

SQLskills Immersion Event

IEPTO2: Performance Tuning and Optimization

Whiteboard Drawings

Kimberly L. Tripp
Kimberly@SQLskills.com



**NOTE: Includes whiteboard drawings from prior IE courses
+ drawings done the week of October 12, 2015 in Dublin, Ireland – IEPTO2 Training**

These next 5 slides come from this Pluralsight course.

Module 4: Statement Caching

[Here's a link to the course.](#)

SQL Server: Optimizing Ad Hoc Statement Performance

Module 4: Statement Caching

Kimberly L. Tripp

Kimberly@SQLskills.com

<http://www.SQLskills.com/blogs/Kimberly>



pluralsight 
hardcore developer training

Stored Procedure Plans and Caching

- A plan is generated when no plan already exists in cache
- Plans are never saved on disk and may not persist within the cache
 - Some operations completely remove / evict the plan from the cache
 - **Server-level:** Server restart | DBCC FREEPROCCACHE | some RECONFIGURE changes
 - See recordings starting with **Side Effect: Plan Cache Flush** for version-specific details and demo
 - **Database-level:** DBCC FLUSHPROCINDB (undocumented) | sp_dbcmptlevel
 - **Procedure-level:** sp_recompile or they're aged out through non-use
 - Some operations cause the plan to be invalidated (but it still remains an object in the cache); these can be tracked
 - Schema of base object changes (ALTER TABLE / ALTER <object>)
 - Including when indexes are added to the base object
 - Statistics of base objects change
 - See recordings starting with **Plan Invalidation** for version-specific details and demos
- When a plan exists in cache, all subsequent executions use that plan

Side Effect: Plan Cache Flush (1)

- In SQL Server 2005 (only), numerous operations cause the entire plan cache to be cleared / flushed (all plans evicted from cache)
 - Database-level operations that cause an entire plan cache flush include:
 - When a database auto-closes (from AUTO_CLOSE option)
 - When a database is detached
 - When a database snapshot is dropped
 - When a database state is changed (OFFLINE, ONLINE, READ_ONLY, READ_WRITE)
 - When a database backup is restored
 - ...
 - **There are quite a few others;** see KB article 917828 for complete list
- In all versions, numerous server-level operations that cause the entire plan cache to be cleared include:
 - max server memory (MB)
 - min server memory (MB)
 - **... There are quite a few others;** more information coming up

Side Effect: Plan Cache Flush (2)

- **From SQL Server 2005 SP2 onward, messages written to the error log show you that the cache was flushed:**
 - SQL Server has encountered 4 occurrence(s) of cachestore flush for the 'Object Plans' cachestore (part of plan cache) due to some database maintenance or reconfigure operations.
 - SQL Server has encountered 4 occurrence(s) of cachestore flush for the 'SQL Plans' cachestore (part of plan cache) due to some database maintenance or reconfigure operations.
 - SQL Server has encountered 4 occurrence(s) of cachestore flush for the 'Bound Trees' cachestore (part of plan cache) due to some database maintenance or reconfigure operations.

Side Effect: Plan Cache Flush (3)

- From SQL Server 2008 onward, many database-level operations that caused the entire plan cache to be flushed no longer do so
 - Instead, they only cause the plans for that database to be flushed
- Database-level operations that cause the database plan cache to be flushed include:
 - DBCC FLUSHPROCINDB (dbid)
 - When a database is detached / dropped / restored
 - When a database (or filegroup with the database) changes between READ_ONLY and READ_WRITE
 - When a database changes between OFFLINE and ONLINE, or auto closes
 - There are others... (for example, changing recovery model ...)

NOTE: There are many places where a few of these are documented incorrectly. Some aren't detailed at all and others say that many of these operations **STILL** cause the entire plan cache to be cleared.

