

`

Plan Caching and Recompilation in SQL Server 2012

SQL Server Technical Article

**Writer:** Greg Low, [SQL Down Under](http://www.sqldownunder.com/)

**Technical Reviewer:** Paul White, Andrew Kelly, Rubén Garrigós, Kalen Delaney, Leigh Stewart, Jack Li

**Published:** March 2013

**Applies to:** SQL Server 2012

**Summary:** This paper explains how SQL Server 2012 allocates memory for plan caching, how query batches are cached and suggests best practices on maximizing reuse of cached plans. It also explains scenarios in which batches are recompiled, and gives best practices for reducing or eliminating unnecessary recompilations and for minimizing plan cache pollution.

Copyright

This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2013 Microsoft. All rights reserved.

Contents

[Introduction 6](#_Toc351106497)

[Definitions 6](#_Toc351106498)

[Query Batch, Script, Batch Terminator 6](#_Toc351106499)

[Plan Cache 7](#_Toc351106500)

[Compilation and Recompilation 7](#_Toc351106501)

[Plan Caching Core Concepts 8](#_Toc351106502)

[Memory Allocated To Plan Caching 8](#_Toc351106503)

[Statement Level Recompilation 9](#_Toc351106504)

[Query Batch Types and Plan Caching 9](#_Toc351106505)

[Ad-hoc queries 9](#_Toc351106506)

[Auto-parameterized queries 10](#_Toc351106507)

[sp\_executesql Procedure 12](#_Toc351106508)

[Prepared Queries 12](#_Toc351106509)

[Stored procedures (Including Triggers) 12](#_Toc351106510)

[Executing queries via EXEC ( …) 13](#_Toc351106511)

[Multiple Levels of Caching 13](#_Toc351106512)

[Execution Contexts 14](#_Toc351106513)

[Mode Changing SET Options (SHOWPLAN-Related and Others) 16](#_Toc351106514)

[Costs associated with query plans and execution contexts 16](#_Toc351106515)

[Achieving Good SQL Server Batch Execution Performance 17](#_Toc351106516)

[Query Plan Reuse Cache-Ability 17](#_Toc351106517)

[Object Name Resolution 17](#_Toc351106518)

[Trigger-Related Issues 18](#_Toc351106519)

[Other Common Non-Cache-Ability Issues 18](#_Toc351106520)

[Plan Reuse-Affecting SET Options 19](#_Toc351106521)

[CREATE WITH RECOMPILE 20](#_Toc351106522)

[EXEC WITH RECOMPILE 20](#_Toc351106523)

[Temporary Objects 20](#_Toc351106524)

[CREATE/UPDATE STATISTICS 21](#_Toc351106525)

[Avoiding Recompilations 21](#_Toc351106526)

[Plan Stability 21](#_Toc351106527)

[Schema Changes 21](#_Toc351106528)

[sp\_recompile 21](#_Toc351106529)

[Flush Entire Plan Cache 22](#_Toc351106530)

[Flush Database-Related Entries in Plan Cache 22](#_Toc351106531)

[CHECK OPTION 22](#_Toc351106532)

[DBCC CHECKDB 22](#_Toc351106533)

[SNAPSHOT Isolation Level 23](#_Toc351106534)

[DBCC FREESYSTEMCACHE 23](#_Toc351106535)

[SET Options 23](#_Toc351106536)

[Plan Optimality 23](#_Toc351106537)

[Trivial Plan 23](#_Toc351106538)

[KEEPFIXED PLAN 23](#_Toc351106539)

[Read-Only Filegroups 24](#_Toc351106540)

[Query Compilation 24](#_Toc351106541)

[Tracking Data Modifications 25](#_Toc351106542)

[Interesting Statistics 26](#_Toc351106543)

[Recompilation Threshold (RT) 27](#_Toc351106544)

[ColModCtr 27](#_Toc351106545)

[Tracking Changes Using colmodctr 28](#_Toc351106546)

[Special Cases 29](#_Toc351106547)

[Special case 1: Statistics created on an empty table or indexed view 29](#_Toc351106548)

[Special case 2: Trigger recompilations 29](#_Toc351106549)

[Best Practices for Achieving Plan Optimality Stability 30](#_Toc351106550)

[Table Variables 30](#_Toc351106551)

[KEEP PLAN 30](#_Toc351106552)

[KEEPFIXED PLAN 30](#_Toc351106553)

[LEAVE AUTOUPDATE STATISTICS ENABLED 31](#_Toc351106554)

[Parameter Sniffing 31](#_Toc351106555)

[Parameter Sniffing Described 31](#_Toc351106556)

[OPTIMIZE FOR Query Hint 31](#_Toc351106557)

[OPTIMIZE FOR UNKNOWN 33](#_Toc351106558)

[TRACE FLAG 4136 33](#_Toc351106559)

[Identifying Recompilations 33](#_Toc351106560)

[Tracing Recompilation Activity 33](#_Toc351106561)

[Recompilation Reasons 36](#_Toc351106562)

[Tools and Commands 38](#_Toc351106563)

[Dynamic Management Objects 38](#_Toc351106564)

[sys.syscacheobjects virtual table 39](#_Toc351106565)

[DBCC FREEPROCCACHE 39](#_Toc351106566)

[DBCC FLUSHPROCINDB( db\_id ) 39](#_Toc351106567)

[DBCC FREESYSTEMCACHE(cache[,resource pool]) 39](#_Toc351106568)

[Extended Events (SQL Server 2008 and later versions) 40](#_Toc351106569)

[Trace Events 40](#_Toc351106570)

[PerfMon Counters 40](#_Toc351106571)

[Memory Cache Counters 41](#_Toc351106572)

[Plan Cache Pollution Issues 41](#_Toc351106573)

[Plan Cache Pollution Described 41](#_Toc351106574)

[Avoiding Plan Cache Pollution 41](#_Toc351106575)

[FORCED PARAMETERIZATION 42](#_Toc351106576)

[OPTIMIZE FOR ADHOC WORKLOADS 42](#_Toc351106577)

[DBCC FREESYSTEMCACHE 42](#_Toc351106578)

[Conclusion 42](#_Toc351106579)

[Appendix A: When Does SQL Server Not Auto-Parameterize Queries? 43](#_Toc351106580)

# Introduction

There are several goals of this white paper. This paper explains how SQL Server allocates memory for plan caching, how query batches are cached and suggests best practices on maximizing reuse of cached plans. It also explains scenarios in which batches are recompiled, and gives best practices for reducing or eliminating unnecessary recompilations and for minimizing plan cache pollution. The white paper explains SQL Server's statement-level recompilation feature (first introduced in SQL Server 2005) and many tools and utilities that are useful as observation tools in the processes of query compilation, query recompilation, plan caching, and plan reuse.

All material in this white paper applies equally to SQL Server 2005, SQL Server 2008, and SQL Server 2012 apart from those areas that are explicitly pointed out as applying to a particular version. You can find previous versions of this whitepaper at:

|  |  |  |
| --- | --- | --- |
| **SQL Server Version** | **Whitepaper** | **Location** |
| 2005 | "Batch Compilation, Recompilation and Plan Caching Issues in SQL Server 2005” by Arun Marathe and Shu Scott | <http://technet.microsoft.com/library/Cc966425> |
| 2008 | “Plan Caching in SQL Server 2008” by Greg Low | <http://msdn.microsoft.com/library/ee343986(v=SQL.100).aspx> |

This paper targets these audiences:

Persons who use, maintain, and develop applications for SQL Server. Those who are new to SQL Server 2012 and those that are migrating from earlier versions of SQL Server will find useful information here. SQL Server developers will find useful background information here.

# Definitions

## Query Batch, Script, Batch Terminator

A query batch is a set of Transact-SQL statements that are submitted to SQL Server for execution in a single call. A batch can contain one or more statements such as the following:

* SELECT, INSERT, UPDATE, DELETE, and MERGE statements
* Stored procedure calls (EXEC)
* Transact-SQL control structures such as SET, IF, WHILE, DECLARE
* DDL statements such as CREATE, DROP, and ALTER
* DCL statements such as GRANT, DENY, and REVOKE

A script is a set of one of more query batches, separated by a batch terminator. The most common batch terminator is the word “GO”. It is important to understand that the batch separator is not a Transact-SQL statement and has no meaning to the database engine. For example, consider the following script:

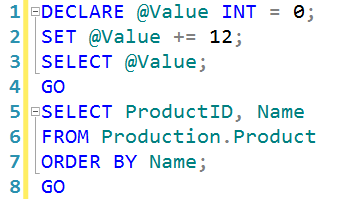


Figure 1: Script

The script contains two batches. When this script is executed within a client tool such as SQL Server Management Studio (SSMS), it might appear that the entire script is sent to the server for execution but this is not the case. SSMS divides the script into batches by locating the word GO. Lines 1 to 3 are sent to the database engine to be executed. When execution completes, lines 5 to 7 are sent to the database engine to be executed.

## Plan Cache

Before a query batch begins execution on SQL Server, the batch is compiled into a plan. The plan is then executed for its effects or to produce results.

The compilation of execution plans is a relatively expensive operation so an attempt is made to avoid these costs by caching the compiled plans in a SQL Server memory region called the Plan Cache. When another query batch needs to be executed, SQL Server searches the Plan Cache for possible plan reuse opportunities. If plan reuse is achieved, the compilation costs are avoided.

Note that in earlier SQL Server literature, the word "Procedure Cache" was often used to describe the Plan Cache. While the term “Procedure Cache” was historically accurate, the term "Plan cache" is now considered to be more accurate because the Plan Cache stores plans for more than just stored procedures. As an example, since SQL Server 2000, plans for some ad-hoc query batches have been stored.

## Compilation and Recompilation

Compilation is the process of creating a compiled plan from a query batch.

When the database engine begins execution of a compiled plan, it first checks that the plan is still valid and optimal. If either of these checks fail, the statement corresponding to the query plan or the entire batch is compiled again. Such compilations are known as "recompilations."

