DBA Handbooks

SQL Server Statistics

Holger Schmeling

ISBN: 978-1-906434-61-8

SQL Server Statistics

by Holger Schmeling

Published 2010 by Simple-Talk Publishing
Cambridge, UK

Credits

Author: Holger Schmeling

Editor: Andrew Clarke

ISBN: 978-1-906434-61-8

Contents

Part 1: Queries, Damned Queries and Statistics

Why should we be interested in SQL Server Statistics, when we just want to get data from a database? The answer is 'performance'. The better the information that SQL Server has about the data in a database, the better its choices will be on how it executes the SQL. Statistics are its' chief source of information. If this information is out of date, performance of queries will suffer.

The SQL queries that you execute are first passed to the SQL Server Query Optimizer. If it doesn't have a plan already stored or 'cached', it will create an execution plan. When it is creating an execution plan, the SQL Server optimizer chooses an appropriate physical operator for any logical operation (such as a join, or a search) to perform. When doing so, the optimizer has some choice in the way that it maps a distinct physical operator to a logical operation. The optimizer can, for example, select either a 'physical' Index Seek or Table Scan when doing a 'logical' search, or could choose between a Hash, - Merge-, and Nested Loop joins in order to do an inner join.

There are a number of factors that influence the decision as to which physical operators the optimizer finally chooses. One of the most important ones are cardinality estimations, the process of calculating the number of qualifying rows that are likely after filtering operations are applied. A query execution plan selected with inaccurate cardinality estimates can perform several orders of magnitude slower than one selected with accurate estimates. These cardinality estimations also influence plan design options, such as join-order and parallelism. Even memory allocation, the amount of memory that a query requests, is guided by cardinality estimations.

In this article we will investigate how the optimizer obtains cardinality estimations by the use of statistics, and demonstrate why, to get accurate cardinality estimates, there must be accurate data distribution statistics for a table or index. You will experience how statistics are created and maintained, including what problems can occur, such as which limits you have to be aware of. I'll show you what statistics can do, why they are good, and also where the statistics may not work as expected.

Prerequisites

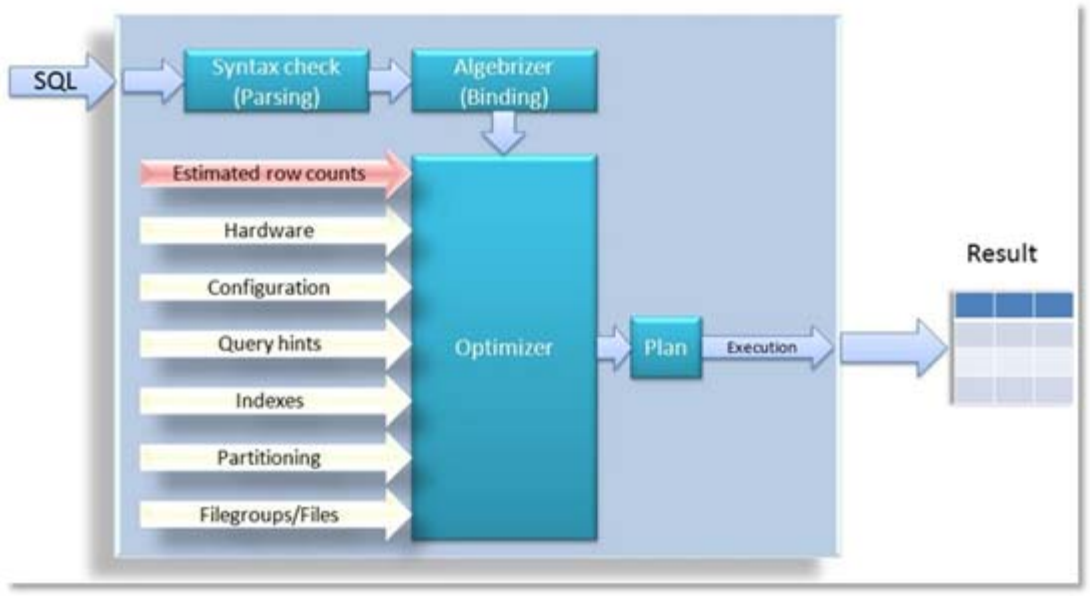
Engineer that I am, I love experimenting. Therefore, I have carefully prepared some experiments for clarifying and illustrating the underlying concepts. If you want to follow up these experiments, you will want to create a separate database with a Numbers table first. The following script will create the sample database and table:

```
if db_id('StatisticsTest') is null
  create database StatisticsTest;
go
alter database StatisticsTest set recovery simple
go
use StatisticsTest
go

-- Create a Numbers table
create table Numbers(n int not null primary key);
go
insert Numbers(n)
     select rn from (select row_number()
                            over(order by current_timestamp) as rn
                       from sys.trace_event_bindings as b1
                           ,sys.trace_event_bindings as b2) as rd
      where rn <= 5000000
```

The Numbers table is very useful as a row provider in a variety of cases. I've first started working with a table of this kind after reading [1]. If you're like me, you'll fall in love with this table very soon! Thank you, Itzik!

Statistics and execution plan generation



Picture 1: Query optimization process

A SQL query goes through the process of parsing and binding and finally a query processor tree is handed over to the optimizer. If it only had this information about the Transact-SQL to perform, however, the optimizer would not have many choices for determining an optimal execution plan. Therefore, additional parameters are considered, as you can see in Picture 1: Even some information about hardware, such as the number of CPU-cores, system configuration or the physical database layout also affects the optimization and the generated execution plan. Every single parameter that are listed as arrows in Picture 1 are relevant to the optimizing process, and therefore affect the overall performance of queries. Cardinality estimations probably have the most influence on the chosen plan.

Picture 1 is slightly inaccurate in that it shows that cardinality estimations are *provided* to the optimizer. This is not quite exact, because the optimizer itself may consider the maintenance of relevant row count estimations.

Introductory Examples

To illustrate some of these points, we will use a table with 100001 rows, which is created by executing the following script:

```
use StatisticsTest;
go

-- Create a test table
if (object_id('T0', 'U') is not null)
  drop table T0;
go

create table T0(c1 int not null, c2 nchar(200) not null default '#')
go

-- Insert 100000 rows. All rows contain the value 1000 for column c1
insert T0(c1)
   select 1000 from Numbers
    where n <= 100000
go

-- Now insert only one row with value 2000
insert T0(c1) values(2000)
go

--create a nonclustered index on column c1
create nonclustered index ix_T0_1 on T0(c1)
```

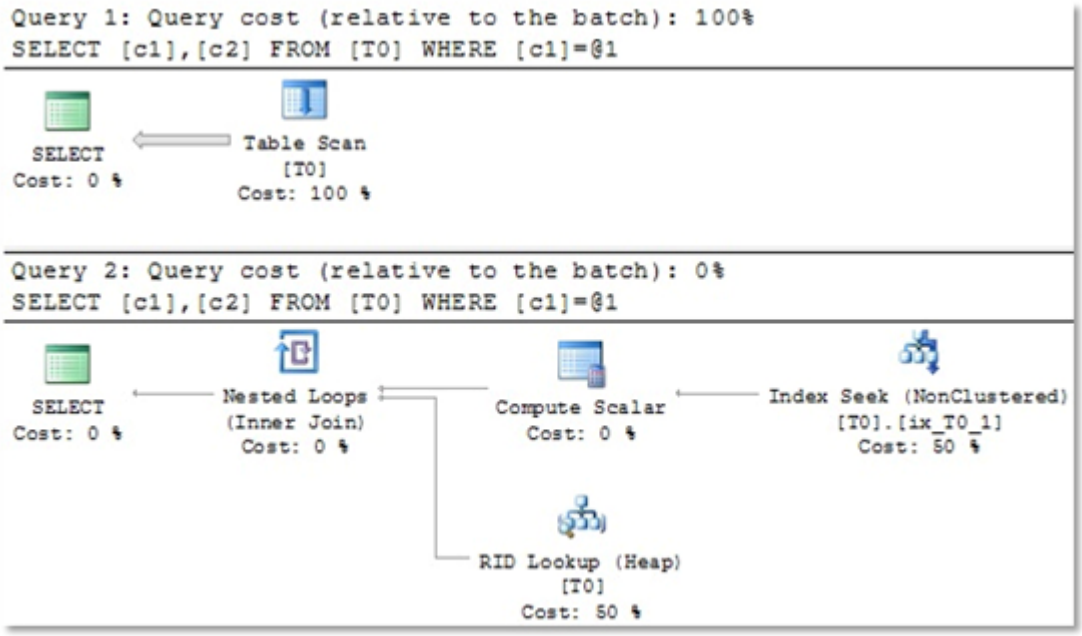
As you can see, we insert 100000 rows with a value of 1000 for column c1 into table T0, and then only one additional row with a value of 2000.

With that table at hand, let's have a look at the estimated execution plans for the following two queries:

```
select c1, c2 from T0 where c1=1000
select c1, c2 from T0 where c1=2000
```

> Note: You can show the estimated execution plan by pressing <CTRL>+L or by selecting Query/Display Estimated Execution Plan from the main menu.

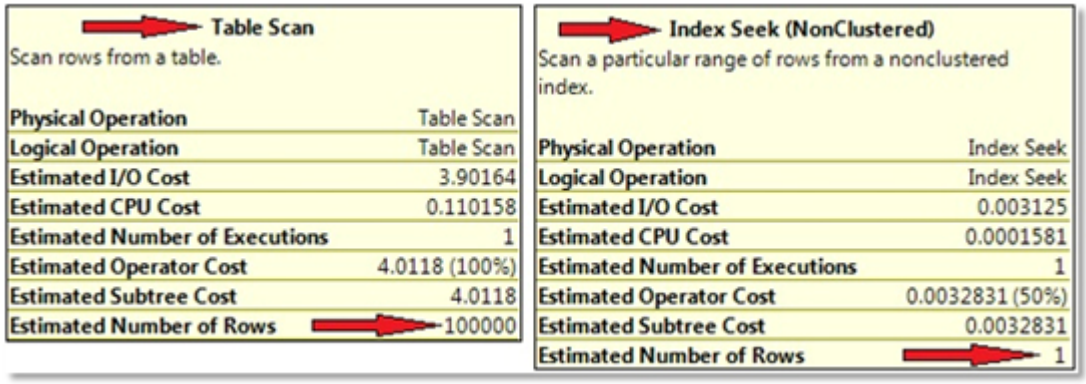
You will see two different execution plans. Have a look at picture 2 to see, what I'm talking about.



Picture 2: For the first query, which will return 100000 rows, the optimizer chooses a Table Scan. The second query yields only one single row, so in this case, an Index Seek is very efficient.

It looks as if the execution plan depends on the value used for the comparison. For each of the two used values (1000 and 2000) we get a tailor-made execution plan.

But how can this happen? Have a look at the estimated row counts for the Scan and Seek operators. You can do so by hovering with the mouse above these operators. Inside the popup window, you will find different row count estimations for each of the two queries (see Picture 3).



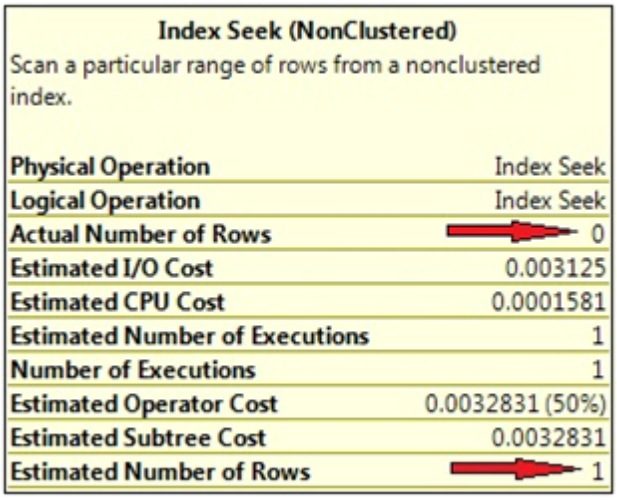
Picture 3: Estimated row counts

Clearly, the plan is adjusted to these row count estimations. But how does the optimizer know these values? If your educated guess here is “due to statistics”: yes, you’re absolutely right! And, surprisingly, the estimated number of rows and the actual number of processed rows are *exactly* the same! In our case, we can check this easily just by executing the two queries. In fact, we know about the number of processed rows already in advance of the query execution, because we’ve just inserted the rows. But you certainly can imagine that it’s not always that easy. Therefore, in many cases, you will want to include the actual execution plan as well. The actual execution plan also contains information about the actual number of rows - those rows that were really processed by every operator.

Now, let’s investigate the actual execution plan for the following query:

```
select c1, c2 from T0 where c1=1500
```

Not surprisingly, we see an Index Seek operator on the top right, just as the second plan in Picture 2 shows. The information for the Index Seek operator in the actual plan, however, is slightly extended compared with the estimated execution plan. In Picture 4 you can see that, this time, additional information about the *actual* number of rows can be obtained from the operator information. In our example the estimated, and actual, row count differs slightly. This has no adverse impact on performance, however, since an Index Seek is used for returning 0 rows, which is quite perfect...

