓词的等效 (ON) 子句定义。通常，查询优化器扫描索引（如果在适当的一组列上存在索引），或在合并联接的下面放一个排序运算符。在极少数情况下，虽然可能有多个等效子句，但只用其中一些可用的等效子句获得合并列。

由于每个输入都已排序，因此 Merge Join 运算符将从每个输入获取一行并将其进行比较。例如，对于内联接操作，如果行相等则返回。如果行不相等，则废弃值较小的行并从该输入获得另一行。这一过程将重复进行，直到处理完所有的行为止。

合并联接操作可以是常规操作，也可以是多对多操作。多对多合并联接使用临时表存储行（会影响效率）。如果每个输入中有重复值，则在处理其中一个输入中的每个重复项时，另一个输入必须重绕到重复项的开始位置。 可以创建唯一索引告诉SqlServer不会有重复值。

如果存在驻留谓词，则所有满足合并谓词的行都将对该驻留谓词取值，而只返回那些满足该驻留谓词的行。

合并联接本身的速度很快，但如果需要排序操作，选择合并联接就会非常费时。然而，如果数据量很大且能够从现有 B 树索引中获得预排序的所需数据，则合并联接通常是最快的可用联接算法。

3. 【Hash Join】，哈希联接可以有效处理未排序的大型非索引输入。它们对复杂查询的中间结果很有用，因为：1. 中间结果未经索引（除非已经显式保存到磁盘上然后创建索引），而且通常不为查询计划中的下一个操作进行适当的排序。2. 查询优化器只估计中间结果的大小。由于对于复杂查询，估计可能有很大的误差，因此如果中间结果比预期的大得多，则处理中间结果的算法不仅必须有效而且必须适度弱化。

哈希联接可以减少使用非规范化。非规范化一般通过减少联接操作获得更好的性能，尽管这样做有冗余之险（如不一致的更新）。哈希联接则减少使用非规范化的需要。哈希联接使垂直分区（用单独的文件或索引代表单个表中的几组列）得以成为物理数据库设计的可行选项。

哈希联接有两种输入：生成输入和探测输入。查询优化器指派这些角色，使两个输入中较小的那个作为生成输入。

哈希联接用于多种设置匹配操作：内部联接；左外部联接、右外部联接和完全外部联接；左半联接和右半联接；交集；联合和差异。此外，哈希联接的某种变形可以进行重复删除和分组，例如 SUM(salary) GROUP BY department。这些修改对生成和探测角色只使用一个输入。

哈希联接又分为3个类型：内存中的哈希联接、Grace 哈希联接和递归哈希联接。

内存中的哈希联接：哈希联接先扫描或计算整个生成输入，然后在内存中生成哈希表。根据计算得出的哈希键的哈希值，将每行插入哈希存储桶。如果整个生成输入小于可用内存，则可以将所有行都插入哈希表中。生成阶段之后是探测阶段。一次一行地对整个探测输入进行扫描或计算，并为每个探测行计算哈希键的值，扫描相应的哈希存储桶并生成匹配项。

Grace 哈希联接：如果生成输入大于内存，哈希联接将分为几步进行。这称为“Grace 哈希联接”。每一步都分为生成阶段和探测阶段。首先，消耗整个生成和探测输入并将其分区（使用哈希键上的哈希函数）为多个文件。对哈希键使用哈希函数可以保证任意两个联接记录一定位于相同的文件对中。因此，联接两个大输入的任务简化为相同任务的多个较小的实例。然后将哈希联接应用于每对分区文件。

递归哈希联接：如果生成输入非常大，以至于标准外部合并的输入需要多个合并级别，则需要多个分区步骤和多个分区级别。如果只有某些分区较大，则只需对那些分区使用附加的分区步骤。为了使所有分区步骤尽可能快，将使用大的异步 I/O 操作以便单个线程就能使多个磁盘驱动器繁忙工作。

