

SQLskills Immersion Event

IE1: Internals and Performance

Module 10: Statistics

Kimberly L. Tripp
Kimberly@SQLskills.com



Overview

- **Cost-based optimization**
- **Data access patterns**
- **Statistics**
 - What do they look like?
 - What are they telling us?
 - How do you see them
 - When/how do they get created
 - When/how do they get updated
- **Problems/solutions with statistics**
 - The histogram
 - Filtered stats
 - Uneven distribution



2

© SQLskills. All rights reserved.
<http://www.SQLskills.com>



Statement Execution Simplified

Optimization = Compilation



1. Parse
2. Standardization/normalization/algebraization
⇒ Query tree (not T-SQL anymore)
3. Cost-based optimization (statistics are used to come up with optimization plan as well as lock granularity)
4. Compilation
5. Execution: at runtime, if the resources don't exist to support the lock level then escalation occurs to TABLE level (by default)

Cost-Based Optimization

- Find a reasonable subset of possible algorithms to access data based on:
 - The query
 - Any joins
 - Any SARGs
 - Data selectivity
 - Join density
- The more information the optimizer has the better!
- How do you provide the BEST information?

Selectivity

- Not just based on the number of rows returned
- Always relative to the number of rows in the table (usually expressed as a percentage)
- Low number of rows = high selectivity
 - Any index is useful if even ONE condition is highly selective!
- High number of rows = low selectivity
 - This is harder!
 - Must consider some form of covering

Understanding Selectivity

How Would YOU Find This Data?

- Imagine a table of employee data – for The Chicago Tribune
- The table is clustered by EmployeeID
- Imagine executing this query?

```
SELECT e.*  
FROM dbo.EmployeesPersonalAddresses AS e  
WHERE e.city = 'Chicago'           not selective enough  
WHERE e.city = 'Glenview'         not an easy answer  
WHERE e.city = 'Peoria'           selective enough
```

- When is an index on "City" useful?
 - When the data is selective ENOUGH...

*More importantly,
how does SQL Server
know?*

How Would SQL Server Know?

- In every example, knowing the number of rows is necessary to estimate I/Os
- How does SQL Server know how many rows without reading the rows?

Statistics

What Do They Look Like?

Statistics Header

Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index
MemberName	Oct 10 2008 1:02AM	10000	10000	26	0	21.5526	YES

Density Vector

All density	Average Length	Columns
0.03846154	5.6154	LastName
0.0001	16.5526	LastName, FirstName
0.0001	17.5526	LastName, FirstName, MiddleInitial
0.0001	21.5526	LastName, FirstName, MiddleInitial, member_no

Histogram

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
ANDERSON	0	385	0	1
BARR	0	385	0	1
CHEN	0	385	0	1
...	...	...	...	...
ZUCKER	0	384	0	1

Overall,
this is
weird
data?!

What Are They Telling You?

- **High-order element (LastName)**
 - Histogram (#4) has great details for this column
 - Anderson 385 rows
 - Barr 385 rows
- **Secondary columns – DENSITY (#3)**
 - Always LEFT-based
 - Rows * All density = estimate of rows
 - Based on even distribution
- **Density information**
 - Density for LastName
 - $10,000 \text{ Rows} * 0.03846154 = 384.6154$
 - Density for LastName, FirstName combo
 - $10,000 \text{ Rows} * 0.0001 = 1$

Are They Really True?

- **For LastName**

```
SELECT AVG(Counts.GroupCounts)
FROM (SELECT COUNT(*) AS GroupCounts
      FROM dbo.member AS m
      GROUP BY m.LastName) AS Counts
```

- **For LastName, FirstName**

```
SELECT AVG(Counts.GroupCounts)
FROM
  (SELECT COUNT(*) AS GroupCounts
   FROM dbo.member AS m
   GROUP BY m.LastName, m.FirstName)
  AS Counts
```

- **What does this tell you about FirstNames?**

What Are the Options?

```
SELECT m.LastName, m.FirstName,  
       m.MiddleInitial, m.Phone_no, m.City  
FROM   dbo.Member AS m  
WHERE  m.FirstName LIKE 'Kim%'
```

- Table scan (always an option)
- No indexes exist for SEEKING (no indexes with FirstName as the high-order element)
- What about SCANNing the NC index which has FirstNames in it...seems like a gamble?