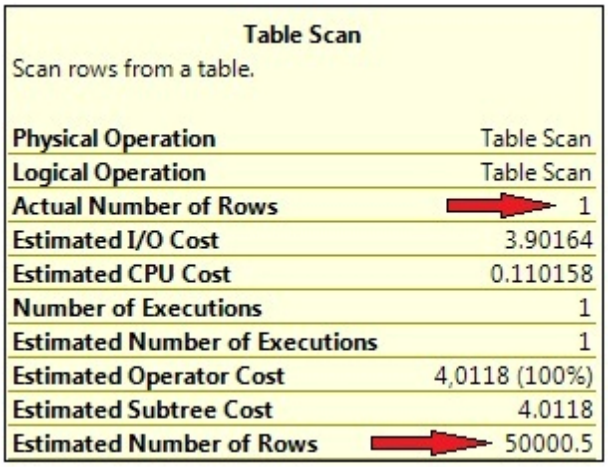


Picture 4: Estimated and actual row counts.

Now, finally let's have a look at the actual execution plan for this query:

```
declare @x int
set @x = 2000
select c1,c2 from T0
 where c1 = @x
```

As we have only one single row with a value of 2000 for c1, you will expect to see an Index Seek here. But surprisingly, a full Table Scan is performed, which is not really a good choice. I'll explain to you later, why this happens. As for now, have a look at the operator information, shown in Picture 5.



Picture 5: Large discrepancy between estimated and actual row count

You can see that the estimated row count is considerably different from the actual number of processed rows. As the plan is, of course, created before the query can be executed, the optimizer has no idea about the actual number of rows during plan generation. In order to compile the optimizer assumes that about 50000 rows are returned. From this perspective it is very clear why a Table Scan is chosen: for this large number of rows, the Table Scan is just more efficient than an Index Seek. So we end up with a sub-optimal execution plan in this case.

Under the hood

So, why do we need statistics in order to estimate cardinality? There's a simple answer to that question: statistics reduce the amount of data that has to be processed during optimization. If the optimizer had to scan the actual table or index data for cardinality estimations, plan generation would be costly and lengthy.

Therefore, the optimizer refers to samples of the actual data called the statistics, or distribution statistics. These samples are, by nature, much smaller than the original amount of data. As such, they allow for a faster plan generation.

Hence, the main reason for statistics is to speed up plan generation. But of course, statistics come at a cost. First, statistics have to be created and maintained, and this requires some resources. Second, the information held in the distribution statistics is likely to become out-of-date if the data changes, so we have to expect common redundancy problems such as change anomalies. As statistics are stored separately from the table or index they relate to, some effort has to be taken to keep them in sync with the original data. Finally, statistics reduce, or summarize, the original amount of information. We know that, due to this 'information reduction', all statistical information involves a certain amount of uncertainty or, as Aaron Levenstein once said:

"Statistics are like bikinis. What they reveal is suggestive, but what they conceal is vital."

Whenever a distribution statistic no longer reflects the source data, the optimizer may make wrong assumption about cardinalities, which in turn may lead to poor execution plans.

Statistics serve as a compromise between accuracy and speed. Statistics enable faster plan generation at the cost of their maintenance. If the maintenance is neglected, then this increases the risk of an inaccurate query plan.

General organization of statistics

The following script creates another test table which we'll use to see in which manner statistics affect query optimization:

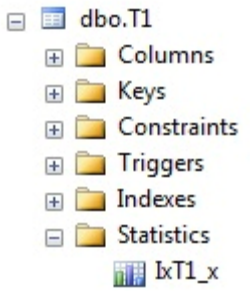
```
use StatisticsTest;
go
if (object_id('T1', 'U') is not null)
  drop table T1
go
create table T1
 (
   x int not null
  ,a varchar(20) not null
  ,b int not null
  ,y char(20) null
 )
go
insert T1(x,a,b)
   select n % 1000,n%3000,n%5000
     from Numbers
    where n <= 100000
go
create nonclustered index IxT1_x on T1(x)
```

By running the script above we will obtain a table with four columns, 100000 rows, and a non-clustered index on column x.

Obtaining information about statistics

Achieving information about statistics using Management Studio

You may open our test table T1 in Object Explorer and find an entry “Statistics” when expanding the folder for the table “dbo.T1” (see Picture 6).

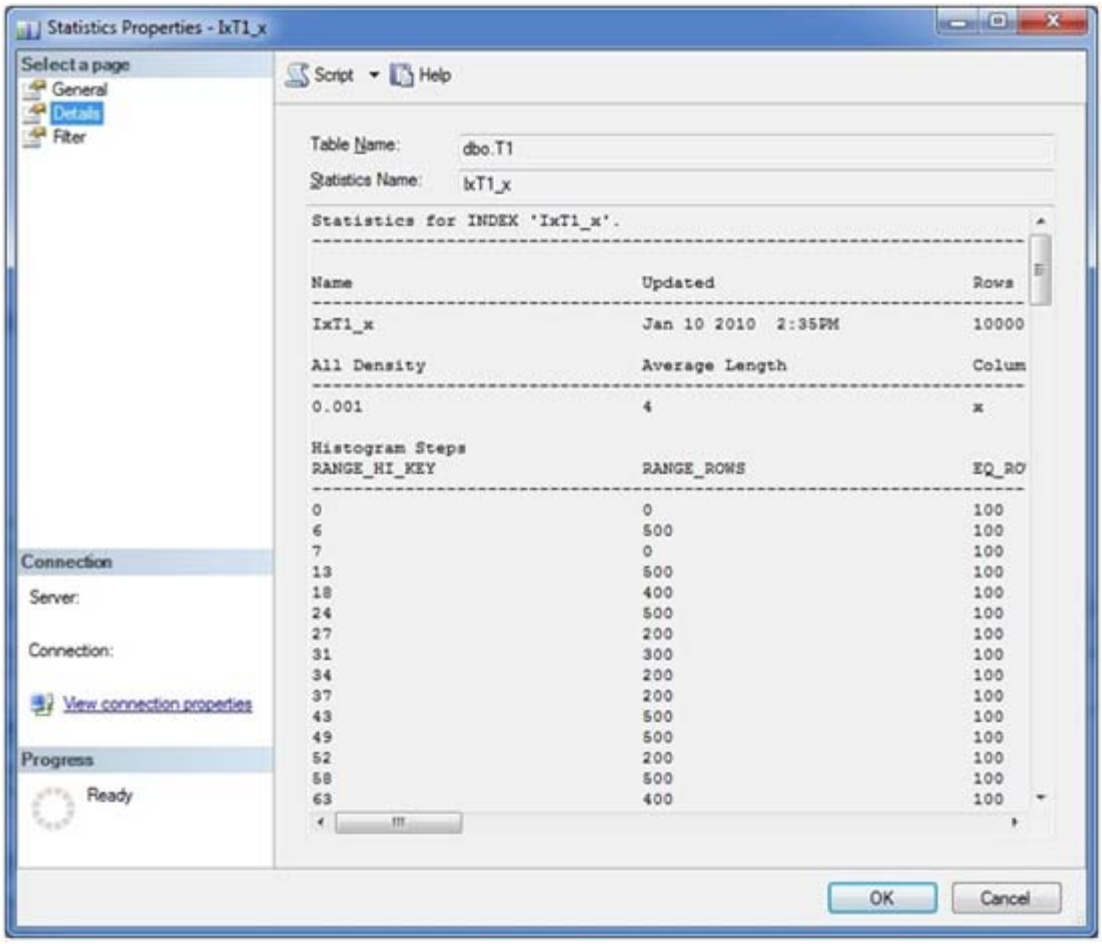


Picture 6: Display of statistics in SSMS Object Explorer

At the moment, there’s one single statistics object with the same name as our non-clustered index. This statistics object has been automatically created at the time that the index was generated. You will find a statistics object like this for every existing index. During index creation or rebuild operations, this statistics object is always refreshed, but it is not done in the course of reorganizing an index. There is one peculiarity with those index-related statistics: if an index is created or rebuilt via CREATE INDEX or ALTER INDEX … REBUILD the sample data that flows into the linked statistics is always obtained by processing *all* table rows. Apparently, for the index generation, all table rows have to be processed anyway. Consequently, the statistics object is created by using all these rows. Usually, this is not the case. In general, for larger tables, only some randomly chosen data are used for the underlying sample. Only for smaller tables, a full scan will always be performed.

SSMS Object Explorer allows the examination of detailed information about a statistics object by opening the properties window. The “General” page provides some data about covered columns and the point in time of the last update. You may also trigger an instant statistics update by selecting the relevant check box.

More interesting data is shown on the “Details” page. Here you will find detailed information about data distributions. Picture 7 shows a sample of what you might see on this page.



Picture 7: Statistics Details Window

Right below the name of the statistics, there are three main areas of information (which unfortunately, do not completely fit inside the provided screen shot):

1. **General information**. In this area you can see information regarding the statistics' creation, and the total number or rows the index encompasses. Also, the number of samples that have been taken from the table for the generation of the histogram is included (see below for an explanation of the histogram).
2. **All Density**. This value represents the general density of the entire statistics object. You may simply calculate this value by the formula:*1/(distinct values)* for the columns comprising the statistics. For single column statistics (as in our example) there's only one row. For multi column statistics, you'll see one row for each left based subset of the included columns. Let's assume we have included three columns c1, c2, and c3 in the statistics. In this case, we'd see three different rows (All Density values): one for (c1), a second one for (c1, c2), and finally a third one for (c1, c2, c3). In our example there's a single column statistics for column x, so there's only one row. There are 1000 distinct values for column x, so the All Density calculates to 0.001. The optimizer will use the 'All Density' value whenever it is not possible to get a precise estimation from the histogram data. We'll come back to this a little later.
3. **Histogram**. This table contains the sample data that has been collected from the underlying table or index. Inside the histogram, the optimizer can discover a more or less exact distribution of a column's data. The histogram is created and maintained (only) for the first column of every statistics – even if the statistics is constructed for more than one column.
   The histogram table will never encompass more than 200 rows. You can find the number of steps inside the first section, containing the general information.
   The histogram columns have a meaning as follows:

Column	Description
RANGE_HI_KEY	This is the column value at the upper interval boundary of the histogram step.

RANGE_ROWS	This is the number of rows inside the interval, excluding the upper boundary.
EQ_ROWS	The value represents the number of rows at the upper boundary (RANGE_HI_KEY). If you like to know the number of rows, including the lower and upper boundary, you can add this value and the number presented through the RANGE_ROWS value.
DISTINCT_RANGE_ROWS	This is the number of distinct interval values, excluding the upper boundary.
AVG_RANGE_ROWS	The value presents the average number of rows for every distinct value inside the interval, again excluding the upper boundary. Hence, this is the result of the expression RANGE_ROWS / DISTINCT_RANGE_ROWS (where DISTINCT_RANGE_ROWS < 0).

Table 1: Histogram columns

Obtaining information about statistics using Transact-SQL

SQL Server provides a large number of system views for retrieving much information on its inner workings. Information regarding statistics is no different. You can use two main system views for viewing information about existing statistics. The **sys.stats** view delivers general facts. You'll see one row for every statistics object. By using the second view, **sys.stats_columns**, you can have a look at the statistics columns, as the name of this view suggests. The following query will return all existing statistics for every user table inside the selected database:

```
select object_name(object_id) as table_name
      ,name as stats_name
      ,stats_date(object_id, stats_id) as last_update
  from sys.stats
 where objectproperty(object_id, 'IsUserTable') = 1
 order by last_update
```

The three columns show table and name of the statistics object, as well as the moment of the last update.

Unfortunately, there's no system view for querying histogram data. If you like to see a statistics' histogram, you'll have to use **DBCC** with the **SHOW_STATISTICS** option like this:

```
DBCC SHOW_STATISTICS(<table_name>, <stats_name>) [WITH <display_option>]
```

yields detailed information on the three areas that you already have seen in Picture 7.

By using the optional **WITH** clause, there's also a possibility of showing just portions of the information that is available. For example, by using **WITH HISTOGRAM**, you'll see only histogram data and the other parts are left out.

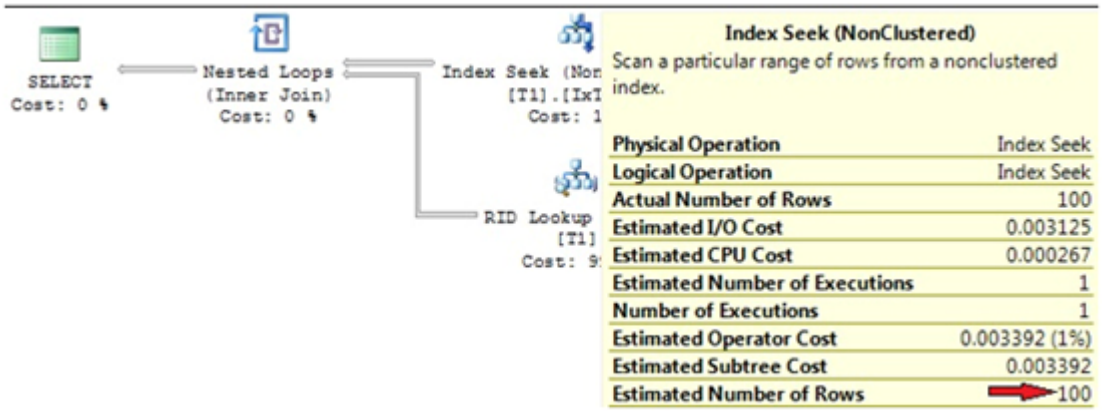
Finally, we also have the stored procedure **sp_helpstats**. This SP will yield general information for statistics on a table. It expects the table's name as parameter. The returned information is very unsubstantial, however, and maybe that's the reason why Microsoft had deprecated this stored procedure.

A more detailed look at the utilization of statistics

So far, we have created a test table **T1** with one index and one statistics (see Picture 7). Now, let's dig a little deeper and see how our existing statistic is used. For the beginning, we start with a very simple statement:

```
select * from T1 where x=100
```

The actual execution plan can be seen in Picture 8.



