IDERA® | SQL Server Whitepaper

SQL SERVER STATISTICS PRIMER

BY ROBERT DAVIS

TABLE OF CONTENTS

INTRODUCTION

This white paper is an introductory guide for DBAs about SQL Server statistics. It covers how to use them, how to maintain them, and how they affect performance.

Statistics, or “stats,” are fundamental components of SQL Server performance, but vastly under-appreciated and misunderstood. They are at the core of query optimization and can have tremendous effect on query plan selection. The query optimizer uses statistics to estimate I/O costs and memory grants. Poor statistics, whether they are skewed or incorrect, can cause massive performance problems when they lead to selection of a bad plan. Regardless, having good statistics is still no guarantee that the plan will be optimal for the query.

Statistics are mostly self-maintaining, though they can require a little care and feeding when they cause poor plan selection. However, caution is warranted when deciding to do regular maintenance on statistics. Sometimes doing maintenance on stats when it is not warranted can cause more harm than good. The key is to make sure you are addressing the problem, and not just the symptom.

Now, let’s get to know statistics. We will take a look at what statistics are, how to view the information they provide, how they are used, and how to maintain them.

WHAT ARE STATISTICS?

Simply put, statistics are descriptions of the data in the column or columns of a table or index. Statistics may cover either a single column or multiple columns in a specific order. There are three types of statistics, and they are categorized by the way they are created. They are:

Index stats: These are created in the background on the key columns of an index

- Created synchronously as part of the index creation process via CREATE INDEX
- May be created as part of a constraint creation that is enforced by creating an index such as a primary key constraint or a unique constraint

Column stats are created via an explicit CREATE STATISTICS statement on the specified columns

Auto-stats: are created during the query optimization process to generate statistics on columns in the criteria of a query, such as the WHERE clause of a JOIN statement

- Can be disabled by turning off the auto-creation of stats at the database level
- ALTER DATABASE [<Database>] SET AUTO_CREATE_STATISTICS ON|OFF;

You can identify how any specific statistics were created by querying the sys.stats system catalog. There are two columns that indicate how a particular set of stats was created. The auto_created column indicates whether the statistics are auto-stats or not. The user_created column indicates that the stats are user-created column stats. If both columns auto_created and user_created are zero, it indicates that the statistics are index stats created via an explicit CREATE INDEX statement.

Generally, you can tell how stats were created by looking at their names. The names for auto-stats will start with "_WA_Sys_." Index stats share a name with the index with which they are associated. User-created column stats do not match the name of an index and should not be named with "_WA_Sys_." (though if you really want to, there's nothing that will prevent it).

You can use the below query to see the stats in the current database and see how the auto_created and user_created columns, as well as the name formats, correlated to the type of statistics.

```
Select TableName = OBJECT_NAME(object_id),
       StatsName = name,
       StatsType = Case
              When auto_created = 1 Then 'Auto-created stats'
              When user_created = 1 Then 'User-created column stats'
              -- auto_created = 0 And auto_created = 0
              Else 'Index stats'
              End
From sys.stats;
```

In addition to the name, you can identify index stats by joining sys.stats to sys.indexes on object_id and on sys.stats.stats_id to sys.indexes.index_id. For example, if I wanted to see all of the index stats for the dbo.DimAccount table in the AdventureWorksDW2012 database, the query would look like the one below.

```
Use AdventureWorksDW2012;

Select StatsName = S.name,
       IndexName = I.name
From sys.stats S
Inner Join sys.indexes I
       On I.object_id = S.object_id
              And I.index_id = S.stats_id
Where S.object_id = OBJECT_ID('dbo.DimAccount');
```

STATSNAME	INDEXNAME
PK_DimAccount	ap-south-1
AK_DimAccount_AccountCodeAlternateKey	ap-northeast-3

You can see from the results of the above query that index stats are automatically given the same name as the index.

A better way to understand how stats are created is to create some statistics of our own. We'll start off by creating a new table by doing a SELECT INTO to create a copy of the sys. databases view. Then, we will combine the two queries we used earlier to see that there are no stats for this new table.

```
-- Create a table with some data in it
Select * Into dbo.AllDatabases
From sys.databases;
Go

-- Are there are stats yet?
Select StatsName = S.name,
       IndexName = I.name,
       StatsType = Case
              When auto_created = 1 Then 'Auto-created stats'
              When user_created = 1 Then 'User-created column stats'
              -- auto_created = 0 And auto_created = 0
              Else 'Index stats'
              End
From sys.stats S
Left Join sys.indexes I
       On I.object_id = S.object_id
              And I.index_id = S.stats_id
Where S.object_id = OBJECT_ID('dbo.AllDatabases');
```

As we said, there are no stats on the table as of yet. Let's now create a primary key constraint on the table, which in turn creates a primary key index. Since we are creating an index, there will also be statistics created with the same name.

```
Alter Table dbo.AllDatabases
       Add Constraint PK_AllDatabases
              Primary Key (database_id);
```

Then, using the same SELECT query as above, we see that there is now one set of statistics with the same name as the index and constraint.

