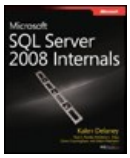


# Microsoft SQL Server 2008 Internals

by Kalen Delaney et al.

# Chapter 9: Plan Caching and Recompilation

***Kalen Delaney***

We've now looked at the query optimization process and the details of query execution in Microsoft SQL Server. Because query optimization can be a complex and time-consuming process, SQL Server frequently and beneficially reuses plans that have already been generated and saved in the plan cache, rather than producing a completely new plan. However, in some cases, a previously created plan may not be ideal for the current query execution, and we might achieve better performance by creating a new plan.

In this chapter, we look at the SQL Server 2008 plan cache and how it is organized. Most of the discussion is relevant to SQL Server 2005 as well, and I will tell you when a behavior or feature is specific to SQL Server 2008. I will tell you about what kinds of plans are saved, and under what conditions SQL Server might decide to reuse them. We look at what might cause an existing plan to be re-created. We also look at the metadata that describes the contents of plan cache. Finally, I describe the ways that you can encourage SQL Server to use an existing plan when it might otherwise create a new one, and how you can force SQL Server to create a new plan when you need to know that the most up-to-date plan is available.

## The Plan Cache

It's important to understand that the plan cache in SQL Server 2008 is not actually a separate area of memory. Releases prior to SQL Server 7 had two effective configuration values to control the size of the plan cache, which was then called the *procedure cache.* One value specified a fixed size for the total usable memory in SQL Server; the other specified a percentage of that memory (after fixed needs were satisfied) to be used exclusively for storing procedure plans. Also, in releases prior to SQL Server 7, query plans for adhoc SQL statements were never stored in cache, only the plans for stored procedures. That is why it was called procedure cache in older versions. In SQL Server 2008, the total size of memory is by default dynamic, and the space used for query plans is also very fluid.

## Plan Cache Metadata

In the first part of this chapter, I explore the different mechanisms by which a plan can be reused, and to observe this plan reuse (or non-reuse), we need to look at only a couple of different metadata objects. There are actually about a dozen different metadata views and functions that give us information about the contents of plan cache, and that doesn't include the metadata that gives us information about memory usage by the plan cache. Later in the chapter, we look at more details available in the plan cache metadata, but for now, we are using just one view and one function. The view is *sys.dm_exec_cached_plans*, which contains one row for each plan in cache, and we look at the columns *usecounts, cacheobjtype,* and *objtype*. The value in *usecounts* allows us to see how many times a plan has been reused. The possible values for *cacheobjtype* and *objtype* are described in the next section. We also use the value in the column *plan_handle* as the parameter when we use the CROSS APPLY operator to join the *sys.dm_exec_cached_plans* view with the table-valued function (TVF) *sys.dm_exec_sql_text.* This is the query we use, which we refer to as the *usecount query:*

```
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text(plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
```

## Clearing Plan Cache

Because SQL Server 2008 has the potential to cache almost every query, the number of plans in cache can become quite large. There is a very efficient mechanism, described later in the chapter, for finding a plan in cache. There is not a direct performance penalty for having lots of cached plans, aside from the memory usage. However, if you have many very similar queries, the lookup time for SQL Server to find the right plan can sometimes be excessive. In addition, from a testing and troubleshooting standpoint, having lots of plans to look at can sometimes make it difficult to find just the plan in which we're currently interested. SQL Server provides a mechanism for clearing out all the plans in cache, and you probably want to do that occasionally on your test servers to keep the cache size manageable and easy to examine. You can use any of the following commands:

- **DBCC FREEPROCCACHE** This command removes all cached plans from memory. SQL Server 2008 added the capability to add parameters to this command, to allow SQL Server to remove a specific plan from cache, all plans with the same *sql_handle* value, or all plans in a specific resource governor resource pool. I discuss the use of this

procedure later in this chapter, when I discuss examining the contents of the plan cache.

- **DBCC FREESYSTEMCACHE** This command clears out all SQL Server memory caches, in addition to the plan caches. I talk a bit more about the different memory caches in the section entitled "Cache Stores," later in this chapter.
- **DBCC FLUSHPROCINDB (<dbid>)** This command allows you to specify a particular database ID, and then clears all plans from that particular database. Note that the *usecount* query that we use in this section does not return database ID information, but the *sys.dm_exec_sql_text* function has that information available, so dbid could be added to the *usecount* query.

**Tip** It is, of course, recommended that you don't use these commands on your production servers because it could affect the performance of your running applications. Usually, you want to keep plans in cache.

### Caching Mechanisms

SQL Server can avoid compilations of previously executed queries by using four mechanisms to make plan caching accessible in a wide set of situations:

- Adhoc query caching
- Autoparameterization
- Prepared queries, using either *sp_executesql* or the prepare and execute method invoked through your API
- Stored procedures or other compiled objects (triggers, TVFs, etc.)

To determine which mechanism is being used for each plan in cache, we need to look at the values in the *cacheobjtype* and *objtype* columns in the *sys.dm_exec_cached_plans* view. The *cacheobjtype* column can have one of six possible values:

- *Compiled Plan*
- *Compiled Plan Stub*
- *Parse Tree*
- *Extended Proc*
- *CLR Compiled Func*
- *CLR Compiled Proc*

In this section, the only values we are looking at are *Compiled Plan* and *Compiled Plan Stub*. Notice that I filter the *usecount* query to limit the results to rows with one of these values.

There are 11 different possible values for the *objtype* column:

- *Proc* (Stored procedure)
- *Prepared* (Prepared statement)
- *Adhoc* (Adhoc query)
- *ReplProc* (Replication-filter-procedure)
- *Trigger*
- *View*
- *Default (*Default constraint or default object)
- *UsrTab* (User table)
- *SysTab* (System table)

- *Check* (CHECK constraint)
- *Rule* (Rule object)

We are mainly examining the first three values, but many caching details that apply to stored procedures also apply to replication filter procedures and triggers.

### Adhoc Query Caching

If the caching metadata indicates a *cacheobjtype* value of *Compiled Plan* and an *objtype* value of *Adhoc*, the plan is considered to be an adhoc plan. Prior to SQL Server 2005, adhoc plans were cached occasionally, but it was not something on which you could depend. However, even when SQL Server caches your adhoc queries, you might not be able to depend on their reuse. When SQL Server caches the plan from an adhoc query, the cached plan is reused only if a subsequent batch matches exactly. This feature requires no extra work to use, but it is limited to *exact* textual matches. For example, if the following three queries are executed in the *Northwind2* database (which can be found on the companion Web site, *http://www.SQLServerInternals.com/companion*), the first and third queries use the same plan, but the second one needs to generate a new plan:

```
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
SELECT * FROM Orders WHERE CustomerID = 'CHOPS';
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
```

You can verify this by first clearing out the plan cache and then running the three queries in separate batches. Then run the *usecount* query referred to previously:

```
USE Northwind2;
DBCC FREEPROCCACHE;
GO
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
GO
SELECT * FROM Orders WHERE CustomerID = 'CHOPS';
GO
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
```

You should get two rows back because the NOT LIKE condition filters out the row for the *usecount* query itself. The two rows are shown here and indicate that one plan was used only once, and the other was used twice:

| *usecounts* | *cacheobjtype* | *objtype* | *text* |
|---|---|---|---|
| 1 | Compiled Plan | Adhoc | `SELECT * FROM Orders WHERE CustomerID = 'CHOPS'` |
| 2 | Compiled Plan | Adhoc | `SELECT * FROM Orders WHERE CustomerID = 'HANAR'` |

**Note** The results shown in this section are obtained with the Optimize for Ad Hoc Workloads configuration option set to 0, which is the default value when you install SQL Server. I discuss this new SQL Server 2008 option later in this chapter.

The results show that with a change of the *CustomerID* value, the same plan cannot be reused. However, to take advantage of reuse of adhoc query plans, you need to make sure that not only are the same *CustomerID* values used in the queries, but also that the queries are identical, character for character. If one query has a new line or an extra space that another one doesn't have, they are not treated the same. If one query contains a comment that the other doesn't have, they are not identical. In addition, if one query uses a different case for either identifiers or keywords, even in a database with a case-insensitive collation, the queries are not the same. If you run the code here, you see that none of the queries can reuse the same plan:

```
USE Northwind2;
DBCC FREEPROCCACHE;
GO
```

```
SELECT * FROM orders WHERE customerID = 'HANAR';
GO
-- Try it again
SELECT * FROM orders WHERE customerID = 'HANAR';
GO
SELECT * FROM orders
WHERE customerID = 'HANAR';
GO
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
GO
select * from orders where customerid = 'HANAR';
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
       CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
      AND [text] NOT LIKE '%dm_exec_cached_plans%';
```

Your results should show five rows in *sys.dm_exec_cached_plans*, each with a *usecounts* value of 1.

**Note** The *SELECT* statements are all in their own batch, separated by GO. If there were no GOs, there would just be one batch, and each batch has its own plan containing the execution plan for each individual query within the batch. For reuse of adhoc query plans, the entire batch must be identical.

There are a few special kinds of statements that are always considered to be adhoc. These constructs include the following:

- A statement used with EXEC, as in `EXEC('SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 6')`
- A statement submitted using *sp_executesql*, if no parameters are supplied

Queries that you submit via your application with *sp_prepare* and *sp_prepexec* are not considered to be adhoc.