Note in particular that the query plans for the batch need not have been cached. Indeed, some types of batches are never cached, but can still cause recompilations. Take, for example, a batch that contains a literal larger than 8000 bytes. Suppose that this batch creates a temporary table, and then inserts 20 rows in that table. The insertion of the seventh row will cause a recompilation, but because of the large literal, the batch is not cached.

Recompilations in SQL Server are performed for good reasons. Some of them are necessary to ensure statement validity (or correctness); others are performed in an attempt to obtain better query execution plans as data in a database changes over time. Recompilations are only an issue if they happen too frequently and slow down batch executions considerably. If that happens, it becomes necessary to reduce how often they occur.

# Plan Caching Core Concepts

## Memory Allocated To Plan Caching

Most memory used by SQL Server is allocated to the Buffer Pool, which is used to store database pages. SQL Server allocates a portion of this memory for use in caching query plans. The overall amount of memory available to SQL Server depends upon the amount of memory installed on the server, the architecture of the server, the version and edition of SQL Server and the amount of memory pressure being experienced by SQL Server. This pressure can be internal (SQL Server resources need memory) or external (operating system needs memory). SQL Server is designed to respond to memory pressure when necessary. For example, it can trim the contents of the plan cache to reduce memory usage.

The Plan Cache comprises four main cache stores: Object Plans, SQL Plans, Bound Trees and Extended Stored Procedures. For details of the structure of the plan cache and of the types of objects that are cached, review the following article: <http://blogs.msdn.com/b/sqlprogrammability/archive/2007/01/09/1-0-structure-of-the-plan-cache-and-types-of-cached-objects.aspx>.

SQL Server decides the appropriate allocation of memory to the Plan Cache from the Buffer Pool. The algorithm used for this has been improved in successive service packs since SQL Server 2005 was introduced.

|  |  |
| --- | --- |
| SQL Server Version | Cache Pressure Limit |
| SQL Server 2012, SQL Server 2008 and SQL Server 2005 SP2 | 75% of visible target memory from 0-4GB + 10% of visible target memory from 4Gb-64GB + 5% of visible target memory > 64GB |
| SQL Server 2005 RTM and SQL Server 2005 SP1 | 75% of visible target memory from 0-8GB + 50% of visible target memory from 8Gb-64GB + 25% of visible target memory > 64GB |
| SQL Server 2000 | SQL Server 2000 4GB upper cap on the plan cache |

Table 1: Plan Cache memory allocation by SQL Server version

Prior to SQL Server 2012, 32-bit systems might have used AWE (Address Window Extensions) memory to extend the available memory beyond the 4GB virtual address space limit of the 32-bit architecture, but that additional memory could only be used for data pages in the Buffer Pool, not for pages in the Plan Cache. It was not considered “visible” memory. The use of AWE memory is discontinued in SQL Server 2012. Note that no such limitation applies to 64-bit systems.

## Statement Level Recompilation

Unlike versions of SQL Server prior to SQL Server 2005, SQL Server can compile statements rather than needing to compile entire batches. Statement-level recompilation minimizes CPU time and memory during batch recompilations, and causes fewer compile locks to be acquired. It avoids the need to break long stored procedures into multiple short stored procedures just to reduce the recompilation penalty of the long stored procedures.

## Query Batch Types and Plan Caching

If a query plan is not cached, its reuse opportunity is zero. Such a plan will be compiled every time it is executed, potentially resulting in poor performance. In some cases, non-caching is a desirable outcome. Examples of these cases are provided later in this whitepaper.

SQL Server can cache query plans for many types of batches. A list of these types if provided here along with details of the necessary conditions for plan reuse for each type. Note that even though these conditions are necessary for plan reuse, they may not be sufficient for plan reuse to occur.

### Ad-hoc queries

An ad-hoc query is a batch that contains one or more SELECT, INSERT, UPDATE, DELETE, or MERGE statements, does not have parameters and is not pre-prepared.

SQL Server requires an exact text match for two ad-hoc queries before reuse can occur. The text match is both case- and space-sensitive, even on case-insensitive servers. For example, the following two queries do not share the same query plan (unless FORCED PARAMETERIZATION is ON):

SELECTProductID

FROMSales.SalesOrderDetail

GROUPBYProductID

HAVINGAVG(OrderQty)>5

ORDERBYProductID;

SELECTproductid

FROMSales.SalesOrderDetail

GROUPBYProductID

HAVINGAVG(OrderQty)>5

ORDERBYProductId;

### Auto-parameterized queries

For certain queries, SQL Server replaces constant literal values before compiling a query plan. If a subsequent query differs in only the values of the constants, it will match against the auto-parameterized query. In general, SQL Server auto-parameterizes those queries whose parameterized form of the query would result in a trivial plan. A trivial plan exists when the query optimizer determines that only one plan is possible. Appendix A contains a list of statement types for which SQL Server does not auto-parameterize.  
As an example of auto-parameterization, the following two queries can reuse a query plan:

SELECTProductID,SalesOrderID

FROMSales.SalesOrderDetail

WHEREProductID>100

ORDERBYProductID;

SELECTProductID,SalesOrderID

FROMSales.SalesOrderDetail

WHEREProductID>200

ORDERBYProductID;

The auto-parameterized form of the above queries is:

SELECT[ProductID],[SalesOrderID]

FROM[Sales].[SalesOrderDetail]

WHERE[ProductID]>@1

ORDERBY[ProductID]ASC

When values of constant literals appearing in a query can influence a query plan, the query is not auto-parameterized. Query plans for such queries are cached, but with the original literal values, not with placeholders such as @1.

SQL Server's SHOWPLAN feature can be used to determine whether auto-parameterization was attempted. For example, the query can be submitted after executing SET SHOWPLAN\_XML ON. If the resulting plan contains such placeholders as @1 and @2, then the query has been auto-parameterized; otherwise not.

Showplans in XML format also contain information about values of parameters at both compile-time ('SHOWPLAN\_XML' and 'STATISTICS XML' modes) and execution-time (' STATISTICS XML ' mode only). To see this using SQL Server Management Studio, right click on the execution plan, and select “Show Execution Plan XML…” and look for the “ParameterList” tag, where you will see the attributes“ParameterCompiledValue”, and “ParameterRuntimeValue”. In SQL Server 2008 and later versions, an additional “Parameterized Text” attribute is available in the "Statement" node in the returned XML plan.

Not all auto-parameterization operations are considered “safe”. Only safe auto-parameterization offers general opportunities for plan reuse with different parameter values. The following query demonstrates an example of an unsafe auto-parameterization:

SELECTa.AddressID

FROMPerson.AddressASa

WHEREa.City=N'Seattle';

The resulting query plan shows that auto-parameterization was attempted, but that cost-based optimization was required (Optimization Level = FULL). Since there are cost-based plan choices, the auto-parameterization attempt was considered unsafe, and only a shell ad-hoc plan is cached, not the Prepared parameterized form. The next query shows the plan cache contents for this query:

WITHXMLNAMESPACES

(

DEFAULT

'http://schemas.microsoft.com/sqlserver/2004/07/showplan'

)

SELECTdeqp.query\_plan.value(

'(//StmtSimple)[1]/@ParameterizedPlanHandle','nvarchar(64)')

ASParameterizedPlanHandle,

deqp.query\_plan.value(

'(//StmtSimple)[1]/@ParameterizedText','nvarchar(max)')

ASParameterizedText,

deqp.query\_plan,

decp.cacheobjtype,

decp.objtype,

decp.plan\_handle,

dest.[text],

decp.refcounts,

decp.usecounts

FROMsys.dm\_exec\_cached\_plansASdecp

CROSSAPPLYsys.dm\_exec\_sql\_text(decp.plan\_handle)ASdest

CROSSAPPLYsys.dm\_exec\_query\_plan(decp.plan\_handle)ASdeqp

WHEREdest.[text]LIKEN'%Address%'

ANDdest.[text]NOTLIKEN'%sys.dm\_exec\_cached\_plans%';

The ParameterizedPlanHandle attribute (added in SQL Server 2008) is NULL, and there is no prepared cached plan, only the ad-hoc stub. If an index is created on the City column, the original query will produce a TRIVIAL plan with Safe auto-parameterization, and a cached prepared plan.

For greater detail on auto-parameterization in query plans, review the following article: <http://blogs.msdn.com/b/sqlprogrammability/archive/2007/01/11/4-0-query-parameterization.aspx>.

### sp\_executesql Procedure

When using sp\_executesql, a user or an application explicitly identifies the parameters. This assists is obtaining plan reuseFor example:

DECLARE@SqlCommandnvarchar(500)

=N'SELECT p.ProductID, p.Name, p.ProductNumber

FROM Production.Product AS p

INNER JOIN Production.ProductDescription AS pd

ON p.ProductID = pd.ProductDescriptionID

WHERE p.ProductID = @ProductID';

DECLARE@ParameterDefinitionnvarchar(100)=N'@ProductID int';

DECLARE@ProductToLocateint;

SET@ProductToLocate=4;

EXECsp\_executesql @SqlCommand,@ParameterDefinition,

@ProductID=@ProductToLocate;

SET@ProductToLocate=320;

EXECsp\_executesql @SqlCommand,@ParameterDefinition,

@ProductID=@ProductToLocate;

The plan reuse opportunities are determined by the match of the query-text (the first argument after sp\_executesql). The parameter values (4 and 320) are not considered as part of the text matches. In the preceding example, plan reuse can happen for the two sp\_executesql statements.

### Prepared Queries

This method — which is similar to the sp\_executesql method— also promotes query plan reuse. The batch text is sent once at the "prepare" time. SQL Server responds by returning a handle that can be used to invoke the batch at execute time. At execute time, a handle and the parameter values are sent to the server. ODBC and OLE DB expose this functionality via SQLPrepare/SQLExecuteand ICommandPrepare. For example, a code snippet using ODBC might look like:

SQLPrepare(hstmt,"SELECT SalesOrderID, SUM(LineTotal) AS SubTotal

FROM Sales.SalesOrderDetail sod

WHERE SalesOrderID < ?

