Microsoft | TechNet

Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005

By Arun Marathe; Revised by Shu Scott

Published: July 2004

Updated: December 2006

Summary: This paper explains how batches are cached and reused in SQL Server 2005, 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.

On This Page

Goals of this white-paper

There are several goals of this white-paper. It explains how batches are cached and reused in SQL Server 2005, 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. The white-paper explains SQL Server 2005's "statement-level recompilation" feature. We also cover many tools and utilities that are useful as observation tools in the processes of query compilation, query recompilation, plan caching, and plan reuse. Throughout the paper, we make comparisons of behaviors between SQL Server 2000 and SQL Server 2005 so that the reader can get a better understanding. Statements made in this document are applicable to SQL Server 2000 *and* SQL Server 2005. Differences in behavior between the two versions of SQL Server are explicitly pointed out.

This paper targets three audiences:

Users: Persons who use, maintain, and develop applications for SQL Server. Users who are new to SQL Server 2005 and those who are migrating from SQL Server 2000 will find useful information here.

Developers: SQL Server developers will find useful background information here.

Top Of Page

Recompilations: Definition

Before a query, batch, stored procedure, trigger, prepared statement, or dynamic SQL statement (henceforth, "batch") begins execution on a SQL Server, the batch gets compiled into a plan. The plan is then executed for its effects or to produce results.

A batch can contain one or more SELECT, INSERT, UPDATE, and DELETE statements; and stored procedure calls possibly interleaved by T-SQL "glue" or control structures such as SET, IF, WHILE, DECLARE; DDL statements such as CREATE, DROP; and permission-related statements such as GRANT, DENY, and REVOKE. A batch can include definition and use of CLR constructs such as user-defined types, functions, procedures, and aggregates.

Compiled plans are stored into a part of SQL Server's memory called ***plan cache***. Plan cache is searched for possible plan reuse opportunities. If a plan reuse for a batch happens, its compilation costs are avoided. Note that in the SQL Server literature, the word "procedure cache" has been used to describe what is called "plan cache" in this paper. "Plan cache" is more accurate because the plan cache stores query plans of not just the stored procedures.

In SQL Server parlance, the compilation process mentioned in the previous paragraph is sometimes confusingly referred to as a "recompilation" although the process is simply a "compilation."

Definition of Recompilation: Suppose that a batch has been compiled into a collection of one or more query plans. Before SQL Server begins executing any of the individual query plans, the server checks for validity (correctness) and optimality of that query plan. If one of the checks fails, the statement corresponding to the query plan or the entire batch is compiled *again*, and a possibly different query plan produced. 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 8 KB. 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.

Most recompilations in SQL Server are performed for good reasons. Some of them are necessary to ensure statement correctness; others are performed to obtain potentially better query execution plans as data in a SQL Server database changes. Sometimes, however, recompilations can slow down batch executions considerably, and then, it becomes necessary to reduce occurrences of recompilations.

Top Of Page

Comparison of Recompilations in SQL Server 2000 and SQL Server 2005

When a batch is recompiled in SQL Server 2000, *all* of the statements in the batch are recompiled, not just the one that triggered the recompilation. SQL Server 2005 improves upon this behavior by compiling only the statement that caused the recompilation, not the entire batch. This **"statement-level recompilation"** feature will improve SQL Server 2005's recompilation behavior when compared to that of SQL Server 2000. In particular, SQL Server 2005 spends less CPU time and memory during batch recompilations, and obtains fewer compile locks.

One benefit of statement-level recompilations should be obvious: it is no longer necessary to break a long stored procedure into multiple short stored procedures just to reduce recompilation penalty of the long stored procedure.

Top Of Page

Plan Caching

Before tackling the issues of recompilations, this paper devotes considerable space to a related and important topic of query plan caching. Plans are cached for possible reuse opportunities. If a query plan is not even cached, its reuse opportunity is zero. Such a plan will be compiled every time it is executed, resulting in poor performance. In rare cases, non-caching is a desirable, and this paper will point out such cases later on.

SQL Server can cache query plans for many types of batches. An enumeration of different types follows. With each type, we describe the *necessary* conditions for plan reuse. Note that these conditions may not be *sufficient*. The reader will get a complete picture later on in this paper.

1. **Ad-hoc queries**. An ad-hoc query is a batch that contains one SELECT, INSERT, UPDATE, or DELETE statement. SQL Server requires exact text match for two ad-hoc queries. The text match is both case- and space-sensitive. For example, the following two queries do not share the same query plan. (All T-SQL code snippets appearing in this white paper are posed on SQL Server 2005's AdventureWorks database.)

   ```
   SELECT ProductID
   FROM Sales.SalesOrderDetail
   GROUP BY ProductID
   HAVING AVG(OrderQty) > 5
   ORDER BY ProductID
   SELECT ProductID

   FROM Sales.SalesOrderDetail
   GROUP BY ProductID
   HAVING AVG(OrderQty) > 5
   ORDER BY ProductId
   ```

2. **Auto-parameterized queries**. For certain queries, SQL Server 2005 replaces constant literal values by variables, and compiles query plans. If a subsequent query differs in only the values of the constants, it will match against the auto-parameterized query. In general, SQL Server 2005 auto-parameterizes those queries whose query plans do no depend on particular values of the constant literals.

   Appendix A contains a list of statement types for which SQL Server 2005 does not auto-parameterize.

   As an example of auto-parameterizaton in SQL Server 2005, the following two queries can reuse a query plan:

   ```
   SELECT ProductID, SalesOrderID, LineNumber
    FROM Sales.SalesOrderDetail
   WHERE ProductID > 1000
   ORDER BY ProductID

   SELECT ProductID, SalesOrderID, LineNumber
    FROM Sales.SalesOrderDetail
   WHERE ProductID > 2000
   ORDER BY ProductID
   ```

   The auto-parameterized form of the above queries is:

   ```
   SELECT ProductID, SalesOrderID, LineNumber
    FROM Sales.SalesOrderDetail
   WHERE ProductID > @p1
   ORDER BY ProductID
   ```

When values of constant literals appearing in a query can influence a query plan, the query is not-autoparameterized. Query plans for such queries *are* cached, but with constants plugged in, not placeholders such as @p1.

SQL Server's "showplan" feature can be used to determine whether a query has been auto-parameterized. For example, the query can be submitted under "set showplan_xml on" mode. If the resulting showplan contains such placeholders as @p1 and @p2, then the query has been auto-parameterized; otherwise not. SQL Server's 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).