### Optimizing for Adhoc Workloads

If most of your queries are adhoc and never be reused, it might seem like a waste of memory to cache their execution plans. Later in this chapter, I talk about how the maximum size of plan cache is determined. It is true that having tens of thousands of cached plans for adhoc queries that have little likelihood of reuse is probably not the best use of SQL Server's memory. For this reason, SQL Server 2008 added a configuration option that you can enable in those cases where you expect most of your queries to be adhoc. Once this option is enabled, SQL Server caches only a stub of your query plan the first time any adhoc query is compiled, and only after a second compilation is the stub replaced with the full plan.

#### Controlling the Optimize for Ad Hoc Workloads setting

Enabling the Optimize for Ad Hoc Workloads option is very straightforward, as shown in the following code:

```
EXEC sp_configure 'optimize for ad hoc workloads', 1;
RECONFIGURE;
```

You can also enable this option using SQL Server Management Studio, in the Advanced page of the Server Properties dialog box, as shown in Figure 9-1.

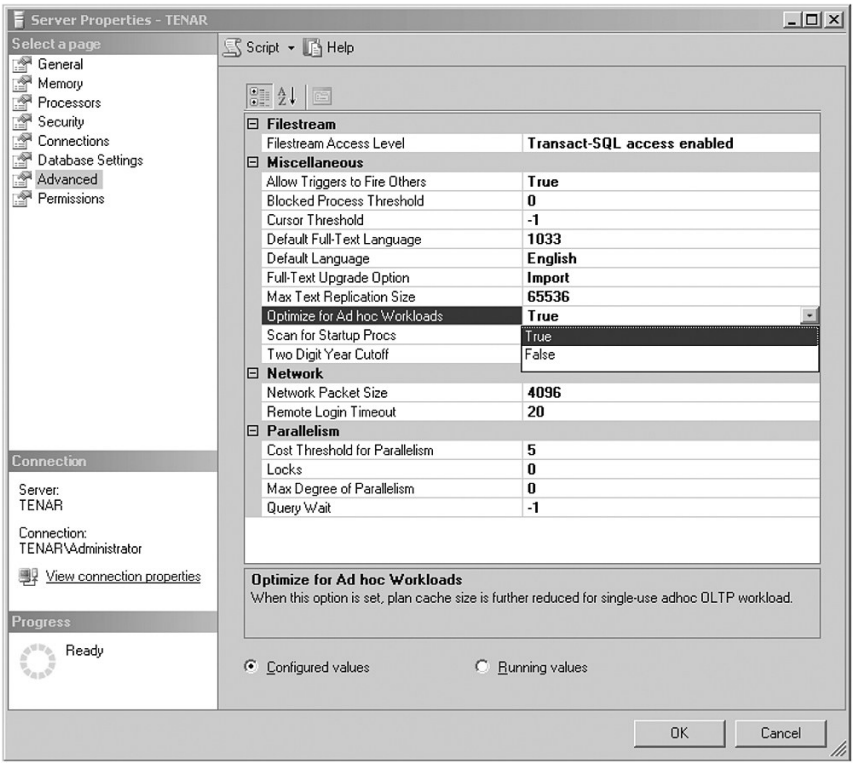


**Figure 9-1:** Using the Server Properties dialog box in Management Studio to enable the Optimize for Ad Hoc Workloads option

#### The Compiled Plan Stub

The stub that SQL Server caches when Optimize for Ad Hoc Workloads is enabled is only about 300 bytes in size and does not contain any part of a query execution plan. It is basically only a placeholder to keep track of whether a particular query has been seen compiled previously. The stub contains the full cache key and a pointer to the actual query text, which is stored in the SQL Manager cache. I discuss cache keys and the SQL Manager in the section entitled "Plan Cache Internals," later in this chapter. The *usecounts* value in the cache metadata is always 1 for compiled plan stubs because they are never reused.

When a query or batch that generated a compiled plan stub is recompiled, the stub is replaced with the full compiled plan. Initially, the *usecounts* value is set to 1 because there is no guarantee that the previous query had exactly the same execution plan. All that is known is that the query itself is the same. I will execute some of the same queries I used in the previous section after enabling the Optimize for Ad Hoc Workloads option, and we see what the *usecounts* query shows us. I need to modify my *usecounts* query slightly, and instead of looking for rows that have a *cacheobjtype* value of *Compiled Plan*, I look for *cacheobjtype* values that start with *Compiled Plan*:

```
EXEC sp_configure 'optimize for ad hoc workloads', 1;
RECONFIGURE;
GO
USE Northwind2;
DBCC FREEPROCCACHE;
GO
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
        CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype LIKE 'Compiled Plan%'
        AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
SELECT * FROM Orders WHERE CustomerID = 'HANAR';
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
        CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype LIKE 'Compiled Plan%'
```

```
        AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
```

The first execution of the *usecounts* query returns the following:

| usecounts | cacheobjtype | objtype | text |
|---|---|---|---|
| 1 | Compiled Plan Stub | Adhoc | SELECT * FROM Orders<br>WHERE CustomerID = 'HANAR' |

The second execution shows the replacement of the stub with the compiled plan:

| usecounts | cacheobjtype | objtype | text |
|---|---|---|---|
| 1 | Compiled Plan | Adhoc | SELECT * FROM Orders<br>WHERE CustomerID = 'HANAR' |

The stub is generated when the plan is compiled, not when it is executed, so you would see this same behavior if you examined the query plan only twice with one of the SHOWPLAN options, without ever executing the query.

If the Optimize for Ad Hoc Workloads option is set to 1 and then is set back to 0 after Compiled Plan Stubs are placed in the plan cache, the stubs are not immediately removed from cache. As when the option was set to 1, any resubmission of the same adhoc T-SQL batch replaces the stub with the compiled plan, and then no further stubs are created.

Even with this new SQL Server 2008 mechanism for improving the caching behavior when your workloads use primarily adhoc queries, this does not mean that adhoc workloads are a good idea. There are times that you have no control over the kind of queries being submitted to your SQL Server, and in that case, you might find this option beneficial. However, if you and your developers have control over the how your queries are submitted, I recommend that you consider another options, such as Prepared Queries or stored procedures, which is discussed later in this chapter.

If you are running my sample queries as you are reading, you might want to turn off the Optimize for Ad Hoc Workloads option at this point:

```
EXEC sp_configure 'optimize for ad hoc workloads', 0;
RECONFIGURE;
GO
```

### Simple Parameterization

For certain queries, SQL Server can decide to treat one or more of the constants as parameters. When this happens, subsequent queries that follow the same basic template can use the same plan. For example, these two queries run in the *Northwind*2 database can use the same plan:

```
SELECT FirstName, LastName, Title FROM Employees
WHERE EmployeeID = 6;
SELECT FirstName, LastName, Title FROM Employees
WHERE EmployeeID = 2;
```

Internally, SQL Server parameterizes these queries as follows:

```
SELECT FirstName, LastName, Title FROM Employees
WHERE EmployeeID = @1
```

You can observe this behavior by running the following code and observing the output of the *usecount* query:

```
USE Northwind2
GO
DBCC FREEPROCCACHE;
GO
SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 6;
GO
SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 2;
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
```

```
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
```

You should get three rows returned, similar to the following:

| *usecounts* | *cacheobjtype* | *objtype* | *text* |
|---|---|---|---|
| 1 | Compiled Plan | Adhoc | `SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 2;` |
| 1 | Compiled Plan | Adhoc | `SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 6;` |
| 2 | Compiled Plan | Prepared | `(@1 tinyint)SELECT [FirstName], [LastName], [Title] FROM [Employees] WHERE [EmployeeID] = @1` |

You should notice that the two individual queries with their distinct constants get cached as adhoc queries. However, these are only considered *shell* queries and are cached only to make it easier to find the parameterized version of the query if the exact same query with the same constant is reused later. These shell queries do not contain the full execution plan but only a pointer to the full plan in the corresponding prepared plan.

> **Tip** Do not confuse a shell query with a plan stub. A shell query contains the complete text of the query, and uses at least 16 KB of memory. Shell queries are created only for those plans that SQL Server thinks are parameterizable. A plan stub, as mentioned previously, only uses about 200 bytes of memory, and is created only for unparameterizable, adhoc queries, and only when the Optimize for Ad Hoc Workloads option is set to 1.

In the output shown previously, the third row returned from *sys.dm_exec_cached_plans* has an *objtype* value of *Prepared.* (The order of the returned rows is not guaranteed. You should have two rows with a *cacheobjtype* value of *Adhoc* and one row with a *cacheobjtype* value of *Prepared*.) The query plan is associated with the prepared plan, and you can observe that the plan was used twice. In addition, the text for that *Prepared* row shows a parameter in place of a constant.

By default, SQL Server is very conservative about deciding when to parameterize automatically. SQL Server automatically parameterizes queries only if the query template is considered to be *safe*. A template is safe if the plan selected does not change even if the actual parameter values change. This ensures that the parameterization won't degrade a query's performance. The *employees* table used in the previous queries has a unique index, so any query that has an equality comparison on *employeeID* is guaranteed to never find more than one row. A plan using a seek on that unique index can be useful no matter what actual value is used.

However, consider a query that has either an inequality comparison or an equality comparison on a nonunique column. In those situations, some actual values may return many rows, and others return no rows, or only one. A nonclustered index seek might be a good choice when only a few rows are returned, but it might be a terrible choice when many rows are returned. So a query for which there is more than one possible best plan, depending on the value used in the query, is not considered safe, and it is not parameterized. By default, the only way for SQL Server to reuse a plan for such a query is to use the adhoc plan caching described in the previous section (which does not happen if the constant values in the query are different).