GROUP BY SalesOrderID

ORDER BY SalesOrderID",SQL\_NTS)

SQLExecute(hstmt)

### Stored procedures (Including Triggers)

Stored procedures are designed to provide a layer of functional abstraction and to promote plan reuse. The plan reuse is based on the stored procedure or trigger name (even though it is not possible to call a trigger directly). Internally, SQL Server converts the name of the stored procedure to an ID, and subsequent plan reuse happens based on the value of that ID. The plan caching and recompilation behavior of triggers differs slightly from that of stored procedures and is identified at appropriate points in this discussion.  
  
When a stored procedure is compiled for the first time (or in fact any parameterized batch), the values of the parameters that were supplied with the execution call are used to optimize the statements within that stored procedure. This process is known as "parameter sniffing." If these values are typical, then most calls to that stored procedure will benefit from an efficient query plan. However, if the parameter values were not typical, reuse of an existing plan is not desirable. We discuss techniques that can be used to prevent caching of query plans with atypical stored procedure parameter values later in this whitepaper.

### Executing queries via EXEC ( …)

SQL Server can cache strings submitted via EXEC for execution. These queries are commonly referred to as "dynamic SQL." For example:

EXEC ('SELECT \*'

+' FROM Production.Product AS pr'

+' INNER JOIN Production.ProductPhoto AS ph'

+' ON pr.ProductID = ph.ProductPhotoID'

+' WHERE pr.MakeFlag = '+@mkflag);

For these queries, plan reuse is based on the final string that is executed, regardless of how it was created. In this example, different values for the variable @mkflag would result in different query plans.

## Multiple Levels of Caching

It is important to understand that cache matches at multiple levels happen independently of one another. Consider the following example:

Suppose that Batch 1 (not a stored procedure) contains the following statement (among others):

EXEC dbo.procA;

Batch 2 (also, not a stored procedure) does not text-match with Batch 1, but contains the exact "EXEC dbo.procA;" referring to the same stored procedure.

In this case, query plans for Batch 1 and Batch 2 do not match. Nevertheless, whenever "EXEC dbo.procA;" is executed in one of the two batches, a possibility for query plan reuse for procA exists if the other batch has executed prior to the current batch, and if the query plan for procA still exists in the plan cache.

The same type of reuse may happen even if dynamic SQL is executed using EXEC, or if an auto-parameterized statement is executed inside Batch 1 and Batch 2. The following types of batches start their own "levels" in which cache matches can happen irrespective of whether a cache match happened at any of the containing levels:

* Stored procedure executions such as "EXEC dbo.stored\_proc\_name;"
* Dynamic SQL executions such as "EXEC query\_string;"
* Auto-parameterized queries
* Query plans and execution contexts

## Execution Contexts

When a cache-able batch is submitted to SQL Server for execution, it is compiled and a query plan for it is put in the plan cache. Query plans are read-only re-entrant structures that are shared by multiple users. While the optimizer may consider both serial and parallel alternatives, one plan is chosen and cached, based on cost. Parallel plans are common for all of the degrees of parallelism. (Strictly speaking, if two identical queries posed by the same user using two different sessions with the same session options arrive at a SQL Server simultaneously, two query plans exists while they execute. However, at the end of their executions, a plan for only one of them is retained in the plan cache).

From a query plan, an execution context is derived. Execution contexts hold the values that are needed for a specific execution of a query plan. Execution contexts are also cached and reused. Each user concurrently executing a batch will have an execution context that holds data (such as parameter values) specific to their execution. Although execution contexts are reused, they are not re-entrant (i.e., they are single-threaded). That is, at any point of time, an execution context can be executing only one batch submitted by a session, and while the execution is happening, the context is not given to any other session or user.

The relationships between a query plan and the execution contexts derived from it are shown in the following diagram. There is one query plan, and three execution contexts are derived from it. The execution contexts contain parameter values and user-specific information. The query plan does not hold parameter values or user-specific information.

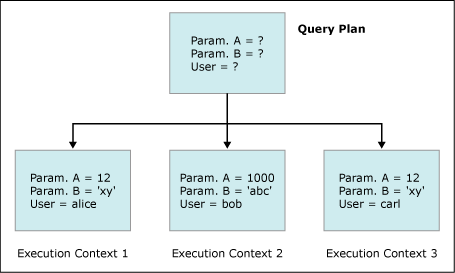


Figure 2: Query plan and execution context relationship

A query plan and multiple associated execution contexts can coexist in Plan Cache. An execution context (without an associated query plan) cannot exist in the Plan Cache. Whenever a query plan is removed from Plan Cache, all of the associated execution contexts are also removed along with it. When Plan Cache is searched for possible plan reuse opportunities, the comparisons are against query plans, not against execution contexts. Once a reusable query plan is found, an available execution context is located (causing execution context reuse) or freshly generated. Query plan reuse does not necessarily imply execution context reuse.

Execution contexts are derived "on the fly". Before a batch execution begins, a skeleton execution context is generated. As execution proceeds, the necessary execution context pieces are generated and put into the skeleton. This means that two execution contexts need not be identical even after user-specific information and query parameters are deleted from them. Because structures of execution contexts derived from the same query plan can differ from one another, the execution context used for a particular execution has slight impact on performance. The impact of the differences diminishes over time as the plan cache gets "hot" and as a steady state is reached.

Example: Suppose that a batch B contains an IF statement. When B begins execution, an execution context for B is generated. Suppose that during this first execution, the TRUE branch of the IF is taken. Further, suppose that B was submitted again by another connection during the first execution. Because the only execution context existing at that moment was in use, a second execution context is generated and given to the second connection. Suppose that the second execution context takes the FALSE branch of the IF. After both executions complete, B is submitted by a third connection. Supposing that the third execution of B chooses the TRUE branch, the execution will complete slightly faster if SQL Server chose the first execution context of B for that connection rather than the second execution context.

Execution contexts of a batch S can be reused even if the calling sequence of S differs. For example, one calling sequence could be "stored proc 1 --> stored proc 2 --> S", whereas a second calling sequence could be "stored proc 3 --> S". The execution context for the first execution of S can be reused for the second execution of S.

If a batch execution generates an error of severity 11 or higher, the execution context is destroyed. Thus, even in the absence of memory pressure — which can cause Plan Cache to shrink— the number of execution contexts (for a given query plan) cached in Plan Cache can go up and down.

If a parallel query plan is compiled, SQL Server may choose to derive a serial execution context from the parallel plan at execution time, by disabling parallelism-specific plan operators.

Execution contexts that are associated with parallel branches of the parallel plan are not cached, however serial branches of the parallel plan (including the branch after the final Gather Streams) are cached. A necessary condition for SQL Server to compile a parallel query plan is that the minimum of the number of processors must be more than one after having survived:

* The processor affinity mask
* The "max degree of parallelism" server-wide option (possibly set using the "sp\_configure" stored procedure)
* (SQL Server 2008 and later) The MAXDOP (Maximum Degree of Parallelism) limits imposed on the resource pool that the query is executing within, by the Resource Governor
* Any MAXDOP hint on the query

## Mode Changing SET Options (SHOWPLAN-Related and Others)

Some SET operations change the mode of operation for SQL Server. In SQL Server 2012, a plan is compiled but not cached when using certain SET options that cause a batch to be evaluated but no query to be executed. This is a change in behavior from earlier versions of SQL Server.

The SET options that have this behavior are:

* SHOWPLAN\_ALL
* SHOWPLAN\_TEXT
* SHOWPLAN\_XML
* NOEXEC
* PARSEONLY

## Costs associated with query plans and execution contexts

With every query plan and execution context, a cost is stored. The cost partially controls how long the plan or context will live in the plan cache. Plans that cost more are more likely to be kept. Apart from ad-hoc queries where the cost is set as zero, the cost of a query plan is a measure of the amount of resources that were required to produce it. Specifically, the cost is calculated as a number of “ticks" with a maximum value of 31, and is composed of three parts:

Cost = I/O cost + context switch cost (a measure of CPU cost) + memory cost

The individual parts of the cost are calculated as follows.

* Two I/Os cost 1 tick, with a maximum of 19 ticks.
* Two context switches cost 1 tick, with a maximum of 8 ticks.
* Sixteen memory pages (128 KB) cost 1 tick, with a maximum of 4 ticks.

As soon as the size of the Plan Cache reaches 50% of the Buffer Pool size, the next plan cache access decrements the ticks of all of the plans by 1. Notice that because this decrement is performed on a thread that accesses the Plan Cache for plan lookup purpose, the decrement can be considered to occur in a lazy fashion. If the sum of the sizes of all of the caches in SQL Server reaches or exceeds 75% of the Buffer Pool size, a dedicated resource monitor thread is activated and is used to decrement tick counts of all of the objects in all of the caches. Each time that a query plan is reused, its cost is reset to its initial value.

For more detail on how the cache responds to memory pressure, review the following article: [http://blogs.msdn.com/sqlprogrammability/archive/2007/01/16/9-0-memory-pressure-limits.aspx.](http://blogs.msdn.com/sqlprogrammability/archive/2007/01/16/9-0-memory-pressure-limits.aspx)

Note that the cost formula and pressure limits apply to most caches in the system, not specifically just to the plan cache.

# Achieving Good SQL Server Batch Execution Performance

To obtain good SQL Server batch execution performance, the following need to happen:

* Query plans should be reused whenever possible and optimal to avoid unnecessary query compilation costs. The reuse of plans also results in better Plan Cache utilization which, in turn, results in better server performance.
* Practices that may cause an increase in the number of query recompilations should be avoided. Reducing recompilation counts saves server resources (CPU and memory), and increases the number of batch executions that can be achieved while maintaining predictable performance.

The following section describes the details of query plan reuse. When appropriate, best practices that result in better plan reuse are given. In a subsequent section, we describe some common scenarios that may cause an increase in the number of recompilations, and give best practices on their avoidance.