Picture 8: Actual execution plan

It shows an Index Seek (NonClustered) with an estimated row count of 100, which is quite accurate, isn't it? We are now able to determine how the optimizer computes this estimation. Have a look at the statistics for the utilized index IxT1_x. Picture 9 reveals an excerpt from the histogram (obtained by using DBCC SHOW_STATISTICS).

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
86	400	100	4	100
92	500	100	5	100
97	400	100	4	100
103	500	100	5	100
109	500	100	5	100

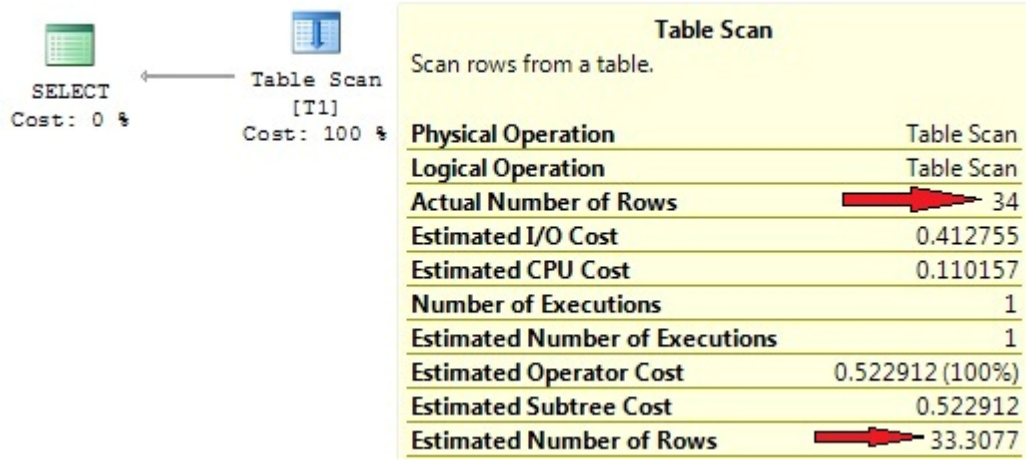
Picture 9: Excerpt from the histogram for statistics IxT1_x

We are querying all rows for a value of x=100. This particular value falls in between the interval bounds 97…103 of the histogram. The average number of rows for each distinct value inside this interval is 100, as we can see in the histogram's **AVG_RANGE_ROWS**. Since we are querying all rows for one distinctive value of x only, and, from the statistics it appears that there is on average 100 rows for each specific value of x inside this interval, the optimizer estimates that there are 100 rows for the value of x=100. That's an excellent estimate, as it concurs precisely with the actual value. The existing histogram works well.

Now have a look at the following query:

```
select * from T1 where a='100'
```

Not surprisingly, you will see a Table Scan in the execution plan, as there is no index on column a (see Picture 10).

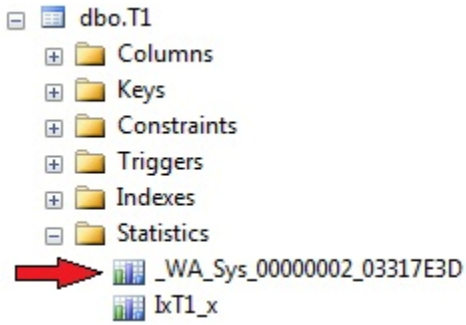


Picture 10: Row count estimation for non-indexed column

The interesting thing here is the “Estimated Number of Rows”, which again is quite exact. How does the optimizer know this value? We don’t have an index for column a, hence there aren’t any statistics. So, where does this estimation come from?

The answer is quite simple. Since there is no statistics, the optimizer will automatically create the missing statistics. In our case, this statistic is created before the plan generation commences, so plan compilation will take slightly longer. There are other options for generation of statistics, which I’ll explain a little further down.

If you open the Statistics folder for table T1 in object explorer, you’ll see the auto-created stats (look at Picture 11).



Picture 11: Automatically created statistics

Auto-created statistics are easily revealed by their automatically created name, which always starts with “_WA”. The **sys.sys_stats** view has a column called **auto_created** that differentiates between automatically and manually created statistics. For index-related statistics, this column will show a value of “false”, even though these statistics are auto-generated as well.

SSMS exposes a problem when it comes to the presentation of detailed data for statistics that are not linked to an index. You will see no detailed data (and thus, no histogram) for these statistics. So, don’t expect to see something like shown in Picture 7 for “pure” column statistics. The good news is that we have an option here: DBCC SHOW_STATISTICS. If you like to see histogram data, make use of this command.

You might wonder what statistics for non-indexed columns are useful for. After all, the only possible search operation is a scan in this case, so why bother about cardinality estimations? Please remember that cardinality estimations not only affect operator selection. There are other issues that are determined by row-count estimations. Samples include: join order (which table is the leading one in a join operation), the utilization of spool operators, and memory grants for things like hash joins and aggregates.

We currently have statistics for columns a (auto created, no related index) and x (linked to index IxT1_x). Both statistics are useful for queries that filter on each of the two columns. But what if we use a filter expression with more that only one column involved? What happens, if we execute the following query?

```
select * from T1 where a='234' and b=1234
```

Row count estimations in the actual execution plan look like shown in Picture 12.

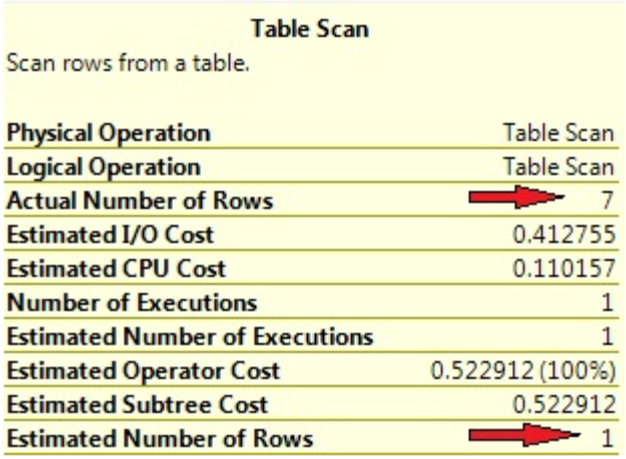


Table Scan

Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Actual Number of Rows	7
Estimated I/O Cost	0.412755
Estimated CPU Cost	0.110157
Number of Executions	1
Estimated Number of Executions	1
Estimated Operator Cost	0.522912 (100%)
Estimated Subtree Cost	0.522912
Estimated Number of Rows	1

Picture 12: Row count estimation for multi-column filter expression

This time, the estimated and actual values differ. In our example, the absolute difference is not dramatic, and doesn't spoil the generated plan. A Table Scan is the best (actually the only) option for performing the query, since we have no index on either of the two columns, a and b, that are used in the predicate. But there may be circumstances where those differences are vital. I'll show you an example in the second part of this article, when we talk about problems with statistics.

The reason for the discrepancy lies in the method for automatically creating statistics. These statistics are never created for more than one column. Although we are filtering on two columns, the optimizer will not establish multi-column statistics for columns (a, b) or (b, a) here. It will only add one single column statistics for column b, since this one does not yet exist. So, the optimizer ends up with two separate statistics for columns a and b. For cardinality estimations, the histogram values for both statistics are combined in some way. I'll explain in detail, how this works a little later. As for now, we can only suppose that this process is somehow fuzzy. But of course, we're talking about estimated values here, which usually will not match exactly with real values (as it was the case in our first experiments).

Nevertheless, you may object that a multi-column statistics would have led to improved estimations here. If the optimizer does not create multi-column statistics, why not do this manually? For this task we can use the CREATE STATISTICS command, as follows:

```
create statistics s1 on T1(b,a)
```

Now, when we execute our last statement again, the estimated row count has improved significantly (see Picture 13).

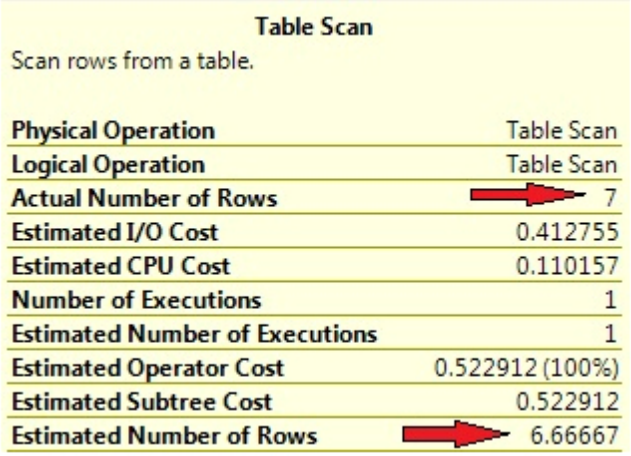


Table Scan

Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Actual Number of Rows	7
Estimated I/O Cost	0.412755
Estimated CPU Cost	0.110157
Number of Executions	1
Estimated Number of Executions	1
Estimated Operator Cost	0.522912 (100%)
Estimated Subtree Cost	0.522912
Estimated Number of Rows	6.66667

Picture 13: Row count estimations with multi column statistics

The two plans itself show no differences, so in our simple case the improved estimation has no impact. But it's not always that easy, so you might consider using multicolumn statistics under certain circumstances.

Please keep in mind that the histogram is always only created for the leading column. So, even with multicolumn statistics, there's only one histogram. The reason for this practice becomes obvious when you consider the sample size again. Remember that the maximum number of rows in a histogram is 200. Now, imagine a two column statistics with separate histograms for each column. Apparently, for each of the 200 samples for the first column, a separate sample for the second column had to be established, so the total sample size for the statistics, as a whole, would equal 200 * 200 = 40,000 samples. For three columns, we'd have already 8,000,000! Doing so would really outweigh the benefits of statistics, namely information-reduction and increasing the speed of plan compilation. In many cases, statistics would really impede the optimization process.

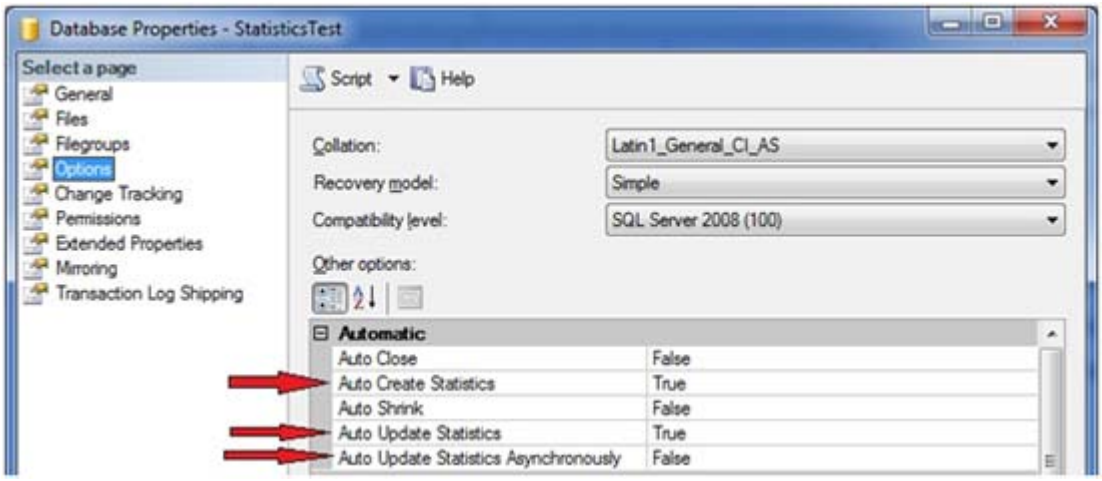
Statistics generation and maintenance

Statistics represent a snapshot of data samples that have come from a table or index at a particular time. They must be kept in sync with the original source data in order to allow suitable cardinality estimations to be made. We will now examine some more of the techniques behind statistics generation and synchronization.

Creation of statistics

You can have your statistics creation done automatically, as is generally the preferable choice. This option is set at the database level, and if you didn't modify your model database in this respect, all of your created databases will have set the AUTO CREATE STATISTICS option to ON initially.

You may query, and set, this option through SSMS. Just open the properties window for your database and switch to the "Options" page. Inside the "Automatic" group you will find the relevant settings (see Picture 14).



Picture 14: Statistics options at the database level

There are three options regarding statistics. As for now, we're only interested in the first one,"Auto Create Statistics", which will turn the automatic statistics creation on or off. I'll explain the other two options in the next section.

Of course, you may also turn the auto-create mechanism on and off using the ALTER DATABASE command, like this:

```
alter database <dbname> set auto_create_statistics [off | on]
```

If you turn off the automatic generation option, your statistics will no longer be created for you by the optimizer. Statistics that belong to an index are not affected by this option, however. As already mentioned, those statistics will always be automatically created during index generation or rebuild.

I suggest setting this option to **on**, and let SQL Server take care of creating relevant statistics. If you, for whatever reason, don't like the automatic mode, you have to create statistics on your own. I'll show you some examples of cases where you might prefer creating statistics manually in some cases, in part two of this article. By no means should you should decide to not use statistics at all. If you opt for working with manually created statistics, there are two options for creating them. The first possibility, which you already know, is the CREATE STATISTICS command. You can create statistics, one-by-one by executing CREATE STATISTICS, which can be painful. Therefore, you may consider another option: **sp_createstats**. This stored procedure will create single-column statistics for all columns in all of your tables and indexed views, depending on options selected. Of course, you are free to use this procedure along with the AUTO CREATE STATISTICS option set to ON. But be aware that there's a chance you end up with a bunch of statistics that never get used, but will have to be maintained (kept in sync with source data). Since we don't have any usage information for statistics, it's almost impossible to detect which of your statistics the optimizer takes advantage of. On the other hand, the chance that the optimizer comes across a missing statistics at run time of a query will be minimized, so missing statistics won't have to be added as part of the optimization process. This may lead to faster plan compilations and overall increased query performance.