In addition to requiring that there only be one possible plan for a query template, there are many query constructs that normally disallow simple parameterization. Such constructs include any statements with the following elements:

- *JOIN*
- *BULK INSERT*
- *IN* lists
- *UNION*
- *INTO*

- *FOR BROWSE*
- *OPTION* <query hints>
- *DISTINCT*
- *TOP*
- *WAITFOR* statements
- *GROUP BY, HAVING, COMPUTE*
- Full-text predicates
- Subqueries
- FROM clause of a *SELECT* statement has a table-valued method or full-text table or *OPENROWSET* or *OPENXML* or *OPENQUERY* or *OPENDATASOURCE*
- Comparison predicate of the form EXPR <> a non-null constant

Simple parameterization is also disallowed for data modification statements that use the following constructs:

- *DELETE/UPDATE* with a FROM clause
- *UPDATE* with a SET clause that has variables

**Forced Parameterization**

If your application uses many similar queries that you know benefit from the same plan but are not autoparameterized, either because SQL Server doesn't consider the plans safe or because they use one of the disallowed constructs, SQL Server 2008 provides an alternative. A database option called PARAMETERIZATION FORCED can be enabled with the following command:

```
ALTER DATABASE <database_name> SET PARAMETERIZATION FORCED;
```

Once this option is enabled, SQL Server treats constants as parameters, with only a very few exceptions. These exceptions, as listed in *SQL Server Books Online,* include the following:

- *INSERT … EXECUTE* statements.
- Statements inside the bodies of stored procedures, triggers, or user-defined functions. SQL Server already reuses query plans for these routines.
- Prepared statements that have already been parameterized on the client-side application.
- Statements that contain XQuery method calls, in which the method appears in a context in which its arguments would typically be parameterized, such as a WHERE clause. If the method appears in a context in which its arguments would not be parameterized, the rest of the statement is parameterized.
- Statements inside a T-SQL cursor. (*SELECT* statements inside API cursors are parameterized.)
- Deprecated query constructs.
- Any statement that is run in the context of ANSI_PADDING or ANSI_NULLS set to OFF.
- Statements that contain more than 2,097 literals.
- Statements that reference variables, such as `WHERE T.col2 >= @p.`
- Statements that contain the RECOMPILE query hint.
- Statements that contain a COMPUTE clause.
- Statements that contain a WHERE CURRENT OF clause.

You need to be careful when setting this option on for the entire database because assuming that all constants should be treated as parameters during optimization and then reusing existing plans frequently gives very poor performance. An alternative that allows only selected queries to be autoparameterized is to use plan guides, which are discussed at the end of this chapter. In addition, plan guides can also be used to override forced parameterization for selected queries, if the database has been set to PARAMETERIZATION FORCED.

**Drawbacks of Simple Parameterization**

A feature of autoparameterization that you might have noticed in the output from the *usecount* query shown previously is that SQL Server makes its own decision as to the data type of the parameter, which might not be the data type you think should be used. In the earlier example, looking at the *employees* table, SQL Server chose to assume a parameter of type *tinyint*. If we rerun the batch and use a value that doesn't fit into the *tinyint* range (that is, a value less than 0 or larger than 255), SQL Server cannot use the same autoparameterized query. The batch below autoparameterizes both *SELECT* statements, but it is not able to use the same plan for both queries. The output from the *usecount* query should show two adhoc shell queries, and two prepared queries. One prepared query has a parameter of type *tinyint,* and the other is *smallint*. As strange as it may seem, even if you switch the order of the queries and use the bigger value first, you get two prepared queries with two different parameter data types:

```
USE Northwind2;
GO
DBCC FREEPROCCACHE;
GO
SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 6;
GO
SELECT FirstName, LastName, Title FROM Employees WHERE EmployeeID = 622;
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
```

The only way to force SQL Server to use the same data type for both queries is to enable PARAMETERIZATION FORCED for the database.

As mentioned, simple parameterization is not always appropriate, which is why SQL Server is so conservative in choosing to use it. Consider the following example. The *BigOrders* table in the *Northwind2* database has 4,150 rows and 105 pages, so we might expect that a table scan reading 105 pages would be the worst possible performance for any query accessing the *BigOrders* table. There is a nonclustered nonunique index on the *CustomerID* column. If we enable forced parameterization for the *Northwind2* database, the plan used for the first *SELECT* is also used for the second *SELECT*, even though the constants are different. The first query returns 5 rows and the second returns 155. Normally, a nonclustered index seek would be chosen for the first *SELECT* and a clustered index scan for the second because the number of qualifying rows exceeds the number of pages in the table. However, with PARAMETERIZATION FORCED, that's not what we get, as you can see when you run the following code:

```
USE Northwind2;
GO
ALTER DATABASE Northwind2 SET PARAMETERIZATION FORCED;
GO
SET STATISTICS IO ON;
GO
DBCC FREEPROCCACHE;
GO
SELECT * FROM BigOrders WHERE CustomerID = 'CENTC'
GO
SELECT * FROM BigOrders WHERE CustomerID = 'SAVEA'
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
ALTER DATABASE Northwind2 SET PARAMETERIZATION SIMPLE;
GO
```

When we run this code, we see that the first *SELECT* required 12 logical reads and the second required 312, almost three times as many reads as would have been required if scanning the table. The output of the *usecount* query, shown here, shows that forced parameterization was applied and the parameterized prepared plan was used twice:

| usecounts | cacheobjtype | objtype | text |
|---|---|---|---|
| 1 | Compiled Plan | Adhoc | `SELECT * FROM BigOrders WHERE CustomerID = 'SAVEA'` |
| 1 | Compiled Plan | Adhoc | `SELECT * FROM BigOrders WHERE CustomerID = 'CENTC'` |
| 2 | Compiled Plan | Prepared | `(@0 varchar(8000))select * from BigOrders where CustomerID = @0` |

In this example, forcing SQL Server to treat the constant as a parameter is not a good thing, and the batch sets the database back to PARAMETERIZATION SIMPLE (the default) as the last step. Note also that while we are using PARAMETERIZATION FORCED, the data type chosen for the parameterized query is the largest possible regular character data type.

So what can you do if you have many queries that should not be parameterized and many others that should be? As we've seen, the SQL Server query processor is much more conservative about deciding whether a template is safe than an application can be. SQL Server guesses which values are really parameters, whereas your application developers should actually know. Rather than rely on SQL Server to parameterize your queries automatically, you can use one of the prepared query mechanisms to mark values as parameters when they are known.

The SQL Server Performance Monitor includes an object called *SQLServer:SQL Statistics* that has several counters dealing with automatic parameterization. You can monitor these counters to determine whether there are many unsafe or failed automatic parameterization attempts. If these numbers are high, you can inspect your applications for situations in which the application developers can take responsibility for explicitly marking the parameters.

## Prepared Queries

As we saw previously, a query that is parameterized by SQL Server shows an *objtype* of *Prepared* in the cached plan metadata. There are two other constructs that have prepared plans. Both of these constructs allow the programmer to take control over which values are parameters and which aren't. In addition, unlike with simple parameterization, the programmer also determines the data type that be used for the parameters. One construct is the SQL Server stored procedure *sp_executesql*, which is called from within a T-SQL batch, and the other is to use the prepare and execute method from the client application.

### The sp_executesql Procedure

The stored procedure *sp_executesql* is halfway between adhoc caching and stored procedures. Using *sp_executesql* requires that you identify the parameters and their data types, but it doesn't require all the persistent object management needed for stored procedures and other programmed objects.

Here's the general syntax for the procedure:

```
sp_executesql @batch_text, @batch_parameter_definitions,
   param1,...paramN
```

Repeated calls with the same values for *@batch_text* and *@batch_parameter_definitions* use the same cached plan, with the new parameter values specified. The plan is reused so long as the plan has not been removed from cache and is still valid. The section entitled "Causes of Recompilation," later in this chapter, discusses those situations in which SQL Server determines that a plan is no longer valid. The same cached plan can be used for all the following queries:

```
EXEC sp_executesql N'SELECT FirstName, LastName, Title
    FROM Employees
    WHERE EmployeeID = @p', N'@p tinyint', 6;
EXEC sp_executesql N'SELECT FirstName, LastName, Title
    FROM Employees
    WHERE EmployeeID = @p', N'@p tinyint', 2;
EXEC sp_executesql N'SELECT FirstName, LastName, Title
    FROM Employees
```

```
    WHERE EmployeeID = @p', N'@p tinyint', 6;
```

Just like forcing autoparameterization, using *sp_executesql* to force reuse of a plan is not always appropriate. If we take the same example used earlier when we set the database to PARAMETERIZATION FORCED, we can see that using *sp_executesql* is just as inappropriate.

```
USE Northwind2;
GO
SET STATISTICS IO ON;
GO
DBCC FREEPROCCACHE;
GO
EXEC sp_executesql N'SELECT * FROM BigOrders
    WHERE CustomerID = @p', N'@p nvarchar(10)', 'CENTC';
GO
EXEC sp_executesql N'SELECT * FROM BigOrders
    WHERE CustomerID = @p', N'@p nvarchar(10)', 'SAVEA';
GO
SELECT usecounts, cacheobjtype, objtype, [text]
FROM sys.dm_exec_cached_plans P
    CROSS APPLY sys.dm_exec_sql_text (plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
    AND [text] NOT LIKE '%dm_exec_cached_plans%';
GO
SET STATISTICS IO OFF;
GO
```

Again, we can see that the first *SELECT* required 12 logical reads and the second required 312. The output of the *usecount* query, seen here, shows the parameterized query being used twice. Note that with *sp_executesql*, we do not have any entries for the adhoc (unparameterized) shell queries.

| *usecounts* | *cacheobjtype* | *objtype* | *text* |
|---|---|---|---|
| 2 | Compiled Plan | Prepared | (@p nvarchar(10))SELECT * FROM BigOrders WHERE CustomerID = @p |

#### The Prepare and Execute Method

This last mechanism is like *sp_executesql* in that parameters to the batch are identified by the application, but there are some key differences. The prepare and execute method does not require the full text of the batch to be sent at each execution. Rather, the full text is sent once at prepare time; a handle that can be used to invoke the batch at execute time is returned. ODBC and OLE DB expose this functionality via *SQLPrepare/SQLExecute* and *ICommandPrepare*. You can also use this mechanism via ODBC and OLE DB when cursors are involved. When you use these functions, SQL Server is informed that this batch is meant to be used repeatedly.