3. **sp_executesql procedure**. This is one of the methods that promote query plan reuse. When using *sp_executesql*, a user or an application explicitly identifies the parameters. For example:

```
EXEC sp_executesql N'SELECT p.ProductID, p.Name, p.ProductNumber
FROM Production.Product p
INNER JOIN Production.ProductDescription pd
ON p.ProductID = pd.ProductDescriptionID
WHERE p.ProductID = @a', N'@a int', 170

EXEC sp_executesql N'SELECT p.ProductID, p.Name, p.ProductNumber
FROM Production.Product p INNER JOIN Production.ProductDescription pd
ON p.ProductID = pd.ProductDescriptionID
WHERE p.ProductID = @a', N'@a int', 1201
```

Multiple parameters can be specified by listing them one after another. The actual parameter values follow the parameter definitions. The plan reuse opportunities are predicated on matches of the query-text (the first argument after *sp_executesql*), and on all of the parameters following the query-text (N'@a int' in the above example). The parameter values (170 and 1201) are not considered for text matches. Therefore, in the preceding example, plan reuse can happen for the two *sp_executesql* statements.

4. **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 2005 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/SQLExecute* and *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)
```

5. **Stored procedures (including triggers)**. Stored procedures are designed to promote plan reuse. The plan reuse is based the stored procedure's or trigger's name. (It is not possible to call a trigger directly, however.) 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. Plan caching and recompilation behavior of triggers differs slightly from that of stored procedures. We will point out the differences at appropriate places in this document.

When a stored procedure is compiled for the first time, the values of the parameters 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. This paper will subsequently discuss techniques that can be used to prevent caching of query plans with atypical stored procedure parameter values.

6. **Batches**. Query plan reuse can happen for batches if the batch text matches exactly. This text match is also case-sensitive and space-sensitive.

7. **Executing queries via EXEC (...)**. SQL Server 2005 can cache strings submitted via EXEC for execution. These are known as "dynamic SQL." For example:

   ```
   EXEC ( 'SELECT *' + ' FROM Production.Product pr
   INNER JOIN Production.ProductPhoto ph' + '
   ON pr.ProductID = ph.ProductPhotoID' +
   ' WHERE pr.MakeFlag = ' + @mkflag )
   ```

   Plan reuse is based on the concatenated string that results after replacing variables such as *@mkflag* in the example above with their actual values when the statement is executed.

Multiple levels of caching

It is important to understand that **cache matches at multiple "levels" happen independently of one another**. Here is an 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 (and execution context reuse, explained later in this paper) 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. Each separate execution of *procA* gets its own execution context, however. That execution context is either freshly generated (if all of the existing execution contexts are in use) or reused (if an unused execution context is available). 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. In short, the following three 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

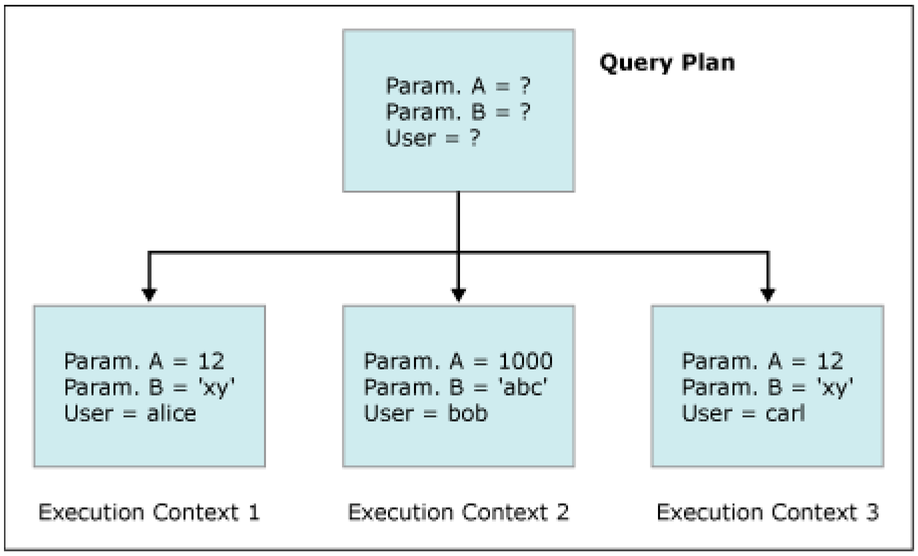
Stored procedures are an **exception** to the above-mentioned rule. For example, query plan and execution context reuse of procA does **not** happen if two different stored procedures contain the statement "EXEC procA".

Query plans and execution contexts

When a cache-able batch is submitted to SQL Server 2005 for execution, it is compiled and a **query plan** for it is put in the plan cache. Query plan is a read-only reentrant structure that is shared by multiple users. There are at most two instances of a query plan at any time in plan cache: one for all of the serial executions and one for all of the parallel executions. The copy for parallel executions is 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 2005 simultaneously, two query plans exists while they execute. However, at the end of their executions, plan for only one of them is retained in the plan cache.)

From a query plan, an **execution context** is derived. An execution context is what is "executed" to produce query results. 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 reentrant (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 is agnostic to both parameter values and user-specific information.



A query plan and multiple associated execution contexts can coexist in plan cache. However, just 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 found (causing execution context reuse) or freshly generated. So query plan reuse does not necessarily imply execution context reuse.

Execution contexts are derived "on the fly" in that 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. Impact of such performance differences diminishes over time as the plan cache gets "hot" and 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. If a batch execution generates a warning (an error with severity 10), the execution context is *not* 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.

Execution contexts for parallel plans are *not* cached. A necessary condition for SQL Server to compile a parallel query plan is that the minimum of the number of processors that have survived the processor affinity mask and the value of the "max degree of parallelism" server-wide option (possibly set using the "sp_configure" stored procedure) is greater than 1. Even if a parallel query plan is compiled, SQL Server's "Query Execution" component may generate a serial execution context out of it. Any execution contexts derived out of a parallel plan — serial or parallel — are not cached. A parallel query plan, however, *is* cached.

Query plan caching and various SET options (showplan-related and others)

Various SET options — most of them showplan-related— affect compilation, caching, and reuse of query plans and execution contexts in complex ways. The following table summarizes the details.

The table should be read as follows. A batch is submitted to SQL Server under a specific mode specified in the first column. A cached query plan may or may not exist in the plan cache for the submitted batch. Columns 2 and 3 cover the cases when a cached query plan exists; columns 4 and 5 cover the cases when a cached query plan does not exist. Within each category, the cases for query plans and execution contexts are separated. The text explains what happens to a structure (query plan or execution context): whether it is cached, reused, and used.

Mode name	**When a cached query plan exists**	**When a cached query plan exists**	**When a cached query plan does not exist**	**When a cached query plan does not exist**
	Query plan	**Exec context**	**Query plan**	**Exec context**
showplan_text, showplan_all, showplan_xml	Reused (no compilation)	Reused	Cached (compilation)	One exec context is generated, not used, and cached
statistics profile, statistics xml, statistics io, statistics time	Reused (no compilation)	Not reused. A fresh exec context is generated, used, and **not** cached	Cached (compilation)	One exec context generated, used, and **not** cached
noexec	Reused (no compilation)	Reused	Cached (compilation)	Execution context is not generated (because of the "noexec" mode).
parseonly (e.g., pressing "parse" button in Query	n/a	n/a	n/a	n/a

Analyzer or Management Studio)				

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. In SQL Server 2000 and SQL Server 2005, the costs are calculated and manipulated differently. Here are the details.

SQL Server 2000: For a query plan, the cost is a measure of the server resources (CPU time and I/O) it takes the query optimizer to optimize the batch. For ad-hoc queries, the cost is zero. For an execution context, the cost is a measure of the server resources (CPU time and I/O) it takes for the server to initialize the execution context so that the individual statements become ready for execution. Note that execution context costs do not include costs (CPU and I/O) incurred during batch executions. Typically, execution context costs are lower than query plan costs.

Here is how a query plan for a batch is costed in SQL Server 2000. The four factors affecting cost are: CPU time spent in generating the plan (*cputime*); number pages read from disk (*ioread*); number of pages written to disk (*iowrite*); and number of memory pages occupied by the query plan of the batch (*pagecount*). The query plan cost can then be expressed as (*f* is a mathematical function):