Sometimes it's useful to monitor 'auto-stats' creation, and this is where SQL Server Profiler comes into play. You can add the event *Performance/Auto Stats* to watch out for the creation and updating of your statistics. There's a chance that you see longer query execution times whenever this event occurs, since the statistics object has to be created before an execution plan can actually be generated. Because the automatic creation and updating of statistics will usually occur without warning, it can be hard to determine why a particular query was performing very poorly, but only at a particular execution or particular point in time. So this is an argument for creating your statistics manually. If you know which statistics will be useful for your system; you may create them manually, so they won't have to be added at run time. But leave the AUTO CREATION option set to ON nevertheless, so the optimizer may add missing statistics that you haven't been aware of.

CREATE STATISTICS gives you an extra option to specify the amount of data that is processed for a statistics' histogram that you don't have to when relying on auto generation. You may specify an absolute or relative value

for the number of rows to process. The more rows that are processed, the more precise your statistics will be. On the other hand, doing full scans in order to deal with all existing rows and create high-quality statistics will take a longer time, so you have to trade-off between statistics quality and utilized resources for statistics creation.

Here's an illustration of how the sample size may be quantified:

```
create statistics s1 on T1(b,a) with fullscan
create statistics s1 on T1(b,a) with sample 1000 rows
create statistics s1 on T1(b,a) with sample 50 percent
```

If you don't specify the sample size, the default setting is used. This should be appropriate in most cases. I've seen that, for smaller tables, a full scan is always used, regardless of the specified sample size. Also, be aware that the provided sample size is not always used, as only whole data pages are being processed, and always all rows of one read data page are incorporated into the histogram.

Keep in mind that automatically-created statistics are always single-column statistics. If you like to take advantage of multi-column statistics, you have to create them manually. There's another reason for creating statistics manually, which takes us to the next section: filtered statistics.

New in SQL Server 2008: filtered statistics

SQL Server 2008 introduces the concept of filtered indexes. These are indexes that are created selectively, that is, only for a subset of a table's rows. Whenever you create a filtered index, a filtered statistic is also created. Moreover, you may also decide to create filtered statistics manually. As a matter of fact, if you like to try using filtered statistics, you'll have to create them manually, since automatic creation always generates unfiltered statistics.

So, why would you like your optimizer to work with filtered statistics? In general, filtered statistics allow more than one statistics for the same column, but for different rows. Thereby, the following capabilities are available:

1. **Advanced granularity:** Please remember the sample size of the histogram, which is limited to 200 steps. Sometimes this might not be sufficient. Imagine a table with one billon rows, providing only 200 samples to the optimizer for cardinality estimations. This can be far too few. Filtered statistics offer a simple opportunity for increasing the sample size. Just create multiple statistics, ideally with mutually exclusive filter restrictions, so the different histograms won't overlap. In doing so, you originate 200 samples for *every* statistics, which obviously increases the statistics' granularity for the table as a whole and allows for more exact row count estimations.
2. **Statistics tailored to specific queries**. If you have a monster killing query slowing down your application every time it runs, you will have to handle this in some way. The creation of specific statistics for a particular support of this query could also be part of a solution. Filtered statistics increase the capabilities for such adjustments. You'll see a basic example for this a little further down.
3. **Statistics for partitioned tables**. If you allocated a table into different partitions, you probably did so for good reasons. Maybe, you have a large table containing 90% infrequently queried historical data and only 10% of current data that is regularly queried during normal operations. You may want to split this table into two partitions, arranged according to the access patterns. Be aware, however, that statistics are not automatically filtered to match created partitions. By default, statistics are always unfiltered, so automatically created column statistics cover the whole table across all partitions. It may be a good idea to adjust your statistics in a way that they correspond to your partitions. If you did so in our example, you had one histogram for a relatively small portion of the table (10% of table data). Apparently this histogram was of higher quality than one for the table as a whole, and therefore your frequently executed queries could profit from improved cardinality estimations.
4. **Decreased maintenance effort for updating statistics.** Think again about our last example with a separate statistics for the frequently used and rather small current table data. Usually, the historical part of the table experiences no updates, so the statistics for this portion never needs updating after creation. Only statistics for 10% of the table's data will have to be updated, which may cut the overall resource usage for updating statistics. (I say "may", since every update for this part of statistics will need fewer resources, but usually will occur more often.) I'll explain this a little deeper in the problems section at the end of the second part of this article series.

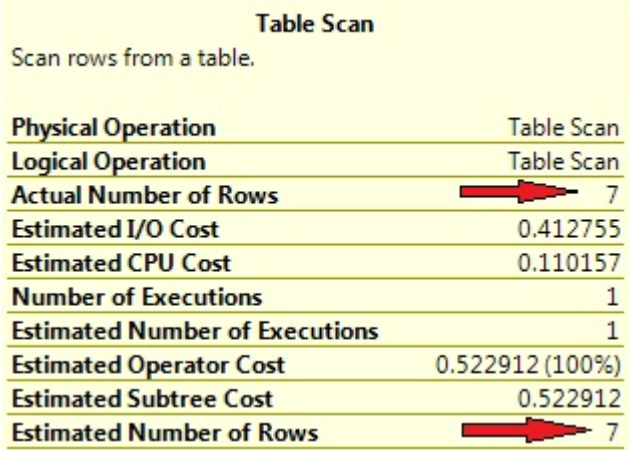
Let's try a simple example. For this example we borrow the following query from a previous experiment:

```
select * from T1 where a='234' and b=1234
```

You can have a look at the actual execution plan in Picture 13. Now, in order to improve the estimated row counts even further, we decide to create a filtered statistics object. We're going to filter on one of the two columns used in the WHERE condition and create the statistics on the other column. This can be very simply accomplished by adding the WHERE clause to the CREATE STATISTICS statement:

```
create statistics s2 on T1(a) where b=1234
```

So this time the statistics is custom-made in respect to our query, and indeed we observe exact row count estimations, as the actual execution plan exposes (see Picture 15).



Picture 15: Filtered statistics and execution plan

You see that filtered statistics can help the optimizer, if they are carefully chosen. Unfortunately, it isn't easy to detect which filtered statistics the optimizer really can take advantage of. You will have to do this by hand which can be fairly painful and also requires some experience. The Database Engine Tuning Advisor will not help you in the process of discovering appropriate filtered statistics, since the creation of filtered statistics is not included into DTA's analysis. As I said: you will have to do this manually!

Also, please be aware that filtered statistics may come at the cost of increased maintenance. If you use filtered statistics for an increased sample size of the histogram, you end up with more statistics and more samples than in the "unfiltered circumstance". All of these additional statistics have to be kept in sync with the table data. This is another reason for an intensified maintenance effort. Remember, if AUTO CREATE STATISTICS is set to ON, always *unfiltered* column statistics will be added, if they are missing. Whenever the optimizer comes across a missing unfiltered statistics for a column, it will add an unfiltered statistics for that column, *regardless of an already existing filtered statistics* that covers the query. So, with manually created filtered statistics and AUTO CREATE STATISTICS set to ON, you will have to maintain filtered *and* non-filtered statistics. This is unlikely to be what you want.

Talking about maintenance, we're now ready to discover how existing statistics can be properly maintained.

Updating statistics

I've already mentioned that statistics only contain redundant data that have to be kept in sync with original table or index data. Of course, it'd generally be possible to revise statistics on the fly, in conjunction with every data update, but I think you agree that doing so would negatively affect all update operations.

Therefore, the synchronization is decoupled from data change operations, which of course also means that you have to beware of the fact that statistics may lag behind original data and are always somewhat out of sync.

SQL Server allows two possibilities of keeping your statistics (almost) in sync. You can leave the optimizer in charge for this task and rely on automatic updates, or you may decide to update your statistics manually. Both methods include advantages and disadvantages and may also be used in combination.

Automatic updates

If you'd like to leave the optimizer in charge of statistics updates, you can generally choose this option at the database level. Picture 14 shows how you may set the AUTO UPDATE STATISTICS with the help of SSMS. Of course, there are also corresponding ALTER DATABASE options available, so you may switch the automatic update on and off via Transact-SQL as well:

```
alter database <dbname> set auto_update_statistics [on | off]
alter database <dbname> set auto_update_statistics_async [on | off]
```

As Picture 14 and the regarding TSQL commands above reveal, SQL Server offers two different approaches of automatic updating:

1. **Synchronous updates**. If the optimizer recognizes any stale statistics, when creating or validating a query plan, those statistics will be updated immediately, *before* the query plan is created or recompiled. This method ensures that query plans are always based on current statistics. The drawback here is that plan generation and query execution will be delayed, because they have to wait for the completion of all necessary statistics' updates.
2. **Asynchronous updates**. In this case, the query plan is generated instantaneously, regardless of any stale statistics. The optimizer won't care about having the latest statistics at hand. It'll just use statistics as they are right at the moment. However, stale statistics are recognized, and updates of those statistics are triggered in the background. The update occurs asynchronously. Obviously, this method ensures for faster plan generation, but may originate only sub-optimal execution plans, since plan compilation may be based on stale statistics. Asynchronous updates are available only as an additional option of the automatic statistics update, thus you can only enable this setting with AUTO UPDATE STATISTICS set to ON.

The above methods offer both advantages and disadvantages. I usually prefer synchronous updates, since I like my execution plans to be as exact as possible, but you may experience improved query performance when you chose 'asynchronous'. After SQL Server installation, the default (and also recommended) setting is automatic and synchronous updates, so if you didn't change your model database, all of your created databases will have this option initially configured.

Of course, there's one open question. If the optimizer is able to discover outdated statistics, what are the criteria for the decision, whether a particular statistics needs a refresh? Which measures are taken into consideration by the optimizer for the evaluation of a statistics' correctness?

SQL Server has to know about the last update of statistics and subsequent data changes in order to detect stale statistics. In earlier versions of SQL Server, this monitoring was maintained at row granularity. Beginning with version 2005, SQL Server now monitors data changes on *columns* that are included in statistics by maintaining a newly introduced, so called ***colmodctr*** for those columns. The value of ***colmodctr*** is incremented every time a change occurs to the regarding column. Every update of statistics resets the value of ***colmodctr*** back to zero.

By the use of the ***colmodctr*** column and the total number of rows in a table, SQL Server is able to apply the following policies to determine stale statistics:

- If the table contained more than 500 rows at the moment of the last statistics update, a statistics is considered as to be stale if at least 20% of the column's data and additional 500 changes have been observed.
- For tables with less or equal than 500 rows from the last statistics update, a statistic is stale if at least 500 modifications have been made.
- Every time the table changes its row count from 0 to a number greater than zero, all of a table's statistics are outdated.
- For temporary tables, an additional check is been made. After every 6 modifications on a temporary table, statistics on temporary tables are also invalidated.

- For filtered statistics the same algorithm applies*, regardless of the filter's selectivity, however*. Column changes are always considered for the entire table, the filtered set is not taken into account. You can easily get trapped by this behavior, as your filtered statistics may "grow old" very fast. I'll explain this more detailed in the problems part at the end of part two of this article.

Unfortunately, ***colmodctr*** is kept secret and not observable, through DMVs. Therefore, you can't watch your statistics in order to uncover those statistics that are likely to be refreshed very soon. (Well, actually you can by connecting via the dedicated administrator connection (DAC) and utilizing undocumented DMVs, but let's just forget about undocumented features and say we don't have access to ***colmodtr*** values.)

It's a common misconception that statistics will be updated automatically whenever the necessary numbers of modifications have been performed, but this is not accurate. Statistics are automatically updated only *when it is necessary in order to get cardinality estimations*. If table T1 contained 100 rows and you insert a million further rows, you'll experience no auto-update statistics during those INSERT operations, but existing statistics will be marked as stale. There's no need for update statistics until cardinality estimations for table T1 have to be made. This will be necessary for instance, if the table is used inside a join, or if one of the columns is used in a WHERE condition.

If you experience problems with too many updates of statistics, you may wish to prevent certain tables or even statistics from automatic updates. I remember a case where we had a table that was used as a command queue. This table had no more than a few rows (by all means less than 500) at any point in time, and usually the command queue was empty quite often and changed very rapidly, since we've processed quite a few commands per second. Every successful processed command was deleted from the queue immediately: Therefore, the row count for this table changed from zero to one quite often. According to the rules I've mentioned above, every time this happened, statistics for this particular table were classified as stale and got updated whenever a command was retrieved from the queue. We were always querying commands from this table via the clustered index, which in this case also represented the primary key, so there was no need to update statistics at all. The optimizer would use the clustered index anyway, which is what we wanted.

There's a stored procedure called sp_**autostats**, which lets you exclude certain tables, indexes or statistics from the automatic updates. So, you can have the AUTO UPDATE STATISTICS set to ON at the database level to be sure, the optimizer can operate with current statistics. If you identify problems with too many updates for certain tables, like in the example above, you can switch the automatic mode off selectively. The CREATE STATISTICS command also offers a solution, if you create statistics manually. This command understands a NORECOMPUTE option that you can apply, if you create a statistics manually and don't want to see automatic updates for this statistics. Likewise, the CREATE INDEX command has an option STATISTICS_NORECOMPUTE than can be switched on, if you don't like to experience automatic updates for the index-linked statistics.

Manual updates

If we have an automatic mode, why would we bother about manual updates at all? Wouldn't it be easier to rely on the automatic process and let SQL Server deal with updates of statistics?