STATSNAME	INDEXNAME	STATSTYPE
PK_AllDatabases	PK_AllDatabases	Index stats

To generate some auto-created statistics, we can execute a query with some criteria in it on a column that is not indexed. In this case, any column other than database_id will do. After running the query, we will use the same SELECT query as before to see the stats that now exist on the table.

```
Select *
From dbo.AllDatabases
Where compatibility_level > 90;
```

STATSNAME	INDEXNAME	STATSTYPE
PK_AllDatabases	PK_AllDatabases	Index stats
_WA_Sys_00000006_031C6FA4	NULL	Auto-created stats

To create the final type of statistics, we can manually create statistics on a specific column. We can use the CREATE STATISTICS statement--quite similarly to how we would create an index on the recovery_model column. The key difference here is that no index is created when we create just statistics.

```
Create Statistics st_AllDatabases_RecoveryModel
       On dbo.AllDatabases(recovery_model);
```

STATSNAME	INDEXNAME	STATSTYPE
PK_AllDatabases	PK_AllDatabases	Index stats
_WA_Sys_00000006_031C6FA4	NULL	Auto-created stats
st_AllDatabases_RecoveryModel	NULL	User-created column stats

If we query the table now with criteria on the recovery_model column, stats already exist, but there is no index for that column. The query optimizer will use the stats I created to estimate the number of rows that will meet the criteria and estimate the cost of the query. Since there is no matching index, it will do a scan on the clustered index for the actual execution. Even in cases where there is an existing index, the query optimizer can use the statistics to generate the query plan—even though the index may not get used in the ensuing query plan.

This is important to understand because it is possible for performance issues to arise when removing an unused index. If you remove an index that is not getting used in the queries, and the query suddenly starts getting a poor query plan, I recommend simply adding stats for the index columns and see if this resolves the performance issue.

VIEWING STATISTICS DATA

Statistics are not something you will need to investigate on a daily basis. You can get great insight into what is in statistics using the DBCC SHOW_STATISTICS command. The command has two required parameters (table name and statistics name, index name, or column name) and several optional parameters used with the WITH keyword. Even though the second parameter allows you to specify a column name, if there are no statistics on the column, it will return an error. Statistics must already exist. They will not be created or calculated on the fly by the DBCC SHOW_STATISTICS command.

There are three sections to the output of DBCC SHOW_STATISTICS; the stats header, the density vector, and the histogram. You can view all three sections at once, or you can specify which section you want to see.

There is a fourth option for DBCC SHOW_STATISTICS introduced in SQL Server 2008, which can only be seen by specifying the option to see the stats stream. Stats stream is not an introductory topic, so for the scope of this white paper, we will focus on the first three options: STAT_HEADER, DENSITY_VECTOR, and HISTOGRAM.

STAT_HEADER

The stats header data shows just the header information for the statistics. Some of the important things the header tells you is the date the stats were last updated, the number of total rows and number of rows sampled in the stats, the number of steps in the statistics, filter expression if filtered, and the number of unfiltered rows. To view the stats header for some index in the dbo.DimAccount table in the AdventureWorksDW2012 database, I would pass in the table name and statistics name to the DBCC SHOW_STATISTICS command along with the STAT_HEADER option.

```
DBCC SHOW_STATISTICS('dbo.DimAccount',
       AK_DimAccount_AccountCodeAlternateKey)
       With STAT_HEADER;
```

The transposed results below demonstrate the information available in the stats header:

COLUMN	VALUE
Name	AK_DimAccount_AccountCodeAlternateKey
Updated	99
Rows	99
Rows Sampled	99
Steps	75
Density	1
Average key length	8
String Index	NO
Filter Expression	NULL
Unfiltered Rows	99

- “Updated” is the date that the stats were created or last updated.
- “Rows” is the total number of rows represented by the stats.
- “Rows Sampled” is the number of rows that were sampled to create the stats. If “Rows” and “Rows Sampled” are equal, it means that the statistics were created or updated with a full scan of all rows.
- “Steps” indicates how many steps are used in the statistics to describe the data. This will be described further when we view the histogram.
- “Density” is often called selectivity, and describes how unique the data is. It is 1 divided by the number of unique values in the leading column only. “Density” is no longer used by the query optimizer and is only provided for backwards compatibility.
- “Average key length” is the average length of the keys.
- “String Index” is a Yes or No value that indicates whether the statistics contain special stats for string values to improve cardinality estimates for LIKE expressions.
- “Filter Expression” is the expression provided if either the statistics or index are filtered.
- “Unfiltered Rows” are the total number of rows in the table. If filter expression is NULL, the unfiltered rows equal the total rows.