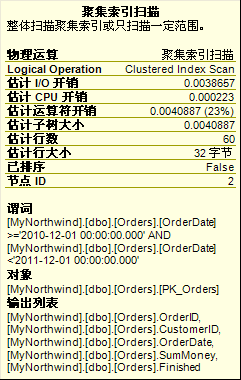
在优化过程中不能始终确定使用哪种哈希联接。因此，SQL Server 开始时使用内存中的哈希联接，然后根据生成输入的大小逐渐转换到 Grace 哈希联接和递归哈希联接。  
如果优化器错误地预计两个输入中哪个较小并由此确定哪个作为生成输入，生成角色和探测角色将动态反转。哈希联接确保使用较小的溢出文件作为生成输入。这一技术称为“角色反转”。至少一个文件溢出到磁盘后，哈希联接中才会发生角色反转。

说明：您也可以显式的指定联接方式，SqlServer会尽量尊重您的选择。比如你可以这样写：inner loop join, left outer merge join, inner hash join   
但是，我还是建议您不要这样做，因为SqlServer的选择基本上都是正确的，不信您可以试一下。

好了，说了一大堆理论东西，再来个实际的例子来解释一下吧。

**更具体执行过程**

前面，我给出一张图片，它反映了SqlServer在执行某个查询的执行计划，但它反映的信息可能不太细致，当然，您可以把鼠标指标移动某个节点上，会有以下信息出现：



刚好，我装的是中文版的，上面都是汉字，我也不多说了。我要说的是另一种方式的执行过程，比这个包含更多的执行信息， 而且是实际的执行情况。（当然，您也可以继续使用图形方式，在运行查询前点击工具栏上的【包括实际的执行计划】按钮）

让我们再次回到【SQL Server Management Studio】，输入以下语句，然后执行。

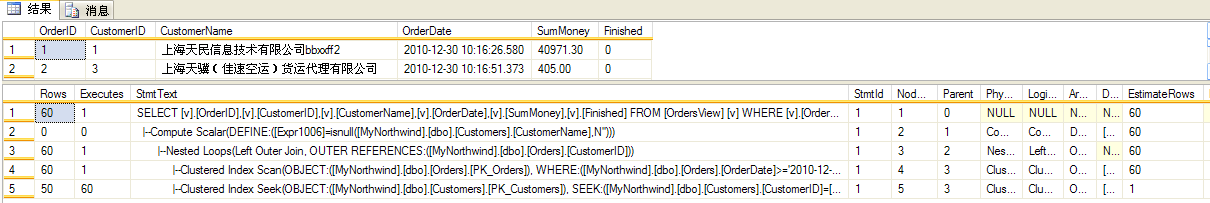
set statistics profile on

select v.OrderID, v.CustomerID, v.CustomerName, v.OrderDate, v.SumMoney, v.Finished

from OrdersView as v

where v.OrderDate >= '2010-12-1' and v.OrderDate < '2011-12-1';

注意：现在加了一句，【set statistics profile on 】，得到的结果如下：



可以从图片上看到，执行查询后，得到二个表格，上面的表格显示了查询的结果，下面的表格显示了查询的执行过程。相比本文的第一张图片， 这张图片可能在直观上不太友好，但是，它能反映更多的信息，而且尤其在比较复杂的查询时，可能看起来更容易，因为对于复杂的查询，【执行计划】的步骤太多，图形方式会造成图形过大，不容易观察。 而且这张执行过程表格能反映2个很有价值的数据（前二列）。

还是来看看这个【执行过程表格】吧。我来挑几个重要的说一下。  
【Rows】：表示在一个执行步骤中，所产生的记录条数。（真实数据，非预期）  
【Executes】：表示某个执行步骤被执行的次数。（真实数据，非预期）  
【Stmt Text】：表示要执行的步骤的描述。  
【EstimateRows】：表示要预期返回多少行数据。

在这个【执行过程表格】中，对于优化查询来说，我认为前三列是比较重要的。对于前二列，我上面也解释了，意思也很清楚。 前二列的数字也大致反映了那些步骤所花的成本，对于比较慢的查询中，应该留意它们。 【Stmt Text】会告诉你每个步骤做了什么事情。对于这种表格，它所要表达的其实是一种树型信息（一行就表示在图形方式下的一个节点）， 所以，我建议从最内层开始去读它们。做为示例，我来解释一下这张表格它所表达的执行过程。

