Troubleshooting stored procedure recompilation

When a user executes a stored procedure, if it is not already available in cache, SQL Server loads the procedure and compiles a query plan. The compiled plan is saved in cache and is reused by subsequent callers of the stored procedure until some action occurs to invalidate the plan and force a recompilation. The following actions may cause recompilation of a stored procedure plan:

| • | Use of a WITH RECOMPILE clause in the CREATE PROCEDURE or EXECUTE statement. |
| --- | --- |
| • | Schema changes to any of the referenced objects, including adding or dropping constraints, defaults, or rules. |
| • | Running **sp\_recompile** for a table referenced by the procedure. |
| • | Restoring the database containing the procedure or any of the objects the procedure references (if you are performing cross-database operations). |
| • | Sufficient server activity causing the plan to be aged out of cache. |

All these reasons for recompiling a stored procedure did exist in earlier versions, and caused the plan to recompile before beginning execution of the procedure. In SQL Server 7.0, a new behavior is introduced that may cause a stored procedure to recompile during execution. This new behavior ensures that the optimizer always has the best possible plan for each specific statement within a procedure. The following events may cause a run-time recompilation of a stored procedure:

| • | A sufficient percentage of data changes in a table that is referenced by the stored procedure. |
| --- | --- |
| • | The procedure interleaves Data Definition Language (DDL) and Data Manipulation Language (DML) operations. |
| • | The procedure performs certain operations on temporary tables. |

Each of these causes is discussed in further detail in this article.   
  
In some cases, the cost of recompiling the stored procedure is more than the benefit derived from doing so, especially for large procedures. It is very important to note that when a recompilation is triggered, the *entire* batch or procedure is recompiled. This means that performance degradation is directly proportional to the size of the procedure or batch. For more information about this topic, see the "Transact-SQL Tips" topic in SQL Server Books Online.  
  
  
The following information in this article focuses on identifying the cause of run-time stored procedure recompilations and discusses methods you can use to prevent them.

| Back to the top | Back to the top |
| --- | --- |

**Best Practice**

It is best to owner qualify stored procedure names when you execute a procedure. This allows for better clarity and easier reuse of the existing execution plan by the current user. For example, if a user who is not the database owner (dbo) executes a dbo-owned stored procedure (called **myProc** in this example) in the **pubs** database, use the following statement:

exec dbo.myProc

Instead of this:

exec myProc

This technique eliminates confusion about other possible versions of the procedure by different owners from a coding and maintenance standpoint, and also allows SQL Server to access the execution plan for the specific procedure more directly.   
  
By not qualifying the owner name, SQL Server enters the compile code and acquires a COMPILE lock on the procedure. However, eventually it determines that a new plan is not required (assuming no other reasons apply), so it does NOT recompile the plan at this point due to the lack of qualification. However, the extra step of getting a COMPILE lock on the procedure can cause blocking contention in severe situations. Refer to Q263889 INF: SQL Blocking Due to [[COMPILE]] Locks for more details on this situation.   
  
If you owner qualify the procedure call with owner.procedure, you do not need to acquire the compile lock, so the contention is reduced.

| Back to the top | Back to the top |
| --- | --- |

**Identifying and Resolving Issues**

If you have not already done so, refer to the following article in the Microsoft Knowledge Base for details on capturing Profiler data to help analyze the performance of your system:

[224587](http://support.microsoft.com/kb/224587/) HOW TO: Troubleshoot Application Performance with SQL Server

**View the Profiler Data**

SQL Server Profiler includes an **SP:Recompile** event that you can use to monitor the number of recompiles occurring. The **SP:Recompile** event occurs whenever a stored procedure recompiles during execution.

