Posted December 22, 2014

Importance of Statistics and How It Works in SQL Server – Part 1

By Arshad Ali

Introduction

Statistics refers to the statistical information about the distribution of values in one or more columns of a table or an index. The SQL Server Query Optimizer uses this statistical information to estimate the cardinality, or number of rows, in the query result to be returned, which enables the SQL Server Query Optimizer to create a high-quality query execution plan. For example, based on these statistical information SQL Server Query Optimizer might decide whether to use the index seek operator or a more resource-intensive index scan operator in order to provide optimal query performance. In this article series, I am going to talk about statistics in detail.

Basics of Statistics

SQL Server Query Optimizer uses statistics to estimate the distribution of values in one or more columns of a table or index views, and the number of rows (called *cardinality*) to create a high-quality query execution plan. Often statistics are created on a single column but it's not uncommon to create statistics on multiple columns.

Each statistics object contains a histogram displaying the distribution of values of the column (or of the first column in the case of multi-column statistics). Multi-column statistics also contains a correlation of values among the columns (called *densities*), which are derived from the number of distinct rows or the column values.

There are different ways you can view the details of the statistics objects. For example, as shown in the query below, you can use the DBCC SHOW_STATISTICS command. DBCC SHOW_STATISTICS shows the header, histogram, and density vector based on data stored in the statistics object.

```
--This shows header, histogram, and density vector based on data   stored
in the statistics object
DBCC SHOW_STATISTICS("SalesOrderDetail", NCI_SalesOrderDetail_ProductID);
```

Related Articles

- The Format() Function in SQL Server 2012
- Sequence Object in SQL Server 2012
- Contained Database Authentication in SQL Server 2012
- Analysis Services PowerShell Provider (SQLAS) in SQL Server 2012

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1	NCI_SalesOrderDetail_ProductID	Nov 19 2014 9:45PM	1213170	1213170	200	0.001008712	4	NO	NULL	1213170

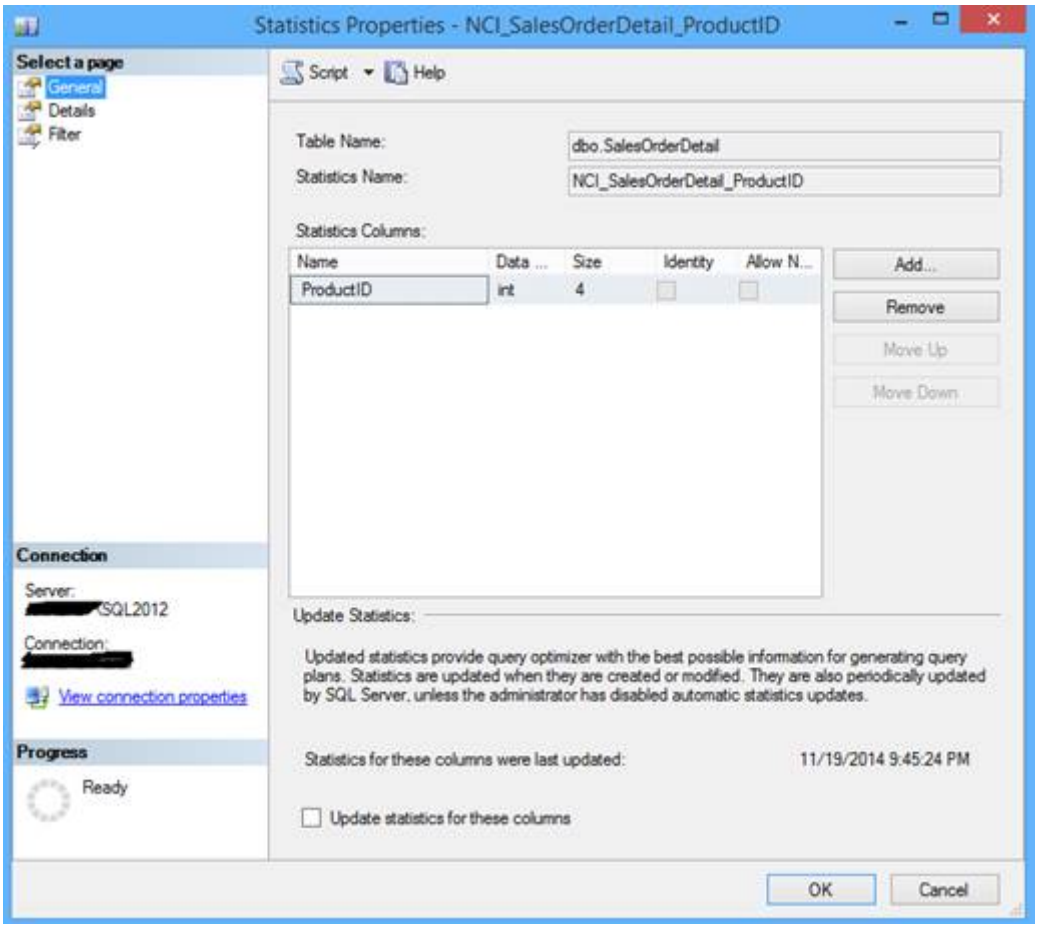
	All density	Average Length	Columns
1	0.003759399	4	ProductID

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	707	0	30830	0	1
2	708	0	30070	0	1
3	710	1880	440	1	1880
4	711	0	30900	0	1
5	712	0	33820	0	1
6	713	0	4290	0	1
7	714	0	12180	0	1
8	715	0	16350	0	1
9	716	0	10760	0	1
10	717	0	2180	0	1
11	718	0	2190	0	1
12	722	440	3920	1	440
13	725	520	3740	1	520
14	726	0	2880	0	1
15	729	480	3750	1	480