第5行：【Clustered Index Seek(OBJECT:([MyNorthwind].[dbo].[Customers].[PK\_Customers]), SEEK:([MyNorthwind].[dbo].[Customers].[CustomerID]=[MyNorthwind].[dbo].[Orders].[CustomerID]) ORDERED FORWARD)】， 意思是说，SqlServer在对表Customers做Seek操作，而且是按照【Clustered Index Seek】的方式，对应的索引是【PK\_Customers】，seek的值来源于[Orders].[CustomerID]

第4行：【Clustered Index Scan(OBJECT:([MyNorthwind].[dbo].[Orders].[PK\_Orders]), WHERE:([MyNorthwind].[dbo].[Orders].[OrderDate]>='2010-12-01 00:00:00.000' AND [MyNorthwind].[dbo].[Orders].[OrderDate]<'2011-12-01 00:00:00.000'))】， 意思是说，SqlServer在对表Customers做Scan操作，即：最差的【表扫描】的方式，原因是，OrderDate列上没有索引，所以只能这样了。

第3行：【Nested Loops(Left Outer Join, OUTER REFERENCES:([MyNorthwind].[dbo].[Orders].[CustomerID]))】， 意思是说，SqlServer把第5行和第4行产生的数据用【Nested Loops】的方式联接起来，其中Outer表是Orders，要联接的匹配操作也在第5行中指出了。

第2行：【Compute Scalar(DEFINE:([Expr1006]=isnull([MyNorthwind].[dbo].[Customers].[CustomerName],N'')))】， 意思是说，要执行一个isnull()函数的调用。具体原因请参考本文前部分中给出视图定义代码。

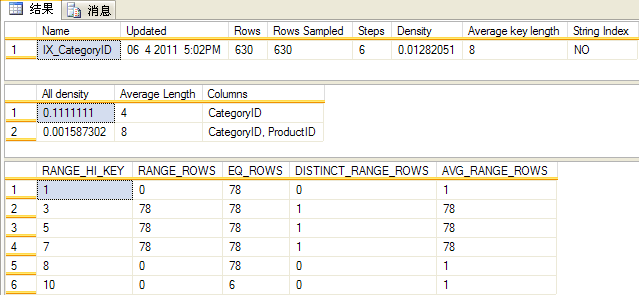
第1行：【SELECT [v].[OrderID],[v].[CustomerID],[v].[CustomerName],[v].[OrderDate],[v].[SumMoney],[v].[Finished] FROM [OrdersView] [v] WHERE [v].[OrderDate]>=@1 AND [v].[OrderDate]<@2】， 通常第1行就是整个查询，表示它的返回值。

**索引统计信息：查询计划的选择依据**

前面一直说到【执行计划】，既然是计划，就表示要在具体执行前就能确定下来的操作方案。那么SqlServer是如何选择一种执行计划的呢？SqlServer怎么知道什么时候该用索引或者用哪个索引？ 对于SqlServer来说，每当要执行一个查询时，都要首先检查有没有这个查询的执行计划是否存在缓存中，如果没有，则要生成一个执行计划， 具体在产生执行计划时，并不是看有哪些索引可用（随机选择），而是会参考一种被称为【索引统计信息】的数据。 如果您仔细地看一下前面的执行计划或者执行过程表格，会发现SqlServer能预估每个步骤所产生的数据量， 正是因为SqlServer能预估这些数据量，SqlServer才能选择一个它认为最合适的方法去执行查询过程， 此时【索引统计信息】就能告诉SqlServer这些数据。 说到这里，您是不是有点好奇呢，为了让您对【索引统计信息】有个感性的认识，我们来看看【索引统计信息】是个什么样子的。 请在【SQL Server Management Studio】，输入以下语句，然后执行。

dbcc show\_statistics (Products, IX\_CategoryID)

得到的结果如下图：



首先，还是解释一下命令：【dbcc show\_statistics】这个命令可以显示我们想知道的【索引统计信息】，它需要二个参数，1. 表名，2. 索引名

再来看看命令的结果，它有三个表格组成：  
1. 第一个表格，它列出了这个索引统计信息的主要信息。

|  |  |
| --- | --- |
| **列名** | **说明** |
| **Name** | 统计信息的名称。 |
| **Updated** | 上一次更新统计信息的日期和时间。 |
| **Rows** | 表中的行数。 |
| **Rows Sampled** | 统计信息的抽样行数。 |
| **Steps** | 数据可分成多少个组，与第三个表对应。 |
| **Density** | 第一个索引列前缀的选择性（不包括 EQ\_ROWS）。 |
| **Average key length** | 所有索引列的平均长度。 |
| **String Index** | 如果为“是”，则统计信息中包含字符串摘要索引，以支持为 LIKE 条件估算结果集大小。仅适用于 **char**、**varchar**、**nchar**和 **nvarchar**、**varchar(max)**、**nvarchar(max)**、**text** 以及 **ntext** 数据类型的前导列。 |