#### Caching Prepared Queries

If your queries have been parameterized at the client using the prepare and execute method, the metadata shows you prepared queries, just as for queries that are parameterized at the server, either automatically or by using *sp_executesql*. However, queries that are not parameterized ( either under simple or forced parameterization) do not have any corresponding adhoc shell queries in cache, containing the unparameterized actual values; they have only the prepared plans. There is no guaranteed way to detect whether a prepared plan was prepared by SQL Server using simple or forced parameterization or by the developer through client-side parameterization. If you see a corresponding shell query, you can know that the query was parameterized by SQL Server, but the opposite is not always true. Because the shell queries have zero cost, they are among the first candidates to be removed when SQL Server is under memory pressure. So a lack of a shell query might just mean that adhoc plan was already removed from cache, not that there never was a shell query.

### Compiled Objects

When looking at the metadata in *sys.dm_exec_cached_plans*, we've seen compiled plans with *objtype* values of *Adhoc* and *Prepared*. The third *objtype* value that we will be discussing is *Proc,* and you will see this type used when executing stored procedures, user-defined scalar functions, and multistatement TVFs. With these objects, you have full control over what values are parameters and what their data types are when executing these objects.

#### Stored Procedures

Stored procedures and user-defined scalar functions are treated almost identically. The metadata indicates that a compiled plan with an *objtype* value of *Proc* is cached and can be reused repeatedly. By default, the cached plan is reused for all successive executions, and as we've seen with the *sp_executesql*, this is not always desirable. However, unlike the plans cached and reused with *sp_executesql*, you have an option with stored procedures and user-defined scalar functions to force recompilation when the object is executed. In addition, for stored procedures, you can create the object so that a new plan is created every single time it is executed.

To force recompilation for a single execution, you can use the EXECUTE … WITH RECOMPILE option. Here is an example in the *Northwind2* database of forcing recompilation for a stored procedure:

```
USE Northwind2;
GO
CREATE PROCEDURE P_Customers
  @cust nvarchar(10)
AS
  SELECT RowNum, CustomerID, OrderDate, ShipCountry
  FROM BigOrders
WHERE CustomerID = @cust;
GO
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
GO
EXEC P_Customers 'CENTC';
GO
EXEC P_Customers 'SAVEA';
GO
EXEC P_Customers 'SAVEA' WITH RECOMPILE;
```

If you look at the output from STATISTICS IO, you see that the second execution used a suboptimal plan that required more pages to be read than would be needed by a table scan. This is the situation that you may have seen referred to as *parameter sniffing.* SQL Server is basing the plan for the procedure on the first actual parameter, in this case, *CENTC,* and then subsequent executions assume the same or a similar parameter is used. The third execution uses the WITH RECOMPILE option to force SQL Server to come up with a new plan, and you should see that the number of logical page reads is equal to the number of pages in the table.

If you look at the results from running the *usecounts* query, shown here, you should see that the cached plan for the *P_Customers* procedure has a *usecounts* value of 2, instead of 3.

| *usecounts* | *cacheobjtype* | *objtype* | *text* |
|---|---|---|---|
| 2 | Compiled Plan | Proc | CREATE PROCEDURE P_Customers<br>@cust nvarchar(10)<br>AS<br>SELECT RowNum, CustomerID, OrderDate,<br>ShipCountry<br>FROM BigOrders WHERE CustomerID = @cust |

The plan developed for a procedure executed with the WITH RECOMPILE option is considered valid only for the current execution; it is never kept in cache for reuse.

**Functions**

User-defined scalar functions can behave exactly the same way as procedures. If you execute them using the *EXECUTE* statement instead of as part of an expression, you can also force recompilation. Here is an example of a function that masks part of a Social Security number. We create it in the *pubs* sample database because the *authors* table contains a Social Security number in the *au_id* column:

```
USE pubs;
GO
CREATE FUNCTION dbo.fnMaskSSN (@ssn char(11))
RETURNS char(11)
AS
BEGIN
```

```
  SELECT @SSN = 'xxx-xx-' + right (@ssn,4);
  RETURN @SSN;
END;
GO
DBCC FREEPROCCACHE;
GO

DECLARE @mask char(11);
EXEC @mask = dbo.fnMaskSSN '123-45-6789';
SELECT @mask;
GO
DECLARE @mask char(11);
EXEC @mask = dbo.fnMaskSSN '123-66-1111';
SELECT @mask;
GO
DECLARE @mask char(11);
EXEC @mask = dbo.fnMaskSSN '123-66-1111' WITH RECOMPILE;
SELECT @mask;
GO
```

If you run the *usecounts* query, you should notice the cached plan for the function has an *objtype* of *Proc* and has a *usecounts* value of 2. If a scalar function is used within an expression, as in the example here, there is no way to request recompilation:

```
SELECT dbo.fnMaskSSN(au_id), au_lname, au_fname, au_id  FROM authors;
```

TVFs may or may not be treated like procedures depending on how you define them. You can define a TVF as an inline function or as a multistatement function. Neither method allows you to force recompilation when the function is called. Here are two functions that do the same thing:

```
USE Northwind2;
GO
CREATE FUNCTION Fnc_Inline_Customers (@cust nvarchar(10))
RETURNS TABLE
AS
 RETURN
 (SELECT RowNum, CustomerID, OrderDate, ShipCountry
 FROM BigOrders
 WHERE CustomerID = @cust);
GO

CREATE FUNCTION Fnc_Multi_Customers (@cust nvarchar(10))
RETURNS @T TABLE (RowNum int, CustomerID nchar(10), OrderDate datetime,
 ShipCountry nvarchar(30))
AS
BEGIN
 INSERT INTO @T
  SELECT RowNum, CustomerID, OrderDate, ShipCountry
  FROM BigOrders
  WHERE CustomerID = @cust
  RETURN
END;
GO
```

Here are the calls to the functions:

```
DBCC FREEPROCCACHE
GO
SELECT * FROM Fnc_Multi_Customers('CENTC');
GO
SELECT * FROM Fnc_Inline_Customers('CENTC');
GO
SELECT * FROM Fnc_Multi_Customers('SAVEA');
GO
SELECT * FROM Fnc_Inline_Customers('SAVEA');
GO
```

If you run the *usecounts* query, you see that only the multistatement function has its plan reused. The inline function is actually treated like a view, and the only way the plan can be reused would be if the exact same query were reexecuted; that is, if the same *SELECT* statement called the function with the exact same parameter.

## Causes of Recompilation

Up to this point, we've been discussing the situations in which SQL Server automatically reuses a plan, and the situation in which a plan may be reused inappropriately so that you need to force recompilation. However, there are also situations in which an existing plan is not reused because of changes to the underlying objects or the execution environment. The reasons for these unexpected recompilations fall into one of two different categories, which we call *correctness-based recompiles* and *optimality-based recompiles.*

### Correctness-Based Recompiles

SQL Server may choose to recompile a plan if it has a reason to suspect that the existing plan may no longer be correct. This can happen when there are explicit changes to the underlying objects, such as changing a data type or dropping an index. Obviously, any existing plan that referenced the column assuming its former data type or that accessed data using the now nonexistent index would not be correct. Correctness-based recompiles fall into two general categories: schema changes and environmental changes. The following changes mark an object's schema as changed:

- Adding or dropping columns to or from a table or view
- Adding or dropping constraints, defaults, or rules to or from a table
- Adding an index to a table or an indexed view
- Dropping an index defined on a table or an indexed view if the index is used by the plan
- Dropping a statistic defined on a table that causes a correctness-related recompilation of any query plans that use that table
- Adding or dropping a trigger from a table

In addition, running the procedure *sp_recompile* on a table or view changes the modification date for the object, which you can observe in the *modify_date* column in *sys.objects*. This makes SQL Server determine that a schema change has occurred so that recompilation takes place at the next execution of any stored procedure, function, or trigger that accesses the table or view. Running *sp_recompile* on a procedure, trigger, or function clears all the plans for the executable object out of cache to guarantee that the next time it is executed, it will be recompiled.