| • | **Group the Profiler trace by Event Class**   | 1. | On the **File** menu, click **Properties**. | | --- | --- | | 2. | On the **Data Columns** tab, use the UP button to move the **Event Class** and **Text** under the **Groups** heading, with **Event Class** first. Use the DOWN button to remove all other columns under the **Groups** heading. | | 3. | Click **OK**. |   Check the number of the **SP:Recompile** events.   You can expand the SP:Recompile group to see the details of the individual occurrences. The **Text** column of the event indicates the name of the stored procedure that recompiled. If multiple procedures are causing recompiles, they are sorted by the number of occurrences. If you have a large number of **SP:Recompile** events and you are experiencing high CPU usage, focus on resolving the procedures that have the highest number of recompiles. Note the system process ID (SPID) and Start Time of the **SP:Recompile** event for one instance of the particular stored procedure(s) and follow the steps below.  If you do not see any **SP:Recompile** events, but you are still experiencing a performance problem, see the following article in the Microsoft Knowledge Base:  [224587](http://support.microsoft.com/kb/224587/) HOW TO: Troubleshoot Application Performance with SQL Server |
| --- | --- | --- | --- | --- | --- | --- | --- |
| • | **Determine the Statement That Triggered the Recompile Event**   | 1. | On the **File** menu, click **Properties**. | | --- | --- | | 2. | On the **Data Columns** tab, use the DOWN button to remove all other columns under the **Groups** heading. | | 3. | On the **Events** tab, remove all events except **SP:Starting**, **SP:StmtStarting**, **SP:Recompile**, and **SP:Completed**. If you did not capture the **SP:StmtStarting** event, you can substitute **SP:StmtCompleted**, but do not include both because doing so doubles the amount of information you need to look through. | | 4. | If you have identified a particular instance of a stored procedure recompilation to examine, you can limit the data you view to the specific SPID and time frame of the occurrence by using the **Filters** tab. | | 5. | Click **OK**. |   The **SP:Recompile** event will be raised directly following the **SP:StmtStarted** event of the stored procedure statement that caused the recompilation. After the recompile event is finished, you will see a repeat of the **SP:StmtStarted** event, indicating that the statement is executing with the newly generated plan.  Consider the following example:  use pubs go drop procedure RecompProc go create procedure RecompProc as create table #t (a int) select \* from #t go exec RecompProc  If you execute this code in Query Analyzer and view the above events in a Profiler trace, you will see the following sequence:   | **Event Class** | **Text** | | --- | --- | | SP:Starting | RecompProc | | SP:StmtStarting | create table #t (a int) | | SP:StmtStarting | select \* from #t | | SP:Recompile | RecompProc | | SP:StmtStarting | select \* from #t | | SP:Completed | RecompProc |   You can tell immediately that the statement that caused the recompilation was:  select \* from #t  Because it appears both before and after the **SP:Recompile** event.  If you had captured only the **SP:StmtCompleted** event, but not the **SP:StmtStarting** event, the **SP:Recompile** will show directly before the statement which caused it as below:   | **Event Class** | **Text** | | --- | --- | | SP:Starting | RecompProc | | SP:Recompile | RecompProc | | SP:StmtCompleted | select \* from #t | | SP:Completed | RecompProc |   You can see that the **SP:Recompile** event is raised before the **SP:StmtCompleted** event for the "select \* from #t" statement, which caused the recompilation. This makes sense, as the statement can not be completed until after the new query plan is generated for the recompile. All the rest of the examples in this article use the **SP:StmtStarting** event. If you have captured only the **SP:StmtCompleted** event, just remember to view the statement after the **SP:Recompile**, as explained above.  Note that if you execute this particular stored procedure multiple times, SQL Server will reuse the existing plan for this procedure. You will only see the recompile event on the first execution of the procedure, or if you drop and re-create the procedure each time you execute the script. The reason for the recompilation in this specific case is discussed in the "Recompilations Due to Interleaving Data Definition Language (DDL) and Data Manipulation Language (DML) Operations" section of this article; this is simply an example to illustrate how to easily determine which statement is causing the recompilation. |

**Recompilations Due to Row Modifications**

If a sufficient percentage of data has changed in a table referenced by a stored procedure since the time the original query plan was generated, SQL Server will recompile the stored procedure to ensure that it has a plan based on the most up-to-date statistical data. As an example, consider the following stored procedure:

drop procedure RowModifications go create procedure RowModifications as -- assume SomeTable exists with the same definition as #t, -- and has over 1000 rows create table #t (a int, b char(10)) select \* from #t insert #t select \* from SomeTable select count(\*) from #t where a = 37 go exec RowModifications exec RowModifications