```
Query plan cost c = f(cputime, ioread, iowrite) / pagecount
```

Here is how an execution context for a batch is costed in SQL Server 2000. The individual costs *c* given by the above equation are calculated for every statement in the batch, and are accumulated. Note, however, that the individual costs are now statement initiation costs, not statement compilation or execution costs.

Occasionally, the lazy-writer process makes sweeps through the plan cache and decrements costs. The costs are divided by four, and rounded down if necessary. (For example, 25 --> 6 --> 1 --> 0.) In case of memory pressure, query plans and execution contexts with costs of 0 are deleted from the plan cache. When a query plan or an execution context is reused, its cost is reset back to its compilation (or execution context generation) cost. The cost of the query plan for an ad-hoc query is always incremented by 1. Thus, query plans for frequently executed batches live in plan cache longer than plans for infrequently executed batches do.

SQL Server 2005: The cost of an ad-hoc query is zero. Otherwise, the cost of a query plan is a measure of the amount of resources required to produce it. Specifically, the cost is calculated in "**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.

In SQL Server 2005, the plan cache is distinct from the data cache. In addition, there are other functionality-specific caches. The lazy-writer process does not decrement costs in SQL Server 2005. Instead, 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 each. Notice that because this decrement is piggybacked 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 2005 reaches or exceeds 75% of the buffer pool size, a dedicated resource monitor thread gets activated, and it decrements tick counts of all of the objects in all of the caches. (So this thread's behavior approximates that of the lazy-writer thread in SQL Server 2000.) A query plan reuse causes the query plan cost to be reset to its initial value.

Top Of Page

Roadmap to the rest of the paper

It should be clear to the reader at this point that in order to obtain good SQL Server batch execution performance, the following two things need to happen:

Query plans should be reused whenever possible. This avoids unnecessary query compilation costs. Plan reuse 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 with 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.

Top Of Page

Query plan reuse

The plan cache contains query plans and execution contexts. A query plan is conceptually linked to its associated execution contexts. Query plan reuse for a batch S is dependent on S itself (for example, the query text or the stored procedure name), and on some factors external to the batch (for example, the user name that generated S, the application which generated S, the SET options of the connection associated with S, and so on). Some of the external factors are plan-reuse-affecting in that if two identical queries that differ only in one such factor will not be able to use a common plan. Other external factors are not plan-reuse-affecting.

Most of the plan-reuse-affecting factors are exposed through columns of the *sys.syscacheobjects* virtual table. The following list describes the factors in "typical usage" scenarios. In some cases, entries simply point out when plans are not cached (and hence never reused) no matter what.

In general, a query plan can be reused if the server, database, and connection settings of the connection that caused the query plan to be cached are the same as the corresponding settings of the current connection. Second, the objects that the batch references do not require name resolutions. For example, *Sales.SalesOrderDetai*l does not require name resolution, whereas *SalesOrderDetail* does because there could be tables named *SalesOrderDetail* in multiple databases. In general, fully qualified object names provide more opportunities for plan reuse.

Factors that affect plan-reuse

Note that if a query plan is not even cached, it cannot be reused. Therefore, we will explicitly point out only non-cachability; non-reuse is then an implication.

1. If a stored procedure is executed in database D1, its query plan is not reused with an execution of the same stored procedure in a different database D2. Note that this behavior applies only to stored procedures, and not to ad-hoc queries, prepared queries, or dynamic SQL.

2. For a trigger execution, the number of rows affected by that execution (*1* versus *n*) — as measured by the number of rows in either *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 SQL Server 2005 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).

3. Bulk insert statements are never cached, but triggers associated with bulk inserts *are* cached.
4. A batch that contains any one literal longer than 8 KB is not cached. Therefore, query plans for such batches cannot be reused. (A literal's length is measured after constant folding is applied.)
5. Batches flagged with the "replication flag" (which is associated with a replication user) are not matched with batches without that flag.
6. A batch called from SQL Server 2005's 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 observation applies to:

   CLR triggers and non-CLR triggers

   Notification queries and non-notification queries

7. Query plans for queries submitted via *sp_resyncquery* are not cached. Therefore, if the query is resubmitted (via *sp_resyncquery* or otherwise), it needs to be compiled again.
8. SQL Server 2005 allows cursor definition on top of a T-SQL batch. If the batch is submitted as a separate statement, then it does not reuse (part of the) plan for that cursor.
9. The following SET options are plan-reuse-affecting.

Number	**SET option name**
1	ANSI_NULL_DFLT_OFF
2	ANSI_NULL_DFLT_ON
3	ANSI_NULLS
4	ANSI_PADDING
5	ANSI_WARNINGS
6	ARITHABORT
7	CONCAT_NULL_YIELDS_NULL

8	DATEFIRST
9	DATEFORMAT
10	FORCEPLAN
11	LANGUAGE
12	NO_BROWSETABLE
13	NUMERIC_ROUNDABORT
14	QUOTED_IDENTIFIER

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 above SET options are plan-reuse-affecting because SQL Server 2000 and SQL Server 2005 perform "constant folding" (evaluating a constant expression at compile time to enable some optimizations) and because settings of these options affects the results of such expressions.

Settings of some of these SET options are exposed through columns of *sys.syscacheobjects* virtual table — for example, "langid" and "dateformat."

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

Using sp_configure stored procedure (for server-wide changes)

Using sp_dboption stored procedure (for database-wide changes)

Using SET clause of the ALTER DATABASE statement

In case of conflicting SET option values, 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 effective, 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.

Best Practice: 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.

10. Batches with unqualified object names result in non-reuse of query plans. For example, in "SELECT * FROM MyTable", MyTable may legitimately resolve to Alice.MyTable if Alice issues this query, and she owns a table with that name. Similarly, MyTable may resolve to Bob.MyTable. In such cases, SQL Server does not reuse query plans. If, however, Alice issues "SELECT * FROM dbo.MyTable", there is no ambiguity because the object is uniquely identified, and query plan reuse can happen. (See the *uid* column in *sys.syscacheobjects*. It indicates the user ID for the connection in which the plan was generated. Only query plans with the same

user ID are candidates for reuse. When *uid* = -2, it means that the query does not depend on implicit name resolution, and can be shared among different user IDs.)

11. When a stored procedure is created with "CREATE PROCEDURE ...WITH RECOMPILE" option, its query plan is not cached whenever that stored procedure is executed. No opportunity of plan reuse exists: every execution of such a procedure causes a fresh compilation.
12. **Best Practice**: "CREATE PROCEDURE ... WITH RECOMPILE" can be used to mark stored procedures that are called with widely varying parameters, and for which best query plans are highly dependent on parameter values supplied during calls.
13. When a stored procedure P is executed using "EXEC ... WITH RECOMPILE", P is freshly compiled. Even if a query plan for P preexists in plan cache, and could be reused otherwise, reuse does not happen. The freshly compiled query plan for P is *not* cached.

Best Practice: 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.

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

1. To avoid multiple query plans for a query that is executed with different parameter values, execute the query using *sp_executesql* stored procedure. This method is useful if the same query plan is good for all or most of the parameter values.
2. Query plans of temporary stored procedures (both session-scoped and global) *are* cached, and therefore, can be reused.
3. In SQL Server 2005, plans for queries that create or update statistics (either manually or automatically) are not cached.

Top Of Page

Causes of Recompilations

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

Correctness-related reasons. A batch must be recompiled if not doing so would result in incorrect results or actions. Correctness-related reasons fall into two sub-categories.

Schemas of objects. A batch B may reference many objects (tables, views, indexes, statistics, UDFs, and so on), and if schemas of some of the objects have changed since B was last compiled, B needs to be recompiled for statement-correctness reasons.

SET options. Some of the SET options affect query results. If the setting of such a plan-reuse-affecting SET option is changed inside of a batch, a recompilation happens.

Plan optimality-related reasons. Data in tables that B references may have changed considerably since B was last compiled. In such cases, B may be recompiled for obtaining a potentially faster query execution plan.

The following two sections describe the two categories in detail.

Correctness-related reasons of batch recompilations

An enumeration of specific actions that cause correctness-related recompilations follows. Because such recompilations must happen, the choice for a user is to not take those actions, or to take them during off-peak hours of SQL Server operation.

Schemas of objects

1. Whenever a schema change occurs for any of the objects referenced by a batch, the batch is recompiled. "Schema change" is defined by 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 (*only if* the index is used by the query plan in question)

   (SQL Server 2000). Manually updating or dropping a statistic (*not* creating!) on a table will cause a recompilation of any query plans that use that table. Such recompilations happen the next time the query plan in question begins execution.

   (SQL Server 2005). Dropping a statistic (not creating or updating!) defined on a table will cause a correctness-related recompilation of any query plans that use that table. Such recompilations happen the next time the query plan in question begins execution. Updating a statistic (both manual and auto-update) will cause an optimality-related (data related) recompilation of any query plans that uses this statistic.

2. Running *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.

3. 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 SQL Server 2000 (on SQL Server 2000)

   Upgrading a database to SQL Server 2005 (on SQL Server 2005 server)

   DBCC FREEPROCCACHE command

   RECONFIGURE command

   ALTER DATABASE ... MODIFY FILEGROUP command

   Modifying a collation using ALTER DATABASE ... COLLATE command

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

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

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.

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