How Do You See Them? (1)

DBCC SHOW_STATISTICS (tname, statname)

- Gives you ALL the statistical details
 - Number of rows and number of rows on which the statistics were based
 - Densities for all LEFT based subsets of column, including the CL key (last – if not already somewhere in the index)
 - Histogram for high order element

sp_autostats tname

Index Name	AUTOSTATS	Last Updated
[member_ident]	ON	2008-08-26 17:18:12.58
[member_corporation_link]	ON	2008-08-26 17:18:12.61
[member_region_link]	ON	2008-08-26 17:18:12.79
[MemberName]	ON	2008-10-29 11:13:29.21
[_WA_Sys_00000003_OCBAEB]	ON	2008-10-29 11:28:32.31

How Do You See Them? (2)

- **Query sys.indexes**
 - Use object_id and index_id as input to stats_date
- **Query sys.stats**
 - Use object_id and stats_id as input to stats_date
- **Use STATS_DATE ([object_id], [index_id]) in a query**
 - Nice quick way to see JUST the date the statistics were last updated
 - Nice to check periodically and automatically
- **Use sys.dm_db_stats_properties (2008R2 SP2+, 2012 SP1+)**

What Kinds of Statistics Exist?

- **Auto-created statistics**
 - Named _WA_SYS_
 - Blog: [How are auto-created column statistics names generated?](#) (Paul Randal)
 - Created automatically when a missing statistics event is encountered
 - Permanent objects in the database, will get auto-updated if database option is set
- **User-created statistics**
 - Created by you...
 - Could have been recommended by DTA and named _dta_stat_
- **Hypothetical indexes**
 - Created during DTA's analysis phase
 - Dropped by letting DTA complete successfully
 - Can be created manually for "what if analysis using auto pilot"

DBCC AUTOPILOT

- Check out the Simple Talk article: Hypothetical Indexes on SQL Server
 - by [Fabiano Amorim](#)

```
CREATE INDEX <name> ON <tablename> (<columns>)
WITH STATISTICS_ONLY = -1
GO
DBCC AUTOPILOT(0, <dbid>, <objectid>, <indexid>)
GO
SET AUTOPILOT ON
GO
RUN QUERY TO TEST INDEX USAGE? (output is showplan xml)
GO
SET AUTOPILOT OFF
```

How Do You See Them? (3)

- Programmatically pull tabular sets from DBCC SHOW_STATISTICS ...
 - STAT_HEADER
 - Number of rows v. rows sampled and number of steps
 - Last updated date
 - DENSITY_VECTOR
 - Left-based density subsets
 - Averages given left-based combinations of index key
 - STAT_HEADER JOIN DENSITY_VECTOR
 - Presents the details from STAT_HEADER and DENSITY_VECTOR as a single row
 - All Density = $\frac{\text{TABLE_CARDINALITY}}{\text{TUPLE_CARDINALITY}}$
(for each left-based subset starting ending at that ordinal position)
 - HISTOGRAM
 - Up to 201 total steps with each step's density information
 - Up to 200 distinct actual values from the table
 - 1 row for NULLs if the column allows NULL values

When Are They Created?

- **Automatically**
 - For all Indexes
 - When “auto create statistics” is ON AND when the optimizer thinks that statistics would be a good idea (often when a column is in a SARG or a join and does not have an index with that column as the high-order element)
- **Manually**
 - sp_createstats
 - Using CREATE STATISTICS

Tip: Leave Auto Create Statistics ON

Manually Creating Statistics

- For secondary columns of an index: can make SCAN choices more likely for highly selective secondary conditions that don't warrant an index (where only some values are selective and where query frequency doesn't warrant the additional index)
- For columns in search arguments and joins where some are selective and others aren't (and again, you've decided not to index)

*While I recommend additional statistics,
they're not always perfect!
True, but a good start...more coming up on this!*

sp_createstats

```
sp_createstats  
@indexonly = 'indexonly'  
, @fullscan = 'fullscan'  
, @norecompute = 'norecompute'
```

- **@indexonly: only create statistics for secondary columns of indexes**
 - This can help to make non-clustered indexes more useful
- **@fullscan: requires more time but will create more accurate statistics**
 - If off-hours this is a good idea
- **@norecompute: the statistics will NOT get automatically updated as distribution of data changes**
 - Generally not recommended
- **Recommendation:**
`sp_createstats 'indexonly', 'fullscan'`

What If the Data Changes?