Yes, this is almost the case. The automatic update is fine in many cases. It ensures your statistics are current, which apparently is what you should be after. But you may also face times when it makes sense to updated statistics manually. Some of the reasons are covered in the problems section below, but here's an outline:

- I've already mentioned that you may decide to exclude statistics from automatic updates. If you do so, instead of never updating those statistics, you could also perform occasional manual updates. This might be best, for example, after bulk loads into a table that experiences no changes beside those bulk loads.
- Remember that for tables with a relevant number of rows, statistics are updated automatically only after about 20% of the table's data were modified. This threshold could be slightly too high. Therefore, you might decide to perform additional manual updates. I'll explain a circumstance where this would be the case in part 2.
- If you rely on the automatic update of statistics, a lot of those updates may happen during operational hours and adversely affect your system's performance. Asynchronous updates provide one solution to this, but it may also be a good idea performing updates of stale statistics at off-peak times during your nightly maintenance window.
- The automatic update will create a histogram only by applying the default, and limited, sample size. Increasing the sample size for selected statistics, and thus the quality of the statistics, may be

necessary if you experience performance problems. If you'd like distinct statistics to be updated with sample size set to full e.g., so all of a table's rows are processed for the histogram, you will have to perform the update manually.
- If you use filtered statistics, you have to be aware that the definition of 'stale' says nothing about the selectivity of the filter. If changes occur that modify the selectivity of any of your filters, you very likely have to perform manual updates of those statistics.
- There's another issue with filtered stats, as the automatic update is very likely to be less than perfect. Trust me; you will want to support automatic updates through manual refreshes when using filtered statistics. You'll get a deeper explanation of this in part two of this article.

If you plan for manual updates, there are two options. You can use the UPDATE STATISTICS command or the stored procedure **sp_updatestats**.

UPDATE STATISTICS allows specific updates of distinct statistics, as well as updates of all existing statistics for a table or index. You may also specify the sample size, which is the amount of table data that is processed for updating the histogram. The somewhat simplified syntax looks like this:

```
update statistics <tablename> [<statsname>]
  [with fullscan | <samplesize> [,norecompute] [,all | ,columns | ,index]]
```

Besides using the FULLSCAN option, which will produce high quality statistics, but may consume a lot of resources, UPDATE STATISTICS also allows the specification of the sample size in terms of the percentage of a table's data, or by itemizing the absolute number of rows to process. You may also direct the update to rely on the provided sample size that has been specified in a prior CREATE or UPDATE STATISTICS operation. Here are some examples:

Update all statistics tor table T1 with full scan:

```
update statistics T1 with fullscan
```

Update all column statistics (that is, statistics without a related index) for table T1 by using the default sample size and exclude those statistics from further automatic updates:

```
update statistics T1 with columns, norecompute
```

Update statistics s1 on table t1 by processing 50 percent of the table's rows:

```
update statistics T1 s1 with sample 50 percent
```

When you decide for manual updates of statistics, please keep in mind that index-rebuild operations also update statistics that are related to a rebuilt index. As an index rebuild will always have to handle all table rows, the linked statistics will, as a consequence, be updated with full scan. Therefore it's a bad idea to perform manual updates of index-related statistics *after* index-rebuild operations. It is not only that you perform the update twice, but the second update can only lead to statistics of declined quality (if you don't use the FULLSCAN option). UPDATE STATISTICS has the COLUMNS option for this scenario. By specifying this option, only statistics without a connected index are affected by the update, which is very useful.

With UPDATE STATISTICS, manual updates can be cumbersome, since you have to execute UPDATE STATISTICS at least at the table level. Also you may want to execute UPDATE STATISTICS only for statistics that really need those updates. Since the ***colmodctr*** value is hidden from the surface, it's impossible to detect those statistics.

SQL Server provides the stored procedure **sp_updatestats** that you may want to consider when you decide to support automatic updates through additional manual updates. **Sp_updatestats** is executed at the database level and will perform updates for all existing statistics, *if necessary*. I've been curious about how **sp_updatestats** figures out, whether an update is required or not, since I'd like to know how the ***colmodctr*** is

accessed. When I carefully investigated the source code for this SP, it became obvious however that only the old ***rowmodctr*** is being used to support the decision, whether an update should be performed or not. So, **sp_updatestats** does not take advantage of the improved possibilities that we have through the ***colmodctr*** since SQL Server 2005.

If you have a maintenance window, you may want to execute **sp_updatestats** also inside this window. But be aware that **sp_updatestats** will update all statistics that have experienced the change of at least one underlying row since the last statistics update. So, **sp_updatestats** will probably update a lot more statistics than usually necessary. Depending on your requirements and available resources during the update, this may or may not be appropriate.

Sp_updatestats knows one string parameter that can be provided as 'resample'. By specifying this parameter the underlying UPDATE STATISTICS command will be executed with the RESAMPLE option, so a previously given sample size is being reused.

Please keep in mind that updating statistics will result in cached plan invalidations. Next time a cached plan with a meanwhile updated statistics is detected during query execution, a recompilation will be performed. Therefore you should avoid unnecessary updates of statistics.

Summary

This article has explained what statistics are, how the optimizer makes use of statistics, and how statistics are created and maintained.

In particular, you should now be familiar with:

- Automatic and manual creation of statistics
- Automatic and manual updating statistics as well as the definition of stale statistics
- Filtered statistics
- Querying information about statistics
- Single- and multi column statistics
- The structure of a statistics' histogram

Now that you know the fundamentals, the second part will be concerned with problems that you may experience and of course also present solutions to those problems.

Part 2: SQL Server Statistics: Problems and Solutions.

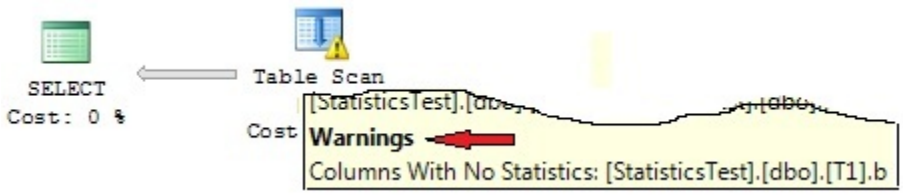
Normally, you do not need to be too concerned about the way that your SQL Queries are executed. These are passed to the Query Optimizer which first checks to see if it has a plan already available to execute it. If not, it compiles a plan. To do so effectively, it needs to be able to estimate the intermediate row counts that would be generated from the various alternative strategies for returning a result. The Database Engine keeps statistics about the distribution of the key values in each index of the table, and uses these statistics to determine which index or indexes to use in compiling the query plan. If, however, there are problems with these statistics then the performance of queries will suffer. So what could go wrong, and what can be done to put things right? We'll go through the most common problems and explain how it might happen and what to do about it.

There is no statistics object at all

If there's no statistics, the optimizer will have to guess row-counts rather than estimate them, and believe me: this is not what you want!

There are several ways of finding out from both the estimated and actual execution plans whether the optimizer comes across missing statistics. In this case you'll see warnings in the plan. The there will be an exclamation

mark in the graphical execution plan and a warning in the extended operator information, just like the one in Picture 1. You won't see this warning for table variables, so watch out for table scans on table variables and their row-count estimations.



Picture 1: Missing statistics warning

If you want to have a look at execution plans for preceding queries, you may request these plans by the use of the **sys.dm_exec_cached_plans** DMV.

SQL Server Profiler offers another option. If you include the event *Errors and Warnings/Missing Column Statistics* in your profile, you'll observe a log entry whenever the optimizer detects a missing statistics. Be aware however that this event will not be fired if AUTO CREATE STATISTICS is enabled for the database or table.

There are several circumstances under which you'll experience missing statistics.

AUTO CREATE STATISTICS is OFF

Problem:

If you've set the option AUTO CREATE STATISTICS OFF and overlooked the task of creating statistics manually, the optimizer will suffer from missing statistics.

Solution:

Rely instead on the automatic creation of statistics by setting AUTO CREATE STATISTICS to ON.

Table variables

Problem:

For table variables, statistics will never be maintained. Keep that in mind: no statistics for table variables! When selecting from table variables, the estimated row-count is always 1, unless a predicate that evaluates to false, and doesn't have any relationship to the table variable, is applied (such as WHERE 1=0), in which case the estimated row-count evaluates to 0.

Solution:

Don't count on using table variables for temp tables if they are likely to contain more than a few rows. As a rule of thumb, use temporary tables (tables with a "#" as their name's first character), rather than table variables, for temp tables with more than 100 rows.

XML and spatial data

Problem:

SQL Server does not maintain statistics for XML and spatial data types. That's just a fact, not a problem. So, don't try to find statistics for these columns, as they're simply not there.

Solution:

If you experience performance problems with queries that have to search through XML data, or filter spatial columns e.g., XML or spatial indexes may help. But that's another story and beyond the scope of this article.

Remote queries

Problem:

Suppose that you query a table from an Oracle database via a linked server connection and join this table to some local database tables. It's perfectly understandable that SQL Server has no idea of any row-counts from that remote table, since it is residing in an Oracle database. This can occur, if you use OPENROWSET or OPENQUERY for remote data access.

But you may also face this issue out of the blue, when working with DMVs. A certain number of SQL Server's DMVs are no more than a shell for querying data from internal tables. You may see a Remote Scan operator for SQL Server 2005 or a Table Valued Function operator for SQL Server 2008 in this case. Those operators have tied default cardinality estimations. (Depending on the server version and the operator, these are 1, 1000, or 10000.)

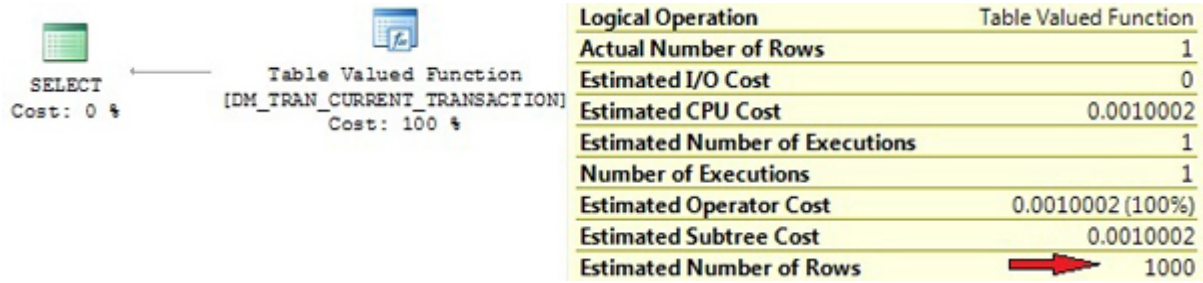
Have a look at the following query:

```
select * from sys.dm_tran_current_transaction
```

Here's what BOL states:

"Returns a single row that displays the state information of the transaction in the current session."

But look at the execution plan in Picture 2. No estimated row-count of 1 there. Apparently the optimizer does not take BOL content into account when generating the plan. You see the internal table DM_TRAN_CURRENT_TRANSACTION that is called by invoking OPENROWSET and an estimated row-count of 1000, which is far from reality.



Picture 2: Row-count estimation for TVF

Solution:

If possible, give the optimizer some support by specifying the row-count through the TOP(n) clause. This will only be effective if n is less than the Estimated Number of Rows, 1000 in our example. So, if you have a guess of how many rows your Oracle query is about to return, just add TOP(n) to your OPENQUERY statement. This will help for better cardinality estimations and be useful if you utilize the OPENROWSET result for further joining or filtering. But be sure that this practice is somewhat dangerous. If you specify a value too low for n, your result set will lose rows, which would be a disaster.

With our example using **sys.dm_tran_current_transaction** , you would do better by rewriting the query like this:

```
select top 1 * from sys.dm_tran_current_transaction
```

Normally this won't be necessary, because the simple SELECT, if used stand-alone, is quick enough. However, if you process the query result further by, perhaps, using it in joins, then the TOP 1 is useful.

If you find that you are using the OPENQUERY results in more complex queries for joining or filtering operations, then it is wise to import the OPENQUERY result into a local table first. Statistics and indexes can be properly maintained for this local table, so the optimizer then has enough information for realistic estimations of row-counts.

The database is read only

Problem:

If your database is set to read-only, the optimizer can't add missing statistics even with AUTO CREATE STATISTICS enabled, because changes to read only databases are generally prevented.

Watch out for the fact that there is also a special kind of read only database. Yes, I'm talking about a snapshot! If the optimizer is missing the statistics for database snapshots, these can't be created automatically. This is likely to happen if snapshots are being used for reporting applications. I often read suggestions to create snapshots for reporting purposes so as to avoid running those long running and resource-intensive reporting queries on the underlying OLPT systems. That may be fine up to a point, but Reporting queries are highly unpredictable and usually differ from normal OLTP queries. Therefore, there's a chance that your reporting queries will suffer from missing statistics or, even worse, missing indexes.

Solution:

If you set your database to read-only, you'll have to create your statistics manually before you do so.

Perfect statistics exist, but can't be used properly

There is always a possibility that the optimizer cannot use statistics even if they are in place and up-to-date. This can be caused by poor TSQL code such as you'll see in this section.

Utilization of local variables in TSQL scripts

Problem:

Let's again have a look at this query form our introductory example:

```
declare @x int
set @x = 2000
select c1,c2 from T0
 where c1 = @x
```

Picture 5 of the first part shows the actual execution plan, which reveals a large discrepancy between the actual and estimated row-count. But what's the reason for this difference?