For the second execution of the **RowModifications** procedure, you will see the following events in Profiler:

| **Event Class** | **Text** |
| --- | --- |
| SP:Starting | RowModifications |
| SP:StmtStarting | create table #t (a int, b char(10)) |
| SP:StmtStarting | select \* from #t |
| SP:StmtStarting | insert #t select \* from SomeTable |
| SP:StmtStarting | select count(\*) from #t where a = 37 |
| SP:Recompile | RowModifications |
| Auto-UpdateStats | a |
| SP:StmtStarting | select count(\*) from #t where a = 37 |
| SP:Completed | RowModifications |

**NOTE:** The first execution will also show a **SP:Recompile** event for the "select \* from #t" statement. The reason for the recompilation in this specific case is discussed in the "Recompilations Due to Interleaving Data Definition Language (DDL) and Data Manipulation Language (DML) Operations" section of this article. For this example, focus on the **SP:Recompile** shown above because it occurs every time the procedure is executed.   
  
In this example, "select count(\*) from #t where a = 37" causes a recompilation of the procedure due to the change in the number of rows since the table was created. The presence of the **Auto-UpdateStats** event confirms that the recompilation was due to row modifications. The **Text** column indicates the column for which the statistics were modified.  
  
When the #t table was created, the number of rows is zero. The plan for the original "select \* from #t" is developed with that row count, as well as the plan for the "select count (\*)" query. However, before the "select count(\*)" is executed, 1,000 new rows are inserted into the #t table. Because a sufficient amount of data has been changed, the optimizer recompiles the procedure to ensure that it chooses the most efficient plan for the statement. This recompilation will occur on every execution of the stored procedure because the insertion of 1,000 rows will always be viewed as significant enough to warrant recompilation.  
  
The algorithm SQL Server uses to determine whether a plan should be recompiled is the same algorithm used for auto-update statistics as described in the following article in the Microsoft Knowledge Base:

[195565](http://support.microsoft.com/kb/195565/EN-US/) INF: How SQL Server 7.0 and SQL Server 2000 Autostats Work

In the example above, the stored procedure is small enough that the recompilation would not have a noticeable effect on performance. However, if you have a large stored procedure that performs similar activities resulting in multiple recompilations, you may notice a performance degradation.  
  
The following methods exist to counteract recompilations due to row modifications:

| • | Execute the statement using **sp\_executesql**. This is the preferred method. Statements executed using the **sp\_executesql** stored procedure are not compiled as part of the stored procedure plan. Therefore, when executing the statement, SQL Server will be free to either use an existing plan in cache for the statement or create a new one at run time. In either case, the plan for the calling stored procedure is unaffected and does not have to be recompiled.   The EXECUTE statement will have the same effect; however, it is not recommended. Using the EXECUTE statement is not as efficient as using **sp\_executesql** because it does not allow for parameterization of the query.  The **RowModifications** procedure given above can be written to use **sp\_executesql** as follows:  drop procedure RowModifications2 go create procedure RowModifications2 as set nocount on -- assume SomeTable exists with the same definition as #t, -- and has over 1000 rows create table #t (a int, b char(10)) select \* from #t insert #t select \* from SomeTable exec sp\_executesql N'select count(\*) from #t where a = @a', N'@a int', @a = 37 go exec RowModifications2 exec RowModifications2  For the second execution of the **RowModifications2** procedure, you will see the following events in Profiler:   | **Event Class** | **Text** | | --- | --- | | SP:Starting | RowModifications2 | | SP:StmtStarting | create table #t (a int, b char(10)) | | SP:StmtStarting | select \* from #t | | SP:StmtStarting | insert #t select \* from SomeTable | | SP:StmtStarting | exec sp\_executesql N'select count(\*) from #t where a = @a', N'@a int', @a = 37 | | SP:Starting |  | | SP:StmtStarting | select count(\*) from #t where a = @a | | Auto-UpdateStats | a | | SP:StmtStarting | select count(\*) from #t where a = @a | | SP:Completed |  | | SP:Completed | RowModifications2 |   Notice that there are no **SP:Recompile** events for the **RowModifications2** procedure. There are complete **SP:Starting** to **SP:Completed** events for the **sp\_executesql** call context, and an **Auto-UpdateStats** event for column **a**. However, because this call is out of the context of the stored procedure, the **RowModifications2** procedure does not need to be recompiled in this case.   For more information on using the **sp\_executesql** stored procedure, see the "sp\_executesql (T-SQL)" and "Using sp\_executesql" topics in SQL Server Books Online. |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| • | Use sub-procedures to execute the statements that are causing the recompilations. In this case, the statement may still cause a recompilation, but instead of recompiling the large calling stored procedure, it will only recompile the small sub-procedure. |
| • | Use the KEEP PLAN option. Temporary tables have special rules regarding recompilations which, in some cases, can be more stringent than the default recompilation algorithm. You can use the KEEP PLAN option to relax the temporary table threshold back to the default algorithm. For more information, see the "Avoiding Recompilation by Using the KEEP PLAN Option" section of this article. |