Side Effect: Plan Cache Flush (4)

- Many server-level configuration changes cause the entire plan cache to be cleared (at the time of RECONFIGURE, not sp_configure)
- Configuration options that could cause the entire cache to be cleared:

<i>Some Versions</i>	<i>Most Versions</i>	<i>SQL Server 2014</i>
cost threshold for parallelism	cross db ownership chaining	access check cache bucket count
max text repl size (B)	index create memory (KB)	access check cache quota
query governor cost limit	max degree of parallelism	clr enabled
remote query timeout (s)	min or max server memory (MB)	max worker threads
	min memory per query (KB)	
	query wait (s)	
	user options	

- Key point: be careful making database-level or server-level configuration changes during production hours
 - Test them in development first
 - sp_configure, then RECONFIGURE – check the cache messages
 - Database changes – check [sys].[dm_exec_procedure_cache]

These next two slides are the two “summary” slides that I discussed when I mentioned troubleshooting performance problems that look like they’re cardinality estimation problems. Yes, they might be statistics BUT it’s always best to check for parameter sniffing FIRST. Then, there are a series of things you can test for cardinality estimation problems – from easy (updating statistics) to more difficult (dealing with skew) to query analysis / rewrites and adding indexes.

These two slides are a summary to the main items to consider and the methodologies.



PASS SUMMIT 2015

Queries Gone Wrong: Statistics, Cardinality, Solutions

DBA-395-P

Tuesday 10/27, 8.30-16.30

Kimberly L. Tripp
President/Founder



Troubleshooting Methodologies

- Is this a CACHED plan method?
 - Stored procedure
 - sp_executesql
- Test to see if the optimal plan varies from the cached plan?

`EXECUTE <procedure> <params>`

`EXECUTE <procedure> <params> WITH RECOMPILE`

NOTE: execute with recompile does NOT allow the *parameterization embedding optimization* [doesn't optimize as effectively as OPTION (RECOMPILE) so there are features that might not be leveraged EXCEPT when using OPTION (RECOMPILE) at the statement level]

- Do you get a different plan for the second?
 - Parameter sensitivity ("sniffing") problem
 - Long term solution is to fix (by changing the code)

Troubleshooting Methodologies

- Is this a poorly performing but NEW (non-cached) plan?
 - AdHoc statement
 - First execution
 - Plan doesn't change when using WITH RECOMPILE
- Use ACTUAL EXECUTION
 - Estimated plan vs. actual plan
 - Be sure to check executions * rows (not just estimated rows vs. actual rows)
 - Problems / solutions that can be exposed by incorrect row estimations
 - Statistics out-of-date -> updating scenarios
 - Estimate is incorrect because of skewed data -> filtered index scenarios
 - Estimate is incorrect because of estimation algorithm -> cardinality estimation scenarios

These next few slides come from the latest version of our IEPTO1 course. And, based on a few of our (IEPTO2) drawings, I've created a few new slides.

SQLskills Immersion Event

IEPTO1: Performance Tuning and Optimization

Module 11: Cardinality Estimation Issues

Kimberly L. Tripp

Kimberly@SQLskills.com



Correlation for Multiple Predicates

(Legacy CE model)

- “Independence” assumption at work...
- Selectivity of conjunctive predicates are computed as the multiplication of individual selectivities

$$p0 * p1 * p2 * p3$$

Correlation for Multiple Predicates

(New CE model)

- Predicates are sorted by selectivity, keeping only the four most selective predicates for use in the calculation
- Successive predicate is then “softened” by taking square roots of next three most selective predicates (at most – four values are used)

$$p0 * \text{sqrt}(p1) * \text{sqrt}(\text{sqrt}(p2)) * \text{sqrt}(\text{sqrt}(\text{sqrt}(p3)))$$

- $p0$ = most-selective predicate
- $p1$ = second most-selective predicate
- ... $p3$ etc...

NOTE: Also known as *exponential back-off*.

Side-by-side CE Example w/Parameters & Variables

```
SELECT ...  
WHERE c1 = @v1 AND c2 = @p1
```

- Selectivity of c1 based on the density_vector (average number of rows for a given c1 value) is 25% or $\frac{1}{4}$
- Selectivity of c2 based on the histogram (parameter sniffing) is $\frac{1}{10}$
- Legacy CE
 - $\frac{1}{4} * \frac{1}{10} = \frac{1}{40}$
- New CE
 - Most selective ($\frac{1}{10}$) times the next most selective with the fraction squared
 - $\frac{1}{10} * \frac{1}{16} = \frac{1}{160}$