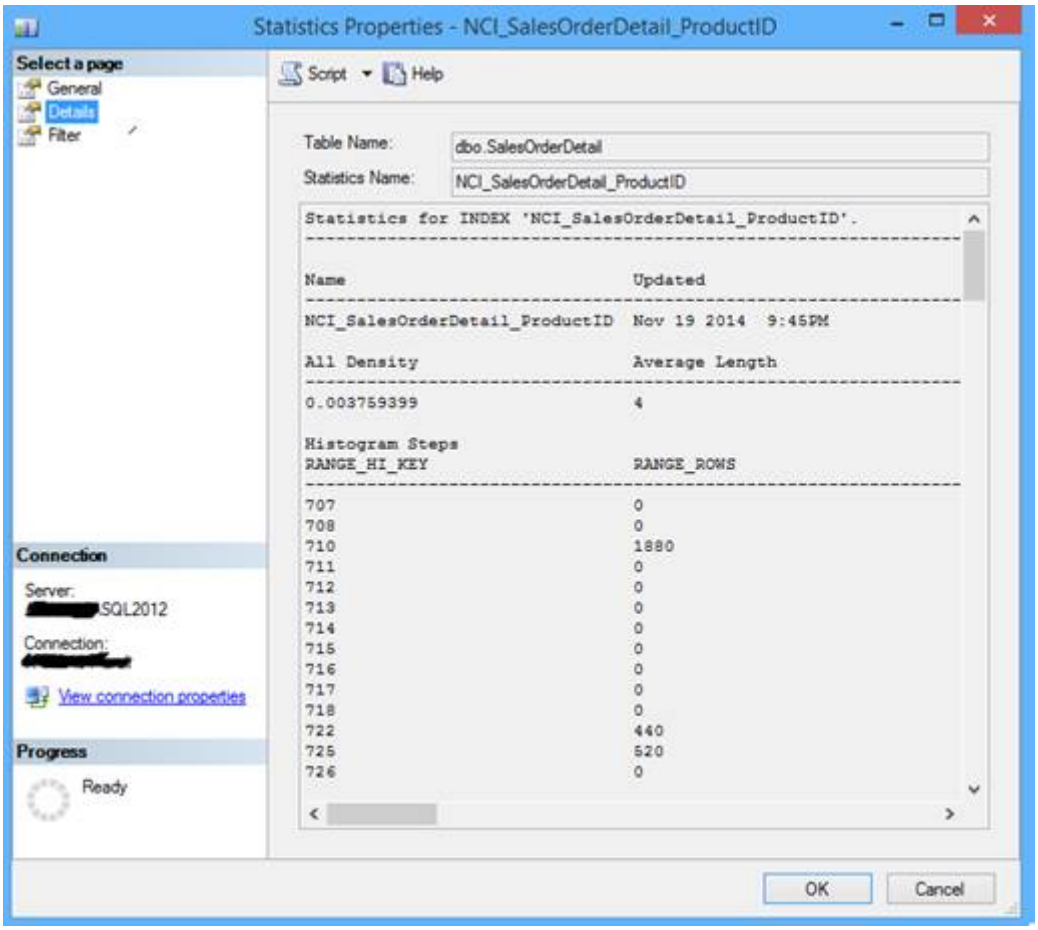
```
--This only shows histogram based on data stored in the  statistics object
DBCC SHOW_STATISTICS("SalesOrderDetail", NCI_SalesOrderDetail_ProductID)
WITH HISTOGRAM;
```

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	707	0	30830	0	1
2	708	0	30070	0	1
3	710	1880	440	1	1880
4	711	0	30900	0	1
5	712	0	33820	0	1
6	713	0	4290	0	1
7	714	0	12180	0	1
8	715	0	16350	0	1
9	716	0	10760	0	1
10	717	0	2180	0	1
11	718	0	2190	0	1
12	722	440	3920	1	440
13	725	520	3740	1	520
14	726	0	2880	0	1
15	729	480	3750	1	480
16	730	0	2880	0	1

You can also view the statistical information by going to the properties page of the statistics object in SQL Server Management Studio as shown below:



Statistics Properties: General



Statistics Properties: Details

The value for all_density (1 / number of distinct values for a column) ranges from 0.0+ to 1, which indicates selectivity (duplicates), 0.0+ being highly selective with less duplicates whereas 1 is less selective with a high number of duplicates; as a rule of thumb, the lesser the better. This actually helps SQL Server Query Optimizer to decide whether to use Index Seek or Index Scan.

The histogram captures the frequency of occurrence for each distinct value in the first key column of the statistics object. SQL Server Query Optimizer creates the histogram by sorting the column values, computing the number of values that match each distinct column value and then aggregating the column values into a maximum of 200 contiguous histogram steps.

Each histogram step includes a range of column values followed by an upper bound column value, which includes all possible column values between boundary values (excluding the boundary values themselves). The lowest of the sorted column values is the upper boundary value for the first histogram step.

- RANGE_HI_KEY - This is also called a key value and represents the upper bound column value for a histogram step.
- RANGE_ROWS - This represents the estimated number of rows whose column value falls within a histogram step, excluding the upper bound.
- DISTINCT_RANGE_ROWS - This represents the estimated number of rows with a distinct column value within a histogram step, excluding the upper bound.

- EQ_ROWS - This represents the estimated number of rows whose column value equals the upper bound of the histogram step.
- AVG_RANGE_ROWS (RANGE_ROWS / DISTINCT_RANGE_ROWS for DISTINCT_RANGE_ROWS > 0) - This represents the average number of rows with duplicate column values within a histogram step, excluding the upper bound.

When to Create or Update Statistics

When to Create Statistics

Often columns being used in JOIN, WHERE, ORDER BY, or GROUP clauses are good candidate to have up-to-date statistics on them. Though the SQL Server Query Optimizer creates single column statistics when the AUTO_CREATE_STATISTICS database property is set to ON or when you create indexes on the table or views (statistics are created on the key columns of the indexes), there might be times when you need to create additional statistics using the CREATE STATISTICS command to capture cardinality, statistical correlations so that it enables the SQL Server Query Optimizer to create improved query plans.

When you find a query predicate containing multiple columns with cross column relationships and dependencies you should create multi-column statistics. These multi-column statistics contain cross-column correlation statistics, often referred to as *densities*, to improve the cardinality estimates when query results depend on data relationships among multiple columns.

When creating multi-column statistics, be sure to put columns in the right order as this impacts the effectiveness of densities for making cardinality estimates. For example, a statistic created on these columns and in order - Name, Age, and Salary. In this case, the statistics object will have densities for the following column prefixes: (Name), (Name, Age), and (Name, Age, Salary). Now if your query uses Name and Salary without using Age, the density is not available for cardinality estimates.