**NOTE:** The **RowModifications** procedure is a very simplified example of a procedure that is recompiled due to row modifications. Please review the following caveats concerning this example:

| • | Although the example uses a temporary table, this situation applies to stored procedures that reference permanent tables as well. If a sufficient amount of data in a referenced table has been altered since the query plan was generated, the stored procedure will be recompiled. The differences in how temporary tables are considered for recompilation purposes are described in the "Avoiding Recompilation by Using the KEEP PLAN Option" section of this article. |
| --- | --- |
| • | The first executions of the above two procedures also cause a recompilation on the first select from the temporary table #t. The reasons for this recompilation are discussed in the "Recompilations Due to Interleaving Data Definition Language (DDL) and Data Manipulation Language (DML) Operations" section of this article. |
| • | A "select count(\*) from #t" statement was used in this example, rather than a simple "select \* from #t" statement. To avoid excessive recompilations, SQL Server does not consider recompiling "trivial plans" (such as a select \* from a table) due to row modifications. |

**Recompilations Due to Interleaving Data Definition Language (DDL) and Data Manipulation Language (DML) Operations**

If DDL operations are performed within a procedure or batch, the procedure or batch is recompiled when it encounters the first subsequent DML operation affecting the table involved in the DDL.   
  
Consider the following example stored procedure:

drop procedure Interleave go create procedure Interleave as -- DDL create table t1 (a int) -- DML select \* from t1 -- DDL create index idx\_t1 on t1(a) -- DML select \* from t1 -- DDL create table t2 (a int) -- DML select \* from t2 go exec Interleave

If you execute this code in Query Analyzer and view the above events in a Profiler trace, you will see the following sequence:

| **Event Class** | **Text** |
| --- | --- |
| SP:Starting | Interleave |
| SP:StmtStarting | create table t1 (a int) |
| SP:StmtStarting | select \* from t1 |
| SP:Recompile | Interleave |
| SP:StmtStarting | select \* from t1 |
| SP:StmtStarting | create index idx\_t1 on t1(a) |
| SP:StmtStarting | select \* from t1 |
| SP:Recompile | Interleave |
| SP:StmtStarting | select \* from t1 |
| SP:StmtStarting | create table t2 (a int) |
| SP:StmtStarting | select \* from t2 |
| SP:Recompile | Interleave |
| SP:StmtStarting | select \* from t2 |
| SP:Completed | Interleave |