DENSITY_VECTOR

Density vector provides density for the column combinations in the statistics columns. The column combinations always start with the leading column and then add each column in order. For example, if the stats were for three columns named Column1, Column2, and Column3, the density vector would provide density information for Column1; Column1 and Column2; and Column1, Column2, and Column3.

Density vector provides information about the selectivity of the combination of stats columns, but since SQL Server 2008, the density information is no longer used by the query optimizer and provides very little useful information.

You can view the density vector for the AK_DimAccount_AccountCodeAlternateKey stats in dbo. DimAccount in AdventureWorksDW2012 using the DBCC SHOW_STATISTICS command with the DENSITY_VECTOR option.

```
DBCC SHOW_STATISTICS('dbo.DimAccount',
       AK_DimAccount_AccountCodeAlternateKey)
       With DENSITY_VECTOR;
```

ALL DENSITY	AVERAGE LENGTH	COLUMNS
0.01010101	4	AccountCodeAlternateKey
0.01010101	8	AccountCodeAlternateKey, AccountKey

HISTOGRAM

The histogram is probably the most useful set of output for the statistics. The histogram shows the steps referred to in the stats header. The histogram is limited to a maximum of 200 steps to describe the entire dataset. For small to medium tables, this is usually more than sufficient to describe the data. The larger the table, the more difficult it is to describe the data accurately. The bigger the gaps are between the steps, the less accurate they tend to be.

The entire output for the histogram can be long, so for this example, I'm going to use a much smaller set of statistics. I'm going to look at the histogram for some auto-created stats on the dbo.DimProduct table on the EnglishProductName column.

```
DBCC SHOW_STATISTICS('dbo.DimProduct',
       EnglishProductName)
       With HISTOGRAM;
```

The results for this histogram are quite lengthy. For the sake of brevity, here are the first ten rows of output, along with an explanation of the columns.

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_ RANGE_ROWS	AVG_ RANGE_ROWS
Adjustable Race	0	1	0	1
AWC Logo Cap	1	3	1	1
Bearing Ball	1	1	1	1
Cable Lock	2	1	2	1
Chainring Bolts	3	1	3	1
Classic Vest, L	1	1	1	1
Classic Vest, S	1	1	1	1
Decal 1	3	1	3	1
Down Tube	1	1	1	1
External Lock Washer 2	1	1	1	1

- The “RANGE_HI_KEY” column is the highest value in the current step.
- “RANGE_ROWS” is the estimated rows with a value in the current step not counting the “RANGE_HI_KEY” value. 0 means that every row in the step is equal to the highest value.
- “EQ_ROWS” is the estimated rows with a value equal to the “RANGE_HI_KEY.” “RANGE_ROWS” plus “EQ_ROWS” equal the total estimated number of rows in the step.
- “DISTINCT_RANGE_ROWS” is the estimated number of distinct values in the step, not counting the “RANGE_HI_KEY” value.
- “AVG_RANGE_ROWS” is the estimated number of rows with a duplicate value in the step, not counting the “RANGE_HI_KEY” value. If “DISTINCT_RANGE_ROWS” is greater than zero, it is “RANGE_ROWS” divided by “DISTINCT_RANGE_ROWS.”

If we wanted to know what the stats tell us about the product named Crown Race, we would see that the value would fall between Classic Vest, S and Decal 1. Since “RANGE_HI_KEY” represents the highest value in the step, Crown Race would be part of step eight, Decal 1.

“RANGE_ROWS” is 3, which tells us that there is an estimated 3 rows in the step that is not equal to Decal 1. “DISTINCT_RANGE_ROWS” is also 3, which means there are 3 distinct values in the step, not equal to Decal 1. Since distinct range rows and range rows are the same number, it tells us that all values in the step have an estimated row count of 1.