Minimum Estimate (Eliminate CE Calculation)

- Trace Flag 4137 introduced in SQL Server 2008 and available through SQL Server 2012
 - See KB: 2658214
FIX: Poor performance when you run a query that contains correlated AND predicates in SQL Server 2008 or in SQL Server 2008 R2 or in SQL Server 2012
 - Not really a “fix” per se but an alternative to how estimates are handled with multiple **conjunctive** predicates (ANDs not ORs)
 - Instead of letting the CE do the estimation, this TF causes SQL Server to use the minimum number (from the two tables) as the estimate (instead of the CE model’s calculation)
- In SQL Server 2014, trace flag 4137 cannot be used unless you’re in a compatibility mode less than 120
- In SQL Server 2014, if you want to use the minimum estimate (for conjunctive predicates) you can use trace flag 9471

Table-valued Parameters and Poor Estimates

- **When a TVP's rowset is estimated (prior to execution), the data set is empty**
 - There is absolutely no way to know how many rows will match (there's no statistics, there's no data, there's just nothing that can be used)
 - SQL Server uses heuristics to estimate rows
 - Legacy CE estimates 1 row
 - New CE reestimates 100 rows
- **Inside of your code, some people (traditionally) have used OPTION (RECOMPILE) to have the statements that use TVPs to get their estimates updated at runtime**
 - Pro: better estimates -> better plans
 - Con: additional CPU to recompile
- **New option (2012SP2+ / 2014CU3+): trace flag 2453 (eliminates recompilation if current value similar to last value)**
 - **New Trace Flag to Fix Table Variable Performance** (<http://bit.ly/1p9Mgzv>)

Cardinality Estimation Models (1 of 2)

- **Legacy CE model: refers to the model introduced with SQL Server 7.0 and used through SQL Server 2012**
 - Will be used in SQL Server 2014 for databases whose compatibility mode is not 120 (120 = SQL Server 2014)
 - 80 = SQL Server 2000
 - 90 = SQL Server 2005
 - 100 = SQL Server 2008 and SQL Server 2008 R2
 - 110 = SQL Server 2012
 - 120 = SQL Server 2014
 - If the database compatibility mode is 120 then you can access the Legacy CE when trace flag 9481 has been supplied
 - Add this to a single query to revert to the Legacy CE for only a specific query
 - **OPTION (QUERYTRACEON 9481)**
 - Run this in a script or a session to test multiple queries more easily
 - **DBCC TRACEON (9481);**

Cardinality Estimation Models (2 of 2)

- **New CE model: refers to the model introduced in SQL Server 2014**
 - Used when the compatibility mode of the database is 120 (SQL Server 2014)
 - Used when the compatibility mode of the database is NOT 120 but trace flag 2312 has been supplied
 - Add this to a single query to test the impact for only a specific query
 - **OPTION (QUERYTRACEON 2312)**
 - Run this in a script or a session to test multiple queries more easily
 - **DBCC TRACEON (2312);**

Migrations / Upgrades / Regressions

■ Option 1 – low impact

- Upgrade existing databases (through backup / restore)
- Leave their existing compatibility level intact
 - Restoring / attaching does not “upgrade” the compatibility level
- When troubleshooting, test trace flag 2312 against queries whose estimates are inaccurate (see if they benefit from the new CE model)
 - If so, leave the QUERYOPTION hint in the specific query that benefits

■ Option 2 – low impact

- Change to compatibility level 120
- Enable trace flag 9481 server-wide

■ Option 3 – high impact

- Change to compatibility level 120
- When troubleshooting, test trace flag 9481 against queries that have regressed
- Use trace flag 9481 for queries that regressed

SQLskills Immersion Event

IEPTO2: Performance Tuning and Optimization

Module 9: Statement Execution, Stored Procedures, and the Plan Cache

Kimberly L. Tripp

Kimberly@SQLskills.com



We had a side discussion on the blocking that you could have with a Sch_M lock. The key point is that ANY lock blocks a SCH_M lock but a SCH_S held from a NOLOCK query that's extremely long running – will cause nasty secondary blocking while the SCH_M waits for ALL locks to drain off (where the SCH_S lock is probably the longest running)

The next slide is from IE1.