## Query Plan Reuse Cache-Ability

If a query plan is not cached, it cannot be reused. Issues that cause non-cache-ability will lead to non-reuse.

### Object Name Resolution

While it is good practice to always schema-qualify object names, it is important to avoid situations that require object name resolution in ad-hoc queries and prepared statements.

For example, in the query "SELECT \* FROM ClientDetails;", the object ClientDetails might resolve to Sales.ClientDetails if the user’s default schema is Sales but may resolve to HR.ClientDetails if the user’s default schema is HR. If, however, the query "SELECT \* FROM Sales.ClientDetails;" had been executed instead, no ambiguity exists because the object is uniquely identified.

For ad-hoc queries and prepared statements, query reuse will not occur if object name resolution needs to occur. Queries that are contained within stored procedures, functions, and triggers do not have this limitation.

However, in general it should be noted that the use of two-part object names (that is, schema.object) provides more opportunities for plan reuse and should be encouraged.

### Trigger-Related Issues

For a trigger execution, the number of rows affected by that execution (as measured by the number of rows in either the inserted or deleted table) is a distinguishing factor in determining a Plan Cache hit. Note that this behavior is specific to triggers, and does not apply to stored procedures.  
  
In INSTEAD OF triggers, the "1-plan" is shared by executions that affect both 0 and 1 row, whereas for non-INSTEAD OF ("AFTER") triggers, "1-plan" is only used by executions that affect 1 row and "n-plan" is used by executions that affect both 0 and n rows (n > 1).

### Other Common Non-Cache-Ability Issues

The execution contexts for bulk insert statements are never cached.

A batch that contains any one literal longer than 8000 bytes is not cached. Therefore, query plans for such batches cannot be reused. Note, however, that a literal's length is measured after constant folding is applied. For example, consider an expression such as:

LEN(long-constant-literal-value)

The value of the long-constant-literal-value is not needed in the plan and can be folded into the length of the value instead.

Batches flagged with the replication flag are not matched with batches without that flag.

A batch called from SQL Server common-language runtime (CLR) is not matched with the same batch submitted from outside of CLR. However, two CLR-submitted batches can reuse the same plan. The same applies to:

* CLR triggers and non-CLR triggers
* Notification queries and non-notification queries

SQL Server allows cursor definition on top of a Transact-SQL batch. If the batch is submitted as a separate statement, then it does not reuse (part of the) plan for that cursor.

### Plan Reuse-Affecting SET Options

SET options can change the way that a query needs to be executed. Because of this, the use of certain SET options affects the chance to reuse plans.

The following SET options are plan-reuse-affecting.

|  |  |
| --- | --- |
| ANSI\_NULL\_DFLT\_OFF | DATEFIRST |
| ANSI\_NULL\_DFLT\_ON | DATEFORMAT |
| ANSI\_NULLS | FORCEPLAN |
| ANSI\_PADDING | LANGUAGE |
| ANSI\_WARNINGS | NO\_BROWSETABLE |
| ARITHABORT | NUMERIC\_ROUNDABORT |
| CONCAT\_NULL\_YIELDS\_NULL | QUOTED\_IDENTIFIER |

Table 3: SET options that are plan-reuse-affecting.

Further, ANSI\_DEFAULTS is plan-reuse-affecting because it can be used to change the following SET options together (some of which are plan-reuse-affecting): ANSI\_NULLS, ANSI\_NULL\_DFLT\_ON, ANSI\_PADDING, ANSI\_WARNINGS, CURSOR\_CLOSE\_ON\_COMMIT, IMPLICIT\_TRANSACTIONS, QUOTED\_IDENTIFIER.

The current usage of many of these SET options can be viewed by querying the sys.dm\_exec\_sessions DMV.

Note that values of some of these SET options can be changed by using several methods:

* Using sp\_configure stored procedure (for server-wide changes)
* Using SET clause of the ALTER DATABASE statement
* Establishing SET options at connection time
* Using the SET statement during a user session

Earlier versions of SQL Server also supported the sp\_dboption stored procedure for making database-wide changes but the use of that stored procedure has been discontinued in SQL Server 2012.

When conflicting SET option values have been configured, user-level and connection-level SET option values take precedence over database and sever-level SET option values. Further, if a database-level SET option is in effect, then for a batch that references multiple databases (which could potentially have different SET option values), the SET options of the database in whose context the batch is being executed takes precedence over SET options of the rest of the databases.

To avoid SET option-related recompilations, establish SET options at connection time, and ensure that they do not change for the duration of the connection.

### CREATE WITH RECOMPILE

When a stored procedure is created with the "CREATE PROCEDURE …WITH RECOMPILE" option, its query plan is not cached whenever that stored procedure is executed. Every execution of such a procedure causes a fresh compilation. This option can be used to mark stored procedures that are called with widely varying parameters, and for which the best query plans are highly dependent on parameter values supplied during calls.

As an alternative to the use of the WITH RECOMPILE option, individual statements within the procedure that are sensitive to parameter values can be recompiled on each execution by adding the OPTION (RECOMPILE) query hint. Plans compiled using this hint enable optimizations for the specific value of the parameter(s) at execution time. The resulting parameter-specific plan is not cached for reuse.

### EXEC WITH RECOMPILE

When a stored procedure is executed using "EXEC … WITH RECOMPILE", the stored procedure is freshly compiled. Even if a query plan for the procedure exists in plan cache (and could be reused otherwise), reuse does not happen. The freshly compiled query plan is not cached.

Note that "EXEC … WITH RECOMPILE" can be used with user-defined functions as well, but only if the EXEC keyword is present.

When executing a stored procedure with atypical parameter values, "EXEC … WITH RECOMPILE" can be used to ensure that the fresh query plan does not replace an existing cached plan that was compiled using typical parameter values. Any existing cache entry for the procedure is left unchanged.

### Temporary Objects

Query plans for temporary stored procedures (both session-scoped and global) are cached, and therefore, can be reused.

If a stored procedure refers to a temporary table that is not created statically within the procedure, the spid (process ID) gets added to the cache key. This means that the plan for the stored procedure would only be reused when executed again from within the same session. Temporary tables created statically within the stored procedure do not cause this behavior.

### CREATE/UPDATE STATISTICS

Plans for queries that create or update statistics (either manually or automatically) are not cached.

## Avoiding Recompilations

After SQL Server begins executing statements in batch B, recompilation occurs when some (or all) of them are compiled again. Reasons for recompilation can be broadly classified into two categories:

Plan Stability: A batch must be recompiled if incorrect results could be returned or if incorrect actions could be taken. Stability-related reasons fall into two sub-categories.

* Schemas of objects. A batch may reference many objects (tables, views, indexes, statistics, user-defined functions, and so on). If the schema of one or more of these objects has changed since the batch was last compiled, the batch needs to be recompiled for correctness reasons.
* SET options. If the setting of a plan- affecting SET option is changed inside of a batch, a recompilation occurs.

Plan Optimality: Data in tables that a batch references may have changed considerably since the batch was last compiled. In such cases, the batch may be recompiled to obtain a potentially faster query execution plan.

The following two sections describe the two categories in detail.

## Plan Stability

A list of specific actions that cause correctness-related recompilations follows.

### Schema Changes

Whenever a schema change occurs for any of the objects referenced by a batch, the batch is recompiled. Schema changes are defined as one or more of the following:

* Adding or dropping columns to a table or view.
* Adding or dropping constraints, defaults, or rules to/from a table.
* Adding an index to a table or an indexed view.
* Dropping an index defined on a table or an indexed view. This only applies if the index is used by the query plan in question.
* Dropping (note: not creating or updating) a statistic defined on a table. This only applies if the query plan uses the table. Updating a statistic (both manual and auto-update) may cause an optimality-related (data related) recompilation of any query plans that uses this statistic. Optimality-related recompilations are described later.

### sp\_recompile

Executing sp\_recompile on a stored procedure or a trigger causes them to be recompiled the next time they are executed. When sp\_recompile is run on a table or a view, all of the stored procedures that reference that table or view will be recompiled the next time they are run. sp\_recompile accomplishes recompilations by incrementing the on-disk schema version of the object in question.

### Flush Entire Plan Cache

The following operations flush the entire plan cache, and therefore, cause fresh compilations of batches that are submitted the first time afterwards:

* Detaching a database
* Upgrading a database to a later database compatibility level
* Restoring a database
* DBCC FREEPROCCACHE command
* RECONFIGURE command (many of the options to this command cause a flush)
* ALTER DATABASE … MODIFY FILEGROUP command
* Modifying a collation using ALTER DATABASE … COLLATE command

### Flush Database-Related Entries in Plan Cache

The following operations flush the plan cache entries that refer to a particular database, and cause fresh compilations afterwards.

* DBCC FLUSHPROCINDB command
* ALTER DATABASE … MODIFY NAME = command
* ALTER DATABASE … SET ONLINE command
* ALTER DATABASE … SET OFFLINE command
* ALTER DATABASE … SET EMERGENCY command
* DROP DATABASE command
* When a database auto-closes

The concept "plan cache entries that refer to a particular database" needs explanation. Database ID is one of the keys for the plan cache. Consider the following command sequence.

USE master;  
GO

-- A query here that references a database called db1  
GO

Suppose that the query is cached in the Plan Cache. The database ID associated with the query’s plan will be the ID of master, not the ID of db1.

### CHECK OPTION

When a view is created with CHECK OPTION, the plan cache entries of the database in which the view is created is flushed.

### DBCC CHECKDB

When DBCC CHECKDB is run, a replica of the specified database is created. As part of DBCC CHECKDB's execution, some queries against the replica are executed, and their plans cached. At the end of DBCC CHECKDB's execution, the replica is deleted and so are the query plans of the queries posed on the replica.