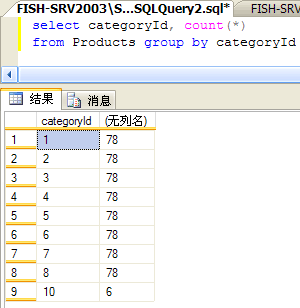
2. 第二个表格，它列出各种字段组合的选择性，数据越小表示重复越性越小，当然选择性也就越高。

|  |  |
| --- | --- |
| **列名** | **说明** |
| **All density** | 索引列前缀集的选择性（包括 EQ\_ROWS）。**注意：这个值越小就表示选择性越高。 如果这个值小于0.1，这个索引的选择性就比较高，反之，则表示选择性就不高了。** |
| **Average length** | 索引列前缀集的平均长度。 |
| **Columns** | 为其显示 **All density** 和 **Average length** 的索引列前缀的名称。 |

3. 第三个表格，数据分布的直方图，SqlServer就是靠它预估一些执行步骤的数据量。

|  |  |
| --- | --- |
| **列名** | **说明** |
| **RANGE\_HI\_KEY** | 每个组中的最大值。 |
| **RANGE\_ROWS** | 每组数据组的估算行数，不包含最大值。 |
| **EQ\_ROWS** | 每组数据组中与最大值相等的行的估算数目。 |
| **DISTINCT\_RANGE\_ROWS** | 每组数据组中的非重复值的估算数目，不包含最大值。 |
| **AVG\_RANGE\_ROWS** | 每组数据组中的重复值的平均数目，不包含最大值，计算公式：RANGE\_ROWS / DISTINCT\_RANGE\_ROWS for DISTINCT\_RANGE\_ROWS > 0 |

为了能让您更好的理解这些数据，尤其是第三组，请看下图：



当时我在填充测试数据时，故意把CategoryId为1到8的组，每组取了78条数据。所以【索引统计信息】的第三个表格的数据也都是正确的， 也正是根据这些统计信息，SqlServer才能对每个执行步骤预估相应的数据量，从而影响Join之类的选择。当然了，在选择Join方式时， 也要参考第二个表格中的字段选择性。最终在为新的查询生成执行计划时， 查询优化器使用这些统计信息并通过估计使用索引评估查询的开销来确定最佳查询计划。

再来个例子来说明一下统计信息对于查询计划选择的重要性。首先多加点数据，请看以下代码：

declare @newCategoryId int;

insert into dbo.Categories (CategoryName) values(N'Test statistics');

set @newCategoryId = scope\_identity();

declare @count int;

set @count = 0;

while( @count < 100000 )

begin

insert into Products (ProductName, CategoryID, Unit, UnitPrice, Quantity, Remark)

values( cast(newid() as nvarchar(50)), @newCategoryId, N'个', 100, @count +1, N'');

set @count = @count + 1;

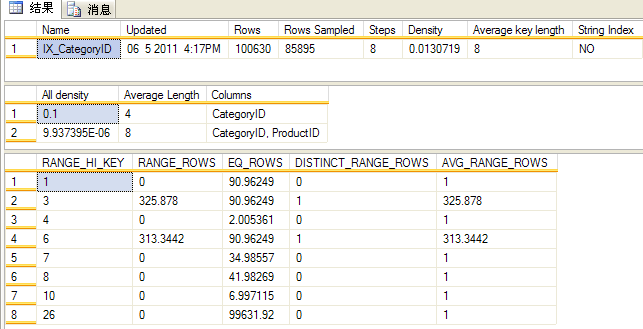
end

go

update statistics Products;