This is a simple example of the logic that the query optimizer goes through to estimate the row counts for the cost of a query. This was an easy example because the values were all unique in the step we looked at. It gets a lot more complicated when the steps contain more rows and more values, especially if one or more value is greatly skewed from the rest of the rows.

MAINTAINING STATISTICS

Fortunately, there is not a lot we need to do to maintain statistics. In fact, I believe that many people do too much statistics maintenance already. SQL Server has a built-in process to update statistics as the data changes reach a certain threshold. The threshold is based on the number of updates to the columns in the statistics. Additionally, when you rebuild an index, the associated statistics are automatically rebuilt with a full scan as well.

One potential issue with auto-updated stats is that it updates with a sampled value, which means it scans a percentage of the rows instead of all rows to generate the statistics. The database engine is sacrificing a little bit of accuracy in favor of processing the statistics update faster. Fewer rows scanned means less work and a faster process time. When auto-update of stats is triggered, it generally means that there is at least one query waiting for the stats to complete the update so it can generate a plan and execute. Speed is of the essence.

As stated above, index rebuilds will rebuild the statistics with a full scan. SQL Server is already doing a full scan for the index rebuild, so there is no reason not to update the statistics at the same time. Short of doing manual filtered statistics, these are the best possible statistics you could want. You should never follow up a rebuild of an index with an UPDATE STATISTICS command. It is just unnecessary work and will only result in statistics of lower quality. This is a common mistake and one for the things you should avoid.

However, index reorganization (defragment) does not update the statistics, because the engine is simply moving blocks of data around—not scanning it. As part of your index maintenance routines, you may want to consider updating the statistics for any indexes

that get reorganized instead of rebuilt if you run into issues with those stats being stale. Additionally, auto-created stats and user-created stats are not tied to an index and thus will never be updated by index maintenance. You should consider updating these stats manually if you encounter issues with the stats being stale.

PARAMETER SNIFFING

When statistics are updated, whether it was automatically triggered or updated manually, it marks all query plans that reference the statistics to be marked for recompilation. If there is considerable data skew that causes statistics to give bad estimations for some data values, it is possible for the query to get a bad plan after the statistics are updated. This is a much less common problem than people think.

A more common problem is the one where the data is skewed from one value to another by a large margin and the optimizer generates a good plan for the first parameter value with which it compiles the plan, but when the plan is reused for other parameter values, it performs poorly. This is often mistakenly thought to be an issue arising from a bad plan due to outdated statistics, as we are conditioned to attempt to update stats as a remediation. As I said before, updating stats causes the plans that reference the statistics to be recompiled and it results in a different plan for the other value.

Typically, this seems to fix it for a while, but the problem always returns. As a result, the DBA usually ends up creating a job to update statistics frequently for a specific table or index. This is actually the hallmark of a problem with parameter sniffing, and is not an issue with statistics. Updating the statistics appears to fix it because it forces the query plan to be recompiled. You could get the same result simply by creating a job that forces a recompile of the plan regularly.

To read more about parameter sniffing and the mistake of updating statistics regularly, see my blog post: "The Misunderstood and Abused UPDATE STATISTICS Command"

UPDATE STATISTICS

The UPDATE STATISTICS command is the command I recommend using to update statistics manually. There is also a system-stored procedure, sp_updatestats, that can be used. The procedure is equivalent to running UPDATE STATISTICS ALL on the tables or database. It blindly performs updates of statistics. It is selective in that it will not update statistics it decides need not be updated, but it does not make any decisions about sampling rates, nor does it understand that some statistics may be more important than others.

I prefer to take a targeted approach and only update the statistics that need additional updates. Updating statistics is a resource-heavy operation. If you have a clear maintenance window with plenty of time to do lots of maintenance, there is no harm in updating statistics regularly—whether they need them or not. If you are, however, running a system that needs to be responsive 24 hours a day, then you should be selective about the extra work you are asking SQL Server to do and only update stats that truly need it.

SUMMARY

Statistics are both complex and simple. Very simply, they are just a description of the data. However, describing hundreds of thousands, or millions or more rows of data using only 200 steps is complex and can lead to problems generating good query plans. We will take a closer look at the problems associated with statistics, how to troubleshoot them, and how to resolve them in a future white paper.

By now, you should understand what statistics are, how SQL Server uses them, and how you can view the statistics to gain a better understanding of the information they contain. You should also have some guidelines on when to perform maintenance on your statistics, and equally important, when to avoid the temptation to update statistics and just leave them alone.