When to Update Statistics

Substantial data change operations (like insert, update, delete, or merge) change the data distribution in the table or indexed view and make the statistics goes stale or out-of-date, as it might not reflect the correct data distribution in a given column or index. SQL Server Query Optimizer identifies these stale statistics before compiling a query and before executing a cached query plan. The identification of stale statistics are done by counting the number of data modifications since the last statistics update and comparing the number of modifications to a threshold as mentioned below.

- A database table with no rows gets a row
- A database table had fewer than 500 rows when statistics was last created or updated and is increased by another 500 or more rows
- A database table had more than 500 rows when statistics was last created or updated and is increased by 500 rows + 20 percent of the number of rows in the table when statistics was last created or updated.

You can find when each statistics object of a database table was updated using the below query:

```
SELECT
        name    AS StatisticsName,
      STATS_DATE(object_id,    stats_id) AS StatisticsUpdatedDate
FROM sys.stats
WHERE OBJECT_NAME(object_id) = 'SalesOrderHeader'
ORDER BY name;
GO
```

	StatisticsName	StatisticsUpdatedDate
1	_WA_Sys_00000008_3C34F16F	2012-03-29 13:52:23.930
2	_WA_Sys_0000000D_3C34F16F	2012-03-29 13:52:33.900
3	_WA_Sys_0000000E_3C34F16F	2012-03-29 13:52:34.037
4	_WA_Sys_0000000F_3C34F16F	2012-03-29 13:52:34.007
5	_WA_Sys_00000010_3C34F16F	2012-03-29 13:52:33.923
6	_WA_Sys_00000011_3C34F16F	2012-03-29 13:52:33.980
7	_WA_Sys_00000013_3C34F16F	2012-03-29 13:52:33.950
8	AK_SalesOrderHeader_rowguid	2012-03-29 13:52:23.853
9	AK_SalesOrderHeader_SalesOrderNumber	2012-03-29 13:52:24.013
10	IX_SalesOrderHeader_CustomerID	2012-03-29 13:52:24.063
11	IX_SalesOrderHeader_SalesPersonID	2012-03-29 13:52:24.097
12	PK_SalesOrderHeader_SalesOrderID	2012-03-29 13:52:20.880

You can also use below query, which uses the dynamic management function (sys.dm_db_stats_properties) to retrieve statistics properties with further details; for example, modification_counter, which shows the total number of modifications on the leading statistics column (the column on which the histogram is built) since the last statistics update.:

```
SELECT
        OBJECT_NAME(stats.object_id) AS TableName,
        stats.name AS StatisticsName,
        stats_properties.last_updated,
        stats_properties.rows_sampled,
        stats_properties.rows,
        stats_properties.unfiltered_rows,
        stats_properties.steps,
        stats_properties.modification_counter
FROM sys.stats stats
OUTER APPLY sys.dm_db_stats_properties(stats.object_id, stats.stats_id) as
stats_properties
WHERE OBJECT_NAME(stats.object_id) = 'SalesOrderHeader'
ORDER BY stats.name;
```

	TableName	StatisticsName	last_updated	rows_sampled	rows	unfiltered_rows	steps	modification_counter
1	SalesOrderHeader	_WA_Sys_00000008_3C34F16F	2012-03-29 13:52:23.9300000	31465	31465	31465	156	0
2	SalesOrderHeader	_WA_Sys_0000000D_3C34F16F	2012-03-29 13:52:33.9000000	31465	31465	31465	10	0
3	SalesOrderHeader	_WA_Sys_0000000E_3C34F16F	2012-03-29 13:52:34.0370000	31465	31465	31465	119	0
4	SalesOrderHeader	_WA_Sys_0000000F_3C34F16F	2012-03-29 13:52:34.0070000	31465	31465	31465	120	0
5	SalesOrderHeader	_WA_Sys_00000010_3C34F16F	2012-03-29 13:52:33.9230000	31465	31465	31465	2	0
6	SalesOrderHeader	_WA_Sys_00000011_3C34F16F	2012-03-29 13:52:33.9800000	31465	31465	31465	158	0
7	SalesOrderHeader	_WA_Sys_00000013_3C34F16F	2012-03-29 13:52:33.9500000	31465	31465	31465	199	0
8	SalesOrderHeader	AK_SalesOrderHeader_rowguid	2012-03-29 13:52:23.8530000	31465	31465	31465	99	0
9	SalesOrderHeader	AK_SalesOrderHeader_SalesOrderNumber	2012-03-29 13:52:24.0130000	31465	31465	31465	156	0
10	SalesOrderHeader	IX_SalesOrderHeader_CustomerID	2012-03-29 13:52:24.0630000	31465	31465	31465	153	0
11	SalesOrderHeader	IX_SalesOrderHeader_SalesPersonID	2012-03-29 13:52:24.0970000	31465	31465	31465	18	0
12	SalesOrderHeader	PK_SalesOrderHeader_SalesOrderID	2012-03-29 13:52:20.8800000	31465	31465	31465	2	0

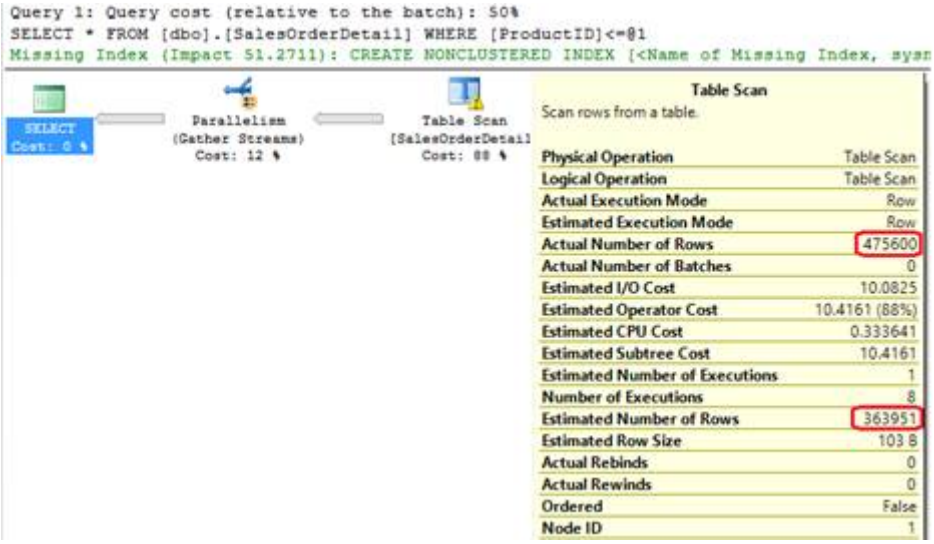
Importance of Statistics in Query Performance

Let's start understanding this with an example. Execute the below query to create a new database and set its AUTO_CREATE_STATISTICS and AUTO_UPDATE_STATISTICS properties to OFF so that automatic statistics creation and updating does not happen on the tables of this database. Next create a table and load data from the SalesOrderDetail table of the AdventureWorks database. I have loaded ten times of the data so that we can see the differences clearly.