- **Automatically updated!**
 - If auto update statistics is ON (for both the DB and the statistic – didn't set "norecompute")
 - If roughly 500 + 20% of the data changes
 - *Complete details in TechNet whitepaper*
- **Manually update statistics**
 - For highly volatile tables where distribution isn't changing significantly and you see a lot of "statistics" events
 - Turn off "auto update" OR turn off auto update by adding STATISTICS_NORECOMPUTE on the index definition (better control)
 - Executing UPDATE STATISTICS

Auto Update Statistics

- **SQL Server 7.0**
 - Invalidated when sysindexes.rowmodctr reached
 - Updated when invalidated
- **SQL Server 2000**
 - Invalidated when sysindexes.rowmodctr reached
 - ✓ Updated when needed
- **SQL Server 2005**
 - Invalidated when sysrowsetcolumns.rcmodified reached
 - ✓ Updated when needed
- **SQL Server 2008**
 - Invalidated when sysrscols.rcmodified reached
 - Updated when needed

Statement Execution: Statistics Events

Optimization = Compilation



Missing Statistics Event

If **auto_create_stats** is enabled then SQL Server (the QO) will WAIT while statistics are created

Invalidated Statistics Event

- If **auto_update_stats_async** is disabled AND **auto_update_statistics** is enabled then SQL Server will WAIT while statistics are created
- If **auto_update_stats_async** is enabled then SQL Server will optimize based on the invalidated statistics AND kick off an update (which will be used by subsequent users)
- If both methods for updating statistics are disabled then the query will optimize using the invalidated statistic and a warning will be generated (visible in [xml] showplan)

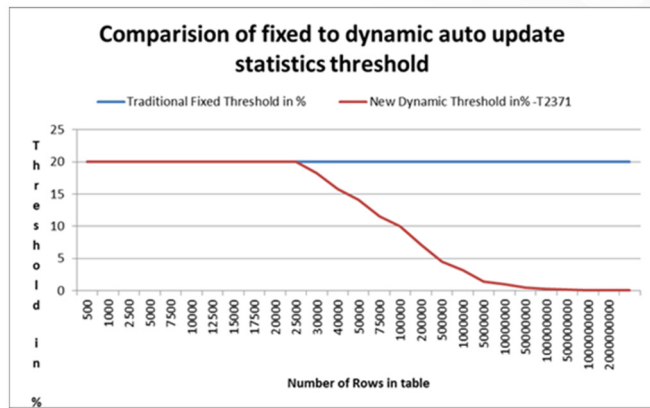
Asynchronous Statistics Updates

- By default statistics are updated before query compilation (as part of compilation/optimization) and this can cause a delay in execution
- Can turn “Async Stats Update” on to have the current execution trigger the update but not wait for the update (but you could get into a vicious cycle where you optimize with old statistics, trigger the update and by the time you execute again, they’re invalid)

```
ALTER DATABASE databasename  
SET AUTO_UPDATE_STATISTICS_ASYNC ON
```

Dynamic Auto Updating Threshold

- Server-wide trace flag 2371
- Released in SQL Server 2008 R2 SP1
- Blogged by:
Juergen
Thomas
(SQLCat)
7 Sep 2011



Updating Statistics

- **Manually: but automated through a job**
 - Executing sp_updatestats
 - Sledgehammer maintenance. Only one row has to have been modified to execute this... not very granular
 - Ola Hallengren's code
 - <http://ola.hallengren.com/>
 - Roll your own?
 - Programmatically evaluate the stat_header (just an idea)
 - If stats_date older than 1 week OR sampled < rows then UPDATE STATS with FULLSCAN
- **Auto update stats: only as a safety measure**
 - As a fallback if your code doesn't catch EVERY statistic
- **Asynchronous update stats**
 - *Unlikely* to cause a problem

Quick FYI – IE2 Slide Plan Invalidation

- **If database option: `auto_update_stats` is ON**
 - Updating statistics causes plan invalidation
- **If database option: `auto_update_stats` is OFF**
 - Updating statistics does NOT cause plan invalidation
- **Recommendation**
 - If you manually update statistics and do not use `auto_update_statistics`, add `sp_recompile` to your stats scripts
- **Erin Stellato's links about auto update stats and plan invalidation:**
 - Statistics and Recompilations: <http://erinstellato.com/2012/01/statistics-recompilations/>
 - Statistics and Recompilations, Part II: <http://erinstellato.com/2012/02/statistics-recompilations-part-ii/>

Sampling: Always Good?