In this case, the stored procedure is recompiled three times during execution. To understand why this happens, consider how the optimizer develops a plan for this stored procedure:

| 1. | During initial compilation of the procedure, the tables t1 and t2 do not exist. Therefore, no plan for the queries referencing these tables can be created. They must be generated at execution time. |
| --- | --- |
| 2. | As the procedure executes for the first time, the first step is to create table t1. The next step is a select from table t1 -- which there is no plan for. Therefore, the procedure is recompiled at this point to develop a plan for the SELECT statement. A plan is generated for the current select from t1, as well as the select from t1 after the index creation. No plan can be generated for the select from t2 because t2 still does not exist yet. |
| 3. | The next step is to create an index on t1. Following that, another select is performed on t1, which now has a plan from the first recompile. However, because the schema of t1 has been changed since that plan was generated, the procedure must be recompiled again to generate a new plan for the select from t1. And because t2 still does not exist, no plan can be generated for the select from t. |
| 4. | Next, the table t2 is created and the select from t2 is executed. Because no plan exists for statement, the procedure is recompiled a final time. |

These recompilations occur on every execution of the stored procedure. To reduce the recompilations, modify the procedure to do all DDL operations first, followed by DML operations, as shown in the following:

drop procedure NoInterleave go create procedure NoInterleave as -- All DDL first create table t1 (a int) create index idx\_t1 on t1(a) create table t2 (a int) -- Then DML select \* from t1 select \* from t1 select \* from t2 go exec NoInterleave exec NoInterleave

The first execution of the **NoInterleave** procedure will show the following events in Profiler:

| **Event Class** | **Text** |
| --- | --- |
| SP:Starting | NoInterleave |
| SP:StmtStarting | create table t1 (a int) |
| SP:StmtStarting | create index idx\_t1 on t1(a) |
| SP:StmtStarting | create table t2 (a int) |
| SP:StmtStarting | select \* from t1 |
| SP:Recompile | NoInterleave |
| SP:StmtStarting | select \* from t1 |
| SP:StmtStarting | select \* from t1 |
| SP:StmtStarting | select \* from t2 |
| SP:Completed | NoInterleave |

In this case all of the DDL statements are done up front. The optimizer compiles this procedure as follows:

| 1. | During initial compilation of the procedure, the tables t1 and t2 do not exist. Therefore, no plan for the queries referencing these tables can be created. They must be generated at execution time. |
| --- | --- |
| 2. | The first steps the procedure performs are the DDL operations, creating tables t1 and t2, as well as the index on t1. |
| 3. | The next step is the first select from t1. Because there is no plan available for this SELECT statement, the procedure is recompiled. Because all of the objects exist, plans are generated for all of the SELECT statements in the procedure at this time. |
| 4. | The rest of the procedure executes using the plans generated. Because there are no changes to the referenced objects, there is no need to recompile the procedure further. |

**NOTE:** The second and subsequent executions make use of the existing query plan and cache, and do not result in any recompilations at all. Procedures that create, alter, or drop tables should be modified to ensure that all DDL statements are located at the beginning of the procedure.

**Recompilations Due to Certain Temporary Table Operations**

Use of temporary tables in a stored procedure may cause the stored procedure to be recompiled every time the procedure is executed.  
  
To avoid this, change the stored procedure so that it meets the following requirements:

| • | All statements that contain the name of a temporary table refer to a temporary table created in the same stored procedure, and not in a calling or called stored procedure, or in a string executed using the EXECUTE statement or **sp\_executesql** stored procedure. |
| --- | --- |
| • | All statements that contain the name of a temporary table appear syntactically after the temporary table in the stored procedure or trigger. |
| • | There are no DECLARE CURSOR statements whose SELECT statements reference a temporary table. |
| • | All statements that contain the name of any temporary table precede any DROP TABLE statement that references a temporary table.  DROP TABLE statements are not needed for temporary tables created in a stored procedure. The tables are automatically dropped when the procedure has completed. |
| • | No statements creating a temporary table (such as CREATE TABLE or SELECT... INTO) appear in a control-of-flow statement such as IF... ELSE or WHILE. |

**Avoiding Recompilation by Using the KEEP PLAN Option**

Temporary table usage within stored procedures introduces certain complexities for the query optimizer. The row count and statistical information of the tables can vary dramatically throughout the lifetime of the stored procedure execution. To ensure that the optimizer uses the best plan in all cases concerning temporary tables, a special algorithm was developed to be more aggressive with recompilations. The algorithm states that if a temporary table created with a stored procedure has changed more than six times, the procedure will be recompiled when the next statement references the temporary table.   
  