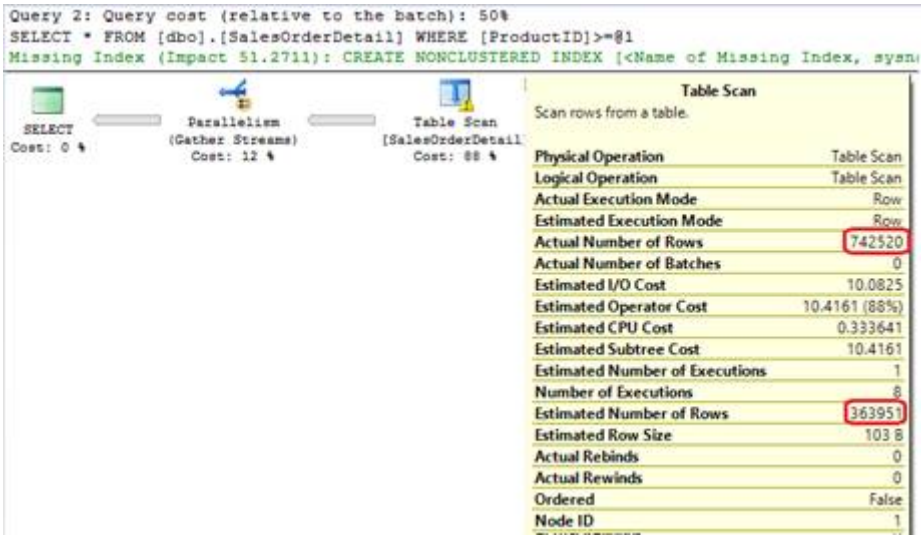
```
CREATE DATABASE [StatisticsTest]
GO

ALTER DATABASE [StatisticsTest] SET   AUTO_CREATE_STATISTICS OFF
ALTER DATABASE [StatisticsTest] SET   AUTO_UPDATE_STATISTICS OFF
ALTER DATABASE [StatisticsTest] SET   AUTO_UPDATE_STATISTICS_ASYNC OFF
GO

USE   [StatisticsTest]
GO
CREATE TABLE [SalesOrderDetail](
        [SalesOrderID]   [int] NOT NULL,
        [SalesOrderDetailID]   [int] NOT NULL,
        [CarrierTrackingNumber]   [nvarchar](25)   NULL,
        [OrderQty]   [smallint] NOT NULL,
        [ProductID]   [int] NOT NULL,
        [SpecialOfferID]   [int] NOT NULL,
        [UnitPrice]   [money] NOT NULL,
        [UnitPriceDiscount]   [money] NOT NULL,
        [LineTotal]   money NOT NULL,
        [rowguid]   [uniqueidentifier] ROWGUIDCOL  NOT NULL,
        [ModifiedDate]   [datetime] NOT NULL,
) ON [PRIMARY]
GO

INSERT INTO [SalesOrderDetail]
SELECT * FROM   [AdventureWorks2008R2].[Sales].[SalesOrderDetail]
GO 10
```

Now let's run these two queries and have a look on their execution plan. Notice the yellow exclamation mark on the "Table Scan" operator; this indicates the missing statistics. Further, notice between the "Actual Number of Rows" and "Estimated Number of Rows" that there is a huge difference. This means, obviously, the execution plan used for query execution was not optimal.

```
select * from [dbo].[SalesOrderDetail]
where   ProductID <= 800;
```

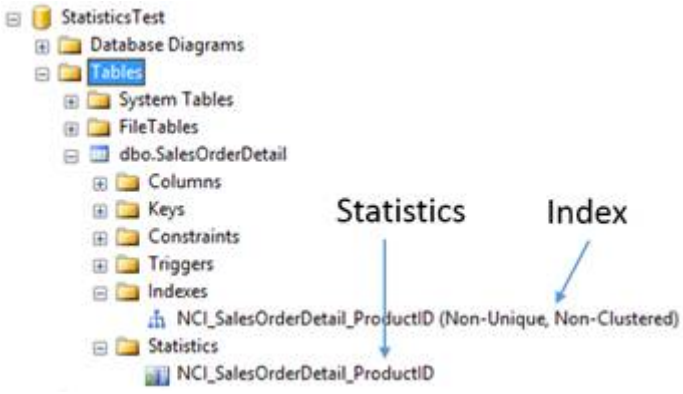



```
select * from [dbo].[SalesOrderDetail]
where   ProductID >= 800;
```



Now let’s create an index on the ProductID column, which will also create statistics on the ProductID with the below query:

```
CREATE NONCLUSTERED INDEX   [NCI_SalesOrderDetail_ProductID]
ON   [dbo].[SalesOrderDetail] ([ProductID])
GO
```



Now if you re-execute the above same queries, you will notice two things. First, there is no warning for missing statistics and second “Actual Number of Rows” and “Estimated Number of Rows” are the same or very close to each other:

```
select * from [dbo].[SalesOrderDetail]
where   ProductID <= 800;
```