- Using showplan tooltip – estimate v. actual rows
- If query performance is poor AND the actual is significantly OFF from the estimate then you might want to verify the statistics creation (rows v. rows scanned)
- If statistics were based on a sampling and performance is improved after statistics have been updated, then you might want to turn off auto update for this index (using STATISTICS_NORECOMPUTE) and schedule an update statistics with FULLSCAN

UPDATE STATISTICS ...
WITH FULLSCAN

The Histogram

- Stores ACTUAL values from the FIRST (and only first) column of the key
- Never stores more than 201 steps (200 actual values plus 1 NULL values row – if the column allows NULLs)
- Each step value is unique
- Values chosen aren't evenly distributed
 - Internally they can do step compression/expansion while building into to track "interesting" anomalies encounter during analysis
- Always has information over the entire table... even when partitioned
- Has the best information – but how good is it?
 - This is really the issue. And, unfortunately, it depends – mostly on how skewed the data distribution is...

Data Distribution Matters

- **Even distribution is easy**
 - Think about “line items per sales order”
 - That’s probably *fairly* consistent at 2 or 3
- **Un-even distribution is HARDER**
 - Think about sales per product
 - This is all over the place – and varies over time as well...
 - Think about sales per customers
 - Again, all over the place – and, again, varies over time...
- **The histogram does a MUCH better job having steps and average distribution per step but what if the table is really big and the data is very skewed...**

Filtered Statistics

- **Not an index, just a statistics blob...**
- **Very useful to help the optimizer assess the usefulness of another index**
- **Take a very skewed example: sales by customer**
 - Sales: 30,923,776
 - Customers: 18,484
 - Average = $30,923,776 / 18,484 = 1,673$
 - All density (from density_vector of statistics) = $5.410084E-05$ multiplied by rows = 1,673
 - Skew $\Rightarrow$ high: 30K+ (only ~6), Low: ~500 (thousands)
 - Not every value can possibly be represented in 200 steps (for histogram)
 - Occasionally, they’ll be wrong – average might not be good enough... enter $\Rightarrow$ filtered statistics

How Many Filtered Indexes/Stats? Per Table

- **SQL Server 2000**
 - Columns: 1,024
 - Nonclustered indexes: 249
 - Statistics: 249 (shared with nonclustered)
- **SQL Server 2005**
 - Columns: 1,024
 - Nonclustered indexes: 249
 - Statistics: 2,000 (referenced in sys.stats)
- **SQL Server 2008/R2**
 - Columns: 30,000
 - Nonclustered indexes: 999
 - Statistics: 10,000 (30,000 in R2)

Valid Index IDs

- **Table always has at least one index id**
 - For a heap the index id is always 0
 - For a clustered table the index id is always 1
 - Nonclustered indexes *can* start at 2
- **999 nonclustered indexes per table**
 - Index ids start with 2 but statistics use the same counter range
 - Index ids 251 through 255 are reserved (from prior use cases)
 - NOTE: 255 was used in earlier releases but 251-254 have never been used (at least not that I know of ☺)
- **30,000 statistics per table (SQL Server 2008 R2+)**
- **Index and statistics ids run from 2 to 250 and from 256 to 31005**
 - NOTE: The datatype used is int (sys.indexes.index_id and sys.stats.stats_id). Technically they *could* support more...

The Bad News...

- **Filtered indexes/filtered stats: stats may get HORRIBLY out of date:**
 - SQL Server ONLY tracks a colmodctr NOT related to the filtered set or size. As a result, the stats (even filtered stats) are only updated when 20% of the table has been modified...
 - For better performance, you need to automate and control their updates (do NOT rely on "auto update statistics")
 - Good news is that they're relatively small and creating a job to automated them is relatively easy!
 - See this blog post: <http://www.sqlskills.com/BLOGS/KIMBERLY/post/Filtered-indexes-and-filtered-stats-might-become-seriously-out-of-date.aspx> (<http://bit.ly/1knEE2>)
- **Forced parameterization (the database option) will generalize statements and many filters won't be eligible during optimization**
- *Similar summary, don't go wild with this feature; it has powerful – but specific – uses!*

Evenly Distributed Data?