Consider the following example:

drop procedure useKeepPlan go create procedure useKeepPlan as create table #t (a int, b char(3)) select \* from #t -- Make greater than 6 changes to #t insert #t values (1, 'abc') insert #t values (2, 'abc') insert #t values (3, 'abc') insert #t values (4, 'abc') insert #t values (5, 'abc') insert #t values (6, 'abc') insert #t values (7, 'abc') -- Now reference #t select count(\*) from #t --option (KEEP PLAN) go exec useKeepPlan exec useKeepPlan

In this case, you will see the following events in Profiler for the second execution:

| **Event Class** | **Text** |
| --- | --- |
| SP:Starting | useKeepPlan |
| SP:StmtStarting | create table #t (a int) |
| SP:StmtStarting | - The seven insert statements - |
| SP:StmtStarting | select count(\*) from #t1 |
| SP:Recompile | useKeepPlan |
| SP:StmtStarting | select count(\*) from #t1 |
| SP:Completed | useKeepPlan |

The procedure is recompiled on the select that occurs after the seven changes to the temporary table #t.   
  
This aggressive recompilation is helpful in cases where the changes to the data distribution of the temporary table can dramatically affect the optimal query plan for the statement referencing it. However, in the case of large procedures that modify temporary tables frequently, but not in a significant manner, the recompilations may result in slower overall performance. The KEEP PLAN option of the SELECT statement was introduced for this situation.   
  
KEEP PLAN eliminates stored procedure recompilations caused by more than six changes to temporary tables within the procedure and reverts back to the standard algorithm for recompilation due to row modifications discussed above in the "Recompilations Due to Row Modifications" section of this article. KEEP PLAN does not prevent recompilations altogether, it simply prevents those caused by more than six changes to temporary tables referenced in the procedure. In the example above, if you remove the comment from the "option (KEEP PLAN)" line in the stored procedure, the **SP:Recompile** event will not be generated.   
  
If you remove the comment from the "option (KEEP PLAN)" line in the above code and execute it, you will see the following events in Profiler:

| **Event Class** | **Text** |
| --- | --- |
| SP:Starting | useKeepPlan |
| SP:StmtStarting | create table #t (a int) |
| SP:StmtStarting | - The seven insert statements - |
| SP:StmtStarting | select count(\*) from #t1 option (KEEP PLAN) |
| SP:Completed | useKeepPlan |

Note there is no **SP:Recompile** event.

| Back to the top | Back to the top |
| --- | --- |

**Recompilations Due to Certain SET Statements Executed in Stored Procedure**

The following five SET options are set to ON by default:

| • | ANSI\_DEFAULTS |
| --- | --- |
| • | ANSI\_NULLS |
| • | ANSI\_PADDING |
| • | ANSI\_WARNINGS |
| • | CONCAT\_NULL\_YIELDS\_NULL |

If you execute the SET statement to set any of these options to OFF, the stored procedure will be recompiled every time it runs. The reason for this is that changing these options may affect the query result that triggered the recompilation.  
  
Consider the following sample code:

Use pubs drop procedure test\_recompile go create procedure test\_recompile as Set ANSI\_DEFAULTS OFF Select au\_lname, au\_fname, au\_id from authors where au\_lname like 'L%' --Option (Keep Plan) Go

In this case, you will see the following events in SQL Profiler for every execution of the stored procedure:

+---------------------------------------------------+ | Event Class | Text | +---------------------------------------------------+ | SP:Starting | test\_recompile | +---------------------------------------------------+ | SP:StmtStarting | Set ANSI\_DEFAULTS OFF | +---------------------------------------------------+ | SP:StmtStarting | select au\_lname, au\_fname, au\_id| +---------------------------------------------------+ | SP:Recompile | test\_recompile | +---------------------------------------------------+ | SP:StmtStarting | select au\_lname, au\_fname, au\_id| +---------------------------------------------------+ | SP:Completed | test\_recompile | +---------------------------------------------------+

Replacing the SET option with any one of the five options listed above will show the same results. Also, using the option of keep plan here won't help to avoid the recompilation because the cause of the recompilation is from the SET statement.  
  