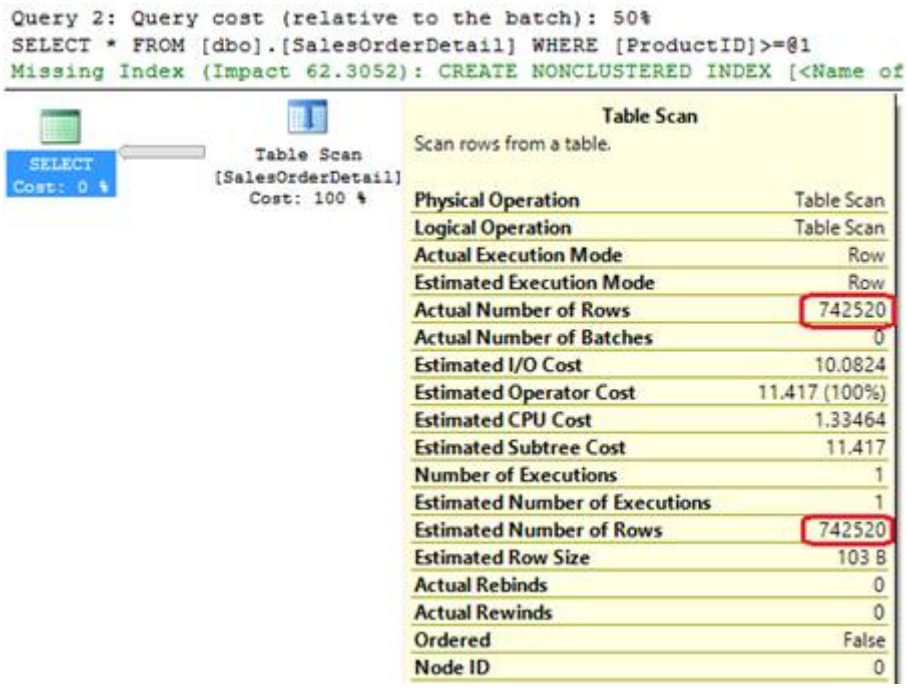
```
Query 1: Query cost (relative to the batch): 50%
SELECT * FROM [dbo].[SalesOrderDetail] WHERE [ProductID]<=@1
Missing Index (Impact 62.3052): CREATE NONCLUSTERED INDEX [<Name of
```

Table Scan

Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Actual Number of Rows	475600
Actual Number of Batches	0
Estimated I/O Cost	10.0824
Estimated Operator Cost	11.417 (100%)
Estimated CPU Cost	1.33464
Estimated Subtree Cost	11.417
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows	475600
Estimated Row Size	103 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0

```
select * from [dbo].[SalesOrderDetail]
where   ProductID >= 800;
```



Conclusion

In this article I discussed the basics of statistics and why it is needed. I also talked about when it needs to be created or updated and how it looks. In the next article of the series, I am going to discuss how performance gets impacted because of missing and out-of-date statistics, how statistics are managed in SQL Server, what are filtered statistics and what is new in SQL Server 2012 related to statistics.

Importance of Statistics and How It Works in SQL Server – Part 2

By Arshad Ali

Introduction

In my last article I discussed the basics of statistics and why it is needed. I also covered when it needed to be created or updated and how it looks. In this article I will talk about how performance gets impacted because of missing and out-of-date statistics, how statistics are managed in SQL Server, what are filtered statistics and what is new in SQL Server 2012 related to statistics.

When the Query Performance Goes for a Toss

As a DBA, have you ever come across a situation where your users start reporting that queries are running slowly? If a user is running a new query with a different set of columns than before, you might have to look for index strategies. But if a user is running the same old query, which ran in an acceptable amount of time earlier but is now taking significantly longer, this might be because of stale or out-of-date statistics on the columns of the table being used in the query. There are certain operations after which you need to ensure statistics are up-to-date for the predictable query response time, for example, when you are performing some operations that change distribution of data significantly, such as truncating a table or performing a bulk insert of a large percentage of the rows or delete a large number of rows etc.

Though if you have set the AUTO_UPDATE_STATISTICS database property to ON, SQL Server Query Optimizer routinely updates statistics when it finds out-of-date statistics being used in the query, but for predictable response time for your user queries its always better to update statistics as part of the bulk data operations itself.

When you see performance issues with your queries, just check if there is any warning for missing statistics or if the “Actual Number of Rows” and “Estimated Number of Rows” have huge differences (this happens because of the out-of-date statistics), in that case you need to update statistics accordingly.

Related Articles

- The Format() Function in SQL Server 2012
- Sequence Object in SQL Server 2012
- Contained Database Authentication in SQL Server 2012
- Analysis Services PowerShell Provider (SQLAS) in SQL Server 2012

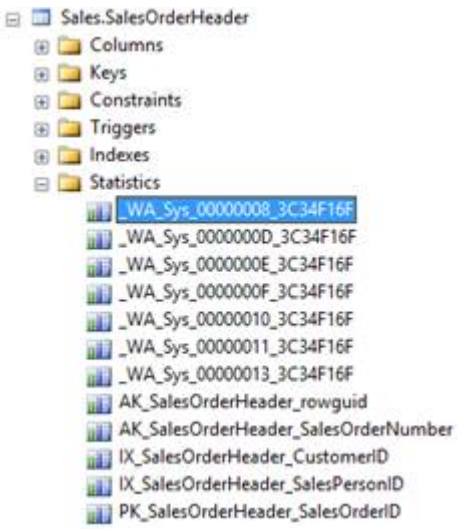
Please note, as part of the maintenance activities if you are performing index rebuild (either using ALTER INDEX REBUILD or DBCC DBREINDEX commands) SQL Server Query Optimizer updates statistics on the respective columns with the index rebuild operations and

you don't need to do it separately. This is applicable only if you are re-building the indexes and does not apply if you are re-organizing indexes.

Statistics Management in SQL Server

There are different ways statistics are created and maintained in SQL Server:

- **By SQL Server engine itself** – There are some database level properties that determine the automatic creation and updating statistics whenever there is a need. For example,
  - AUTO_CREATE_STATISTICS property of the database, if set to TRUE, lets SQL Server (or more specifically SQL Server Query Optimizer) routinely create single-column statistics for query predicate columns as necessary, to improve cardinality estimates for the query execution plan if that specific column does not already have a histogram in an existing statistics object. The name for these statistics starts with _WA as you can see in the figure below, as an example for a table.