go

再来看看索引统计信息：



再来看看同一个查询，但因为查询参数值不同时，SqlServer选择的执行计划：

select p.ProductId, t.Quantity

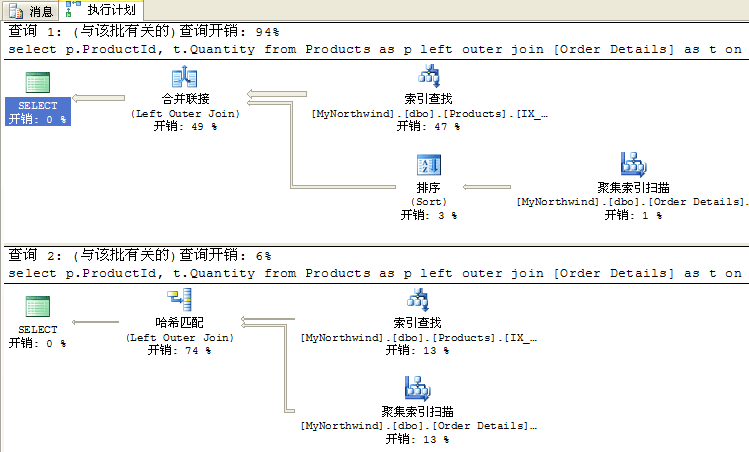
from Products as p left outer join [Order Details] as t on p.ProductId = t.ProductId

where p.CategoryId = 26; -- 26 就是最新产生的CategoryId，因此这个查询会返回10W条记录

select p.ProductId, t.Quantity

from Products as p left outer join [Order Details] as t on p.ProductId = t.ProductId

where p.CategoryId = 6; -- 这个查询会返回95条记录

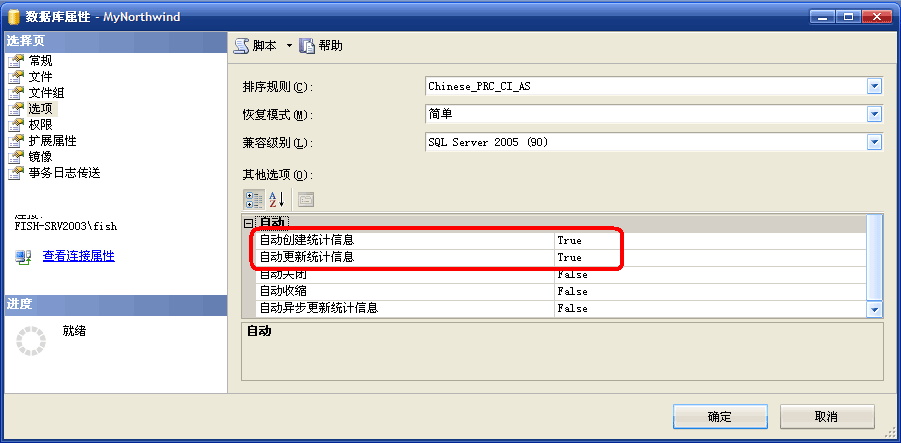


从上图可以看出，由于CategoryId的参数值不同，SqlServer会选择完全不同的执行计划。统计信息重要性在这里体现地很清楚吧。

创建统计信息后，数据库引擎对列值（根据这些值创建统计信息）进行排序， 并根据这些值（最多 200 个，按间隔分隔开）创建一个“直方图”。直方图指定有多少行精确匹配每个间隔值， 有多少行在间隔范围内，以及间隔中值的密度大小或重复值的发生率。

SQL Server 2005 引入了对 char、varchar、varchar(max)、nchar、nvarchar、nvarchar(max)、text 和 ntext 列创建的统计信息收集的其他信息。这些信息称为“字符串摘要”，可以帮助查询优化器估计字符串模式中查询谓词的选择性。 查询中有 LIKE 条件时，使用字符串摘要可以更准确地估计结果集大小，并不断优化查询计划。 这些条件包括诸如 WHERE ProductName LIKE '%Bike' 和 WHERE Name LIKE '[CS]heryl' 之类的条件。

既然【索引统计信息】这么重要，那么它会在什么时候生成或者更新呢？事实上，【索引统计信息】是不用我们手工去维护的，SqlServer会自动去维护它们。而且在SqlServer中也有个参数来控制这个更新方式：