- **Scenario**
 - You have 90 rows in a table
 - 10 rows have a value of x for column 6
 - Where (physically) in this table are these 10 rows?
- **They could be evenly distributed throughout the table...**



- **But, what if they're not?**



Scenario: Unevenly Distributed Data

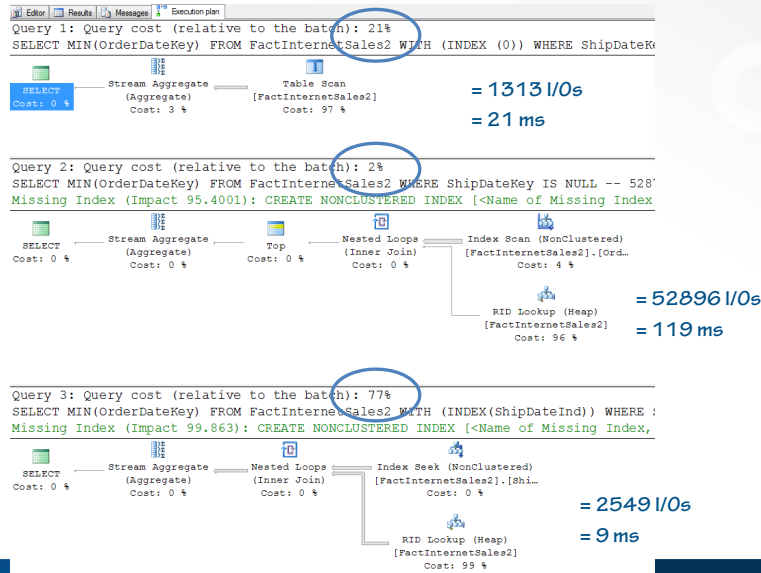
- **Scenario**
 - AdventureWorksDW2012: FactInternetSales has 60,398
 - 2,541 rows have a null for ShipDateKey
 - $60,398 / 2,541 = 23.7$ (1 in 23.7 rows is NULL)
 - Nonclustered index on ShipDateKey
 - Nonclustered index on OrderDateKey
- **What does SQL Server do?**

```
SELECT MIN(fis.OrderDateKey)
FROM dbo.FactInternetSales2 AS fis
WHERE fis.ShipDateKey IS NULL
```

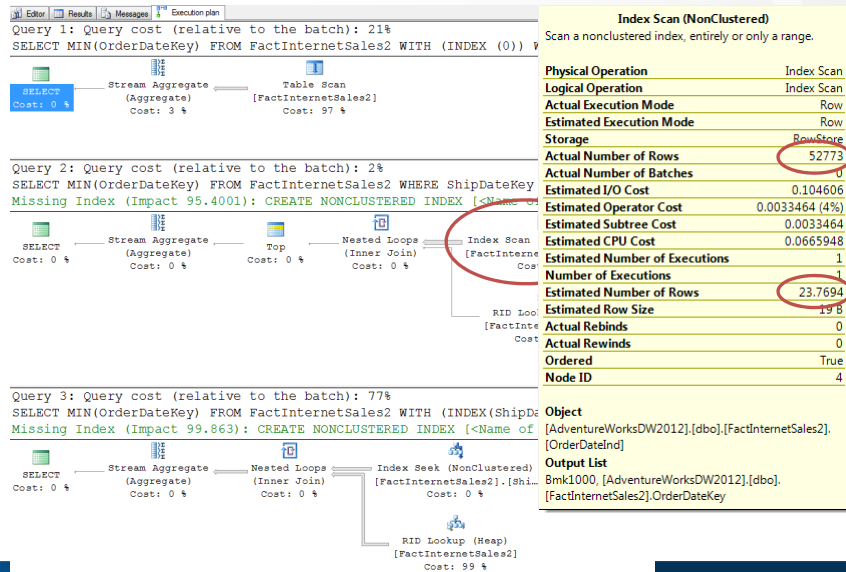
Even Distribution: Always Even??

- **Table scan is always an option**
- **Use an index on ShipDateKey to look up the actual date for all orders where ShipDateKey is NULL**
 - This means a bookmark lookup must be run for EVERY NULL so that we can get the OrderDateKey
 - The worktable then needs to be sorted to find the lowest order date
- **Use an index on OrderDateKey as only 1 on 23.7 sales have a NULL for ShipDateKey**
 - SQL Server estimates that they'll find a NULL within 23.7 rows and they won't need a worktable
 - This sounds better...
- **But the rows are not evenly distributed!**

Evenly Distributed Data??