Statistics

You can also use the below query to find out all of those statistics created by SQL Server Query Optimizer for a specific table:

```
SELECT
  OBJECT_NAME(stats.object_id) AS TableName,
  COL_NAME(stats_columns.object_id,
stats_columns.column_id) AS ColumnName,
  stats.name AS StatisticsName
FROM sys.stats AS stats
JOIN sys.stats_columns AS stats_columns ON stats.stats_id
= stats_columns.stats_id
        AND stats.object_id = stats_columns.object_id
WHERE OBJECT_NAME(stats.object_id) = 'SalesOrderHeader'
AND stats.name like '_WA%'
ORDER BY stats.name;
```

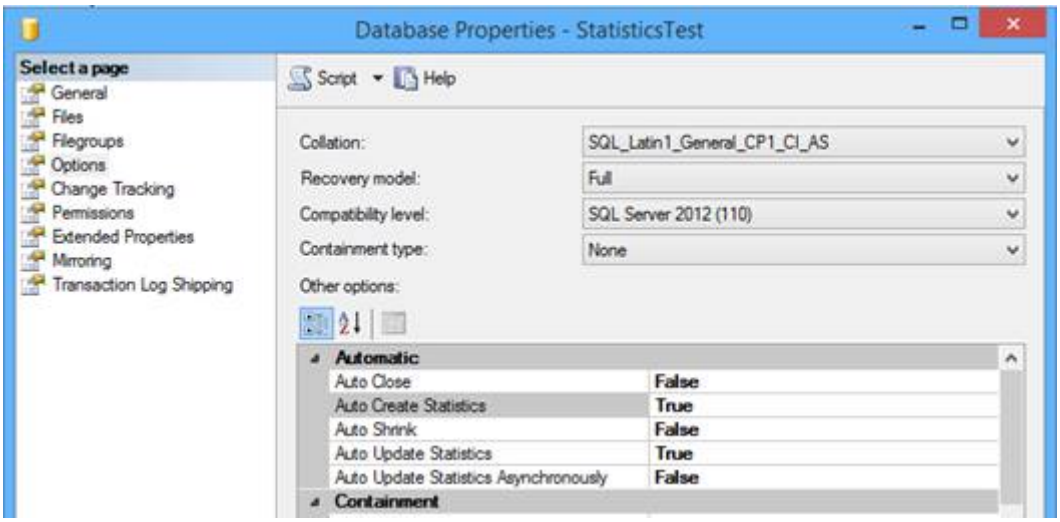
	TableName	ColumnName	StatisticsName
1	SalesOrderHeader	SalesOrderNumber	_WA_Sys_00000008_3C34F16F
2	SalesOrderHeader	TerritoryID	_WA_Sys_0000000D_3C34F16F
3	SalesOrderHeader	BillToAddressID	_WA_Sys_0000000E_3C34F16F
4	SalesOrderHeader	ShipToAddressID	_WA_Sys_0000000F_3C34F16F
5	SalesOrderHeader	ShipMethodID	_WA_Sys_00000010_3C34F16F
6	SalesOrderHeader	CreditCardID	_WA_Sys_00000011_3C34F16F
7	SalesOrderHeader	CurrencyRateID	_WA_Sys_00000013_3C34F16F

Statistics Created by SQL Server Query Optimizer - Results

- AUTO_UPDATE_STATISTICS property of the database, if set to TRUE lets SQL Server (or more specifically SQL Server Query Optimizer) routinely update the statistics being used by the query when they are stale (out-of-date) . Unlike AUTO_CREATE_STATISTICS, which applies for creating single column statistics only. AUTO_UPDATE_STATISTICS updates statistics objects created for indexes, single-columns in query predicates, filtered statistics and statistics created using the CREATE STATISTICS command.
- By default, identified stale statistics are updated synchronously, which means the query being executed will be put on hold until the required statistics are updated in order to ensure the query always compiles and executes with up-to-date statistics. Sometimes this wait could be longer, especially when a table involved in the query is bigger in size, and might cause the client request time-out. In order to deal with such a situation, SQL Server has AUTO_UPDATE_STATISTICS_ASYNC property of the database, which if set to TRUE lets the current running query compile with existing statistics even if the existing statistics are stale (chooses suboptimal query plan) and initiates a process in the background asynchronously to update the stale statistics in order to ensure subsequent query compilation and execution uses up-to-date statistics.

There are different ways you can change these properties; for example, you can use a script to change or use the Database Properties dialog box in SQL Server Management Studio to change it as shown below:

```
ALTER DATABASE [StatisticsTest] SET       AUTO_CREATE_STATISTICS ON
ALTER DATABASE [StatisticsTest] SET       AUTO_UPDATE_STATISTICS ON
ALTER DATABASE [StatisticsTest] SET       AUTO_UPDATE_STATISTICS_ASYNC OFF
GO
```

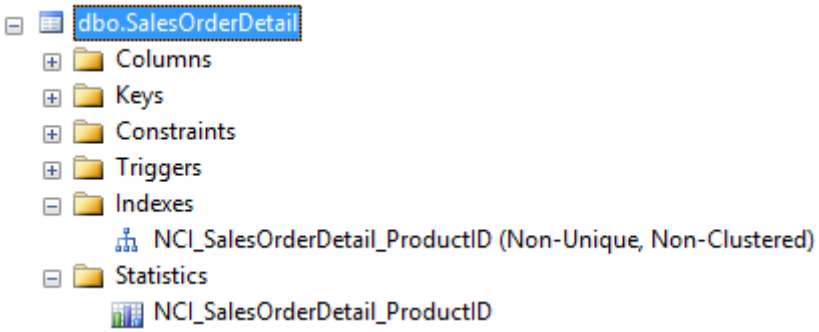


Database Properties – Statistics Test

- When you create an index – When you create indexes on a table or view, statistics are automatically created on the key columns of the indexes. This applies for filtered indexes as well. In other words, when you create filtered indexes, even filtered statistics are created along with that too.