If you're familiar with the different steps of query execution, you'll realize the answer to this. Before a query is executed, a query plan has to be generated, and right at the time of the compilation of the

query, SQL Server does not know the value of variable @x. Yes of course, in our case it'd be easy to determine the value of @x, but there may be more complicated expressions that prevent the calculation of @x's value at compile time In this case, the optimizer has no knowledge of the actual value of @x and is therefore unable to do a reasonable estimation of cardinality from the histogram.

But wait! At least there's a statistics for column c1, so if the optimizer can't browse the histogram, it may probably fall back to other, more general quantities. That's exactly what happens here.

In case the optimizer can't take advantage of a statistic's histogram, the estimations of cardinality are determined by examining the average density, the total number of rows in the table, and possibly also the predicate's operator(s). If you look at the example from the first part, you'll see that we inserted 100001 rows into our test table, where column c1 had a value of 2000 for only one row and a value of 1000 for the remaining 100000 rows. You can inspect the average density of the statistics by executing DBCC SHOW_STATISTICS, but remember that this value calculates to *1/distinct number of values*, which evaluates to 1/2=0.5 in our case. Hence, the optimizer calculates the average number of rows per distinct value of c1 to 100001 rows * 0.5 = 50000.5 rows. With this value at hand, the predicate's operator comes into play, which is "=" in our example. For the exact comparison, the optimizer assumes that the average number of rows for one value of c1 will be returned, thus the expected row-count is 50000.5 (again, see Picture 5 of part one).

Other comparison operators lead to different estimates of selectivity, where the average density may or may not be considered. If 'greater' or 'less than' operators are applied, then it's just assumed that 30% of all table rows will be returned. You can easily verify this by playing around with our little test script.

Solution(s):

If possible, avoid the using local variables in TSQL scripts. As this may not always be practicable, there are also other options available.

First, you may consider introducing stored procedures. These are perfectly designed for working with parameters by using a technique that's called parameter sniffing. When a procedure is called for the first time, the optimizer will figure out any provided parameters and adjust the generated plan (and cardinality estimations, of course) according to these parameters. Although you might face other problems with this approach (see below), it is a perfect solution in our case.

If we embed our query into a stored procedure like this:

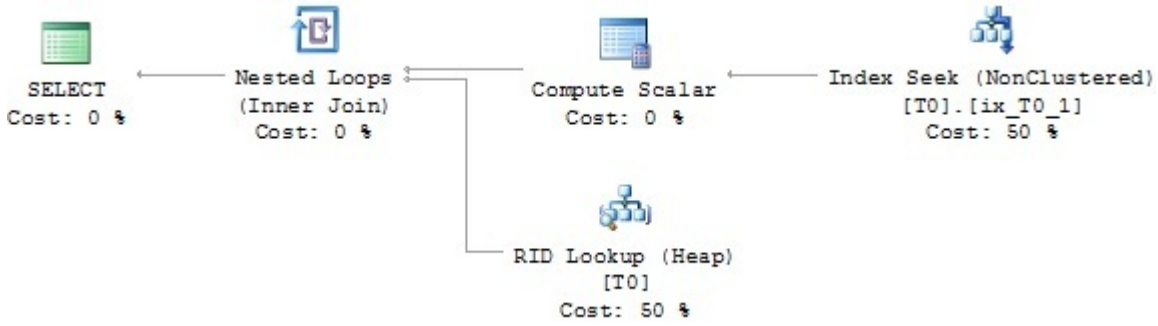
```
create procedure getT0Values(@x int) as
 select c1,c2 from T0
  where c1 = @x
```

And then just execute this procedure by invoking:

```
exec getT0Values 2000
```

The execution plan will expose an Index Seek, and thus look as it should. That's because the optimizer knows it has to generate the plan for a value of @x=2000.

Picture 3 reveals the plan. Compare this to the original plan presented in Picture 5 of part one.



Picture 3: Execution plan of stored procedure

Secondly, you may consider solving the issue by the use of dynamic SQL. Ok, just to clarify this: No, no, no, I do not suggest using dynamic SQL in general and extensively. By no means would that be appropriate! Dynamic SQL has very bad side effects, such as possible Plan cache pollution, vulnerability to SQL injection attacks, and possibly increased CPU- and memory utilization. But look at this:

```
declare @x int
       ,@cmd nvarchar(300)
set @x = 2000
set @cmd = 'select c1,c2 from T0 where c1='
               + cast(@x as nvarchar(8))
exec (@cmd)
```

The execution plan is flawless now (same as in Picture 3), since it is created during the execution of the EXEC command, and the provided command string is passed as parameter to this command. In reality, you'd have to balance the consequences before deciding to use dynamic SQL. But you can see that, if carefully chosen and selectively applied, dynamic SQL is not bad in all cases. Altogether it's very simple: just be aware of what you do!

More often than not the extended stored procedure **sp_executesql** will help you using dynamic SQL, while also excluding some of the dynamic SQL drawbacks at the same time.

Here's our example again, this time re-written for **sp_executesql**:

```
exec sp_executesql N'select c1,c2 from T0 where c1=@x'
                  ,N'@x int'
                  ,@x = 2000
```

Once more the execution plan looks like the one presented in Picture 3.

Providing expressions in predicates

Problem:

Using expressions in predicates may also prevent the optimizer from using the histogram. Look at the following example:

```
select c1,c2 from T0
 where sqrt(c1) = 100
```

The execution plan is shown in Picture 4.

Picture 4: Bad cardinality estimation because of expression in predicate

So, although a statistics for columns c1 exists, the optimizer has no idea of how to apply this statistics to the *expression* POWER(c1,1) and therefore has to guess the row-count here. This is very similar to the problem of the missing statistics we've mentioned already, because there's simply no statistics for the expression POWER(c1,1). In terms of the optimizer, POWER(c1,1) is called a non-foldable expression. Please refer to this article for more information.

Solution(s):

If possible, re-write your SQL Code so that comparisons are only done with "pure" columns. So e.g. instead of specifying

```
where sqrt(c1) = 100
```

You would do better to write:

```
where c1 = 10000
```

Fortunately the optimizer is quite smart when evaluating expressions and in some cases will do the re-write internally (again see here for some more information about this).

If re-writing your query is not possible, I suggest you ask a colleague for assistance. If it's still not feasible, you may take calculated columns into account. Calculated columns will work out fine, since statistics are also maintained for calculated columns. Furthermore you may also create indexes for calculated columns, which you can't do for expressions.

Parameterization issues

Problem:

If you use parameterized queries, such as the stored procedure in a previous example, you might face another problem. You'll remember that the query plan is actually generated on the first *call* of the procedure, not the CREATE PROCEDURE statement. That's the way, parameter sniffing works. The plan will be generated by utilizing row-count estimations for the provided parameter values that were provided at the first call. The problem here is quite obvious. What, if the first call is done with uncharacteristic parameter values? Cardinality estimations will use these values and the created execution plan then goes into the plan cache. The plan will have a poor estimation of row-counts, so subsequent reuse of the plan with more usual parameter values are likely to suffer poor performance.

Have a look at our stored procedure call:

```
exec getT0Values 2000
```

If this is the first call of the procedure, the plan is generated for the filter WHERE c1=2000. Since there's only one estimated row for c1=2000, an Index Seek is performed. If we'd call the procedure a second time like this:

```
exec getT0Values 1000
```

The cached plan is re-used and again an Index Seek is performed. This is a very bad choice, because the query will now return 100000 rows. There is a big difference between the estimated and actual row-count! A Table Scan had been much more efficient here.

There's another issue with parameterization that is very similar to the problem of using local variables in TSQL scripts that we've already written about. Consider this stored procedure:

```
create procedure getT0Values(@x int) as
 set @x = @x * 2
 select c1,c2 from T0
  where c1 = @x
```

Modifying the value of @x is not ideal. The parameter sniffing technique will not trace any modifications of @x, so the execution plan is adjusted to the *provided* value of @x only, *not* the actual value that is used inside the query. If you call the above stored procedure like this:

```
exec getT0Values 1000
```

Then, the plan actually gets optimized for the filter WHERE c1=1000 and *not* WHERE c1=2000!

You really fool the optimizer by this practice, and unsatisfactory estimations of cardinality are highly likely to happen.

Solution:

If you are having problems that are caused by parameter sniffing, you can consider using query plan hints, such as OPTIMIZE FOR, or WITH RECOMPILE. This topic is outside the scope of this article, however.

Try not to modify parameter values within stored procedures. Doing so will outfox parameter sniffing. If you have to change parameter values for further processing, you might consider splitting the stored procedure into several smaller stored procedures. Introduce stored procedures in the sense of sub-routines. Modify any parameter values in the frame procedure and call your sub-routine procedure by using the changed value. As for every stored procedure, a separate plan is generated: This practice will circumvent the issue.

Statistics is too imprecise

Generally we have to accept a certain amount of uncertainty with distribution statistics. After all, statistics perform some data reduction, and information losses are inevitable. But if the information that we lose is crucial for the optimizer to make a proper estimate of cardinality, we must look for some other solution. How can statistics objects become just too imprecise to be useful?

Insufficient sample size

Problem:

Imagine a table with some million rows. Whenever SQL Server automatically creates or updates statistics for one of the columns of this table, it won't take all of the table's rows into account. To prevent exhaustive use of resources, like CPU and IO, usually only some sample rows are processed for the maintenance of statistics. This could lead to a histogram that poorly represents the overall distribution of data. If the optimizer estimates the cardinality, it may not be able to find enough information to generate an efficient query plan.

Solution:

The general solution is very simple. You'll have to interfere in the automatic update or creation of statistics by manual updates or generation of statistics. Remember that you can specify the sample size or even perform a full scan when invoking CREATE STATISTICS or UPDATE STATISTICS.

Remember that an index rebuild will always force the production of statistics that are created with a full scan. Be aware though that an index rebuild has no effect on column statistics but only regards index-related statistics.

Statistics' granularity is too broad

Problem:

Let's come back to our table with multi-million rows again. Although the maximum size of the histogram is always limited to 200 entries, we know that we only have 200 clues for row-count estimations of a, let's say a 4 million rows table. That's an average of 4000000 rows/200 steps = 20000 rows per histogram step. If the column's values were equally shared across all rows this won't be a problem, but what if they aren't? The statistics will probably be too coarse-grained and lead to inaccurate estimations of cardinality.

Solution:

Apparently it would be helpful to increase the histogram size, but we can't do that can we? No, there's no way of enlarging the size of a histogram to contain more than 200 entries. But if we can't include more rows into one histogram, how about simply using more than one histogram? Of course we can achieve that by introducing filtered statistics. Since the histogram is always tied to a single statistics, we can have multiple histograms for the same column when using filtered statistics.

You will have to create these filtered statistics manually, and you'll have to be aware of the consequences that I'll explain a bit later.

Let's just return to a previous example and imagine that we have a table containing 90% of historical data (that never get changed) and only 10% of active data. Gladly we decide on having two filtered statistics, and separate these two statistics objects by applying a filter condition to some date column. (You might prefer two filtered indexes but, for this illustration, it doesn't matter whether it's filtered indexes or statistics.) Additionally, we can disable automatic updates for the statistics that represent only historical data (the 90% part): This is because it is only our active part, the 10%-part, for which the statistics need to be regularly refreshed.

At this point we hit a problem: With CREATE AUTO STATISTICS set to ON, the optimizer will always add an unfiltered statistics object as well, even if filtered statistics for the same column already exist. This automatically-added statistics will also have the 'automatic update' option enabled. Therefore, although it may seem that the two filtered statistics are the only ones in place, one for the historical, and another one for the active part, you'll end up with a total of three statistics, since another (unfiltered) one has been automatically added. Not only do we now have the superfluous unfiltered statistics but there will also be pointless automatic updates to it. Of course we could disable the automatic creation of statistics for the entire table, but this would put the onus on us for creating all appropriate statistics. I don't like this idea very much. If there's an automatic option, why not rely on it whenever possible.

I think the best way to overcome this issue is to create the unfiltered statistics manually by specifying the NORECOMPUTE option. This will prevent the optimizer from adding this statistics object and will also avoid further automatic updates.

The steps necessary are as follows:

Create one unfiltered statistics with NORECOMPUTE set to ON in order to prevent automatic updates. This statistics is only needed to "mislead" the optimizer.

Create another filtered statistics for the 10% active part, but this time with automatic updates enabled.

If desirable, create another filtered statistics for the 90% part and again, disable automatic updates.

If you follow the above steps, you will obtain more decent statistics for the active part of your table. But unfortunately this solution is not free of hitches, as the next section will reveal.

Stale statistics

I've already mentioned that the synchronization of statistics always lags behind actual data modifications. Therefore, almost every statistics object is stale – at least to a certain level. In many cases this behavior is absolutely acceptable, but there are also situations where the deviation between source data and statistics may be too great.

Problem:

You already know that for tables with more than 500 rows at least 20% of a column's data had to be changed in order to invalidate any linked statistics, so that these statistics receive an update next time they're needed. That threshold of "at least 20%" can be far too large under several circumstances. This is best explained by introducing another sample.

Let's say we have a product-table that looks like this:

```
if (object_id('Product', 'U') is not null)
  drop table Product
go
create table Product
 (
   ProductId int identity(1,1) not null
  ,ListPrice decimal(8,2) not null
  ,LastUpdate date not null default current_timestamp
  ,filler nchar(500) not null default '#'
 )
go
alter table Product add constraint PK_Product
  primary key clustered (ProductId)
```

Among a surrogate primary key and some other columns, there's also a column that holds the last modification of the product. Later on, we're going to search products for distinct dates or date ranges. Therefore we also create a nonclustered index for the **LastUpdate** column:

```
create nonclustered index ix_Product_LastUpdate on Product(LastUpdate)
```

Now, let's add 500000 products to our table:

```
insert Product(LastUpdate, ListPrice)
 select dateadd(day, abs(checksum(newid())) % 3250,'20000101')
       ,0.01*(abs(checksum(newid())) % 20000)
   from Numbers where n <= 500000
go
update statistics Product with fullscan
```

We use some randomly calculated values for **LastUpdate** and **ListPrice** and also update all existing statistics after the INSERT has been completed.