The recommended way to avoid the recompilation is not to use any of these five SET statement in a stored procedure. For additional information, see the following article in the Microsoft Knowledge Base:

[294942](http://support.microsoft.com/kb/294942/EN-US/) PRB: SET CONCAT\_NULL\_YIELDS\_NULL May Cause Stored Procedures to Recompile

However, as not recommended, running the SET statement to reset the connection option to the same value as the stored procedure, it can also avoid the recompile, doing it as:

Set ANSI\_DEFAULTS OFF exec test\_recompile

The SQL Profiler trace will show no more SP:Recompile events.   
  
The following table lists some common SET statements and whether or not changing the SET statement in a stored procedure causes a recompile:

| Set Statement | Recompile |
| --- | --- |
| Set quoted\_identifier | No |
| Set arithabort | Yes |
| Set ansi\_null\_dflt\_on | Yes |
| Set ansi\_defaults | Yes |
| Set ansi\_warnings | Yes |
| Set ansi\_padding | Yes |
| Set concat\_null\_yields\_null | Yes |
| Set numeric\_roundabort | No |
| Set nocount | No |
| Set rowcount | No |
| Set xact\_abort | No |
| Set implicit\_transactions | No |
| Set arithignore | No |
| Set lock\_timeout | No |
| Set fmtonly | No |

| Back to the top | Back to the top |
| --- | --- |

**References**

[308737](http://support.microsoft.com/kb/308737/EN-US/) INF: How to Identify the Cause of Recompilation in a SP:Recompile Event

For information about using SQL Server Profiler, see SQL Server Books Online.

# **PRB: SET CONCAT\_NULL\_YIELDS\_NULL May Cause Stored Procedures to Recompile**

View products that this article applies to.

| Article ID | : | 294942 |
| --- | --- | --- |
| Last Review | : | October 16, 2003 |
| Revision | : | 1.0 |

This article was previously published under Q294942

**SYMPTOMS**

If a stored procedure contains the statement SET CONCAT\_NULL\_YIELDS\_NULL, and the statement changes the current setting of CONCAT\_NULL\_YIELDS\_NULL, the procedure will be recompiled every time that it is executed. This may lead to performance problems.

**CAUSE**

Recompilation is required because changing the setting of CONCAT\_NULL\_YIELDS\_NULL changes both the result and the way that commands are interpreted. In SQL Server 7.0 and SQL Server 2000, CONCAT\_NULL\_YIELDS\_NULL is set to ON by default, in accordance with ANSI standards.  
  
Setting CONCAT\_NULL\_YIELDS\_NULL to OFF at the database level does not yield the desired result because ODBC and OLE DB set this property to ON when they connect. (Because this is specified by a bit in the connection packet, SQL Profiler does not show that this property is being set.)

**WORKAROUND**

Do not set CONCAT\_NULL\_YIELDS\_NULL inside of the stored procedure. The recommended solution is to use the ISNULL function when concatenating strings.  
  
For example, a query should be written as follows:

SELECT \* FROM mytable WHERE name LIKE 'Xavier ' + ISNULL(@lastname,'') + '%'

rather than like this:

SELECT \* FROM mytable WHERE name LIKE 'Xavier ' + @lastname + '%'

Although not recommended, an alternate solution would be to issue the SET CONCAT\_NULL\_YIELDS\_NULL statement before executing the stored procedure.

**MORE INFORMATION**

This article addresses only one reason that stored procedures may be recompiled; for additional information, click the article number below to view the article in the Microsoft Knowledge Base:

[243586](http://support.microsoft.com/kb/243586/EN-US/) INF: Troubleshooting Stored Procedure Recompilation