<http://sqlmag.com/blog/forced-parameterization-when-should-i-use-it>

[SQL Server Questions Answered](http://sqlmag.com/blog/sql-server-questions-answered)

# Forced parameterization–when should I use it?

May 12, 2011 by [Paul S. Randal](http://sqlmag.com/blogger/paul-s-randal) in [SQL Server Questions Answered](http://sqlmag.com/blog/sql-server-questions-answered)

* [RSS](http://sqlmag.com/blog/10118/feed)
  + [EMAIL](http://sqlmag.com/forward?path=node%2F10237)
* [in**Share**](javascript:void(0);)
* [Tweet](http://twitter.com/share)

[Comments](http://sqlmag.com/blog/forced-parameterization-when-should-i-use-it#comments) [**1**](http://sqlmag.com/blog/forced-parameterization-when-should-i-use-it#comments)

**QUESTION:** When it is a appropriate to turn on the database option FORCED parameterization?

**ANSWER:** Before I give a specific answer, I’d like to start with a few cautionary words. First, I don’t often recommend using this database option. So, I’d really caution you to do your homework and analyze your database characteristics to ensure that this option is desirable. Second, if you do turn this on – I’d also make sure that you further analyze the impact in production so that you can switch back if necessary. However, if you find that there are a few statements that break the mold (either a few that NEED forced parameterization OR a few that do not (and should stay with SIMPLE) then you can also use these settlings more granularly using plan guides.

**Related:** [Controlling Parameterization](http://sqlmag.com/database-performance-tuning/controlling-parameterization)

Let’s start with the general concepts of statement execution. When you submit a SQL statement, SQL Server goes through multiple phases (which are simplified here):

**(1) Parse –** if SQL Server doesn’t know what you want – they can’t do much with your statement.

**(2) Standardize/normalize/algebrize –** This phase puts the statement into a format the optimizer can understand and where it can begin to break down the number of tables and types of operators. It’s also where they “expand views” to their base tables, where they expand \* to the list of all columns and where they might be able to remove redundancies exposed in the query. If I’m being honest, the people that care the most about this phase all work in Redmond.

**(3) Optimization –** This is the phase that we care about the most. This is the phase that we want to help the most (to come up with a good plan) but that we also don’t want to have to do repeatedly because it can be costly (especially in terms of CPU). So, while this is an important phase we need to see if we can reduce it.

**(4) Compilation –** One a plan is defined (and often this actually means two plans – a serial plan and a parallel plan) SQL Server can put this plan into cache and POTENTIALLY reuse it saving the cost of optimization again. However, here’s where things are tricky. For a plan to be good to reuse, SQL Server needs to deem that statement as “safe.” Unfortunately, the rules for a statement being safe are pretty strict. In my experience, most statements aren’t safe. As a result, each execution causes the statement to go through each of these phases – again and again and again.

**(5) Execution**… (do I really need to explain this one?)

The worst part about the statements going through this process again and again is that each execution costs you in terms of CPU and memory. Additionally, each statement shows up in cache individually. Prior to [SQL Server 2008](http://www.sqlmag.com/sql-server/sql-server-2008), there were no easy ways to figure out if a particular statement was being executed repeatedly (just with different parameters). What you really wanted to find was a “class of query” and the only way to do that was with [PSS’s RML utilities](http://blogs.msdn.com/b/psssql/archive/2007/12/18/rml-utilities-for-microsoft-sql-server-released.aspx), third party tools (like [ClearTrace](http://www.scalesql.com/cleartrace/)) or “rolling your own” by tracing statements to a table and then running [sp\_get\_query\_template](http://technet.microsoft.com/en-us/library/ms186908.aspx) on the statements to parameterize them and then aggregate over the text. The point is that finding a statement that is costly not because of the cost of each individual execution but instead the cumulative cost – was extremely challenging. In SQL Server 2008 you can find and tune similar queries using the new query\_hash and query\_plan\_hash which are available in the sys.dm\_exec\_query\_stats DMV. In fact, there’s an excellent topic in the BOL titled: [Finding and Tuning Similar Queries by Using Query and Query Plan Hashes](http://msdn.microsoft.com/en-us/library/cc645887.aspx).

OK, so what does this have to do with FORCED parameterization? If you analyze your plan cache and end up finding that you have a lot of “single-use plans” but that many of those statements are actually the same – just with different parameters, then you might be able to take advantage of forcing SQL Server to parameterize. SQL Server has a lot of rules that determine if your statements are “safe.” And, while these rules are important you might actually have an environment where your statements actually are safe but just not in terms of SQL Server’s definition. What you need to analyze is how many plans exist for each query\_hash. If most of your query\_hash statements all have the same query\_plan\_hash then these “unsafe” statements are actually safe (but SQL Server just doesn’t know it). By telling SQL Server to force the parameterization you are making SQL Server define and then reuse a plan. If this is the case then you might be able to significantly reduce optimization/compilation time.

However, if one or two statements are NOT safe and you force SQL Server to parameterize and reuse the plan that gets into cache first, then you can create very unpredictable behaviors. Sometimes a good plan gets into cache and everything seems to run well. However, occasionally the first person that executes after the plan has fallen out of cache happens to execute with non-standard values which cause an uncommon plan to be placed in cache. While that plan is probably good for that particular execution, it might be horribly bad for other users. This happens with stored procedures all the time but now will treat every statement the same way.

To help you better understand plan caching, you might want to check out my blog post category called [Optimizing Procedural Code](http://www.sqlskills.com/BLOGS/KIMBERLY/category/Optimizing-Procedural-Code.aspx). Above all, realize that “one-size-fits-all” options can be very problematic. And, if you find that FORCED parameterization works really well except for a few statements then you can consider using Plan Guides for the statements that should use SIMPLE parameterization. For example,

EXEC sp\_create\_plan\_guide  
    'Plan Guide Name&rsquo;,  
    @UNSafeQuery,  
    N'TEMPLATE',  
    NULL,  
    @Parameters,  
    N'OPTION(PARAMETERIZATION SIMPLE)';

You really have quite a few options for dealing with statement caching but understanding your workload and properly tuning it takes time, analysis and adjustments. It’s really never finished but you can get closer and closer to stabilizing your environment but taking the time to get to know and understand your workloads.