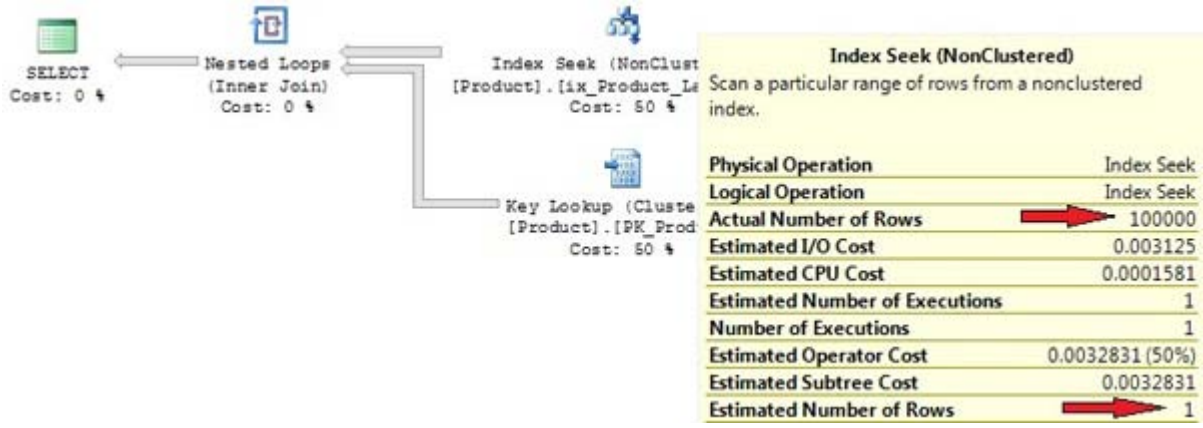
One fine day, after some tough negotiations, we're pleased to announce the takeover of our main competitor by the first of January 2010. Very happily we add all of their 100000 products to our portfolio:

```
insert Product(LastUpdate,ListPrice)
 select '20100101', 100 from Numbers where n <= 100000
```

Just to make sure, which products have been added, we check all inserted rows by executing the following statement:

```
select * from Product where LastUpdate = '20100101'
```

See the actual execution plan for the above statement in Picture 5.



Picture 5: Execution plan created by use of stale statistics

Due to stale statistics, there's a huge discrepancy between the actual and estimated row-count. The 100000 rows that we've added are below the required threshold of 20% modifications; which means that an automatic update isn't executed. The Index Seek that was used is really not the best choice for retrieving 100000 rows, with additional Key Lookups for every returned key from the Index Seek. The statement took about 300000 logical reads on my PC. A Table or Clustered Index Scan would have been a much better option here.

Solution:

Of course we have the opportunity to provide our knowledge to the optimizer by the use of query hints. In our case, if we knew a Clustered Index Scan was the best choice, we could have specified a query hint, like this:

```
select * from Product with (index=0)
 where LastUpdate = '20100101'
```

Although the query hint would do it, specifying query hints is somewhat dangerous. There's always a possibility that query parameters are modified or underlying data change. When this happens, your formerly useful query hint may now negatively affect performance. An update statistics is a much better choice here:

```
update statistics Product with fullscan
```

After that, a Clustered Index Scan is performed and we see only about 86000 logical reads now.

So, please don't rely solely on automatic updates. Switch it on, but be prepared to support the automatic process by manual updates, probably at off-peak times during your maintenance window. This is especially important for statistics on columns that contain constantly increasing values such as IDENTITY columns. For every added row the IDENTITY column will be set to a value that is above the highest item in the histogram, making it difficult or impossible for the optimizer obtaining proper row-count estimations for such column values. Usually you will want to update statistics on these columns more frequently than only after 20% of changes.

Problem

Filtered statistics pose two very particular problems when it comes to automatic updates.

First, any data modifications that change the selectivity of the filter are not taken into account to qualify for the automatic invalidation of existing statistics.

Second and more important, the “20% rule” is applied to all of a table’s rows, *not* only to the filtered set. This fact can outdate your filtered statistics very rapidly. Let’s again return to our example with 10% of active data inside a table and a filtered statistics on this portion of the table. If all data of the filtered set have been modified, although this is 100% of the filtered set (the active data), it’s only 10% of the table. Even if we’d change all of the 10% part once again, regarding the table, we only have 20% of data changes. Still our filtered statistics will not be considered as being outdated, albeit we’ve already modified 200% of the data! Please, also keep in mind that this likewise applies to filtered statistics that are linked to filtered indexes.

Solution

For most of the problems I’ve mentioned so far, a reasonable solution involves the manual update or creation of statistics. If you introduce filtered indexes or filtered statistics, then manual updates become even more important. You shouldn’t rely solely on the automatic update of filtered statistics, but should perform additional manual updates rather frequently.

No automatic generation of multi-column statistics

Problem:

If you rely on the automatic creation of statistics, you’ll need to remember that these statistics are always single-column statistics. In many situations the optimizer can take advantage of multi-column statistics and retrieve more exact row-count estimations, if multi-column statistics are in place. You’ve seen an example of this in the introduction.

Solution:

You have to add multi-column statistics manually. If, as a result of some query analysis, your suspect multi-column statistics will help the optimizer, just add them. Finding supportive multi-column statistics can be quite difficult, but the Database Engine Tuning Advisor (DTA) can help you with this task.

Statistics for correlated columns are not supported

There is one particular variation where statistics may not work as expected, because they’re simply not designed to: correlated columns. By correlated columns we mean columns which contain data that is related. Sometimes you will come across two or more columns where values in different columns are not independent of each other; such examples might include a child’s age and shoe size, or gender and height.

Problem:

In order to show why this might cause a problem, we’ll try a simple experiment. Let’s create the following test table, containing a list of rental cars:

```
create table RentalCar
 (
  RentalCarID int not null identity(1,1)
      primary key clustered
 ,CarType nvarchar(20) not null
 ,DailyRate decimal(6,2)
```

```
 ,MoreColumns nchar(200) not null default '#'
 )
```

This table has two columns, one for the car type and the other one for a daily rate, along with some other data that is of no particular significance for this experiment.

We know that applications are going to query for 'car type' and 'daily rate', so it may be a good idea to create an index on this columns:

```
create nonclustered index Ix_RentalCar_CarType_DailyRate
   on RentalCar(CarType, DailyRate
```

Now, let's add some test data. We will include four different car types with daily rates adjusted to the car's type: the better the car, the higher the rate of course. The following script will take care of this:

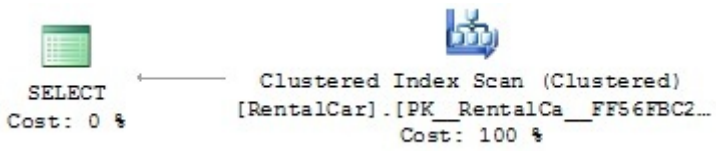
```
with CarTypes(minRate, maxRate, carType) as
 (
  select 20, 39, 'Compact'
   union all select 40, 59, 'Medium'
   union all select 60, 89, 'FullSize'
   union all select 90, 140, 'Luxory'
 )
insert RentalCar(CarType, DailyRate)
  select carType, minRate+abs(checksum(newid()))%(maxRate-minRate)
    from CarTypes
         inner join Numbers on n <= 25000
go
update statistics RentalCar with fullscan
```

As you can see, we will have daily rates for luxury cars in the range between $90 and $140, whereas compact cars are given with daily rates between $20 and $39, e.g. In total the script adds 100000 rows to the table.

Now, let's imagine that a customer asks for a luxury car. Since customers always demand everything to be very low-priced, she doesn't want to spend more than $90 per day for this car. Here's the query:

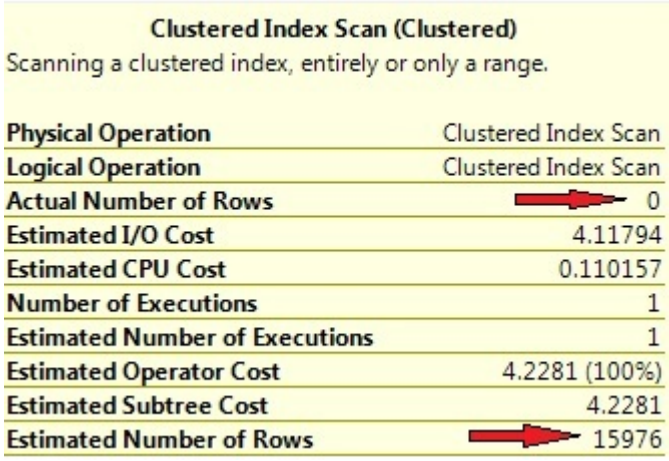
```
select * from RentalCar
 where CarType='Luxory'
   and DailyRate < 90
```

We know that we haven't a car like this in our database, so the query returns zero rows. But have a look at the actual execution plan (see Picture 6).



Picture 6: Correlated columns and Clustered Index Scan

Why isn't our index being utilized here? The statistics are up to date, since we've explicitly executed an UPDATE STATISTICS command after inserting the 100000 rows. The query returns 0 rows, so the index *is* selective and should be used, right? A look at the row-count estimations gives the answer. Picture 7 shows the operator information for the clustered index scan.



Clustered Index Scan (Clustered)

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Number of Rows	0
Estimated I/O Cost	4.11794
Estimated CPU Cost	0.110157
Number of Executions	1
Estimated Number of Executions	1
Estimated Operator Cost	4.2281 (100%)
Estimated Subtree Cost	4.2281
Estimated Number of Rows	15976

Picture 7: Wrong row-count estimations for correlated columns

Look at the huge discrepancy between the estimated and actual number of rows. The optimizer expects the result set to be of about 16000 rows large, and from this point of view it's perfectly understandable that a clustered index scan has been chosen.

So, where's the reason for this strange behavior? We have to inspect the statistics in order to give an answer to this question. If you open the "Statistics" folder inside Object Explorer, the first thing you may notice is an automatically created statistics for the non-indexed column **DailyRate**. You can then easily calculate the expected number of rows for the condition 'DailyRate<90' from the histogram for this statistics (see Picture 8).

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS
86.00	653.5186	174.1223
87.00	653.5186	174.1223
88.00	653.5186	174.1223
91.00	653.5186	33.91992
92.00	653.5186	171.1072

Picture 8: Excerpt from the statistics for the DailyRate column

Just sum up the values for RANGE_ROWS and EQ_ROWS where RANGE_HI_KEY < 90, and you get the row-count for 'DailyRate < 90'. Actually, 'DailyRate=89' would need some special treatment, since this value isn't contained as a distinct histogram step. Therefore the calculated value won't be 100% precisely, but it gives you an idea of the way that the optimizer uses the histogram. Of course, we may simply calculate the number of rows by executing the following query:

```
select count(*) from RentalCar where DailyRate < 90
```

Look at the estimated row-count in the execution plan, which is 62949.8 in my case. (Your numbers may differ, since we've added random values for the daily rate.)

Now we do the same thing with our statistics for the indexed column **CarType** (see Picture 9 for the histogram).

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS
Compact	0	24472.51
FullSize	0	24606.24
Luxory	0	25378.9
Medium	0	25542.35

Picture 9: Histogram for the CarType column

Apparently, there are 25378.9 estimated luxury cars in our table. (Note, what a funny thing statistics are! They use two decimal places for exposing estimated values.)

And here's how the optimizer calculates the cardinality estimation for our SELECT statement:

With 62949.8 estimated rows for DailyRate<90, and 100000 total table rows, the density for this filter computes to 62949.8/100000=0.629498.

The same calculation is being done for the second filter condition CarType='Luxory'. This time, the estimated density is 25378.9/100000=0.253789.

To determine the total number of rows for the overall filter condition, both density values are simply multiplied. Doing this, we receive 0.629498*0.253789=0.15976. The optimizer takes this value combined with the number of table rows to determine the estimated row-count, which finally calculates to 0.15976*100000=15976. That's exactly the value that Picture 7 shows.

To understand the problem, you need to recall some school mathematics. By just multiplying the two separate computed densities, the optimizer assumes that values in both participating columns are independent of each other, which apparently isn't the case. Column values in one column are not equally distributed for all column values in the second column. Mathematically it's not correct therefore, just to multiply densities (probabilities) for both columns. It's just wrong to conclude that the total density for 'DailyRate<90' for all values of the **CarType** column will be equally rational for a value of "CarType='Luxory'". At the moment, the optimizer does not take column dependencies like this into consideration, but we have some options to deal with this kind of problems, as you'll see in the following solutions section.

Solution(s):

1) Using an index hint

Of course, we know that the execution plan is unsatisfactory, and this is easy to prove if we force the optimizer to use the existing index on (CarType, DailyRate). We can do this straightforward by adding a query hint like this:

```
select * from RentalCar with (index=Ix_RentalCar_CarType_DailyRate)
 where CarType='Luxory'
   and DailyRate < 90
```

The execution plan will reveal an index seek now. Note that the estimated row-count has not changed however. Since the execution plan is generated *before* the query executes, cardinality estimations are the same in both experiments.

Nevertheless, the number of necessary reads (monitored with SET STATISTICS IO ON) has been drastically reduced. In my environment I had 5578 logical reads for the clustered index scan and only 4 logical reads, if an index seek is used. The improvement-factor computes to about 1400!