Other correctness-based recompiles are invoked when the environment changes by changing one of a list of SET options. Changes in certain SET options can cause a query to return different results, so when one of these values changes, SQL Server wants to make sure a plan is used that was created in a similar environment. SQL Server keeps track of which SET options are set when a plan is executed, and you have access to a bitmap of these SET options using the DMF called *sys.dm_exec_plan_attributes*. This function is called by passing in a plan handle value that you can obtain from the *sys.dm_exec_cached_plans* view and returns one row for each of a list of plan attributes. You need to make sure you include *plan_handle* in the list of columns to be retrieved, not just the few columns we used earlier in the *usecounts* query. Here's an example of retrieving all the plan attributes when we supply a *plan_handle* value. Table 9-1 shows the results of running this code:

```
SELECT * FROM sys.dm_exec_plan_attributes
(0x06001200CF0B831CB821AA0500000000000000000000000000)
```

**Table 9-1: Attributes Corresponding to a Particular plan_handle**

| Attribute | Value | is_cache_key |
|---|---|---|
| *set_options* | 4347 | 1 |
| *objectid* | 478350287 | 1 |
| *dbid* | 18 | 1 |
| *dbid_execute* | 0 | 1 |
| *user_id* | -2 | 1 |
| *language_id* | 0 | 1 |
| *date_format* | 1 | 1 |
| *date_first* | 7 | 1 |

| | | |
|---|---|---|
| *Compat_level* | 100 | 1 |
| *status* | 0 | 1 |
| *required_cursor_options* | 0 | 1 |
| *acceptable_cursor_options* | 0 | 1 |
| *merge_action_type* | 0 | 1 |
| *is_replication_specific* | 0 | 1 |
| *optional_spid* | 0 | 1 |
| *optional_clr_trigger_dbid* | 0 | 1 |
| *optional_clr_trigger_objid* | 0 | 1 |
| *inuse_exec_context* | 0 | 0 |
| *free_exec_context* | 1 | 0 |
| *hits_exec_context* | 0 | 0 |
| *misses_exec_context* | 0 | 0 |
| *removed_exec_context* | 0 | 0 |
| *inuse_cursors* | 0 | 0 |
| *free_cursors* | 0 | 0 |
| *hits_cursors* | 0 | 0 |
| *misses_cursors* | 0 | 0 |
| *removed_cursors* | 0 | 0 |
| *sql_handle* | 0x02000000CF0B831CBBE70632EC8A 8F7828AD6E6 | 0 |

Later in the chapter, when we explore cache management and caching internals, you learn about some of these values in which the meaning is not obvious and I also go into more detail about the metadata that keeps track of your plans. To get the attributes to be returned in a row along with each *plan_handle*, you can use the PIVOT operator and list each of the attributes that you want to turn into a column. In this next query, we want to retrieve the *set_options*, the *object_id,* and the *sql_handle* from the list of attributes:

```
SELECT plan_handle, pvt.set_options, pvt.object_id, pvt.sql_handle
FROM (SELECT plan_handle, epa.attribute, epa.value
      FROM sys.dm_exec_cached_plans
          OUTER APPLY sys.dm_exec_plan_attributes(plan_handle) AS epa
      WHERE cacheobjtype = 'Compiled Plan'
     ) AS ecpa
PIVOT (MAX(ecpa.value) FOR ecpa.attribute
   IN ("set_options", "object_id", "sql_handle")) AS pvt;
```

We get a value of 4347 for *set_options* which is equivalent to the bit string 1000011111011. To see which bit refers to which SET options, we could change one option and then see how the bits have changed. For example, if we clear the plan cache and change ANSI_NULLS to OFF, the *set_options* value change to 4315, or binary 1000011011011. The difference is the sixth bit from the right, which has a value of 32, the difference between 4347 and 4315. If we didn't clear the plan cache, we would end up with two plans for the same batch, one for each *set_options* value.

Not all changes to SET options cause a recompile, although many of them do. The following is a list of the SET options that cause a recompile when changed:

- 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

Two of the SET options in this list have a special behavior in relationship to objects, including stored procedures, functions, views, and triggers. The SET option settings for ANSI_NULLS and QUOTED_IDENTIFIER are actually saved along with the object definition and the procedure or function always executes with the SET values as they were when the object was first created. You can determine what values these two SET options had for your objects by selecting from the *OBJECTPROPERTY* function, as shown:

```
SELECT OBJECTPROPERTY(object_id('<object name>'), 'ExecIsQuotedIdentOn');
SELECT OBJECTPROPERTY(object_id('<object name>'), 'ExecIsAnsiNullsOn');
```

A returned value of 0 means the SET option is OFF, a value of 1 means the option is ON, and a value of *NULL* means that you typed something incorrectly or that you don't have appropriate permissions. However, even though changing the value of either of these options does not cause any difference in execution of the objects, SQL Server may still recompile the statement that accesses the object. The only objects for which recompilation is avoided is for cached plans with an *objtype* value of *Proc,* namely stored procedures and multistatement TVFs. For these compiled objects, the *usecounts* query shows you the same plan being reused and does not show additional plans with different *set_options* values. Inline TVFs and views create new plans if these options are changed, and the *set_ options* value indicates a different bitmap. However, the behavior of the underlying *SELECT* statement does not change.

#### Optimality-Based Recompiles

SQL Server may also choose to recompile a plan if it has reason to suspect that the existing plan is no longer optimal. The primary reasons for suspecting a nonoptimal plan deal with changes to the underlying data. If any of the statistics used to generate the query plan have been updated since the plan was created, or if any of the statistics are considered stale, SQL Server recompiles the query plan.

**Updated Statistics** Statistics can be updated either manually or automatically. Manual updates happen when someone runs *sp_updatestats* or the *UPDATE STATISTICS* command. Automatic updates happen when SQL Server determines that existing statistics are out of date or stale, and these updates happen only when the database has the option AUTO_UPDATE_STATISTICS or AUTO_UPDATE_STATISTICS_ASYNC set to ON. This could happen if another batch had tried to use one of the same tables or indexes used in the current plan, detected the statistics were stale, and initiated an *UPDATE STATISTICS* operation.

**Stale Statistics** SQL Server detects out-of-date statistics when it is first compiling a batch that has no plan in cache. It also detects stale statistics for existing plans. Figure 9-2 shows a flowchart of the steps involved in finding an existing plan and checking to see if recompilation is required. You can see that SQL Server checks for stale statistics after checking to see if there already are updated statistics available. If there are stale statistics, the statistics are updated, and then a recompile begins on the batch. If AUTO_UPDATE_STATISTICS_ASYNC is ON for the database, SQL Server does not wait for the update of statistics to complete; it just recompiles based on the stale statistics.

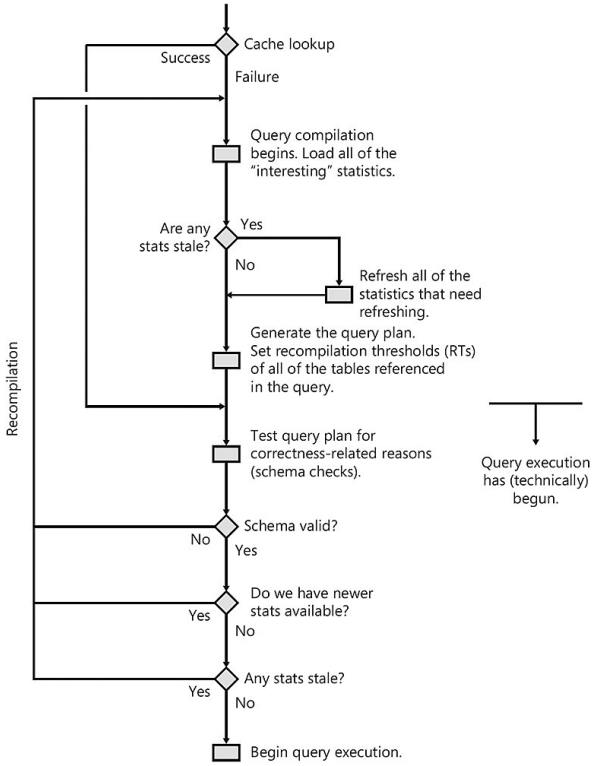


**Figure 9-2:** Checking an existing plan to see if recompilation is necessary

Statistics are considered to be stale if a sufficient number of modifications have occurred on the column supporting the statistics. Each table has a recompilation threshold (RT) that determines how many changes can take place before any statistics on that table are marked as stale. The RT values for all the tables referenced in a batch are stored with the query plans of that batch.

The RT values depend on the type of table, that is, whether it is permanent or temporary, and on the current number of rows in the table at the time a plan is compiled. The exact algorithms for determining the RT values are subject to change with each service pack, so I show you the algorithm for the RTM release of SQL Server 2008. The formulas used in the various service packs will be similar to this, but are not guaranteed to be exactly the same. *N* indicates the cardinality of the table.

- For both permanent and temporary tables, if *N* is less or equal to 500, the RT value is 500. This means that for a relatively small table, you must make at least 500 changes to trigger recompilation. For larger tables, at least 500 changes must be made, plus 20 percent of the number of rows.
- For temporary tables, the algorithm is the same, with one exception. If the table is very small or empty (*N* is less than six prior to any data modification operations), all we need are six changes to trigger a recompile. This means that a procedure that creates a temporary table, which is empty when created, and then inserts six or more rows into that table, will have to be recompiled as soon as the temporary table is accessed.