### SNAPSHOT Isolation Level

When SQL Server's transaction-level snapshot isolation level is on, plan reuse happens as usual. Whenever a statement in a batch under snapshot isolation level refers to an object whose schema has changed since the snapshot isolation mode was turned on, a statement-level recompilation happens if the query plan for that statement was cached and was reused. The freshly compiled query plan is cached, but the statement itself fails (as per that isolation level's semantics). If a query plan was not cached, a compilation happens, the compiled query plan is cached, and the statement itself fails.

### DBCC FREESYSTEMCACHE

The query DBCC FREESYSTEMCACHE() clears a specific cache store, in SQL Server 2008 and later. For example, the query DBCC FREESYSTEMCACHE(‘SQL Plans’) clears the store associated with ad-hoc queries. DBCC FREESYSTEMCACHE(‘ALL’,’somepool’) clears all cache entries associated with the Resource Governor resource pool named ‘somepool’.

### SET Options

As mentioned earlier, changing one or more of the following SET options after a batch has started execution will cause a recompilation: ANSI\_NULL\_DFLT\_OFF, ANSI\_NULL\_DFLT\_ON, ANSI\_NULLS, ANSI\_PADDING, ANSI\_WARNINGS, ARITHABORT, CONCAT\_NULL\_YIELDS\_NULL, DATEFIRST, DATEFORMAT, FORCEPLAN, LANGUAGE, NO\_BROWSETABLE, NUMERIC\_ROUNDABORT, QUOTED\_IDENTIFIER.

## Plan Optimality

SQL Server is designed to generate optimal query execution plans as data in databases changes. Data changes are tracked using statistics (histograms) in SQL Server's query processor. Therefore, plan optimality-related reasons have close association with the statistics.

Before describing plan optimality-related reasons in detail, we enumerate the conditions under which plan optimality-related recompilations do not happen.

### Trivial Plan

A trivial plan results when the query optimizer determines that, given the tables referenced in the query and the existing indexes on them, only one plan is always optimal.Only one plan is always optimal. Obviously, a recompilation would be futile in such a case.

A query that has generated a trivial plan may not always continue to generate a trivial plan. For example, a new index might be created an underlying table, providing another potential access path to the query optimizer. Additions of such indexes would be detected and an optimality -related recompilation might replace the trivial plan with a non-trivial one.

### KEEPFIXED PLAN

When a query contains the KEEPFIXED PLAN query hint, its plan is not recompiled for plan optimality-related reasons.

### Read-Only Filegroups

When all of the tables referenced in the query plan reside in read-only filegroups or databases, the plan is not recompiled.

# Query Compilation

The following flowchart describes the batch compilation and recompilation process in SQL Server. The main processing steps are as follows (individual steps are described in detail later on in this document):

* SQL Server begins compiling a query. As mentioned previously, a batch is the unit of compilation and caching, but individual statements in a batch are compiled one after another.
* All of the interesting statistics that may help to generate an optimal query plan are loaded from disk into memory.
* If any of the statistics are outdated, they are updated one-at-a-time. The query compilation waits for the updates to finish. Statistics may optionally be updated asynchronously. That is, the query compilation thread is not blocked by statistics updating threads. The compilation thread proceeds with stale statistics.
* The query plan is generated. Recompilation thresholds of all of the tables referenced in the query are stored along with the query plan.
* At this point, query execution has technically begun. The query plan is now tested for the correctness-related reasons described earlier.
* If the plan is not correct for any of the correctness-related reasons, a recompilation is started. Notice that because query execution has technically begun, the compilation just started is a recompilation.
* If the plan is correct, then various recompilation thresholds are compared with either table cardinalities or table modification counters (colmodctr).
* If any of the statistics are deemed out-of-date, a recompilation results.
* If all of the comparisons succeed, actual query execution begins.

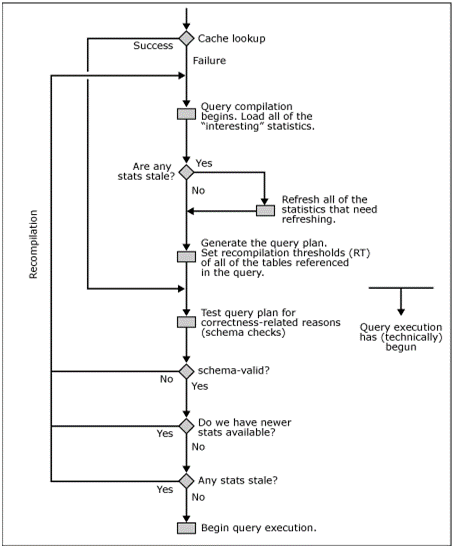


Figure 3: Batch compilation and recompilation process

## Tracking Data Modifications

Each SELECT, INSERT, UPDATE, DELETE and MERGE statement accesses one or more tables. Table contents change because of such operations as INSERT, UPDATE, DELETE and MERGE. SQL Server's query processor is designed to adapt to such changes by generating potentially different query plans, each optimal at the time it is generated. Table contents are tracked directly using table cardinality, and indirectly using statistics (histograms) on table columns. Each column has a recompilation threshold (RT) associated with it. RT is a function of the number of rows in a table.

During query compilation, the query processor loads zero or more statistics defined on tables referenced in a query. These statistics are known as interesting statistics. For every table referenced in a query, the compiled query plan contains:

* A list of all of the statistics loaded during query compilation. For each such statistic, a snapshot value of a counter that counts the number of table modifications is stored. The counter is called colmodctr. A separate colmodctr exists for each table column (except computed non-persisted columns).
* The threshold crossing test — which is performed to decide whether to recompile a query plan — is defined by the formula:

ABS( colmodctr(current) – colmodctr(snapshot)) ) >= RT

colmodctr(current) refers to the current value of the modification counter, and colmodctr(snapshot) refers to the value of the modification counter when the query plan was last compiled. If threshold crossing succeeds for any of the interesting statistics, the query plan is recompiled. Only the query in question is recompiled.

If a table or an indexed view has no statistic on it, or none of the existing statistics on it are considered "interesting" during a query compilation, the following threshold-crossing test, based purely on T's cardinality, is still performed.

ABS( card(current) – card(snapshot)) ) >= RT

where card(current) denotes the number of rows in the indexed view at present, and card(snapshot) denotes the row count when the query plan was last compiled.

## Interesting Statistics

With every query plan, the optimizer stores the IDs of the statistics that were loaded to generate the plan. Note that the loaded set includes both:

* Statistics that are used as cardinality estimators of the operators appearing in the plan
* Statistics that are used as cardinality estimators in query plans that were considered during query optimization but were discarded in favor of the plan.

In other words, the query optimizer considers all of the loaded statistics as "interesting" for one reason or another.

Statistics can be created or updated either manually or automatically. Statistics updates also happen because of executions of the following commands:

* CREATE INDEX … WITH DROP EXISTING
* sp\_createstats stored procedure
* sp\_updatestats stored procedure
* ALTER INDEX REBUILD (but not REORGANIZE)

Given multiple statistics on the same set of columns in the same order, how does the query optimizer decide which ones to load during query optimization? The answer is not simple, but the query optimizer uses such guidelines as giving preference to recent statistics over older statistics; giving preference to statistics computed using FULLSCAN option to those computed using sampling; and so on.

## Recompilation Threshold (RT)

The recompilation threshold for a table partly determines the frequency with which queries that refer to the table recompile. The RT depends on the table type (permanent versus temporary), and the number of rows in the table (cardinality) when a query plan is compiled. The recompilation thresholds for all of the tables referenced in a batch are stored with the query plans of that batch.

RT is calculated as follows. (n refers to a table's cardinality when a query plan is compiled.)

|  |  |
| --- | --- |
| Permanent table | If n <= 500, RT = 500. If n > 500, RT = 500 + 0.20 \* n. |
| Temporary table | If n < 6, RT = 6. If 6 <= n <= 500, RT = 500. If n > 500, RT = 500 + 0.20 \* n. |
| Table variable | RT does not exist. Therefore, recompilations do not happen because of changes in cardinalities of table variables. |

Note that the use of Trace Flag 2371 alters this calculation as described here: <http://blogs.msdn.com/b/saponsqlserver/archive/2011/09/07/changes-to-automatic-update-statistics-in-sql-server-traceflag-2371.aspx>. The following knowledgebase article describes usage guidelines: <http://support.microsoft.com/kb/2754171>.

## ColModCtr

As mentioned previously, the RT is compared against the number of modifications that a column has undergone. The number of modifications that a table has undergone is tracked using a counter (for each column) known as colmodctr. This counter is not transactional. For example, if a transaction starts, inserts 100 rows into a table, and then is rolled back, the changes to colmodctr will not be rolled back.

A colmodctr value is stored for each table column except for computed columns that are not persisted. Colmodctr values for non-persisted computed columns do not exist. They are derived from the columns that participate in the computation. Persisted computed columns however, do have colmodctrs, the same as other columns. Using colmodctrvalues, changes to a table can be tracked with fine granularity.

Note: While there are methods such as using the dedicated admin connection to view colmodctr values, they are not normally available to users and are typically only available to the query processor.

When a statistic is created or updated (either manually or automatically by the auto-stats feature) on one or more columns of a table or indexed view, the snapshot value of the colmodctr of the leftmost column is stored in the stats-blob.

Colmodctr(current), mentioned in the "threshold-crossing" test, is the value persisted in SQL Server's metadata when the test is performed during query compilation.

Colmodctr's values are an ever-increasing sequence: colmodctr values are never reset to 0.

Beginning with Service Pack 2 for SQL Server 2008 and Service Pack 1 for SQL Server 2012, the DMV sys.dm\_db\_stats\_properties has been added. This DMV returns properties of the statistics for database objects and is particularly interesting in relation to colmodctr values as they are exposed by the returned column modification\_counter. As an example, the following query returns all statistics for a table named Person.Address:

SELECTsp.stats\_id,name,filter\_definition,

last\_updated,[rows],rows\_sampled,

steps,unfiltered\_rows,

modification\_counter

FROMsys.statsASs

CROSSAPPLYsys.dm\_db\_stats\_properties(s.object\_id,s.stats\_id)ASsp

WHEREs.object\_id=object\_id('Person.Address');

## Tracking Changes Using colmodctr

Because colmodctr values are used to make recompilation decisions, their values are modified as the data in a table changes. In the following description, we only refer to tables. However, identical observations apply to indexed views.

