SQL Server:
Optimizing Ad Hoc Statement Performance

Module 3: Estimates and Selectivity

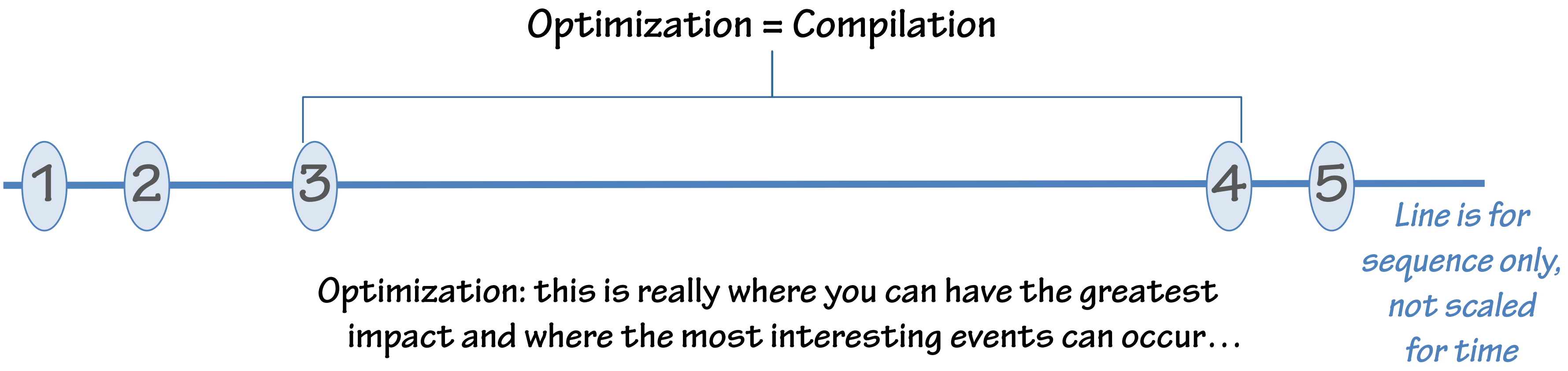
Kimberly L. Tripp
Kimberly@SQLskills.com
http://www.SQLskills.com/blogs/Kimberly

pluralsight
hardcore developer training

Course Overview

- **Statement execution methods**
- **Estimates and selectivity**
  - Statement execution simplified
  - Cost-based optimization
  - Understanding selectivity
  - Estimates, statistics, and heuristics
  - How do you see statistics?
  - What do statistics tell us about our data?
  - When and how does SQL Server use statistics?
- **Statement caching**
- **Plan cache pollution**
- **Statement execution summary**

Statement Execution Simplified



Optimization: this is really where you can have the greatest impact and where the most interesting events can occur…

Line is for sequence only, not scaled for time

1. Parse
2. Standardization/normalization/algebrization
   ⇒ query tree (not Transact-SQL anymore)
3. Cost-based optimization (statistics are used to come up with an optimal plan, as well as other things)
4. Compilation
5. Execution

Cost-Based Optimization

- **Find a reasonable subset of possible algorithms to access data based on:**

□ The query	Sometimes a rewrite helps...
□ Any joins	Sometimes a derived table (sub-query in the FROM clause)…
□ Any SARGs	Your SARGs need to be well-defined…
□ Data selectivity	
□ Join density	

- **The more information the optimizer has the better…**
- **How do you provide the BEST information?**

- **One of the best ways to “influence” your query plans is through effective statistics (and better indexes)**