Evenly Distributed Data??

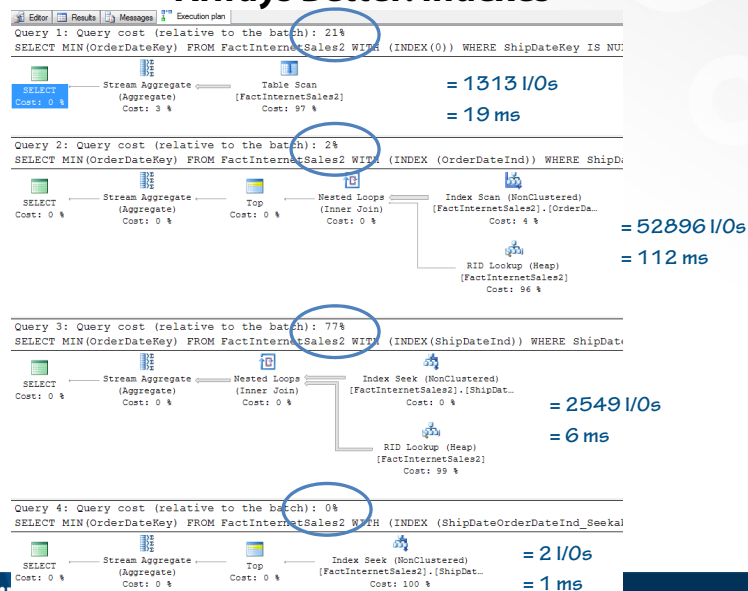


Always Better: Indexes

- **Statistics cannot be used to directly access data but:**
 - They can only HELP the optimizer determine the best plan
 - They can help determine best join order
- **Statistics are just estimates: they help MOST of the time but:**
 - There are limitations
 - They take time to create, update, store...
- **Indexing can often be A LOT better...**

```
CREATE INDEX ShipDateOrderDateInd_SeekableForMin
ON FactInternetSales2 (ShipDateKey, OrderDateKey)
```

Always Better: Indexes



Always Better: The RIGHT Indexes

- Every SINGLE QUERY can be indexed to make THAT SINGLE query fast
- You can't index EVERY query unless you're read-only/decision support, and even then you're limited by disk space (but that's about it...lots of indexes – yeah!)
- But a better choice is prioritizing and determining which queries really need indexes!

Review

- **Cost-based optimization**
- **Data access patterns**
- **Statistics**
 - What do they look like?
 - What are they telling us?
 - How do you see them
 - When/how do they get created
 - When/how do they get updated
- **Problems/solutions with statistics**
 - The histogram
 - Filtered stats
 - Uneven distribution

Questions!



Stats Trace Flags of Interest

- 9204: statistics used by query
- 9292: statistics header loaded
- 2371: dynamic auto updating of statistics
- 2388: Additional statistics details (see next slide)
- 2389: don't rely on "max" value – query the table to compute the highest value of the column. Useful for ascending columns that are queried long before their statistics are updated. Statistics must have been updated at least 3 times to be "branded" as ascending (Column: Leading Column Type from 2388 output)
- 2390: Read highest value from column disregard branding

Query-level Trace Flags

```
SELECT m.*  
FROM dbo.Member AS m  
WHERE m.firstname LIKE 'Kim%'  
OPTION  
    (QUERYTRACEON 3604  
    , QUERYTRACEON 9204  
    , QUERYTRACEON 9292  
    , RECOMPILE);
```

- Only set for that execution
- Affect only that query

Additional Statistics Output

DBCC TRACEON(2388)

```
DBCC SHOW_STATISTICS  
(  
    'member',  
    'member_ident'  
)
```

DBCC TRACEOFF(2388)

Returns additional details from statistics blob. Columns:

- ❑ Updated
- ❑ Table Cardinality
- ❑ Snapshot Ctr
- ❑ Steps
- ❑ Density
- ❑ Rows Above
- ❑ Rows Below
- ❑ Squared Variance Error
- ❑ Inserts Since Last Update
- ❑ Deletes Since Last Update
- ❑ Leading column Type