We had a side discussion on the blocking that you could have with a Sch_M lock. The key point is that ANY lock blocks a SCH_M lock but a SCH_S held from a NOLOCK query that's extremely long running – will cause nasty secondary blocking while the SCH_M waits for ALL locks to drain off (where the SCH_S lock is probably the longest running) The next slide talks about my script (PreventLongBlockingChains.sql). The following slide is a graphic from IEPTO1.

**Be sure to check out the script:
PreventLongBlockingChains.sql**

Nasty Blocking CHAINS

Relaxed FIFO isn't Always Enough

We had a side discussion on the blocking that you could have with a Sch_M lock. This slide is from IEPTO1.

- Imagine someone's running a NOLOCK query against a very large table (the query takes 4 minutes to run [ok, not that nasty])
- This query's running off-hours but the standard is to use NOLOCK
- An **sp_recompile tname** is requested

Unfortunately now...
everybody waits.....

Relaxed FIFO only let's through the folks that are compatible with ALL pending locks (not just those that are currently held).

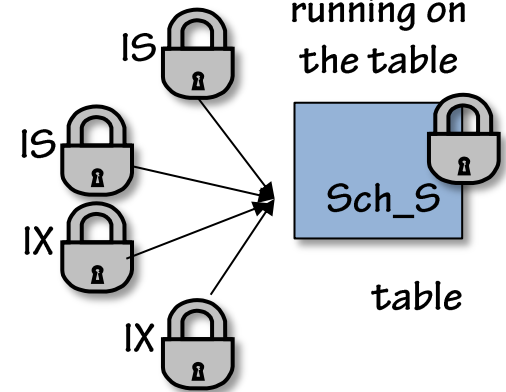
This can create terribly long blocking chains.



Then an
sp_recompile tname
is executed

Other readers
AND writers are
allowed

A NOLOCK
query is
running on
the table



Time

Using EXEC (string) [DSE] and/or sp_ExecuteSql [ES] inside of stored procedures

DSE and ES are not the same.

DSE is unknown until runtime. At that point the statement is treated as though it's adhoc. Because most statements are **not** going to be safe, the statement will end up being recompiled (and optimized) for each execution.

ES is forced parameterization. When a statement is stable you can use this to reduce compilation / CPU costs.

OSFA Dialog (An attempt at ***One Size Fits All*** Programming)

Multipurpose screens/dialog boxes that lead to multipurpose procedures

They looked/sounded good at the time?

But, they're often VERY hard to optimize with where clauses that look like:

```
WHERE (Column1 = @Value1 OR @Value1 IS NULL)
      AND (Column2 = @Value2 OR @Value2 IS NULL)
```

...

Check out the demo scripts in order to fully see how to better optimize this – using `sp_executesql` (for combinations that are stable) and `sp_executesql WITH RECOMPILE` for combinations that could generate different plans.

If you turn on “optimize for ad hoc workloads” then the number of “stubs” that can fit in cache is much higher.

At that point, searching to find whether or not the statement has executed is harder... this is why you want to make sure that you don't let this cache get really large (clearing it at 1 or 2GB).

Options for different executions

If we largely execute with “standard” users and we absolutely want THAT plan to be in cache and to be re-used then we could:

- 1) Use option (optimize FOR...) and use a literal that will generate a plan that’s great for the “typical” scenario. However, the performance won’t be great for the atypical (super user) scenario.
- 2) We could use EXEC for the typical scenario and EXEC proc WITH RECOMPILE for the super user scenario. The benefit of this is:
 - 1) We get a cached plan for the typical user
(no CPU/plan stability)
 - 2) We get a specific plan for the atypical user (the super user) as opposed to a possibly inefficient plan (so, we have to compile but we’re only compiling this one type of plan)

SQL Server “Optimizing the Process of Optimization” Resulting in an Inoptimal Plan

- To simplify the process of optimization, SQL Server optimizes all statements that are optimizable (at compilation)
 - Parameters are known (because they initiated the execution)
 - Literals are known (because they're hard-coded in the logic of the procedure)
 - Variables won't be known until the logic of the procedure is executed (which is after we optimize / compile)
 - Conditional logic / branching is unknown until runtime
- All statements that CAN be optimized, WILL be optimized – based on the information that they can interrogate
 - Parameters and literals can be sniffed (SQL Server can use the histogram)
 - Variables cannot be sniffed (SQL Server has to use the density_vector)