Of course this solution also exposes a great drawback. Despite of the fact that an index hint is apparently helpful *in this case*, you should generally avoid using index hints (or query hints in general) whenever possible. Index hints reduce the optimizer's potential options, and may lead to unsatisfactory execution plans when the table data have been changed in a way that the index is no longer useful. Even worse, there's a chance that the query becomes totally invalid, in case the index has been deleted or renamed.

There are superior solutions, which are presented in the next paragraphs.

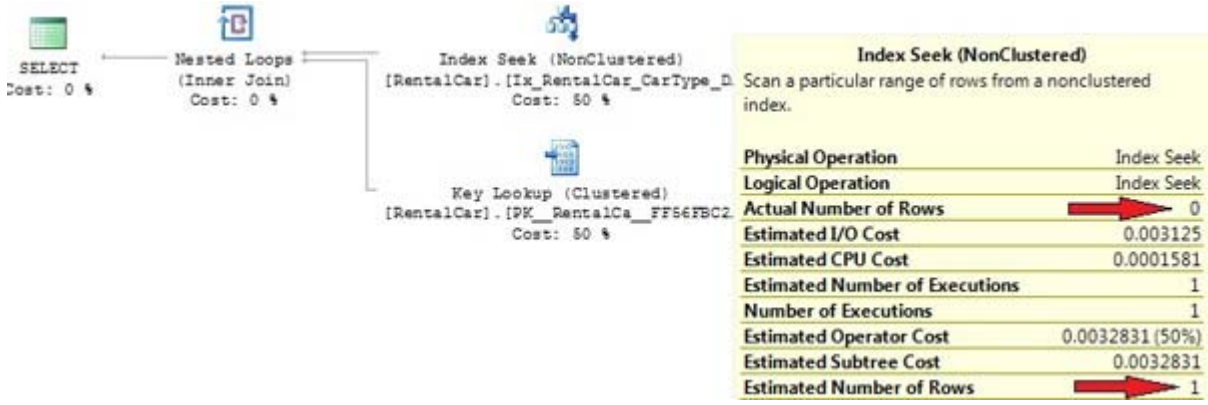
2) Using filtered indexes

As of SQL Server 2008, we have the opportunity of working with filtered indexes. That's quite perfect for our query. Since we only have 4 distinct values for the CarType column, we may create four different indexes, one for every car type. A special index for CarType='Luxory' will look as follows:

```
create nonclustered index Ix_RentalCar_LuxoryCar_DailyRate
    on RentalCar(DailyRate)
 where CarType='Luxory'
```

For the remaining three values of CarType we'd create identical indexes with adjusted filter conditions.

We end up with four indexes and also four statistics that are pretty much adjusted to our queries. The optimizer can take advantage of these tailor-made indexes. You see perfect cardinality estimations along with a flawless execution plan in Picture 10 below.



Picture 10: Improved execution plan with filtered indexes

Filtered indexes provide a very elegant solution to the correlated-columns problem for no more than two participating columns, whenever there are only a handful of values for one contributing column. In case you need to incorporate more than two columns and/or your column values vary widely and unforeseen, you might experience difficulties in detecting proper filter conditions. Also, you should ensure that filter conditions don't overlap, since this may puzzle the optimizer.

3) Using filtered statistics

I'd like you to reflect on the last section. By creating a filtered index, the optimizer was able to create an optimal plan. Finally it could do so, because we provided it with much improved cardinality estimations. But wait! Cardinality estimations are not gained from indexes. Actually, it's the index-linked statistics that accounts for the amended appraisal.

So, why not just leave the original index in place and create filtered statistics instead? Indeed that will do it. We can remove the filtered index and create a filtered statistics as an alternative:

```
drop index Ix_RentalCar_LuxoryCar_DailyRate on RentalCar
```

```
go
create statistics sf on RentalCar(DailyRate)
 where CarType='Luxory'
```

If we do the same for the other three values of the CarType column, this again will result in four different histograms, one for each distinct value of this column. Execute our test query again, and you'll see the same execution plan as the one displayed in Picture 10.

Please be prepared to face identical obstacles, like the ones than were already mentioned in the last section concerning filtered indexes: you may experience difficulties in detecting proper filter conditions.

4) Using covering indexes

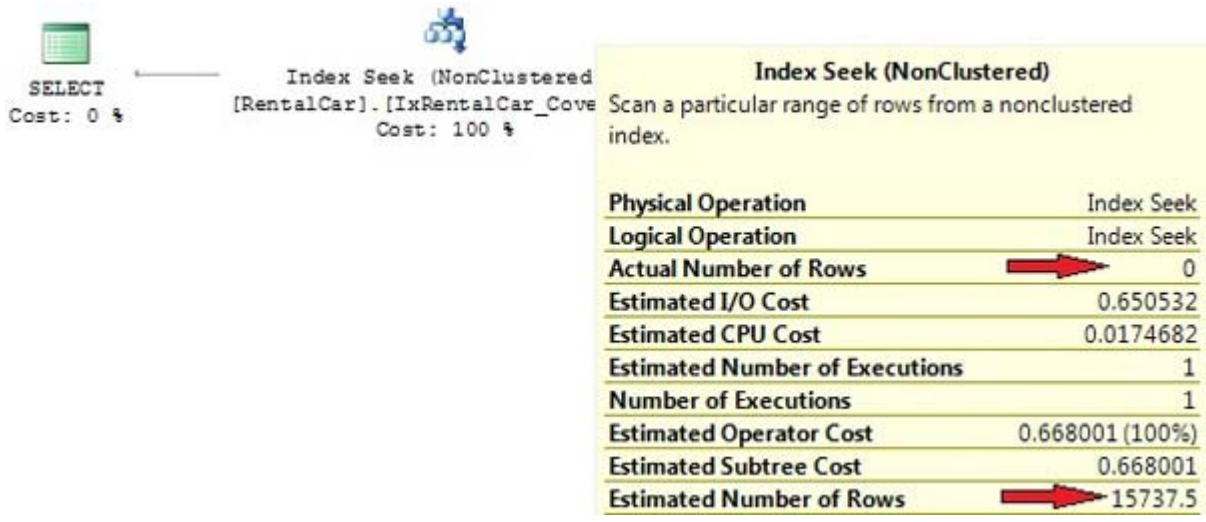
I've already mentioned that you might face situations where optimal filtered indexes, or statistics are hard to determine. It'll not always be that easy as in our more educational example. If, you're unable to uncover subtle filter expressions, there's another option, you will want to take into consideration: covering indexes.

If the optimizer finds an index where all the required columns and rows can be retrieved from the index without referencing the related table, an index that covers the query, then that index will be used, regardless of cardinality estimations.

Let's construct an index like that. At first, we remove the filtered statistics to ensure, the optimizer will not rely on this. Afterwards we construct a covering index that the optimizer can benefit from:

```
drop statistics RentalCar.sf
go
create index IxRentalCar_Covering
  on RentalCar(CarType, DailyRate)
 include (MoreColumns)
```

If you execute our test query again, you'll see that the covering index is utilized. Picture 11 displays the execution plan.



Picture 11: Index Seek with covering index

(Just in case you wonder why we didn't include the **RentalCarID** column too, the reason is that we've created the clustered index on this column, so it will be included in every nonclustered index anyway.)

Note however that the row-count estimation is still far away from reality! Row-count appraisals are unimportant here. The index is used because our query uses a search argument on the first index column *and* the index covers the query. One could actually say that the execution plan is just coincidentally efficient.

Please bear in mind that there's also one special form of a covering index. I'm talking about the clustered index that every table can (actually should) have. If your query is designed in a way that searches on the leading index columns of the clustered index are performed, searching the clustered index will be favorable, irrespective of row-count estimations.

Updates of statistics come at a cost

Problem:

Actually this is not a problem, but just a fact that you have to take into account when planning database maintenance tasks: It's best to avoid automatic updates of statistics for that 5 million rows table during normal OLTP operations.

Solution(s):

Again, the solution is to supplement automatic updates with manual updates. You should add manual updates of statistics to your database maintenance task-list. Keep in mind, though, that updates of statistics will also invalidate any cached query plans and cause them to be re-compiled, so you don't want to update too much. If you still face the problem of costly automatic updates during normal operation, you may want to switch to asynchronous updates.

Memory allocation problems

Every query needs a certain amount of memory for execution. The amount of memory required is calculated and requested by the optimizer, where the optimizer takes estimated row-counts and also estimated row sizes into account. If either of the two is incorrect, the optimizer may over- or underestimate the required memory. This is a problem regarding sorts and hash joins. Memory allocation problems may be divided into the following two issues.

An Overestimate of memory requirements

Problem:

With row-count estimations too large, the allocated memory will be too high. That's simply a waste of memory, since parts of the allocated memory will never get used during query execution. If the system already experiences memory allocation contention, this may also lead to increased wait times.

Solution

Of course the best method is adjusting statistics or re-writing the SQL code. If that's not possible, you may use query hints (like OPTIMIZE FOR) for providing a better idea of cardinality estimations to the optimizer.

An Underestimate of memory requirements

Problem

That case is even worse! If the requested memory is underestimated, it isn't possible to acquire additional memory on the fly, that is during query execution. As a result, the query will swap intermediate results to **tempdb**, causing a performance degrade by factor 8-10!

Solution:

SQL Server Profiler knows the events *Errors and Warnings/Sort Warnings* and *Errors and Warnings/Hash Warnings* that will fire, every time a sort operation or hash join makes use of **tempdb**. Whenever you see this happen, it's probably worth further investigations. If you suspect underestimated row-counts as the cause for **tempdb** swapping, see if updating statistics or SQL review and edit will help. If not, consider query hints for solving the issue.

Additionally, you may also increase the minimum amount of memory that a query allocates by adjusting the configuration option *min memory per query (KB)*. The default is 1 MB. Please make sure that there's no other solution before you opt for this. Every time you change configuration options, it's good knowing exactly what you're doing. Modifying configuration options should generally be your last choice in solving problems.

Best practices

I've extracted all of the advices given so far and put them into the following best practices list. You may consider this as a special kind of summary.

- Be lazy. If there are automatic processes for creating and updating statistics, make use of them. Let SQL Server do the majority of the work. The option of automatically creating and updating statistics should work out fine in almost all cases.
- If you experience problems with query performance, this is very often caused by stale or poor quality statistics. In many situations you won't find the time for deeper analysis, so simply perform an update statistics first. Be sure not to do this update for all existing tables, but only for the participating tables or indexes of poor performing queries. If this doesn't help, you may be the point at which you consider updating distinct statistics with full scan. Watch out for differences between actual and estimated row-counts in the actual execution plan. The optimizer should estimate, not guess! If you see considerable differences, this is very often an indicator for non-conforming statistics – or poor TSQL code, of course.
- Use the built-in automatic mechanisms, but don't rely solely on these. This is particularly true for updates. Support automatic updates through additional manual ones, if necessary.
- Rebuild fragmented indexes when necessary. This will also update existing index-linked statistics with full scan. Be sure *not* to update those index-linked statistics again right after an index rebuild has been performed. Not only is this unnecessary, it will even downgrade those statistics' quality if the default sample rate is applied.
- Carefully inspect, if your queries can make use of multi-column statistics. If so, create them manually. You may utilize the Database Engine Tuning Advisor (DTA) for regarding analysis.
- Use filtered statistics when you need more that just 200 histogram entries. When introducing filtered statistics, you will want to perform manual updates; otherwise your filtered statistics might get stale very soon.
- Don't create more than one statistics on the same column unless, of course, they are filtered statistics. SQL Server will not prevent you from generating multiple statistics for one column: Likewise, you can have several identical indexes on the same column. Not only will this increase the maintenance effort, it will also increase the optimizer's workload. Moreover, since the optimizer will always use one particular statistics for estimations of cardinality, it has to choose one statistics out of a set. It will do this by rating statistics and selecting the "best" one. This could be the one with the newest last update date, or perhaps the one with a larger sample size.
- And last but not least: improve your TSQL Code. Avoid using local variables in TSQL Scripts or overwriting parameter values inside stored procedures. Don't use expressions in search arguments, joins, or comparisons.

Bibliography

[1] *Ben-Gan, Itzik*: Inside Microsoft SQL Server 2005 T-SQL Querying. Microsoft Press, 2006

[2] Statistics Used by the Query Optimizer in Microsoft SQL Server 2005

[3] Statistics Used by the Query Optimizer in Microsoft SQL Server 2008

About Red Gate

You know those annoying jobs that spoil your day whenever they come up?

Writing out scripts to update your production database, or trawling through code to see why it's running so slow.

Red Gate makes tools to fix those problems for you. Many of our tools are now industry standards. In fact, at the last count, we had over 650,000 users.

But we try to go beyond that. We want to support you and the rest of the SQL Server and .NET communities in any way we can.

First, we publish a library of free books on .NET and SQL Server. You're reading one of them now. You can get dozens more from www.red-gate.com/books

Second, we commission and edit rigorously accurate articles from experts on the front line of application and database development. We publish them in our online journal Simple Talk, which is read by millions of technology professionals each year.

On SQL Server Central, we host the largest SQL Server community in the world. As well as lively forums, it puts out a daily dose of distilled SQL Server know-how through its newsletter, which now has nearly a million subscribers (and counting).

Third, we organize and sponsor events (about 50,000 of you came to them last year), including SQL in the City, a free event for SQL Server users in the US and Europe.

So, if you want more free books and articles, or to get sponsorship, or to try some tools that make your life easier, then head over to www.red-gate.com