  You can get around this frequent recompilation of batches that create temporary tables by using the KEEP PLAN query hint. Use of this hint changes the recompilation thresholds for temporary tables and makes them identical to those for permanent tables. So if changes to temporary tables are causing many recompilations, and you suspect that the recompilations are affecting overall system performance, you can use this hint and see if there is a performance improvement. The hint can be specified as shown in this query:

  ```
  SELECT <column list>
  FROM dbo.PermTable A INNER JOIN #TempTable B ON A.col1 = B.col2
  WHERE <filter conditions>
  OPTION (KEEP PLAN)
  ```

- For table variables, there is no RT value. This means that you will not get recompilations caused by changes in the number of rows in a table variable.

**Modification Counters** The RT values discussed here are the number of changes required for SQL Server to recognize that statistics are stale. In versions of SQL Server prior to SQL

Server 2005, the *sysindexes* system table keeps track of the number of changes that had actually occurred in a table in a column called *rowmodctr*. These counters keep track of any changes in any row of the table or index, even if the change was to a column that was not involved in any index or useful statistics. SQL Server 2008 now uses a set of Column Modification Counters or *colmodctr* values, with a separate count being maintained for each column in a table, except for computed nonpersisted columns. These counters are not transactional, which means that if a transaction starts, inserts thousands of rows into a table, and then is rolled back, the changes to the modification counters are *not* rolled back. Unlike the *rowmodctr* values in *sysindexes*, the *colmodctr* values are not visible to the user. They are only available internally to the Query Optimizer.

**Tracking Changes to Tables and Indexed Views Using colmodctr Values** The *colmodctr* values that SQL Server keeps track of are continually modified as the table data changes. Table 9-2 describes when and how the *colmodctr* values are modified based on changes to your data, including *INSERT, UPDATE, DELETE, BULK INSERT,* and *TRUNCATE TABLE* operations. Although we are only mentioning table modifications specifically, keep in mind the same *colmodctr* values are kept track of for indexed views.

**Table 9-2: Factors Affecting Changes to the Internal colmodctr Values**

| Statement | Changes to *colmodctr* Values |
|---|---|
| *INSERT* | All *colmodctr* values increased by 1 for each row inserted. |
| *DELETE* | All *colmodctr* values increased by 1 for each row deleted. |
| *UPDATE* | If the update is to nonkey columns: *colmodctr* values for modified columns are increased by 1 for each row updated. If the update is to key columns: *colmodctr* values are increased by 2 for *all* the columns in the table, for each row updated. |
| *BULK INSERT* | Treated like *N INSERT* operations. All *colmodctr* values increased by *N* where *N* is the number of rows bulk inserted. |
| *TRUNCATE TABLE* | Treated like *N DELETE* operations. All *colmodctr* values increased by *N* where *N* is the table's cardinality. |

#### Skipping the Recompilation Step

There are several situations in which SQL Server bypasses recompiling a statement for plan optimality reasons. These include the following:

- When the plan is a trivial plan. A trivial plan is one for which there are no alternative plans, based on the tables referenced by the query, and the indexes (or lack of indexes) on those tables. In these cases, where there really is only one way to process a query, any recompilation would be a waste of resources, no matter how much the statistics had changed. Keep in mind that there is no assurance that a query will continue to have a trivial plan just because it originally had a trivial plan. If new indexes have been added since the query was last compiled, there may now be multiple possible ways to process the query.
- If the query contains the OPTION hint KEEPFIXED PLAN, SQL Server will not recompile the plan for any optimality-related reasons.
- If automatic updates of statistics for indexes and statistics defined on a table or indexed view are disabled, all plan optimality-related recompilations caused by those indexes or statistics will stop.

  **Caution** Turning off the auto-statistics feature is usually not a good idea because the Query Optimizer would no longer be sensitive to data changes in those objects, and suboptimal query plans could easily result. You can consider using this technique only as a last resort after exhausting all of the other alternative ways to avoid recompilation. Make sure you thoroughly test your applications after changing the auto-statistics options to verify that you are not hurting performance in other areas.

- If all the tables referenced in the query are read-only, SQL Server will not recompile the plan.

#### Multiple Recompilations

In the previous discussion of unplanned recompilation, we primarily described situations in which a cached plan would be recompiled prior to execution. However, even if SQL Server calculates that it can reuse an existing plan, there may be cases where stale statistics or schema changes are discovered after the batch begins execution, and then a recompile occurs after execution starts. Each batch or stored procedure can contain multiple query plans, one for each optimizable statement. Before SQL Server begins executing any of the individual query plans, it checks for correctness and optimality

of that plan. If one of the checks fails, the corresponding statement is compiled *again*, and a possibly different query plan is produced.

In some cases, query plans may be recompiled even if the plan for the batch was not cached. For example, if a batch contains a literal larger than 8 KB, it is never cached. However, if this batch creates a temporary table, and then inserts multiple rows into that table, the insertion of the seventh row causes a recompilation because of passing the recompilation threshold for temporary tables. Because of the large literal, the batch was not cached, but the currently executing plan needs to be recompiled.

In SQL Server 2000, when a batch was recompiled, *all* the statements in the batch were recompiled, not just the one that initiated the recompilation. SQL Server 2005 introduced statement-level recompilation, which means that only the statement that causes the recompilation has a new plan created, not the entire batch. This means that SQL Server spends less CPU time and memory during recompilations.

**Removing Plans from Cache**

In addition to needing to recompile a plan based on schema or statistics changes, SQL Server needs to compile plans for batches if all previous plans have been removed from the plan cache. Plans are removed from cache based on memory pressure, which we talk about in the section entitled "Cache Size Management," later in this chapter. However, other operations can cause plans to be removed from cache. Some of these operations remove all the plans from a particular database, and others remove all the plans for the entire SQL Server instance.

The following operations flush the entire plan cache so that all batches submitted afterwards will need a fresh plan. Note that although some of these operations affect only a single database, the entire plan cache is cleared.

- Upgrading any database to SQL Server 2008
- Running the *DBCC FREEPROCCACHE* or *DBCC FREESYSTEMCACHE* commands
- Changing any of the following configuration options:
  - cross db ownership chaining
  - index create memory
  - cost threshold for parallelism
  - max degree of parallelism
  - max text repl size
  - min memory per query
  - min server memory
  - max server memory
  - query governor cost limit
  - query wait
  - remote query timeout
  - user options

The following operations clear all plans associated with a particular database:

- Running the *DBCC FLUSHPROCINDB* command
- Detaching a database
- Closing or opening an auto-close database
- Modifying a collation for a database using the *ALTER DATABASE … COLLATE* command

- Altering a database with any of the following commands:
  - *ALTER DATABASE … MODIFY_NAME*
  - *ALTER DATABASE … MODIFY FILEGROUP*
  - *ALTER DATABASE … SET ONLINE*
  - *ALTER DATABASE … SET OFFLINE*
  - *ALTER DATABASE … SET EMERGENCY*
  - *ALTER DATABASE … SET READ_ONLY*
  - *ALTER DATABASE … SET READ_WRITE*
  - *ALTER DATABASE … COLLATE*
- Dropping a database

Clearing a single plan from cache can be done in a couple of different ways. First, you can create a plan guide that exactly matches the SQL text for the cached plan, and then all plans with that text will be removed automatically. SQL Server 2008 provides an easy way of creating a plan guide from plan cache. We look at plan guides in detail later in the chapter. The second method of removing a single plan from cache is new in SQL Server 2008 and uses new options for *DBCC FREEPROCCACHE.* The syntax is illustrated in the following code:

```
DBCC FREEPROCCACHE [ ( { plan_handle | sql_handle | pool_name } ) ] [ WITH NO_INFOMSGS ]
```

This command now allows you to specify one of three parameters to indicate which plan or plans you want to remove from cache:

- **plan_handle** By specifying a *plan_handle*, you can remove the plan with that handle from cache. (The *plan_handle* is guaranteed to be unique for all currently existing plans.)
- **sql_handle** By specifying a *sql_handle*, you can remove the plans with that handle from cache. You can have multiple plans for the same SQL text if any of the cache key values are changed, such as SET options. The following code illustrates this:

```
USE Northwind2;
GO
DBCC FREEPROCCACHE;
GO
SET ANSI_NULLS ON
GO
SELECT * FROM orders WHERE customerid = 'HANAR';
GO
SELECT * FROM Orders WHERE CustomerID = 'CENTC';
GO
SET ANSI_NULLS OFF
GO
SELECT * FROM orders WHERE customerid = 'HANAR';
GO
SET ANSI_NULLS ON
GO

-- Now examine the sys.dm_exec_query_stats view and notice two different rows for the
-- query searching for 'HANAR'
SELECT execution_count, text, sql_handle, query_plan
FROM sys.dm_exec_query_stats
   CROSS APPLY sys.dm_exec_sql_text(sql_handle) AS TXT
       CROSS APPLY sys.dm_exec_query_plan(plan_handle)AS PLN;
GO
-- The two rows containing 'HANAR'should have the same value for sql_handle;
-- Copy that sql_handle value and paste into the command below:
DBCC FREEPROCCACHE(0x02000000CECDF507D9D4D70720F581172A42506136AA80BA);
GO
-- If you examine sys.dm_exec_query_stats again, you see the rows for this query
```

```
-- have been removed
SELECT execution_count, text, sql_handle, query_plan
FROM sys.dm_exec_query_stats
   CROSS APPLY sys.dm_exec_sql_text(sql_handle) AS TXT
       CROSS APPLY sys.dm_exec_query_plan(plan_handle)AS PLN;
GO
```

- **pool_name** By specifying the name of a Resource Governor pool, you can clear all the plans in cache that are associated with queries that were assigned to workload group using the specified resource pool. (The Resource Governor, workload groups, and resource pools were discussed in Chapter 1, “SQL Server 2008 Architecture and Configuration.”)