Conditional logic and modularization

This is another way of looking at the same thing from the prior diagram. Really, don't think of the "logic" in your procedure; think only of the procedure as a list of statements (regardless of conditional logic). SQL Server tries to optimize ANY statement that's "*optimizable*" with the parameters that were supplied (regardless of those specific parameters apply for *that* execution). This is another case where you can end up with an inefficient plan because of parameter sniffing.

Conditional Logic example from the demo

When we ran our procedure to get members tied to the input of the name supplied – the conditional logic would execute only ONE branch... but, SQL Server would have already had to optimize BOTH branches

If the first execution was EXEC GetMembers '%e%' then the wildcard statement can be optimized using a table scan. The equality case will know that '%e%' (as a LITERAL name) is HIGHLY selective so there, they'll use an index...

However, if the first execution happens to be 'Tripp' then BOTH statements will be using an index and this will make the first statement inefficient.

The point, you can't control who gets there first and you never want statements to be optimized for parameters that weren't intended for them. **SOLUTION? Modularize your code!**

Conditional logic and modularization

SQL Server will never step into a sub-procedure unless it's executed and in this case it will only be executed with exactly the right parameters.

Parameters are “sniffed”

SQL Server uses parameters to optimize. Sniffing implies that SQL Server uses the histogram to evaluate the data selectivity. For the FIRST execution, parameter sniffing is great. It's the subsequent executions that can suffer from parameter sniffing (and therefore end up with PSP = parameter sniffing problems).

Variables are “unknown”

Variables are only known at runtime as the statements execute and the variables are assigned. As a result, they are unknown during optimization. Without an actual value, SQL Server cannot use the histogram. Instead SQL Server uses the density vector for the “average” numbers of rows that meet that criteria.

SQL Server optimizes everything that's optimizable (sp?) 😊

Again, parameters are known. Parameters can be sniffed. SQL Server will optimize every statement that it can – regardless of what actually ends up executing. At the time of optimization they do not know what will or will not execute and they do not know the value of a variable.

Dynamic String Execution and SQL Injection

To prevent SQL Injection, SQL Server requires that strings execute under the context of the caller.

Dynamic String Execution and Execute As

I have a love/hate relationship with EXECUTE AS. On the one hand, you can give someone rights to something that you wouldn't otherwise be able to give...

However, be careful giving elevated privs around DSE. Generally, my recommendation around DSE is to only create a login-less user with very low privs.

Without Dynamic String Execution permissions flow through the ownership chain

The idea is that when the ownership chain is unbroken then direct permissions (for example, to do a delete) are not necessary. As an owner you are giving rights to a process; this procedure can be protected with more business rules/logic and even reduced/isolated to specific rows. The end user does NOT need direct delete permissions to be able to execute the procedure as long as they've been given execute rights on the sproc.

Dynamic String Execution, EXECUTE AS and SQL Injection

People complained that granting permissions directly to the user was a problem... so, they wanted an alternative. Enter: EXECUTE AS.

While it solves the problem of permissions, it adds more potential for SQLInjection.

So..... Check out these posts:

[Little Bobby Tables, SQL Injection and EXECUTE AS](#)

["EXECUTE AS" and an important update your DDL Triggers \(for auditing or prevention\)](#)

Stable plans vs. unstable plans – how do you know?

- You could test different combinations using execute with recompile.
- You might know because of selectivity?

Options that better balancing recompiling too much vs. not recompiling enough

This is from the multipurpose procedure demo. There are 3 parameters and 7 possible combinations that can be submitted. Instead of adding `OPTION (RECOMPILE)` to the statement (which doesn't always work [remember, it has a very checkered past]) you can build the statement dynamically and then programmatically (by setting a `RECOMPILE` flag to 0 or 1) decide whether or not `OPTION(RECOMPILE)` should be added to the dynamically constructed statement. Then, you can use `sp_executesql` to place ALL seven statements in cache. Those without the `OPTION(RECOMPILE)` clause will be cached. Those with `OPTION (RECOMPILE)` will get a plan on each execution.