A table can change because of the following statements: INSERT, DELETE, UPDATE, MERGE, BULK INSERT, and TRUNCATE TABLE. The following table defines how colmodctr values are modified in each case. (Note: n is the number of rows).

|  |  |
| --- | --- |
| Statement | SQL Server 2008 and higher |
| INSERT | All colmodctr += 1 \* n |
| DELETE | All colmodctr += 1 \* n |
| UPDATE | Typically an update to a row is counted as a modification. However, the execution plan may split updates into deletes and inserts and not collapse them back again. In this case, each modification would count as two modifications. |
| MERGE | Total of the INSERT, UPDATE and DELETE behavior of the MERGE statement. |
| Bulk insert | Like n INSERTs. All colmodctr += n. (n is the number of rows bulk inserted.) |
| Table truncation | Like n DELETEs. All colmodctr += n. (n is the table's cardinality.) |

Table 4: SQL Server statements that modify colmodctr values.

## Special Cases

Plan optimality-related recompilations are handled differently in the following two special cases.

### Special case 1: Statistics created on an empty table or indexed view

A user creates an empty table. She then creates a statistic on one or more columns of the table. Because the table is empty, the stats-blob (histogram) is NULL, but the statistic has been created on the table. Suppose that the statistic has been found interesting during a query compilation. As per the "500 row" rule for recompilation threshold, the table would be expected to cause recompilations on SQL Server only after the table contains at least 500 rows. Therefore, a user would potentially have suffered from sub-optimal plans until the table contains at least 500 rows.

SQL Server detects this special case, and handles it differently. The recompilation threshold for such a table or indexed view is 1. In other words, even the insertion of one row in the table can cause a recompilation. When such a recompilation happens, the statistic is updated, and the histogram for the statistic is no longer NULL. After this recompilation, however, the usual rule for recompilation threshold (500 + 0.20 \* n) is followed. The recompilation threshold is 1 even when the table has no statistics or the table has no statistics that are considered interesting during a query compilation.

### Special case 2: Trigger recompilations

All of the plan optimality-related reasons for recompilations are applicable to triggers. In addition, plan optimality-related recompilations for triggers can also happen because of the number of rows in the inserted or deleted tables changing significantly from one trigger execution to the next.

Recall that triggers that affect one row versus multiple rows are cached independently of each other. The numbers of rows in the inserted and deleted tables are stored with the query plan for a trigger. These numbers reflect the row counts for the trigger execution that caused plan caching. If a subsequent trigger execution results in the inserted or deleted table having sufficiently different row counts, then the trigger is recompiled (and a fresh query plan with the new row counts is cached).

Sufficiently different is defined by the formula:

if m > n then ABS( log10(n) –log10(m) ) > 1 else ABS( log10(n) –log10(m) ) > 2.1

where n is the row count of the inserted or deleted table in the cached query plan and m is the row count of the corresponding table for the current trigger execution. If both the inserted and deleted tables have rows, the above-mentioned test is separately performed for both of them. As an example of the calculation, a row count change from 10 to 100 does not cause a recompilation, whereas a change from 10 to 101 does.

# Best Practices for Achieving Plan Optimality Stability

## Table Variables

Because a change in cardinality of a table variable does not cause recompilations, consider using a table variable instead of a temporary table when faced with a problem of excessive recompilations. However, the use of table variables can lead to poor query plans. Distribution statistics are not stored for table variable and cardinality is only available during recompiles, not during initial compilation. It is important to experiment to determine whether this is the case or not, and make an appropriate trade-off decision. In general, if a large number of rows are expected, table variables should not be used. This is explained further in the following article: <http://blogs.msdn.com/b/psssql/archive/2010/08/24/query-performance-and-table-variables.aspx>.

## KEEP PLAN

The KEEP PLAN query hint changes the recompilation thresholds for temporary tables, and makes them identical to those for permanent tables. Therefore, if changes to temporary tables are causing many recompilations, this query hint can be used. The hint can be specified using the following syntax:

SELECTb.col4,SUM(a.col1)

FROMdbo.PermTableASa

INNERJOIN#TempTableASb

ONa.col1=b.col2

WHEREb.col3<100

GROUPBYb.col4

OPTION (KEEPPLAN);

## KEEPFIXED PLAN

To avoid recompilations due to plan optimality-related (statistic update-related) reasons totally, the KEEPFIXED PLAN query hint can be specified using the syntax:

SELECTc.TerritoryID,COUNT(\*)ASNumber,c.SalesPersonID

FROMSales.StoreASs

INNERJOINSales.CustomerASc

ONs.CustomerID=c.CustomerID

WHEREs.NameLIKE'%Bike%'

ANDc.SalesPersonID>285

GROUPBYc.TerritoryID,c.SalesPersonID

ORDERBYNumberDESC

OPTION (KEEPFIXEDPLAN);

With this option in effect, recompilations can only happen because of correctness-related reasons — for example, the schema of a table referenced by a statement changes, or a table is marked with the sp\_recompile procedure.

Consider the situation where a query with OPTION(KEEPFIXED PLAN) hint is being compiled for the first time, and compilation causes auto-creation of a statistic. If SQL Server can get a "stats lock," a recompilation happens and the statistic is auto-created. If the "stats lock" cannot be obtained, there is no recompilation, and the query is compiled without that statistic.

## LEAVE AUTOUPDATE STATISTICS ENABLED

Do not turn off automatic updates of statistics for indexes and statistics defined on a table or indexed view. While doing so will ensure that plan optimality-related recompilations caused by those objects will stop, the query optimizer will no longer be sensitive to data changes in those objects and sub-optimal query plans might result. Turn off automatic update of statistics only as a last resort after exhausting all of the other alternatives.

# Parameter Sniffing

## Parameter Sniffing Described

"Parameter Sniffing" refers to a process whereby SQL Server's execution environment "sniffs" the current parameter values during compilation or recompilation, and passes them to the query optimizer so that they can be used to generate potentially faster query execution plans. The word "current" refers to the parameter values present in the statement call that caused a compilation or a recompilation. Parameter values are sniffed during compilation or recompilation for the following types of batches:

* Stored procedures
* Queries submitted via sp\_executesql
* Prepared queries
* OPTION(RECOMPILE) query hint.

For such a query (which could be SELECT, INSERT, UPDATE, DELETE or MERGE), both the parameter values and the current values of local variables are sniffed. Note that without OPTION(RECOMPILE), only parameter values are sniffed. The values of parameters and local variables that are sniffed are those that exist at the place in the batch just before the statement with the OPTION(RECOMPILE) hint. In particular, for parameters, the values that came along with the batch invocation call are not sniffed.

In general, parameter sniffing is desirable. However, if the value of a query’s parameters can greatly affect the choice of appropriate query plan, parameter sniffing can lead to the reuse of inappropriate query plans.

## OPTIMIZE FOR Query Hint

The OPTIMIZE FOR query hint can assist in avoiding parameter sniffing problems caused by the first execution of a procedure where atypical parameter values are passed. For example, consider the query plans for the following two queries:

SELECTsoh.SalesOrderID,soh.DueDate,

sod.OrderQty,sod.ProductID

FROMSales.SalesOrderHeaderASsoh

INNERJOINSales.SalesOrderDetailASsod

ONsoh.SalesOrderID=sod.SalesOrderID

WHEREsoh.CustomerID>30117;

SELECTsoh.SalesOrderID,soh.DueDate,

sod.OrderQty,sod.ProductID

FROMSales.SalesOrderHeaderASsoh

INNERJOINSales.SalesOrderDetailASsod

ONsoh.SalesOrderID=sod.SalesOrderID

WHEREsoh.CustomerID>10000;

In these queries, the plan produced is highly dependent on the value of the CustomerID. Only one SalesOrderHeader row matches the predicate soh.CustomerID > 30117 but a large number of rows match the predicate soh.CustomerID > 10000. If this SELECT statement was included in a stored procedure, the effectiveness of the query plan produced would be significantly influenced by the value passed to the query on its first execution. If a particular value is known to be typical, SQL Server can be instructed to optimize for that value via a query hint as shown in the following procedure definition:

CREATEPROCEDUREdbo.ProductOrderDetails

@CustomerIDint

AS

SELECTsoh.SalesOrderID,soh.DueDate,

sod.OrderQty,sod.ProductID

FROMSales.SalesOrderHeaderASsoh

INNERJOINSales.SalesOrderDetailASsod

ONsoh.SalesOrderID=sod.SalesOrderID

WHEREsoh.CustomerID>@CustomerID

OPTION (OPTIMIZEFOR (@CustomerID=30117));

Compare the query plans from the following two executions:

EXECdbo.ProductOrderDetails 30117;

EXECdbo.ProductOrderDetails 10000;

You will see that the query plans are now identical. It is important to note however that this hint should only ever be used to supply typical parameter values. In the case above, we have optimized the plan for a CustomerID of 30117 but created a poor plan for a CustomerID of 10000.

## OPTIMIZE FOR UNKNOWN

For values that vary widely, SQL Server 2008 and later versions offer an additional option. OPTIMIZE FOR UNKNOWN requests SQL Server to use statistical distributions to determine the plan rather than the supplied value. For example, we will modify our previous procedure via the following code:

ALTERPROCEDUREdbo.ProductOrderDetails

@CustomerIDint

AS

SELECTsoh.SalesOrderID,soh.DueDate,

sod.OrderQty,sod.ProductID

FROMSales.SalesOrderHeaderASsoh

INNERJOINSales.SalesOrderDetailASsod

ONsoh.SalesOrderID=sod.SalesOrderID

WHEREsoh.CustomerID>@CustomerID

OPTION (OPTIMIZEFORUNKNOWN);

Once again compare the query plans from the following two executions:

EXECdbo.ProductOrderDetails 30117;

EXECdbo.ProductOrderDetails 10000;

To see further details on the OPTIMIZE FOR query hint, review the following article: <http://msdn.microsoft.com/en-us/library/ms181714(v=SQL.110).aspx>.