### Plan Cache Internals

Knowing when and how plans are reused or recompiled can help you design high-performing applications. The more you understand about optimal query plans, and how different actual values and cardinalities require different plans, the more you can determine when recompilation is a useful thing. When you are getting unnecessary recompiles, or when SQL Server is not recompiling when you think it should, your troubleshooting efforts will be easier the more you know about how plans are managed internally. In this section, we explore the internal organization of the plan cache, the metadata available, how SQL Server finds a plan in cache, plan cache sizing, and the plan eviction policy.

### Cache Stores

The plan cache in SQL Server is made up of four separate memory areas, called *cache stores.* There are actually other stores in its memory, which can be seen in the DMV called *sys.dm_os_memory_cache_counters*, but only four that contain query plans. The names in parentheses below are the values that can be seen in the *type* column of *sys.dm_os_memory_cache_counters:*

- **Object Plans (CACHESTORE_OBJCP)** Object Plans include plans for stored procedures, functions, and triggers .
- **SQL Plans (CACHESTORE_SQLCP)** SQL Plans include the plans for adhoc cached plans, autoparameterized plans, and prepared plans. The memory clerk that manages the SQLCP cache store is also used for the SQL Manager, which manages all the T-SQL text used in your adhoc queries.
- **Bound Trees (CACHESTORE_PHDR)** Bound Trees are the structures produced by the algebrizer in SQL Server for views, constraints, and defaults.
- **Extended Stored Procedures (CACHESTORE_XPROC)** Extended Procs (Xprocs) are predefined system procedures, like *sp_executesql* and *sp_tracecreate,* that are defined using a dynamic link library (DLL), not using T-SQL statements. The cached structure contains only the function name and the DLL name in which the procedure is implemented.

Each plan cache store contains a hash table to keep track of all the plans in that particular store. Each bucket in the hash table contains zero, one, or more cached plans. When determining which bucket to use, SQL Server uses a very straightforward hash algorithm. The hash key is computed as (*object_id* * *database_id*) mod (hash table size). For plans that are associated with adhoc or prepared plans, the *object_id* is an internal hash of the batch text. The DMV *sys.dm_os_memory_cache_hash_tables* contains information about each hash table, including its size. You can query this view to retrieve the number of buckets for each of the plan cache stores using the following query:

```
SELECT type as 'plan cache store', buckets_count
FROM sys.dm_os_memory_cache_hash_tables
WHERE type IN ('CACHESTORE_OBJCP', 'CACHESTORE_SQLCP',
   'CACHESTORE_PHDR', 'CACHESTORE_XPROC');
```

You should notice that the Bound Trees store has about 10 percent of the number of hash buckets of the stores for Object Plans and SQL Plans. (On a 64-bit system, the number of buckets for the Object Plan and SQL Plan stores is about 40,000 and on a 32-bit system, the number is about 10,000.) The number of buckets for the Extended Stored Procedures store is always set to 127 entries. We will not be discussing Bound Trees and Extended Stored Procedures further. The rest of the chapter dealing with caching of plans is concerned only with Object Plans and SQL Plans.

To find the size of the stores themselves, you can use the view *sys.dm_os_memory_objects*. The following query returns the size of all the cache stores holding plans, plus the size of the SQL Manager, which stores the T-SQL text of all the adhoc and prepared queries:

```
SELECT type AS Store, SUM(pages_allocated_count) AS Pages_used
FROM sys.dm_os_memory_objects
WHERE type IN ('MEMOBJ_CACHESTOREOBJCP', 'MEMOBJ_CACHESTORESQLCP',
  'MEMOBJ_CACHESTOREXPROC', 'MEMOBJ_SQLMGR')
GROUP BY type
```

Finding a plan in cache is a two-step process. The hash key described previously leads SQL Server to the bucket in which a plan might be found, but if there are multiple entries in the bucket, SQL Server needs more information to determine if the exact plan it is looking for can be found. For this second step, it needs a cache key, which is a combination of several attributes of the plan. Earlier, we looked at the DMF *sys.dm_exec_plan_attributes*, to which we could pass a *plan_handle.* The results obtained were a list of attributes for a particular plan, and a Boolean value indicating whether that particular value was part of the cache key. Table 9-1 included 17 attributes that comprise the cache key, and SQL Server needs to make sure all 17 values match before determining that it has found a matching plan in cache. In addition to the 17 values found in *sys.dm_exec_plan_attributes,* the column *sys.dm_exec_cached_plans.pool_id* is also part of the cache key for any plan.

### Compiled Plans

There are two main types of plans in the Object and SQL plan cache stores: compiled plans and execution plans. Compiled plans are the type of object we have been looking at up to this point when examining the *sys.dm_exec_cached_plans* view. We have already discussed the three main *objtype* values that can correspond to a compiled plan: *Adhoc, Prepared,* and *Proc.* Compiled plans can be stored in either the Object Store or the SQL Store depending on which of those three *objtype* values they have. The compiled plans are considered valuable memory objects, since they can be costly to re-create. SQL Server attempts to keep them in cache. When SQL Server experiences heavy memory pressure, the policies used to remove cache objects ensure that our compiled plans are not the first objects to be removed.

A compiled plan is generated for an entire batch, not just for a single statement. For a multistatement batch, you can think of the compiled plan as an array of plans, with each element of the array containing a query plan for an individual statement. Compiled plans can be shared between multiple sessions or users. However, you should be aware that not every user executing the same plan will get the same results, even if there is no change to the underlying data. Unless the compiled plan is an adhoc plan, each user has his or her own parameters and local variables, and the batch may build temporary tables or worktables specific to that user. The information specific to one particular execution of a compiled plan is stored in another structure called the *executable plan.*

### Execution Contexts

Executable plans, or execution contexts, are considered to be dependent on compiled plans and do not show up in the *sys.dm_exec_cached_plans* view. Executable plans are run-time objects created when a compiled plan is executed. Just as for compiled plans, executable plans can be Object Plans, stored in the Object Store, or SQL Plans, stored in the SQL Store. Each executable plan exists in the same cache store as the compiled plan on which it depends. Executable plans contain the particular run-time information for one execution of a compiled plan, and include the actual run-time parameters, any local variable information, object IDs for objects created at run time, the user ID, and information about the currently executing statement in the batch.

When SQL Server starts executing a compiled plan, it generates an executable plan from that compiled plan. Each individual statement in a compiled plan gets its own executable plan, which you can think of as a run-time query plan. Unlike compiled plans, executable plans are for a single session. For example, if 100 users are executing the same batch simultaneously, there will be 100 executable plans for the same compiled plan. Executable plans can be regenerated from their associated compiled plan, and they are relatively inexpensive to create. Later in this section, we look at the *sys.dm_exec_cached_plan_dependent_objects* view, which contains information about your executable plans. Note that Compiled Plan Stubs, generated when the Optimize for Ad Hoc Workloads configuration option is set to 1, do not have associated execution contexts.

### Plan Cache Metadata

We have already looked at some of the information in the *sys.dm_exec_cached_plans* DMV when we looked at *usecount* information to determine whether or not our plans were being reused. In this section, we look at some of the other metadata objects and discuss the meaning of some of the data contained in the metadata.

### Handles

The *sys.dm_exec_cached_plans* view contains a value called a *plan_handle* for every compiled plan. The *plan_handle* is a hash value that SQL Server derives from the compiled plan of the entire batch, and it is guaranteed to be unique for every currently existing compiled plan. (The *plan_handle* values can be reused over time.) The *plan_handle* can be used as an identifier for a compiled plan. The *plan_handle* remains the same even if individual statements in the batch are recompiled because of the correctness or optimality reasons discussed earlier.

As mentioned, the compiled plans are stored in the two cache stores, depending on whether the plan is an Object Plan or a SQL Plan. The actual SQL Text of the batch or object is stored in another cache called the *SQL Manager Cache (SQLMGR).* The T-SQL Text associated with each batch is stored in its entirety, including all the comments. The T-SQL Text cached in the SQLMGR cache can be retrieved using a data value called the *sql_handle.* The *sql_handle* contains a hash of the entire batch text, and because it is unique for every batch, the *sql_handle* can serve as an identifier for the batch text in the SQLMGR cache.

Any specific T-SQL batch always has the same *sql_handle*, but it may not always have the same *plan_handle*. If any of the values in the cache key change, we get a new *plan_handle* in plan cache. Refer back to Table 9-1 to see which plan attributes make up the cache keys. The relationship between *sql_handle* and *plan_handle,* therefore, is 1:*N*.

We've seen that *plan_handle* values can be obtained easily from the *sys.dm_exec_cached_ plans* view. We can get the *sql_handle* value that corresponds to a particular *plan_handle* from the *sys.dm_exec_plan_attributes* function that we looked at earlier. Here is the same query we discussed earlier to return attribute information and pivot it so that three of the attributes are returned in the same row as the *plan_handle* value:

```
SELECT plan_handle, pvt.set_options, pvt.object_id, pvt.sql_handle
FROM (SELECT plan_handle, epa.attribute, epa.value
     FROM sys.dm_exec_cached_plans
        OUTER APPLY sys.dm_exec_plan_attributes(plan_handle) AS epa
    WHERE cacheobjtype = 'Compiled Plan'
    ) AS ecpa
PIVOT (MAX(ecpa.value) FOR ecpa.attribute
   IN ("set_options", "object_id", "sql_handle")) AS pvt;
```

The *sys.dm_exec_query_stats* view contains both *plan_handle* and *sql_handle* values, as well as information about how often each plan was executed and how much work was involved in the execution. The value for *sql_handle* is very cryptic, and it's sometimes difficult to determine which of our queries each *sql_handle* corresponds to. To get that information, we can use another function.