**统计信息自动功能工作方式**

创建索引时，查询优化器自动存储有关索引列的统计信息。另外，当 AUTO\_CREATE\_STATISTICS 数据库选项设置为 ON（默认值）时， 数据库引擎自动为没有用于谓词的索引的列创建统计信息。

随着列中数据发生变化，索引和列的统计信息可能会过时，从而导致查询优化器选择的查询处理方法不是最佳的。 例如，如果创建一个包含一个索引列和 1,000 行数据的表，每一行在索引列中的值都是唯一的， 则查询优化器将把该索引列视为收集查询数据的好方法。如果更新列中的数据后存在许多重复值， 则该列不再是用于查询的理想候选列。但是，查询优化器仍然根据索引的过时分布统计信息（基于更新前的数据），将其视为好的候选列。

当 AUTO\_UPDATE\_STATISTICS 数据库选项设置为 ON（默认值）时，查询优化器会在表中的数据发生变化时自动定期更新这些统计信息。 每当查询执行计划中使用的统计信息没有通过针对当前统计信息的测试时就会启动统计信息更新。 采样是在各个数据页上随机进行的，取自表或统计信息所需列的最小非聚集索引。 从磁盘读取一个数据页后，该数据页上的所有行都被用来更新统计信息。 常规情况是：在大约有 20% 的数据行发生变化时更新统计信息。但是，查询优化器始终确保采样的行数尽量少。 对于小于 8 MB 的表，则始终进行完整扫描来收集统计信息。

采样数据（而不是分析所有数据）可以将统计信息自动更新的开销降至最低。 在某些情况下，统计采样无法获得表中数据的精确特征。可以使用 UPDATE STATISTICS 语句的 SAMPLE 子句和 FULLSCAN 子句， 控制按逐个表的方式手动更新统计信息时采样的数据量。FULLSCAN 子句指定扫描表中的所有数据来收集统计信息， 而 SAMPLE 子句用来指定采样的行数百分比或采样的行数

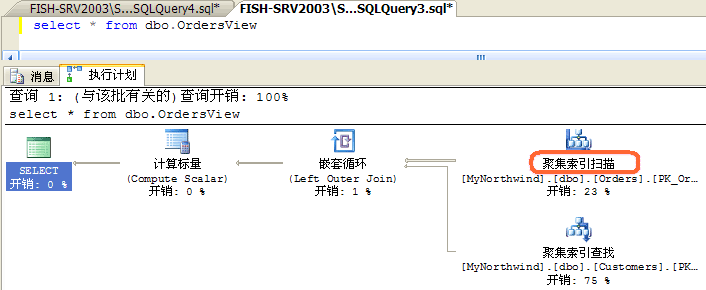
在 SQL Server 2005 中，数据库选项 AUTO\_UPDATE\_STATISTICS\_ASYNC 提供了统计信息异步更新功能。 当此选项设置为 ON 时，查询不等待统计信息更新，即可进行编译。而过期的统计信息置于队列中， 由后台进程中的工作线程来更新。查询和任何其他并发查询都通过使用现有的过期统计信息立即编译。 由于不存在等待更新后的统计信息的延迟，因此查询响应时间可预测；但是过期的统计信息可能导致查询优化器选择低效的查询计划。 在更新后的统计信息就绪后启动的查询将使用那些统计信息。这可能会导致重新编译缓存的计划（取决于较旧的统计信息版本）。 如果在同一个显式用户事务中出现某些数据定义语言 (DDL) 语句（例如，CREATE、ALTER 和 DROP 语句），则无法更新异步统计信息。

AUTO\_UPDATE\_STATISTICS\_ASYNC 选项设置于数据库级别，并确定用于数据库中所有统计信息的更新方法。 它只适用于统计信息更新，而无法用于以异步方式创建统计信息。只有将 AUTO\_UPDATE\_STATISTICS 设置为 ON 时， 将此选项设置为 ON 才有效。默认情况下，AUTO\_UPDATE\_STATISTICS\_ASYNC 选项设置为 OFF。