For example, as you can see in the script below I created a non-clustered index on ProductID and accordingly, statistics on the same key column has been created with the same name as of index:

```
CREATE NONCLUSTERED INDEX       [NCI_SalesOrderDetail_ProductID]
ON       [dbo].[SalesOrderDetail] ([ProductID])
GO
```



Dbo.SalesOrderDetail

- **Manually creating statistics on strategic columns** – As discussed above, statistics are created when you create indexes or are created automatically by SQL Server Query Optimizer on the single column when the AUTO_CREATE_STATISTICS database property is set to ON. But there might be times when you need to create additional statistics using the CREATE STATISTICS command, syntax mentioned below, to capture statistical correlations so that it enables the SQL Server Query Optimizer to create improved query plans and drop it when not needed using DROP STATISTICScommand.
- `CREATE STATISTICS statistics_name`

```
ON { table_or_indexed_view_name } ( column1, column2,
...n )
            [ WHERE
<filter_predicate_for_filtered_statistics> ]
            [ WITH
                [ [ FULLSCAN
                  | SAMPLE number { PERCENT | ROWS }
                  | STATS_STREAM = stats_stream ] ]
                [ [ , ] NORECOMPUTE ]
                [ [ , ] INCREMENTAL = { ON |       OFF } ]
            ] ;
```

- WHERE - with this clause you can specify a filter predicate for filtered statistics.
- FULLSCAN - With this clause, you specify to compute statistics by scanning all rows in the table or indexed view or use SAMPLE to create a based on sample.
- NORECOMPUTE - With this clause, you specify to disable the automatic statistics update by SQL Server when AUTO_STATISTICS_UPDATE is ON. It's recommended to not use this feature often and let SQL Server Query Optimizer decide and update statistics as and when needed.
- INCREMENTAL - This is a new clause available in SQL Server 2014, its default value is OFF which means stats are combined for all partitions. If you set to ON, the statistics created are per partition statistics.

For example, the below given script creates statistics on the Employee table for Name, Age and Salary columns by scanning all the rows of the table. In this case, histogram is generated for Name, densities for the following column prefixes: (Name), (Name, Age), and (Name, Age, Salary).

```
CREATE STATISTICS statis_Employee_Name_Age_Salary
ON       [dbo].[Employee] (Name, Age, Salary)
WITH FULLSCAN;
```

Filtered Index and Filtered Statistics

Normally, statistics are created by considering all the values for all rows. But starting with SQL Server 2008, like filtered index you can also create filtered statistics on a subset of rows from a table. This comes in handy when your queries select only a subset of rows as these subset of rows will have completely different data distribution.

Filtered statistics are created either when you create a filtered index (on the same subset of rows specified for the filtered index) or by using the CREATE STATISTICS command along with the WHERE clause to specify the filter predicate.

What's New in SQL Server 2012 with Respect to Statistics

In earlier versions of SQL Server, if you use a database in read-only mode or a database snapshot, your queries will compile with the existing statistics even though it is not up-to-date. SQL Server will not be able to create statistics when it is missing or has become stale as changes to the database are not allowed, which means SQL Server will continue executing your queries with a sub-optimal plan.

Starting with SQL Server 2012, SQL Server Query Optimizer creates and maintains temporary statistics in tempdb database for the read-only database or read-only snapshot database or readable secondaries in the AlwaysOn cluster in order to ensure your queries perform better. These temporary statistics are created and maintained by SQL Server Query Optimizer only; though you can delete them when not needed. These statistics are suffixed with "_readonly_database_statistic" to differentiate it from the regular statistics.

Please note, as the tempdb database is re-created every time SQL Server service is restarted, all your temporary statistics will disappear after restart.

Conclusion

Statistics in SQL Server plays a pivotal role in efficient query execution. In this article, I discussed how the presence of up-to-date statistics plays an important role in the generation of high-quality query execution plans. I discussed the details of statistics, when to create and when to update it. I also talked about filtered statistics (introduced with SQL Server 2008) and new statistics enhancements in SQL Server 2012.

Execute UPDATE STATISTICS for all SQL Server Databases

By: Tim Ford | Read Comments (8) | Related Tips: More > Maintenance

Problem

If you're like me, you have a SQL Agent job in place to rebuild or reorganize only the indexes in your databases that truly require such actions. If you rely on the standard maintenance plans in Microsoft SQL Server, a *Scorched-Earth* policy of rebuilding all indexes occurs. That is whether such actions are required or not for a specific index, a rebuild of the index and all the locking and churning in the logs occurs. That is why so many of us "roll our own" index maintenance solutions as it were. It is also one of my biggest peeves with Microsoft - why was this not built into SQL Server 2008? Ah, at any rate, by only maintaining indexes that are fragmented, statistics updates do not occur globally against the tables/indexes in your databases. What we need is a quick solution for updating all the statistics for every database on our SQL Server instance.

Solution

Before you go ahead and state the fact that you have AUTO_UPDATE_STATISTICS ON for your databases remember that does not mean that they are being updated!

Bah! You say? Well think about this for a second. I've touched upon this in an earlier tip, SQL Server's engine will update the statistic when:

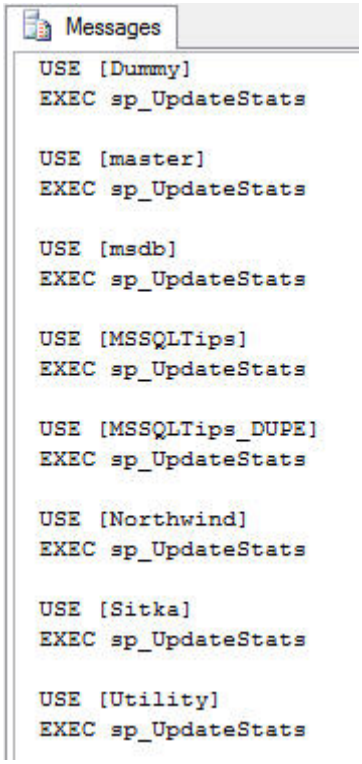
- When data is initially added to an empty table
- The table had > 500 records when statistics were last collected and the lead column of the statistics object has now increased by 500 records since that collection date
- The table had < 500 records when statistics were last collected and the lead column of the statistics object has now increased by 500 records + 20% of the row count from the previous statistics collection date

Based upon this criteria, there will be many cases where the underlying data changes in such a way or in such levels that the statistics that exist for an index will not be indicative of the actual data in the database. Because of this you simply can not rely on the engine to keep you statistics in check and current. Here is a simple block of code that will iterate through all your databases in order to build the sp_updatestats command that can then be copied and pasted into a new query window for execution. This code will work with all current and previous versions of SQL Server back through SQL 7.0.

```
DECLARE @SQL VARCHAR(1000)
DECLARE @DB sysname

DECLARE curDB CURSOR FORWARD_ONLY STATIC FOR
  SELECT [name]
  FROM master..sysdatabases
```

```
  WHERE [name] NOT IN ('model', 'tempdb')
  ORDER BY [name]

OPEN curDB
FETCH NEXT FROM curDB INTO @DB
WHILE @@FETCH_STATUS = 0
  BEGIN
     SELECT @SQL = 'USE [' + @DB +']' + CHAR(13) + 'EXEC sp_updatestats' + CHAR(13)
     PRINT @SQL
     FETCH NEXT FROM curDB INTO @DB
  END

CLOSE curDB
DEALLOCATE curDB
```

On my test server this code yields the following results.