```
use master
go
<-- A query Q that references a database called db1 -->
go
```

Suppose that Q is cached in the plan cache. The database ID associated with Q's plan will be that of the "master," and *not* that of "db1."

When SQL Server 2005'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.

SET options

As already mentioned in Section 6, 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-related reasons of batch recompilations

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, let us enumerate the conditions under which plan optimality-related recompilations do **not** happen.

When the plan is a "trivial plan." A trivial plan results when the query optimizer determines that given the tables referenced in the query and the indexes existing on them, **only one** plan is possible. Obviously, a recompilation would be futile in such a case. A query that has generated a trivial plan may not always generate a trivial plan, of course. For examples, new indexes might be created on the underlying tables, and so multiple access paths become available to the query optimizer. Additions of such indexes would be detected as mentioned in Section 7.1, and a correctness-related recompilation might replace the trivial plan with a non-trivial one.

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

When all of the tables referenced in the query plan are read-only, the plan is not recompiled.

This point is only applicable to SQL Server 2000. Consider a query plan is being compiled (not recompiled), and as a part of compilation, the query processor decides to update a statistic S on a table T. The query processor attempts to obtain a special "statistic lock" on T. If some other process is updating *some* statistic on T (not necessarily S!), the query processor cannot obtain a statistic lock on T. In such a case, the query processor does not update S. Furthermore, the query plan in question is **never** recompiled again for plan optimality-related reasons. It is as if the query were submitted with a "KEEPFIXED PLAN" hint.

This case is identical to the one mentioned in the previous bullet except that in this case, the query plan is cached. In other words, this case is about a recompilation, as opposed to the previous case which is about a compilation. In case of such a recompilation, suppose that the query processor attempts to obtain a "statistic lock" on T and fails. In this case, the query processor skips updating the statistic S; uses a stale statistic S; and proceeds with the other recompilation steps/checks as usual. Thus, a recompilation is avoided at the cost of a potentially slower query execution plan.

High-level overview of query compilation

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

1. 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.)
2. All of the "interesting" statistics that may help to generate an optimal query plan are loaded from disk into memory.
3. If any of the statistics are outdated, they are updated one-at-a-time. The query compilation waits for the updates to finish. An important difference between SQL Server 2000 and SQL Server 2005 regarding this step is that in SQL Server 2005, 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.
4. The query plan is generated. Recompilation thresholds of all of the tables referenced in the query are stored along with the query plan.
5. At this point, query execution has technically begun. The query plan is now tested for correctness-related reasons. Those reasons were described in Section 7.1.
6. 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*.
7. If the plan is "correct," then various recompilation thresholds are compared with either table cardinalities or various table modification counters (*rowmodctr* in SQL Server 2000 or *colmodctr* in SQL Server 2005).

8. If any of the statistics are deemed out-of-date as per the comparisons performed in Step 7, a recompilation results.
9. If all of the comparisons in Step 7 succeed, actual query execution begins.

Plan optimality-related recompilations: The Big Picture

Each SELECT, INSERT, UPDATE, and DELETE statement accesses one or more tables. Table contents change because of such operations as INSERT, UPDATE, and DELETE. 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 table 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:

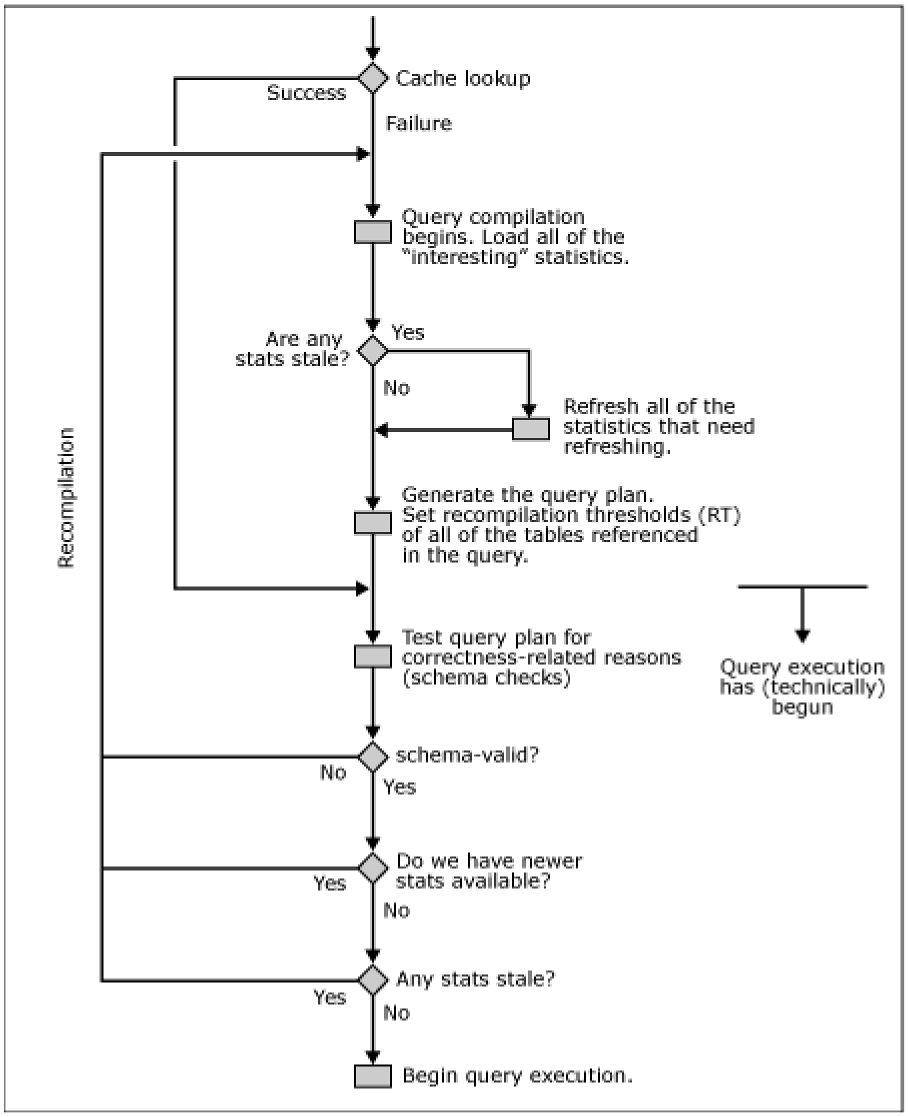
Recompilation threshold

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 *rowmodctr* in SQL Server 2000, and *colmodctr* in SQL Server 2005. 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:

```
| modctr(snapshot) – modctr(current) | >= RT
```

modctr(current) refers to the current value of the modification counter, and *modctr*(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. In SQL Server 2000, the *entire batch* containing the query is recompiled; in SQL Server 2005, only the query in question is recompiled.


Cache lookup
Success
Failure
Query compilation begins. Load all of the "interesting" statistics.
Are any stats stale?
Yes
No
Refresh all of the statistics that need refreshing.
Generate the query plan. Set recompilation thresholds (RT) of all of the tables referenced in the query.
Recompilation
Test query plan for correctness-related reasons (schema checks)
Query execution has (technically) begun
schema-valid?
No
Yes
Do we have newer stats available?
Yes
No
Any stats stale?
Yes
No
Begin query execution.

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

```
| card(snapshot) - card(current) | >= RT
```

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

The following sections describe the important concepts introduced in the "big picture."

Concept of "interesting" statistics

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

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

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

Recall that 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

DBCC DBREINDEX (but not DBCC INDEXDEFRAG!)

Recompilation threshold (RT)

The recompilation threshold for a table partly determines the frequency with which queries that refer to the table recompile. 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.

Table modification counters (*rowmodctr* and *colmodctr*)

As mentioned previously, RT is compared against the number of modifications that a table has undergone. The number of modifications that a table has undergone is tracked using counters called *rowmodctr* (in SQL Server 2000) and *colmodctr* (in SQL Server 2005). Neither of these counters is transactional. For example, if a transaction starts, inserts 100 rows into a table, and then is rolled back, the changes to the modification counters will not be rolled back.

Rowmodctr (SQL Server 2000)

One *rowmodctr* is associated with each table. Its value is available from *sysindexes* system table. A snapshot value of the *rowmodctr* is associated with every statistic created on one or more columns of a table or an indexed view T. Whenever this statistic is updated — either manually or automatically (by SQL Server's auto-stats feature) — the snapshot value of the *rowmodctr* is also refreshed.

Rowmodctr(current), mentioned in the "threshold-crossing" test, is the value persisted in the *sysindexes* system table (for either heap or clustered index) when the test is performed during query compilation.

rowmodctr is available in SQL Server 2005 servers, but its values is always 0.

As an aside, in SQL Server 2000, when a *rowmodctr* is 0, it cannot cause a recompilation.

Colmodctr (SQL Server 2005)

Unlike *rowmodctr*, a *colmodctr* value is stored *for each table column* (except for computed non-persisted columns). Persisted computed columns have *colmodctr*s, just like ordinary columns do. Using *colmodctr* values, changes to a table can be tracked on a finer granularity. *Colmodctr* values are not available to users; they are 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 T, 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 2005's metadata when the test is performed during query compilation.

Unlike *rowmodctr*'s, *colmodctr*'s values are an ever-increasing sequence: *colmodctr* values are never reset to 0.

Colmodctr values for non-persisted computed columns do not exist. They are derived from the columns that participate in the computation.

Tracking changes to tables and indexed views using rowmodctr and colmodctr

Because *rowmodctr* and *colmodctr* values are used to make recompilation decisions, their values are modified as 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, bulk insert, and table truncation. The following table defines how rowmodctr and colmodctr values are modified.

Statement	SQL Server 2000	SQL Server 2005
INSERT	*rowmodctr* += 1	All *colmodctr* += 1
DELETE	*rowmodctr* += 1	All *colmodctr* += 1
UPDATE	*rowmodctr* += 2 or 3. Rationale for "2": 1 for delete + 1 for insert.	If the update is to non-key columns: *colmodct*r += 1 for all of the updated columns. If the update is to key columns: *colmodctr* += 2 for *all* of the columns.
Bulk insert	No change.	Like *n* INSERTs. All colmodctr += *n*. (*n* is the number of rows bulk inserted.)

Table truncation	No change.	Like *n* DELETEs. All *colmodctr* += *n*. (*n* is the table's cardinality.)

Two 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

SQL Server 2005 handles the following scenario **differently** from SQL Server 2000. A user creates an empty table T. She then creates a statistic S on one or more columns of T. Because T is empty, the stats-blob (histogram) is NULL, but the statistic has been created on T. Suppose that S has been found "interesting" during a query compilation. As per the "500 row" rule for recompilation threshold, T will cause recompilations on SQL Server 2000 only after T contains at least 500 rows. Therefore, a user may suffer from sub-optimal plans until T contains at least 500 rows.

SQL Server 2005 detects this special case, and handles it differently. In SQL Server 2005, recompilation threshold for such a table or indexed view is **1**. In other words, even the insertion of one row in T can cause a recompilation. When such a recompilation happens, S is updated, and the histogram for S is no longer NULL. After this recompilation, however, the usual rule for recompilation threshold (500 + 0.20 * *n*) is followed.

In SQL Server 2005, the recompilation threshold is 1 even when: (1) T has no statistics; or (2) T 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 *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).

In SQL Server 2005, "sufficiently different" is defined by:

```
| log10(n) - log10(m) | > 1          if m > n
| log10(n) - log10(m) | > 2.1       otherwise
```

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 "inserted" and "deleted" tables exist, 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.

In SQL Server 2000, "sufficiently different" is defined by:

```
| log10(n+5) - log10(m+5) | >= 1
```

where *n* and *m* are defined as before. Notice that as per this formula, in SQL Server 2000, a change in cardinality of either *inserted* or *deleted* table from 5 to 95 will cause a recompilation, whereas a change from 5 to 94 will not.

Identifying statistics-related recompilations

Statistics-related recompilations can be identified by the "EventSubClass" column of the profiler trace (to be described later in this paper) containing the string "Statistics changed".

Closing remarks

An issue not directly related to the topic of this document is: 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: Give preference to recent statistics over older statistics; Give preference to statistics computed using FULLSCAN option to those computed using sampling; and so on.

There is a potential of confusion regarding the "cause and effect" relationship between plan optimality-related compilations, recompilations, and statistics creation/updates. Recall that statistics can be created or updated manually or automatically. Only compilations and recompilations cause automatic creation or updates of statistics. On the other hand, when a statistic is created or updated (manually or automatically), there is an increased chance of recompilation of a query plan which might find that statistic "interesting."

Best practices

Four best practices for reducing plan optimality-related batch recompilations are given next:

Best Practice: Because a change in cardinality of a table variable does not cause recompilations, consider using a table variable instead of a temporary table. However, because the query optimizer does not keep track of a table variable's cardinality and because statistics are not created or maintained on table variables, non-optimal query plans might result. One has to experiment whether this is the case, and make an appropriate trade-off.

Best Practice: 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:

```
SELECT B.col4, sum(A.col1)
FROM dbo.PermTable A INNER JOIN #TempTable B ON A.col1 = B.col2
WHERE B.col3 < 100
GROUP BY B.col4
OPTION (KEEP PLAN)
```

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

```
SELECT c.TerritoryID, count(*) as Number, c.SalesPersonID
FROM Sales.Store s INNER JOIN Sales.Customer c
ON s.CustomerID = c.CustomerID
WHERE s.Name LIKE '%Bike%' AND c.SalesPersonID > 285
GROUP BY c.TerritoryID, c.SalesPersonID
ORDER BY Number DESC
OPTION (KEEPFIXED PLAN)
```

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

In SQL Server 2005, there is a slight change in behavior as described below. Suppose that 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 2005 can get a special "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. In SQL Server 2000, a query with OPTION (KEEPFIXED PLAN) is *never* recompiled because of statistics-related reasons, and therefore, in this scenario, no attempt is made to get a "stats lock" or to auto-create the statistic.

Best Practice: Turning off automatic updates of statistics for indexes and statistics defined on a table or indexed view will ensure that plan optimality-related recompilations caused by those objects will stop. Note, however, that turning off the "auto-stats" feature using this method is usually not a good idea because the query optimizer is no longer sensitive to data changes in those objects, and sub-optimal query plans might result. Adopt this method only as a last resort after exhausting all of the other alternatives.

Top Of Page

Compilations, Recompilations, and Concurrency

In SQL Server 2000, compilations and recompilations of stored procedures, triggers, and dynamic SQL are serialized. For example, suppose that a stored procedure is submitted for execution using "EXEC dbo.SP1". Suppose that while SQL Server is compiling SP1, another request "EXEC dbo.SP1" referring to the same stored procedure is received. The second request will wait for the first request to complete the compilation of SP1, and will attempt to reuse the resulting query plan. In SQL Server 2005, compilations are serialized, but recompilations are **not** serialized. In other words, two concurrent recompilations of the same stored procedures may continue. The recompilation request that finishes last will replace the query plan generated by the other one.

Top Of Page

Compilations, Recompilations, and Parameter Sniffing

"Parameter sniffing" refers to a process whereby SQL Server's execution environment "sniffs" the current parameter values during compilation or recompilation, and passes it along 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. Both in SQL Server 2000 and SQL Server 2005, parameter values are sniffed during compilation or recompilation for the following types of batches:

- Stored procedures
- Queries submitted via *sp_executesql*
- Prepared queries

In SQL Server 2005, the behavior is extended for queries submitted using the OPTION(RECOMPILE) query hint. For such a query (could be SELECT, INSERT, UPDATE, or DELETE), **both** the parameter values **and** the current values of local variables are sniffed. The values sniffed (of parameters and local variables) 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.

Top Of Page

Identifying Recompilations

SQL Server's Profiler makes it easy to identify batches that cause recompilations. Start a new profiler trace 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

In addition, to detect statistics-update-related recompilations, the "Auto Stats" event under "Objects" class can be selected.

Now start SQL Server 2005 Management Studio, and execute the following T-SQL code:

```
use AdventureWorks          -- On SQL Server 2000, say "use pubs"
go
drop procedure DemoProc1
go
create procedure DemoProc1 as
create table #t1 (a int, b int)
select * from #t1
go
exec DemoProc1
go
exec DemoProc1
go
```

Pause the profiler 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	

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.

An identical behavior is observed in SQL Server 2000.

Batches 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	

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 profiler trace events henceforth.

To see all of the possible recompilation reasons reported for SP:Recompile event, issue the following query on SQL Server 2005:

```
select v.subclass_name, v.subclass_value
from sys.trace_events e inner join sys.trace_subclass_values v
on e.trace_event_id = v.trace_event_id
where e.name = 'SP:Recompile'
```

The output of the above query is as follows. (Only the unshaded columns are output; the shaded column is for additional details.)

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.
Query notification environment changed	8	(NEW in SQL Server 2005!)
Partition view changed	9	SQL Server 2005 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. (NEW in SQL Server 2005!)

In SQL Server 2000, the EventSubClass column contains integer values 1 through 6 with the same meanings as those mentioned in the above table. The last two categories are new in SQL Server 2005.

For both of the examples presented in this section, trace output on SQL Server 2000 is identical to that on SQL Server 2005, except that on SQL Server 2000, EventSubClass column contains 3, not the string "Deferred compile". Internally, statement-level recompilations happen on SQL Server

2005. Thereby, only the "select * from #t1" is recompiled on SQL Server 2005, whereas on SQL Server 2000, the entire DemoProc1 is recompiled.

Recompilations due to mixing DDL and DML

Mixing Data Definition Language (DDL) and Data Manipulation Language (DML) statements within a batch or stored procedure is a bad idea because it can cause unnecessary recompilations. The following example illustrates this using a stored procedure. (The same phenomenon happens for a batch also. However, because SQL Server 2005 Profiler does not provide the necessary tracing events, we cannot observe it in action.) Create the following stored procedure.

```
drop procedure MixDDLDML
go
create procedure MixDDLDML as
create table tab1 (a int)            -- DDL
select * from tab1                   -- DML
create index nc_tab1idx1 on tab1(a)  -- DDL
select * from tab1                   -- DML
create table tab2 (a int)            -- DDL
select * from tab2                   -- DML
go
exec MixDDLDML
go
```

In the profiler trace output, the following sequence of events can be observed.

EventClass	TextData	EventSubClass
SP:Starting	exec MixDDLDML	
SP:StmtStarting	-- MixDDLDML create table tab1 (a int) --DDL	
SP:StmtStarting	-- MixDDLDML select * from tab1 -- DML	
SP:Recompile		Deferred compile
SP:StmtStarting	-- MixDDLDML select * from tab1 -- DML	
SP:StmtStarting	-- MixDDLDML create index nc_tab1idx1 on tab1(a) -- DDL	
SP:StmtStarting	-- MixDDLDML select * from tab1 -- DML	
SP:Recompile		Deferred compile
SP:StmtStarting	-- MixDDLDML select * from tab1 -- DML	
SP:StmtStarting	-- MixDDLDML create table tab2 (a int) --DDL	

SP:StmtStarting	-- MixDDLDML select * from tab2 -- DML	
SP:Recompile		Deferred compile
SP:StmtStarting	-- MixDDLDML select * from tab2 -- DML	
SP:Completed	exec MixDDLDML	

Here is how the MixDDLDML is compiled.

1. During the first compilation (not recompilation) of MixDDLDML, a skeleton plan for it is generated. Because tables tab1 and tab2 do not exist, plans for the three "select" statements cannot be produced. The skeleton contains plans for the two "create table" statements and the one "create index" statement.
2. When the procedure begins execution, table tab1 is created. Because there is no plan for the first "select * from tab1", a statement-level recompilation happens. (In SQL Server 2000, a plan for the second "select * from tabl" will also be generated by this recompilation.)
3. The second "select * from tab1" causes a recompilation because a plan for that query does not yet exist. In SQL Server 2000, this recompilation happens but for a different reason: the schema for "tab1" has changed because of the creation of a non-clustered index on "tab1".
4. Next, "tab2" gets created. "select * from tab2" causes a recompilation because a plan for that query did not exist.

In conclusion, three recompilations happen in both SQL Server 2000 and SQL Server 2005 for this example. The SQL Server 2005 recompilations, however, are cheaper than SQL Server 2000 ones because they are statement-level rather than stored procedure-level.

If the stored procedure is written as follows, an interesting phenomenon is observed.

```
create procedure DDLBeforeDML as
create table tab1 (a int)              -- DDL
create index nc_tab1idx1 on tab1(a)    -- DDL
create table tab2 (a int)              -- DDL
select * from tab1                     -- DML
select * from tab1                     -- DML
select * from tab2                     -- DML
go
exec DDLBeforeDML
go
```

In the profiler trace output, the following sequence of events can be observed.

EventClass	TextData	EventSubClass
SP:Starting	exec DDLBeforeDML	
SP:StmtStarting	-- DDLBeforeDML create table tab1 (a int) -- DDL	

SP:StmtStarting	-- DDLBeforeDML create index nc_tab1idx1 on tab1(a) -- DDL	
SP:StmtStarting	-- DDLBeforeDML create table tab2 (a int) -- DDL	
SP:StmtStarting	-- DDLBeforeDML select * from tab1 --DML	
SP:Recompile		Deferred compile
SP:StmtStarting	-- DDLBeforeDML select * from tab1 --DML	
SP:StmtStarting	-- DDLBeforeDML select * from tab1 --DML	
SP:Recompile		Deferred compile
SP:StmtStarting	-- DDLBeforeDML select * from tab1 --DML	
SP:StmtStarting	-- DDLBeforeDML select * from tab2 -- DML	
SP:Recompile		Deferred compile
SP:StmtStarting	-- DDLBeforeDML select * from tab2 -- DML	
SP:Completed	exec DDLBeforeDML	

In SQL Server 2005, because of statement-level recompilations, three recompilations still happen. When compared with the MixDDLDML stored procedure, the number of recompilations has not reduced. If the same example is tried on SQL Server 2000, the number of recompilations is reduced from 3 to 1. In SQL Server 2000, recompilations are stored procedure-level, and therefore, the three "select" statements could be compiled in one shot. In conclusion, in SQL Server 2005, the recompilation effort has not increased but the number of recompilations has increased when compared to SQL Server 2000.

Next, consider the following T-SQL code snippet:

```
-- dbo.someTable will be used to populate a temp table
-- subsequently.
create table dbo.someTable (a int not null, b int not null)
go
declare @i int
set @i = 1
while (@i <= 2000)
begin
    insert into dbo.someTable values (@i, @i+5)
    set @i = @i + 1
end
go
```

```
-- This is the stored procedure of main interest.
create procedure dbo.AlwaysRecompile
as
set nocount on

-- create a temp table
create table #temp1(c int not null, d int not null)

select count(*) from #temp1

-- now populate #temp1 with 2000 rows
insert into #temp1
select * from dbo.someTable

-- create a clustered index on #temp1
create clustered index cl_idx_temp1 on #temp1(c)

select count(*) from #temp1
go
```

In SQL Server 2000, when this stored procedure is executed the first time, the first SP:Recompile event is generated for the first "select" statement. This is a deferred compile, and not a true recompilation. The second SP:Recompile event is for the second "select". When the first recompilation happened, the second "select" was also compiled because compilations are batch-level in SQL Server 2000. Then, during execution, schema of #temp1 changed because of the newly created clustered index. Therefore, reason for the second SP:Recompile is schema change.

Recompilations due to number of row modifications

Consider the following stored procedure and its execution.

```
use AdventureWorks   -- or say "use pubs" on SQL Server 2000
go
create procedure RowCountDemo
as
begin
    create table #t1 (a int, b int)

    declare @i int
    set @i = 0    while (@i < 20)
    begin
       insert into #t1 values (@i, 2*@i - 50)

       select a
       from #t1
       where a < 10 or ((b > 20 or a >=100) and (a < 10000))
       group by a

       set @i = @i + 1
    end
end
go
exec RowCountDemo
go
```

Recall that the recompilation threshold for a temporary table is 6 when the table is empty when the threshold is calculated. When RowCountDemo is executed, a "statistics changed"-related recompilation can be observed after #t1 contains exactly 6 rows. By changing the upper bound of the "while" loop, more recompilations can be observed.

Recompilations due to SET option changes

Consider the following stored procedure.

```
use AdventureWorks
go
create procedure SetOptionsDemo as
begin
    set ansi_nulls off
    select p.Size, sum(p.ListPrice)
    from Production.Product p
         inner join Production.ProductCategory pc
          on p.ProductSubcategoryID = pc.ProductCategoryID
    where p.Color = 'Black'
    group by p.Size
end
go
exec SetOptionsDemo    -- causes a recompilation
go
exec SetOptionsDemo    -- does not cause a recompilation
go
```

When SetOptionsDemo is executed, the "select" query is compiled with "ansi_nulls" ON. When SetOptionsDemo begins execution, the value of that SET option changes because of "set ansi_nulls off", and therefore the compiled query plan is no longer "valid." It is therefore recompiled with "ansi_nulls" OFF. The second execution does not cause a recompilation because the cached plan is compiled with "ansi_nulls" OFF.

Another example of more recompilations in SQL Server 2005 than in SQL Server 2000

Consider the following stored procedure.

```
use AdventureWorks     -- say "use pubs" on SQL Server 2000
go
create procedure CreateThenReference as
begin
   -- create two temp tables
   create table #t1(a int, b int)
   create table #t2(c int, d int)

   -- populate them with some data
   insert into #t1 values (1, 1)
   insert into #t1 values (2, 2)
   insert into #t2 values (3, 2)
   insert into #t2 values (4, 3)

         -- issue two queries on them
   select x.a, x.b, sum(y.c)
   from #t1 x inner join #t2 y on x.b = y.d
   group by x.b, x.a
   order by x.b

   select *
   from #t1 z cross join #t2 w
   where w.c != 5 or w.c != 2
end
go
exec CreateThenReference
go
```

In SQL Server 2005, CreateThenReference's first execution causes six statement-level recompilations: four for the "insert" statements and two for the "select" queries. When the stored procedure starts executing, the initial query plan does not contain plans for either "insert" or "select" statements because the objects that they reference — temporary tables #t1 and #t2 — do not yet exist. After #t1 and #t2 are created, query plans for "insert" and "select" are compiled, and these compilations count as recompilations. In SQL Server 2000, because the entire stored procedure is recompiled at once, only one (stored procedure level) recompilation — the one that is caused when the first "insert" begins execution — happens. At that time, the entire stored procedure is recompiled, and because #t1 and #t2 already exist, the subsequent "insert"s and "select"s can all be compiled in one attempt. Obviously, the number of statement-level recompilations in SQL Server 2005 can be increased without bound by adding more statements that reference such objects as #t1 and #t2.

Top Of Page

Tools and Commands

This section contains descriptions of various tools and commands that exist in observing and debugging recompilation-related scenarios.

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 *objtype* column: 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.

Please refer to BOL for more information on this virtual table.

DBCC FREEPROCCACHE

This command removes all of the cached query plans and execution contexts from the plan cache. It is 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.

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 because it can adversely affect performance of running applications.

Profiler Trace Events

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

'Cursors: CursorRecompile' (new in SQL Server 2005) 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' (new in SQL Server 2005) 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' (new in SQL Server 2005) 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' fires when a recompilation happens. 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

Values of the following perfmon counters are relevant when debugging performance problems that may be caused by excessive compilations and recompilations.

Performance object	**Counters**
SQLServer: Buffer Manager	Buffer cache hit ratio, Lazy writes/sec, Procedure cache pages, Total pages
SQLServer: Cache Manager	Cache Hit Ratio, Cache Object Counts, Cache Pages, Cache Use Counts/sec
SQLServer: Memory Manager	SQL Cache Memory (KB)
SQLServer:SQL Statistics	Auto-Param Attmpts/sec, Batch Requests/sec, Failed Auto-Params/sec, Safe Auto-Params/sec, SQL Compilations/sec, SQL Re-Compilations/sec, Unsafe Auto-Params/sec

Top Of Page

Conclusion

SQL Server 2005 caches query plans for a variety of statement types submitted to it for execution. Query plan caching causes 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 get non-reusable for two reasons: (a) Schema of an object appearing in a query plan can change thereby making the plan invalid; and (b) 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. Bad T-SQL coding practices can increase recompilation frequency and adversely affect SQL Server's performance. Such situations can be debugged and corrected in many cases.

Top Of Page

Appendix A: When does SQL Server 2005 not auto-parameterize queries?

Auto-parameterization is a process whereby SQL Server replaces literal constants appearing in a SQL statement with such parameters as @p1 and @p2. 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. Values of some of the perfmon counters mentioned in Section 11.5 ('SQLServer: SQL Statistics' category) report statistical information on auto-parameterization.

The following list enumerates the statement types for which SQL Server 2005 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.*

Statements submitted via *sp_executesql*, *sp_prepare*, and *sp_prepexec* without parameters **are** auto-parameterized under TF 447.

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.

Top Of Page