Understanding Selectivity

- Imagine a table of employee data – for a Chicago company
- The table is clustered by *EmployeeID*
- Imagine executing this query:

```
SELECT [e].*
FROM [dbo].[EmployeesAddresses] 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?

Demo Summary: Estimates and Selectivity

- **Estimates come from:**
  - Statistics – if they exist or if they can be (auto) created, using:
    - The histogram: when the value can be "sniffed" (parameters)
    - The density vector: when the value cannot be "sniffed" (variables)
  - Heuristics – if there are no statistics available and SQL Server cannot auto create them
    - These are internal "magic" numbers (cannot be changed)
    - They often result in very poor plans (LEAVE *AUTO_CREATE_STATISTICS* ON)
    - Sometimes this is the only option when better estimations cannot occur (comparison between columns (e.g. *col1* > *col2*))
- **Statistics have to be reasonably small to be fast/useful**
- **They're just estimates**
  - They're not always guaranteed to be accurate
  - They're just meant to get us closer to the right value

How Do You See Statistics?

- `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 the column, including the cluster key (last – if not already somewhere in the index)
    - Histogram for the high-order element

- `sp_autostats tname`

Index Name	AUTOSTATS	Last Updated
[member_ident]	ON	2008-08-26 17:18:12.593
[member_corporation_link]	ON	2008-08-26 17:18:12.673
[member_region_link]	ON	2008-08-26 17:18:12.793
[MemberName]	ON	2008-10-29 11:13:29.220
[_WA_Sys_00000003_0CBAE877]	ON	2008-10-29 11:28:32.313

What Do Statistics Tell Us About Our Data? (1)

1 2

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

3

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

4

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

What Do Statistics Tell Us About Our Data? (2)

- **Statistics date (#1)**
  - This is the date that the statistics were last updated:
    - Through SQL Server's auto-updating mechanism
      - Database option: *AUTO_UPDATE_STATISTICS*
    - Manually, by executing one of the following:
      - *sp_updatestats*
      - *UPDATE STATISTICS*
  - Or, if they've never been updated then it represents date they were created:
    - Through SQL Server's auto-create mechanism
      - Database option: *AUTO_CREATE_STATISTICS*
    - Manually, by executing one of the following:
      - *sp_createstats*
      - *CREATE STATISTICS*
  - Can also get this information from the function: *STATS_DATE()* and the new (new in SQL 2008 R2 SP2 / and SQL Server 2012 SP1) DMV:

```
SELECT *
FROM [sys].[dm_db_stats_properties](object_id, index_id)
```

What Do Statistics Tell Us About Our Data? (3)

- **Data analyzed to build the statistics (#2)**
  - Rows – number of rows in the table at the time the statistics were built
  - Rows Sampled – the number of rows that were analyzed to generate the statistic
- **Sampling**
  - Does not directly indicate a problem with statistics
  - Could be a problem if your data is heavily skewed
- **Is it a problem?**
  - Using showplan tooltip – estimate vs. 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 sampled)
    - 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*

What Do Statistics Tell Us About Our Data? (4)

- **Density vector (#3)**
  - Shows the average distribution of data given the LEFT-based subsets of the entire key
  - Rows * all density = average number of rows returned
    - Based on that left-based subset of columns supplied
    - Density information
      - Density for *LastName*
        - 10,000 Rows * 0.03846154 = 384.6154
      - Density for *LastName, FirstName* combined
        - 10,000 Rows * 0.0001 = 1
- **Histogram (#4)**
  - Only stores data for the leading column of the index (sometimes referred to as the high-order element (e.g. *LastName*))
  - Has actual values with details about that "step"
    - Anderson 385 rows
    - Barr 385 rows

    …

When and How Does SQL Server Use Statistics?

- **Estimation comes from “sniffing” the value**
  - **Result:** estimate comes from *HISTOGRAM*
  - **Pro:** estimate is usually more accurate
  - **Pro:** *that* execution gets a plan designed for *that* value
  - **Con:** If/when this plan is saved – subsequent executions are prone to “parameter sniffing problems” (PSP)
- **Value cannot be “sniffed”**
  - **Result:** estimate comes from the *DENSITY_VECTOR*
  - **Depends:** estimate is an “average”
  - **Depends:** the plan generated is designed for the “average” value – not *that* value
  - **Pro:** If/when this plan is saved – subsequent executions are NOT prone to PSP
  - **Con:** When your data is NOT [relatively] evenly distributed, this plan might not be good for anyone

Summary: Estimates and Selectivity

- **Method: ad hoc statement**
  - Can have literals
    - Can be “sniffed” and estimated using the *HISTOGRAM*
    - Can generate an optimal plan
  - Can have variables
    - Cannot be “sniffed” (they are unknown during optimization/compilation)
    - Optimizes based on the average distribution of data (using the *DENSITY_VECTOR*)
  - Can be parameterized and cached but it’s extremely unlikely (only when safe)
    - Requires CPU/compilation on every execution
- **Method: *sp_executesql***
  - Can generate an optimal plan for the *first* execution
    - Saves CPU/compilations costs for subsequent executions
  - Can be prone to parameter sniffing problems (PSP)
    - When the optimal plan varies (based on the parameters passed) then subsequent executions may suffer by using the plan chosen by the *first* execution’s parameters
- **Method: Dynamic String Execution through *EXEC (@string)***
  - Turns the statement into an ad hoc statement
  - It behaves exactly like an ad hoc statement