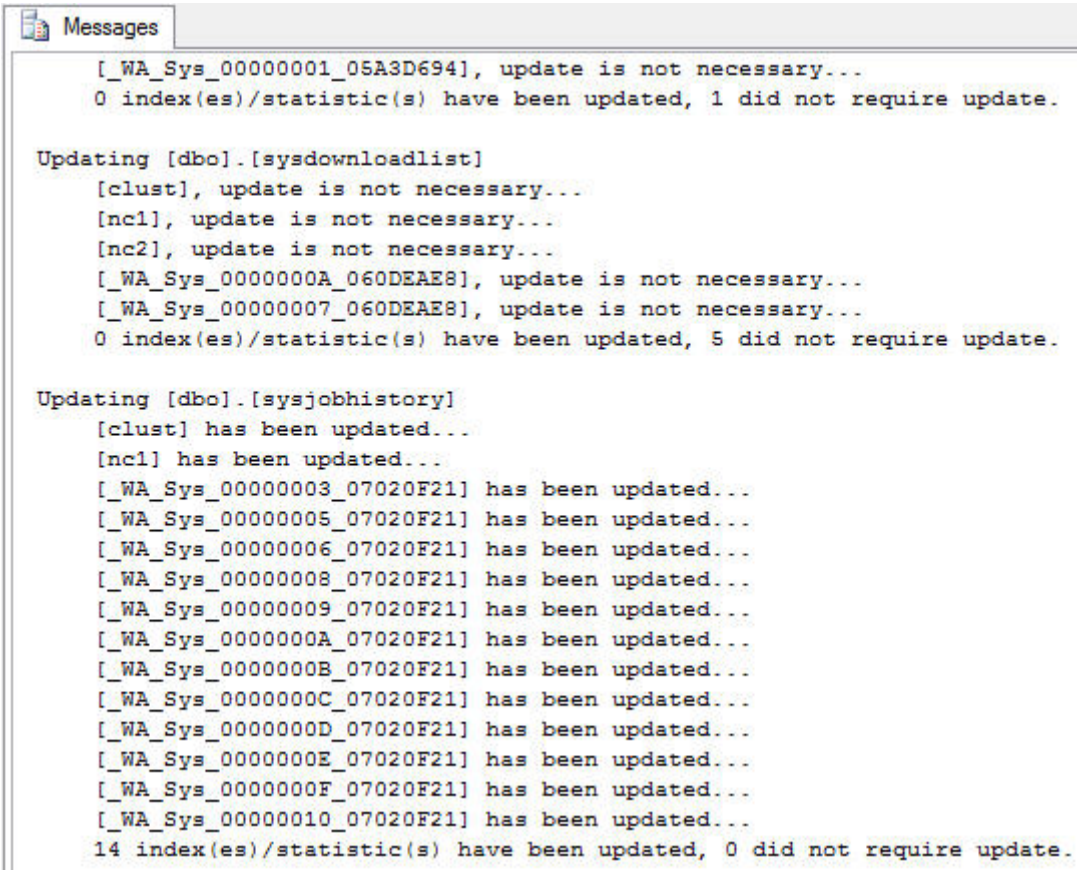


The next step is to copy this text, paste it into a query window in SQL Server Management Studio, then execute it against the instance. Alternately you may choose only to execute it against select databases, but that is entirely up to you.

```
USE [Dummy]
EXEC sp_UpdateStats

USE [master]
EXEC sp_UpdateStats
```

```
USE [msdb]
EXEC sp_UpdateStats

USE [MSSQLTips]
EXEC sp_UpdateStats

USE [MSSQLTips_DUPE]
EXEC sp_UpdateStats

USE [Northwind]
EXEC sp_UpdateStats

USE [Sitka]
EXEC sp_UpdateStats

USE [Utility]
EXEC sp_UpdateStats
```

I've provided a sample of the output generated by the collection of SQL statements executed above. As you can see, the engine still will review the statistics to see if they warrant updating. It will ignore those that are acceptable and will update only the statistics that require such action. I can assure you that each of the databases in the listing above had AUTO_UPDATE_STATISTICS and AUTO_CREATE_STATISTICS both set to ON, yet the following results are indicative of statistics that can become outdated.


Messages
[_WA_Sys_00000001_05A3D694], update is not necessary...
0 index(es)/statistic(s) have been updated, 1 did not require update.
Updating [dbo].[sysdownloadlist]
[clust], update is not necessary...
[nc1], update is not necessary...
[nc2], update is not necessary...
[_WA_Sys_0000000A_060DEAE8], update is not necessary...
[_WA_Sys_00000007_060DEAE8], update is not necessary...
0 index(es)/statistic(s) have been updated, 5 did not require update.
Updating [dbo].[sysjobhistory]
[clust] has been updated...
[nc1] has been updated...
[_WA_Sys_00000003_07020F21] has been updated...
[_WA_Sys_00000005_07020F21] has been updated...
[_WA_Sys_00000006_07020F21] has been updated...
[_WA_Sys_00000008_07020F21] has been updated...
[_WA_Sys_00000009_07020F21] has been updated...
[_WA_Sys_0000000A_07020F21] has been updated...
[_WA_Sys_0000000B_07020F21] has been updated...
[_WA_Sys_0000000C_07020F21] has been updated...
[_WA_Sys_0000000D_07020F21] has been updated...
[_WA_Sys_0000000E_07020F21] has been updated...
[_WA_Sys_0000000F_07020F21] has been updated...
[_WA_Sys_00000010_07020F21] has been updated...
14 index(es)/statistic(s) have been updated, 0 did not require update.