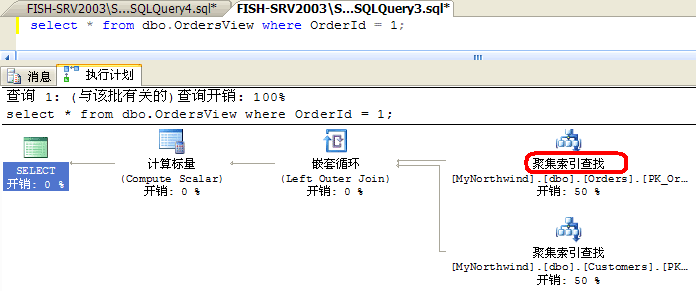
从以上说明中，我们可以看出，对于大表，还是有可能存在统计信息更新不及时的时候，这时，就可能会影响查询优化器的判断了。  
有些人可能有个经验：对于一些慢的查询，他们会想到重建索引来尝试解决。其实这样做是有道理的。 因为，在某些时候一个查询突然变慢了，可能和统计信息更新不及时有关，进而会影响查询优化器的判断。 如果此时重建索引，就可以让查询优化器知道最新的数据分布，自然就可以避开这个问题。 还记得我前面用【set statistics profile on】显示的执行过程表格吗？注意哦，那个表格就显示每个步骤的实际数据量和预估的数据量。要不要重建索引，其实我们可以用【set statistics profile on】来看一下，如果实际数据量和预估的数据量的差值比较大， 那么我们可以考虑手工去更新统计信息，然后再去试试。

**优化视图查询**

再来说说优化视图查询，虽然视图也是由一个查询语句定义的，本质上也是一个查询，但它和一般的查询语句在优化时，还是有点要区分的地方。 这里主要的区别在于，视图虽然是由一个查询语句定义的，但如果只去分析这个查询定义，可能得到的意义不大，因为视图多数时候就不是直接使用， 而是在使用前，会加上where语句，或者放在其它语句中被from所使用。下面还是举个例子吧，在我的演示数据库中有个视图OrdersView，定义代码前面有。 我们来看看，如果直接使用这个视图，会有什么样的执行计划出来：



从这个视图可以看出，SqlServer会对表Orders做全表扫描，应该是很低效的。再来看看下面这个查询：



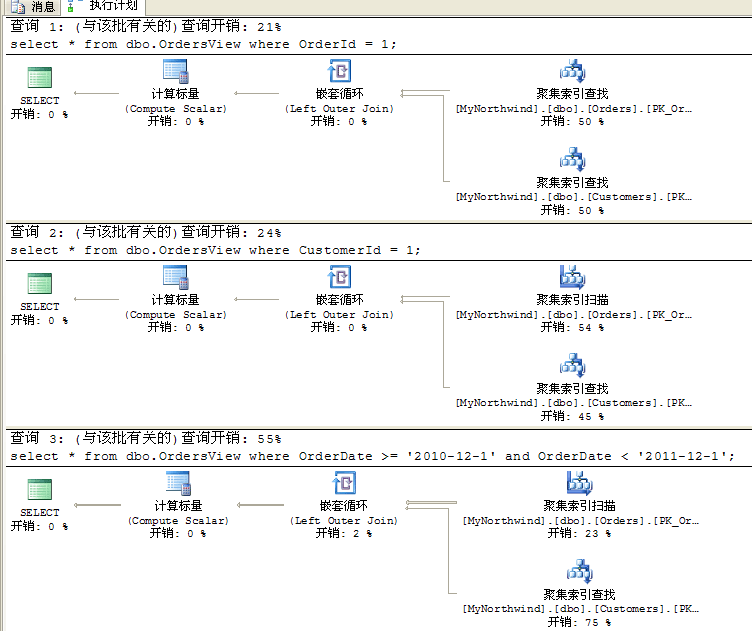
从这个执行计划可以看出，与上面那个就不一样了。前一个查询中对Orders表的查找是使用【Clustered Index Scan】的方式， 而现在在使用【Clustered Index Seek】的方式了，最右边二个步骤的成本的百分比也发生了改变。这样就足以说明，**优化视图时， 最好能根据实际要求，应用不同的过滤条件，再来决定如何去优化。**

再来一个由三个查询组成的情况来看看这个视图的执行计划。

select \* from dbo.OrdersView where OrderId = 1;

select \* from dbo.OrdersView where CustomerId = 1;

select \* from dbo.OrdersView where OrderDate >= '2010-12-1' and OrderDate < '2011-12-1';



很明显，对于同一个视图，在不同的过滤条件下，执行计划的差别很明显。