How many plans COULD be generated?

When you're testing your procedures you might have 15 test cases. Executing each of the procedures (with the 15 different combinations of parameters) WITH RECOMPILE. Then, review the number of different plans

If there are 15 different plans for 15 different executions – Use OPTION (RECOMPILE)

If there are 2 primary plans that are fairly split – see if you can create subprocedures and then control the behavior that way (especially if they either don't need to recompile OR only one needs to recompile).

If there are only 2 plans where one is 14 of the executions (and, is the norm) then you might use OPTION (OPTIMIZE FOR ...) but it depends on how bad the performance is the for 15th. Can you control that through [other] parameters or programmatically?

How many plans COULD be generated?

This is another way to describe what was on the prior page...

These are the pros/cons of the different procedure (and statements within) strategies:

The default

- Prone to parameter sniffing problems
- Possibly HORRIBLE plans for subsequent executions
- No guarantee of what plan actually gets into cache
- + Less CPU (not always though – you do save compilation time but if the execution is really expensive then you might lose all of the gains of having compiled/saved a plan)

Forcing a recompile for EVERY execution

- More CPU
- + Plan specific to those parameters

Optimize for UNKNOWN

- Guarantees an AVERAGE plan
- + Works well when the data is evenly distributed (but, the problems are less likely as well – but, if plans were getting into cache from atypical values then this might solve the problem)

SQLskills Immersion Event

IEPTO2: Performance Tuning and Optimization

Module 10: Index Analysis

Kimberly L. Tripp
Kimberly@SQLskills.com



Index 1

Index 2

Index 3

Should you drop redundant indexes? It depends. 😊

It's TRUE that query that uses a left-based subset of one index (let's say that a query uses index 2) that is COULD use a wider version of the same (left-based) index. It could use Index 1 instead of 2.

The problem is that there could be specific uses that warrant the smaller version. Questions to ask: Is the smaller version used by scans? Are they executed VERY frequently? How important are they?

Index consolidation

No production environment can support THE BEST index for every query, you'd have too many indexes. To be really successful with indexing – you need to find a balance. This includes consolidating some of your index recommendations to have one index handle multiple purposes!

Consolidation?

You can CONSIDER removing indexes that are left-based subsets of others (in terms of the key). However, be sure to preserve the key as that's used for navigation/seeking. The columns that are included can be combined in any order.

Index Health – Checking for fragmentation

To get the value: **avg_fragmentation_in_percent** for **sys.dm_db_index_physical_stats**, SQL Server uses a scan of the first intermediate level (index level 1). All of the modes (limited, sampled, and detailed), use this same method. Then, SAMPLED and DETAILED do additional work (on the leaf level of the index) to give you things like average page density, etc. If your analysis/rebuild scripts only use **avg_fragmentation_in_percent** to determine fragmentation then you should only do a LIMITED mode scan. SAMPLED and DETAILED offer more page-specific details (page density, forwarding pointers, etc.) and if you're NOT using those during your automated maintenance process (and most people aren't) don't waste time!

Adding more indexes

Here we were discussing the order of columns. If there are range-based predicates than it doesn't usually matter which column is second. You tend to want to put the most selective first (in terms of the PREDICATES supplied). If all of the predicates are equality-based then it doesn't really matter. You're best ordering them by the LEFT-based combination that's used most commonly.

Multi-column, column-level statistics

Only DTA recommends multi-column, column-level statistics...

If you are about to trust (and create) a “green hint” recommendation (which comes from the Missing Index DMVs) then you might want to first run the query through DTA. If you end up creating the index that DTA recommends then you should also create the recommended stats (for that same table). These statistics can be helpful to the optimizer to better understand non-left-based subsets of the index for other possible uses.

This was around our discussion about the limitation of SQL Server's knowledge of the data with gaps in the step values.

If the step describes the range from 101 to 200 (not including those values)

- There are 250K rows over that range
- There are 5 distinct values

The average will show 50K and EVERY value supplied between 101 and 199 will get an estimate of 50K. There's no way for SQL Server to know WHICH values actually exist.