## TRACE FLAG 4136

OPTIMIZE FOR UNKNOWN is useful when working with individual queries that suffer from parameter sniffing issues. For situations where these issues are widespread or where it is not possible to modify the queries that are being executed, SQL Server 2008 R2 Cumulative Update 2, SQL Server 2008 Service Pack 1 (SP1) Cumulative Update 7, and SQL Server 2005 Service Pack 3 (SP3) Cumulative Update 9 introduced trace flag 4136. TF4136 disables parameter sniffing at the instance level rather than at the query level. The knowledgebase article covering this trace flag is here: <http://support.microsoft.com/kb/980653>.

# Identifying Recompilations

## Tracing Recompilation Activity

SQL Server's tracing capabilities make it easy to identify batches that cause recompilations. Start a new trace (using either SQL Server Profiler or an Extended Events session) and select the following events under Stored Procedures event class. (To reduce the amount of data generated, it is recommended that you de-select any other events.)

* SP:Starting
* SP:StmtStarting
* SP:Recompile
* SP:Completed
* SQL:StmtRecompile

In addition, to detect statistics-update-related recompilations, select the "Auto Stats" event under "Objects" class. (You may also wish to use SQL:StmtRecompile to see finer detail). Start SQL Server Management Studio, and execute the following Transact-SQL code:

DROPPROCEDUREDemoProc1;

GO

CREATEPROCEDUREDemoProc1

AS

CREATETABLE#t1 (aint,bint);

SELECT\*FROM#t1;

GO

EXECDemoProc1;

GO

EXECDemoProc1;

GO

Pause the trace, and you will see the following sequence of events.

|  |  |  |
| --- | --- | --- |
| **EventClass** | **TextData** | **EventSubClass** |
| SP:Starting | EXEC DemoProc1 |  |
| SP:StmtStarting | -- DemoProc1 CREATE TABLE #t1 (a int, b int) |  |
| SP:StmtStarting | -- DemoProc1 SELECT \* FROM #t1 |  |
| SP:Recompile |  | Deferred compile |
| SP:StmtStarting | -- DemoProc1 SELECT \* FROM #t1 |  |
| SP:Completed | EXEC DemoProc1 |  |
| SP:Starting | EXEC DemoProc1 |  |
| SP:StmtStarting | -- DemoProc1 CREATE TABLE #t1 (a int, b int) |  |
| SP:StmtStarting | -- DemoProc1 SELECT \* FROM #t1 |  |
| SP:Completed | EXEC DemoProc1 |  |

Table 5: Events generated from executing the Transact-SQL code example.

The event sequence indicates that "SELECT \* FROM #t1" was the statement that caused the recompilation. The EventSubClass column indicates the reason for the recompilation. In this case, when DemoProc1 was compiled before it began execution, the "CREATE TABLE" statement could be compiled. The subsequent "SELECT" statement could not be compiled because it referred to a temporary table #t1 that did not exist at the time of the initial compilation. The compiled plan for DemoProc1 was thus incomplete. When DemoProc1 started executing, #t1 got created and then the "SELECT" statement could be compiled. Because DemoProc1 was already executing, this compilation counts as a recompilation as per our definition of recompilation. The reason for this recompilation is correctly given as "deferred compile."

It is interesting to note that when DemoProc1 is executed again, the query plan is no longer incomplete. The recompilation has inserted a complete query plan for DemoProc1 into the plan cache. Therefore, no recompilations happen for the second execution.

The statements causing recompilations can also be identified by selecting the following set of trace events.

* SP:Starting
* SP:StmtCompleted
* SP:Recompile
* SP:Completed

If the above example is run after selecting this new set of trace events, the trace output looks like the following.

|  |  |  |
| --- | --- | --- |
| **EventClass** | **TextData** | **EventSubClass** |
| SP:Starting | EXEC DemoProc1 |  |
| SP:StmtCompleted | -- DemoProc1 CREATE TABLE #t1 (a int, b int) |  |
| SP:Recompile |  | Deferred compile |
| SP:StmtCompleted | -- DemoProc1 SELECT \* FROM #t1 |  |
| SP:Completed | EXEC DemoProc1 |  |
| SP:Starting | EXEC DemoProc1 |  |
| SP:StmtCompleted | -- DemoProc1 CREATE TABLE #t1 (a int, b int) |  |
| SP:StmtCompleted | -- DemoProc1 SELECT \* FROM #t1 |  |
| SP:Completed | EXEC DemoProc1 |  |

Table 6: Events generated from executing the Transact-SQL code example.

Notice that in this case, the statement causing the recompilation is printed after the SP:Recompile event. This method is somewhat less obvious than the first one. Therefore, we shall trace the first set of trace events henceforth.

## Recompilation Reasons

To see all of the possible recompilation reasons reported for the SP:Recompile event, issue the following query:

SELECTdxmv.map\_key,

dxmv.map\_value

FROMsys.dm\_xe\_map\_valuesASdxmv

WHEREdxmv.name=N'statement\_recompile\_cause'

ORDERBYdxmv.map\_key;

The output of the above query is as follows:

|  |  |  |
| --- | --- | --- |
| **SubclassName** | **SubclassValue** | **Detailed reason for recompilation** |
| Schema changed | 1 | Schema, bindings, or permissions changed between compile and execute. |
| Statistics changed | 2 | Statistics changed. |
| Deferred compile | 3 | Recompile because of DNR (Deferred Name Resolution). Object not found at compile time, deferred check to run time. |
| Set option change | 4 | Set option changed in batch. |
| Temp table changed | 5 | Temp table schema, binding, or permission changed. |
| Remote rowset changed | 6 | Remote rowset schema, binding, or permission changed. |
| For browse permissions changed | 7 | Permissions changed in FOR BROWSE (deprecated DBLIB option) |
| Query notification environment changed | 8 | Query notification environment changed |
| Partition view changed | 9 | SQL Server sometimes adds data-dependent implied predicates to WHERE clauses of queries in some indexed views. If the underlying data changes, such implied predicates become invalid, and the associated cached query plan needs recompilation. |
| Cursor options changed | 10 | Change in cursor options |
| Option (Recompile) requested | 11 | Recompile was requested |
| Parameterized plan flushed | 12 | Parameterized plan was flushed from cache (SQL Server 2008 and later) |
| Test plan linearization | 13 | (SQL Server 2008 and later) For internal test only |
| Plan affecting database version changed | 14 | (SQL Server 2008 and later) |

Table 7: Recompilation reasons reported for the SP:Recompile event.

## Tools and Commands

There are a variety of tools and commands that are useful when observing and debugging recompilation-related scenarios.

### Dynamic Management Objects

A number of dynamic management views and functions are useful when exploring plan reuse. In particular, the following objects are most helpful:

* sys.dm\_exec\_cached\_plans
* sys.dm\_exec\_query\_plan
* sys.dm\_exec\_query\_stats
* sys.dm\_exec\_plan\_attributes
* sys.dm\_exec\_sql\_text
* sys.dm\_exec\_cached\_plan\_dependent\_objects
* sys.dm\_exec\_procedure\_stats
* sys.dm\_exec\_trigger\_stats

As an example, the following query is useful in exploring the current plan cache contents:

SELECTcp.objtypeASPlanType,

OBJECT\_NAME(st.objectid,st.dbid)ASObjectName,

cp.refcountsASReferenceCounts,

cp.usecountsASUseCounts,

st.textASSQLBatch,

qp.query\_planASQueryPlan

FROMsys.dm\_exec\_cached\_plansAScp

CROSSAPPLYsys.dm\_exec\_query\_plan(cp.plan\_handle)ASqp

CROSSAPPLYsys.dm\_exec\_sql\_text(cp.plan\_handle)ASst;

In SQL Server 2012, the sys.dm\_exec\_query\_stats DMV has additional columns (total\_rows, last\_rows, min\_rows and max\_rows) that can help when you have network constraints or when you want to detect plans that return sets with very variable number of rows. For example, a common symptom of parameter sniffing problems can be identified by a situation where the same plan sometimes returns a single row but at other times, a large number of rows.

### sys.syscacheobjects virtual table

This virtual table conceptually exists only in the master database, although it can be queried from any database. The cacheobjtype column of this virtual table is particularly interesting. When cacheobjtype= "Compiled Plan", the row refers to a query plan. When cacheobjtype = "Executable Plan", the row refers to an execution context. As we have explained, each execution context must have its associated query plan, but not vice versa. One other column of interest is the objtypecolumn: it indicates the type of object whose plan is cached (for example, "Adhoc", "Prepared", and "Proc"). The setopts column encodes a bitmap indicating the SET options that were in effect when the plan was compiled. Sometimes, multiple copies of the same compiled plan (that differ in only setopts column) are cached in a plan cache. This indicates that different connections are using different sets of SET options, a situation that is often undesirable. The usecounts column stores the number of times a cached objects has been reused since the time the object was cached.

This virtual table is maintained for backwards compatibility with SQL Server 2000. Dynamic Management Objects should now be used in preference. The virtual table is documented here: <http://msdn.microsoft.com/en-us/library/ms187815.aspx>.

### DBCC FREEPROCCACHE

This command removes all of the cached query plans and execution contexts from the Plan Cache. It is generally not advisable to run this command on a production server because it can adversely affect performance of running applications. This command is useful to control plan cache's contents when troubleshooting a recompilation issue. In SQL Server 2012, it can be used to evict an individual plan or the contents of a pool:

DBCC FREEPROCCACHE [ ( { plan\_handle | sql\_handle | pool\_name } ) ] [ WITH NO\_INFOMSGS ];

### DBCC FLUSHPROCINDB( db\_id )

This command removes all of the cached plans from the plan cache for a particular database. It is not advisable to run this command on a production server unless its effect is fully understood as it can adversely affect performance of running applications.