ABOUT THE AUTHOR

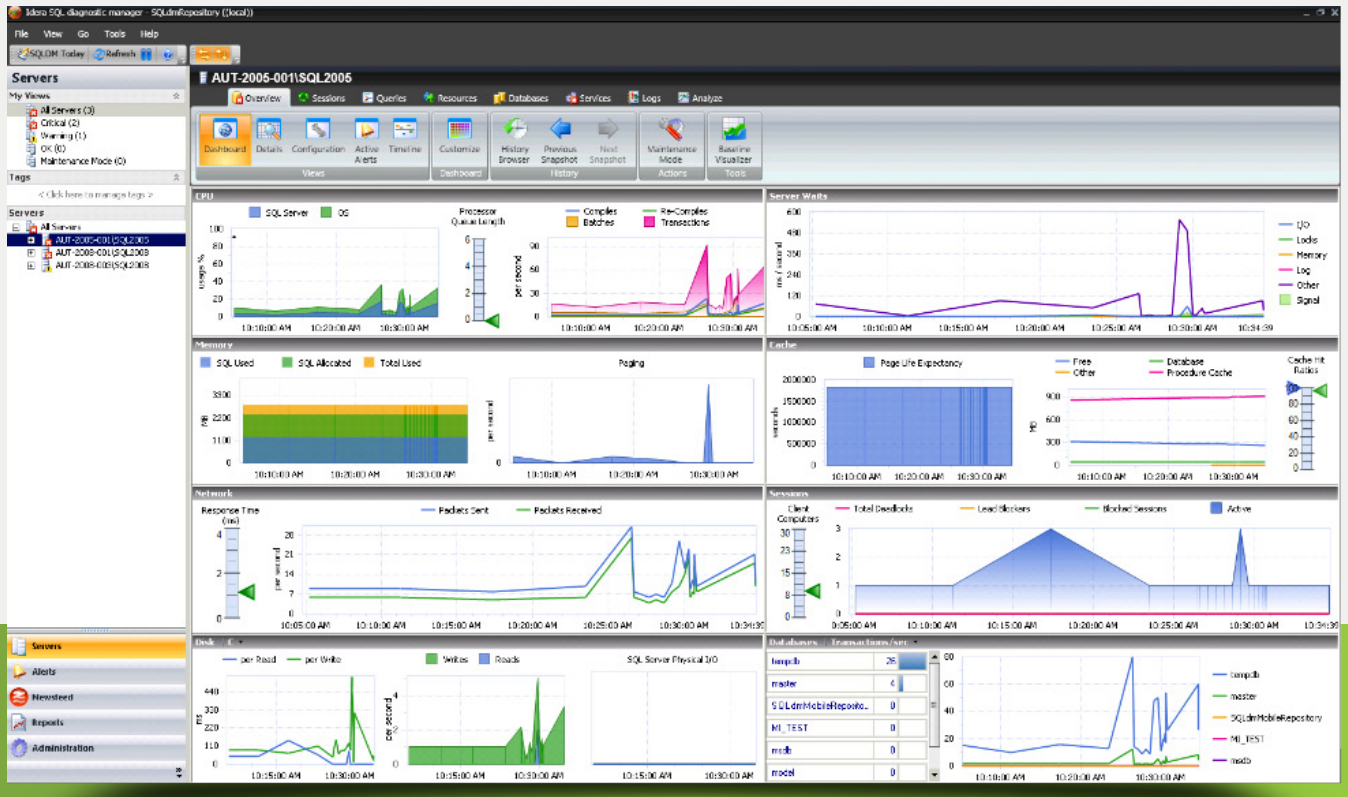
Robert L. Davis was a SQL Server 2008 Certified Master and an experienced Database Administrator, SQL Server evangelist, speaker, writer, and trainer. He worked with SQL Server for more than 15 years. His positions included Principal Database Administrator at Outerwall, Senior Product Consultant and Chief SQL Server Evangelist for Idera Software, Program Manager for the SQL Server Certified Master Program at Microsoft Learning, and Production Database Administrator at Microsoft.

ACHIEVE 24/7 SQL MONITORING

WITH SQL DIAGNOSTIC MANAGER

- Monitor performance for physical, virtual, and cloud environments.
- Monitor queries and query plans to see the causes of blocks and deadlocks.
- Monitor application transactions with SQL Workload Analysis add-on.
- Optimize SQL queries to maximize application performance with SQL Query Tuner add-on.
- View expert recommendations from SQL Doctor to optimize performance.
- Alert predictively with settings to avoid false alerts.
- View summary of top issues and alerts with the web console add-on.

Start for FREE



IDERA

IDERA.com