### DBCC FREESYSTEMCACHE(cache[,resource pool])

This command removes all the plans in the specified cache store. The cache store specified can be one of the following:

* Object Plans
* SQL Plans
* Bound Trees
* Extended Stored Procedures
* Temporary Tables & Table Variables

The value ‘ALL’ can also be provided for the cache. Optionally, in SQL Server 2008 and later versions, the effect of this command can be limited by a Resource Governor resource pool name. This latter option could be useful for removing ad-hoc query plans associated with a specific resource pool, when the ‘SQL Plans’ cache option is also provided. It is not advisable to run this command on a production server unless its effect is fully understood as it can adversely affect performance of running applications.

### Extended Events (SQL Server 2008 and later versions)

SQL Server 2008 introduced a new light-weight eventing system called Extended Events. It allows tracing of events as they occur and details of events as they are fired to be written to a variety of synchronous and asynchronous targets. A number of these events are related to plan caching. You can see the full list of available events by executing the following command:

SELECTdxp.[name]ASPackage,

dxo.[name]ASEventName,

dxo.capabilities\_descASCapabilities,

dxo.[description]ASDescription

FROMsys.dm\_xe\_packagesASdxp

INNERJOINsys.dm\_xe\_objectsASdxo

ONdxp.[guid]=dxo.package\_guid

WHEREdxo.object\_type='event'

ORDERBYPackage,EventName;

You can find details on the Extended Events system at <http://msdn.microsoft.com/en-us/library/bb630354.aspx>.

### Trace Events

The following trace events are relevant for observing and debugging plan caching, compilation, and recompilation behaviors.

* 'Cursors: CursorRecompile' for observing recompilations caused by cursor-related batches.
* 'Objects: Auto Stats' for observing the statistics updates caused by SQL Server's "auto-stats" feature.
* 'Performance: Show Plan All For Query Compile' is useful for tracing batch compilations. It does not distinguish between a compilation and a recompilation. It produces showplan data in textual format (similar to the one produced using "set showplan\_all on" option).
* 'Performance: Show Plan XML For Query Compile' is useful for tracing batch compilations. It does not distinguish between a compilation and a recompilation. It produces showplan data in XML format (similar to the one produced using "set showplan\_xml on" option).
* 'Stored Procedures: SP: Recompile' and SQL:StmtRecompile for detecting when recompilation occurs. Other events in the "Stored Procedures" category are also useful — for example, SP:CacheInsert, SP:StmtStarting, SP:CacheHit, SP:Starting, and so on.

### PerfMon Counters

A wide variety of useful counters are available in PerfMon. In particular, it is worth investigating the SQL Server: Plan Cache Object performance counters. In addition to general PerfMon counters related to system performance, SQL Server provides statistics on forced parameterization and plan guides. For information on these, review the following article: <http://msdn.microsoft.com/en-us/library/ms190911.aspx>

### Memory Cache Counters

In SQL Server 2012, you can view the current usage of the cache objects using the following DMV-based query:

SELECT

domcc.name,

domcc.[type],

domcc.pages\_kb,

domcc.pages\_in\_use\_kb,

domcc.entries\_count,

domcc.entries\_in\_use\_count

FROMsys.dm\_os\_memory\_cache\_countersASdomcc

WHEREdomcc.[type]IN

(

N'CACHESTORE\_OBJCP',-- Object Plans

N'CACHESTORE\_SQLCP',-- SQL Plans

N'CACHESTORE\_PHDR',-- Bound Trees

N'CACHESTORE\_XPROC',-- Extended Stored Procedures

N'CACHESTORE\_TEMPTABLES'-- Temporary Tables & Table Variables

);

# Plan Cache Pollution Issues

## Plan Cache Pollution Described

While the plan cache generally is self-maintaining, poor application coding practices can cause the plan cache to become polluted with a large number of query plans that are unlikely to be reused.

As an example, while it is convenient to use the AddWithValue(parametername, parametervalue) method of the Parameters collection of the SqlCommand object in .NET coding, doing so does not specify the data type of the parameter. For string parameters, this can be particularly troubling. If the parameter value is initially “hello”, a query plan with an nvarchar parameter length of 5 will be cached after the command is executed. When the query is re-executed with a parameter value of “trouble”, the command will appear to be different as it has an nvarchar parameter with a length of 7.

The more the command is executed, the more the plan cache will become full of plans for different length string parameters. This is particularly troubling for commands with multiple string parameters as plans will end up being stored for all combinations of all lengths of all the parameters.

## Avoiding Plan Cache Pollution

To work around such a problem, the application could use a method to add the parameter that allows specifying the data type precisely. As an example, nvarchar(100) might be used as the data type for each execution in the above example, if we know that all possible parameter lengths are less than 100. Ad-hoc queries generated by end-user query tools can also cause a similar problem where many combinations of similar queries can end up becoming cached.

There are three additional options that can help in dealing with plan cache pollution issues:

### FORCED PARAMETERIZATION

FORCED PARAMETERIZATION can be set at the database level. This makes SQL Server become much more aggressive in deciding which queries to auto-parameterize. The down-side of this option is that it could potentially introduce parameter-sensitivity problems. (This option was added in SQL Server 2005).

### OPTIMIZE FOR ADHOC WORKLOADS

OPTIMIZE FOR ADHOC WORKLOADS is an sp\_configure server level option. When set, SQL Server only caches a plan stub on the first execution of an ad-hoc query. The next time the same query is executed, the full plan is stored. Plan stubs are much smaller than query plans and this option ensures that the plan cache is not filled by query plans that have never been reused. (This option was added in SQL Server 2008).

### DBCC FREESYSTEMCACHE

DBCC FREESYSTEMCACHE can be used to clear the cache of plans associated with a particular Resource Governor resource pool. This could be useful when executed periodically if ad-hoc queries are able to be isolated into identifiable resource pools. (This command was first available in SQL Server 2005 but the option to clear a specific resource pool was added in SQL Server 2008).

# Conclusion

SQL Server caches query plans for a variety of statement types submitted to it for execution. Query plan caching allows for query plan reuse, avoids compilation penalty, and utilizes plan cache better. Some coding practices hinder query plan caching and reuse, and therefore, should be avoided. SQL Server detects opportunities for query plan reuse.

In particular, query plans can be non-reusable for two reasons:

* Schema of an object appearing in a query plan can change thereby making the plan invalid
* Data in tables referred to by a query plan can change enough to make a plan sub-optimal.

SQL Server detects these two classes of conditions at query execution time, and recompiles a batch or pieces of it as necessary. Poor Transact-SQL coding practices both at server and at application data access layer can increase recompilation frequency or plan cache pollution and adversely affect SQL Server's performance. Such situations can be debugged and corrected in many cases.

# Appendix A: When Does SQL Server Not Auto-Parameterize Queries?

Auto-parameterization is a process whereby SQL Server replaces literal constants appearing in a SQL statement with such parameters as @1 and @2. The SQL statement's compiled plan is then cached in plan cache in parameterized form so that a subsequent statement that differs only in the values of the literal constants can reuse the cached plan. As mentioned in Section 4, only those SQL statements for which parameter values do not affect query plan selection are auto-parameterized.

SQL Server's LPE (Language Processing and Execution) component auto-parameterizes SQL statements. When QP (query processor) component realizes that values of literal constants does not affect query plan choice, it declares LPE's attempt of auto-parameterization "safe" and auto-parameterization proceeds; otherwise, auto-parameterization is declared "unsafe" and is aborted.

The following list enumerates the statement types for which SQL Server does not auto-parameterize.

* Queries with IN clauses are not auto-parameterized. For example:

WHERE ProductID IN (707, 799, 905)

* BULK INSERT statement.
* UPDATE statement with a SET clause that contains variables. For example:

UPDATE Sales.Customer

SET CustomerType = N'S'

WHERE CustomerType = @a;

* A SELECT statement with UNION.
* A SELECT statement with INTO clause.
* A SELECT or UPDATE statement with FOR BROWSE clause.
* A statement with query hints specified using the OPTION clause.
* A SELECT statement whose SELECT list contains a DISTINCT.
* A statement with the TOP clause.
* A WAITFOR statement.
* A DELETE or UPDATE with FROM clause.
* When FROM clause has one of the following:
  + More than one table
  + TABLESAMPLE clause
  + Table-valued function or table-valued variable
  + Full-text table
  + OPENROWSET
  + XMLUNNEST
  + OPENXML
  + OPENQUERY
  + IROWSET
  + OPENDATASOURCE
  + Table hints or index hints
* When a SELECT query contains a sub-query
* When a SELECT statement has GROUP BY, HAVING, or COMPUTE BY
* Expressions joined by OR in a WHERE clause.
* Comparison predicates of the form expr <> non-null-constant.
* Full-text predicates.
* When the target table in an INSERT, UPDATE, or DELETE is a table-valued function.
* Statements submitted via EXEC string.
* When query notification is requested.
* When a query contains a common table expression list.
* When a query contains FOR UPDATE clause.
* When an UPDATE contains an ORDER BY clause.
* When a query contains the GROUPING clause.
* INSERT statement of the form: INSERT INTO T DEFAULT VALUES.
* INSERT… EXEC statement.
* When a query contains comparison between two constants. For example, WHERE 20 > 5
* If by doing auto-parameterization, more than 1000 parameters can be created.

**For more information:**

<http://www.microsoft.com/sqlserver/>: SQL Server Web site

<http://technet.microsoft.com/en-us/sqlserver/>: SQL Server TechCenter

<http://msdn.microsoft.com/en-us/sqlserver/>: SQL Server DevCenter

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

* Are you rating it high due to having good examples, excellent screen shots, clear writing, or another reason?
* Are you rating it low due to poor examples, fuzzy screen shots, or unclear writing?

This feedback will help us improve the quality of white papers we release.

[Send feedback](mailto:sqlfback@microsoft.com?subject=Plan%20Caching%20and%20Recompilation%20in%20SQL%20Server%202012).