### sys.dm_exec_sql_text

The function *sys.dm_exec_sql_text* can take either a *sql_handle* or a *plan_handle* as a parameter, and it returns the SQL Text that corresponds to the handle. Any sensitive information that might be contained in the SQL Text, like passwords, are blocked when the SQL is returned. The text column in the functions output contains the entire SQL batch text for adhoc, prepared, and autoparameterized queries, and for objects like triggers, procedures, and functions, it gives the full object definition.

Viewing the SQL Text from *sys.dm_exec_sql_text* is useful in quickly identifying identical batches that may have different compiled plans because of several factors, like SET option differences. As an example, consider the following code, which executes two identical batches. This example is similar to the one we saw previously when I discussed using *DBCC FREEPROCCACHE* with a *sql_handle*, but this time, we see the *sql_handle* and *plan_handle* values. The only difference between the two consecutive executions is that the value of the SET option QUOTED _IDENTIFIER has changed. It is OFF in the first execution and ON in the second. After executing both batches, we examine the *sys.dm_exec_query_stats* view:

```
USE Northwind2;
DBCC FREEPROCCACHE;
SET QUOTED_IDENTIFIER OFF;
GO
-- this is an example of the relationship between
-- sql_handle and plan_handle
SELECT LastName, FirstName, Country
FROM Employees
WHERE Country <> 'USA';
GO
```

```
SET QUOTED_IDENTIFIER ON;
GO
-- this is an example of the relationship between
-- sql_handle and plan_handle
SELECT LastName, FirstName, Country
FROM Employees
WHERE Country <> 'USA';
GO
SELECT st.text, qs. sql_handle, qs.plan_handle
FROM sys.dm_exec_query_stats qs
   CROSS APPLY sys.dm_exec_sql_text(sql_handle) st;
GO
```

You should see two rows with the same text string and *sql_handle*, but with different *plan_handle* values, as shown here. (In our output, the difference between the two *plan_handle* values is only a single digit so it may be hard to see, but in other cases, the difference may be more obvious.)

| *text* | *sql_handle* | *plan_handle* |
|---|---|---|
| -- this is an example of the<br>-- relationship between<br>-- sql_handle and plan_handle<br>SELECT LastName, FirstName,<br>Country FROM Employees<br>WHERE Country <> 'USA' | 0x0200000012330<br>B0EEA82077439354E7A<br>5B12E1B7E37A1361 | 0x0600120012330B0EB82<br>187050000000000000000<br>00000000 |
| -- this is an example of the<br>-- relationship between<br>-- sql_handle and plan_handle<br>SELECT LastName, FirstName,<br>Country FROM Employees<br>WHERE Country <> 'USA' | 0x0200000012330<br>B0EEA82077439354E7A<br>5B12E1B7E37A1361 | 0x0600120012330B0EB82<br>186050000000000000000<br>00000000 |

We can see that we have two plans corresponding to the same batch text, and this example should make clear the importance of making sure that all the SET options that affect plan caching should be the same when the same queries are executed repeatedly. You should verify whatever changes your programming interface makes to your SET options to make sure you don't end up with different plans unintentionally. Not all interfaces use the same defaults for the SET option values. For example, the OSQL interface uses the ODBC driver, which sets QUOTED_IDENTIFIER to OFF for every connection, whereas Management Studio uses ADO.NET, which sets QUOTED_IDENTIFIER to ON. Executing the same batches from these two different clients results in multiple plans in cache.

### sys.dm_exec_query_plan

The function *sys.dm_exec_query_plan* is a table-valued function that takes a *plan_handle* as a parameter and returns the associated query plan in XML format. If the plan is for an object, the TVF includes the database ID, object ID, procedure number, and encryption state of the object. If the plan is for an adhoc or prepared query, these additional values are NULL. If the *plan_handle* corresponds to a *Compiled Plan Stub*, the query plan will also be NULL. I have used this function in some of the preceding examples.

### sys.dm_exec_text_query_plan

The function *sys.dm_exec_text_query_plan* is a table-valued function that takes a *plan_handle* as a parameter and returns the same basic information as *sys.dm_exec_query_plan*. The differences between the two functions are as follows:

- *sys.dm_exec_text_query_plan* can take optional input parameters to specify the start and end offset of statements with a batch.
- The output of *sys.dm_exec_text_query_plan* returns the plan as text data, instead of XML data.
- The XML output for the query plan returned by *sys.dm_exec_query_plan* is limited to 128 levels of nested elements. If the plan exceeds that, a NULL is returned. The text output for the query plan returned by

*sys.dm_exec_text_query_plan* is not limited in size.

### sys.dm_exec_cached_plans

The *sys.dm_exec_cached_plans* view is the one we use most often for troubleshooting query plan recompilation issues. It's the one I used in the first section to illustrate the plan reuse behavior of adhoc plans compared to autoparameterized and prepared plans. This view has one row per cached plan, and in addition to the *plan_handle* and *usecounts*, which we've looked at already, this DMV has other useful information about the cached plans, including the following:

- **size_in_byte** The number of bytes consumed by this cache object
- **cacheobjtype** The type of the cache object; that is, if it's a Compiled Plan, or a Parse Tree or an Extended Proc
- **memory_object_address** The memory address of the cache object, which can be used to get the memory breakdown of the cache object

Although this DMV does not have the SQL Text associated with each compiled plan, we've seen that we can find it by passing the *plan_handle* to the *sys.dm_exec_sql_text* function. We can use the query below to retrieve the *text*, *usecounts*, and *size_in_bytes* of the compiled plan and *cacheobjtype* for all the plans in cache. The results are returned in order of frequency, with the batch having the most use showing up first:

```
SELECT st.text, cp.plan_handle, cp.usecounts, cp.size_in_bytes,
  cp.cacheobjtype, cp.objtype
FROM sys.dm_exec_cached_plans cp
  CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) st
ORDER BY cp.usecounts DESC;
```

### sys.dm_exec_cached_plan_dependent_objects

The *sys.dm_exec_cached_plan_dependent_objects* function returns one row for every dependent object of a compiled plan when you pass a valid *plan_handle* in as a parameter. If the *plan_handle* is not that of a compiled plan, the function returns NULL. Dependent objects include executable plans, as discussed previously, as well as plans for cursors used by the compiled plan. The example shown here uses *sys.dm_exec_cached_plan_dependent_objects,* as well as *sys.dm_exec_cached_plans,* to retrieve the dependent objects for all compiled plans, the *plan_handle,* and their *usecounts.* It also calls the *sys.dm_exec_sql_text* function to return the associated T-SQL batch:

```
SELECT text, plan_handle, d.usecounts, d.cacheobjtype
FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
CROSS APPLY
   sys.dm_exec_cached_plan_dependent_objects(plan_handle) d;
```

### sys.dm_exec_requests

The *sys.dm_exec_requests* view returns one row for every currently executing request within your SQL Server instance and is useful for many purposes in addition to tracking down plan cache information. This DMV contains the *sql_handle* and the *plan_handle* for the current statement, as well as resource usage information for each request. For troubleshooting purposes, you can use this view to help identify long-running queries. Keep in mind that the *sql_handle* points to the T-SQL for the entire batch. However, the *sys.dm_exec_requests* view contains the *statement_start_offset* and *statement_end_offset* columns, which indicate the position within the entire batch where the currently executing statement can be found. The offsets start at 0, and an offset of –1 indicates the end of the batch. The statement offsets can be used in combination with the *sql_handle* passed to *sys.dm_exec_sql_text* to extract the query text from the entire batch text, as demonstrated in the following code. This query returns the 10 longest-running queries currently executing:

```
SELECT TOP 10 SUBSTRING(text, (statement_start_offset/2) + 1,
   ((CASE statement_end_offset
        WHEN -1
           THEN DATALENGTH(text)
        ELSE statement_end_offset
   END - statement_start_offset)/2) + 1) AS query_text, *
FROM sys.dm_exec_requests
   CROSS APPLY sys.dm_exec_sql_text(sql_handle)
ORDER BY total_elapsed_time DESC;
```

Note that including the '*' in the *SELECT* list indicates that this query should return *all* of the columns from the

*sys.dm_exec_requests* view. You should replace the '*' with the columns that you are particularly interested in, such as *start_time, blocking_session_id,* and so on.

### sys.dm_exec_query_stats

Just as the text returned from the *sql_handle* is the text for the entire batch, the compiled plans that are returned are for the entire batch. For optimum troubleshooting, we can use *sys.dm_exec_query_stats* to return performance information for individual queries within a batch. This view returns performance statistics for queries, aggregated across all executions of the same query. This view also returns both a *sql_handle* and a *plan_handle,* as well as the start and end offsets like we saw in *sys.dm_exec_requests*. The following query returns the top 10 queries by total CPU time, to help you identify the most expensive queries on your SQL Server instance:

```
SELECT TOP 10 SUBSTRING(text, (statement_start_offset/2) + 1,
   ((CASE statement_end_offset
        WHEN -1
           THEN DATALENGTH(text)
        ELSE statement_end_offset
   END - statement_start_offset)/2) + 1) AS query_text, *
FROM sys.dm_exec_query_stats
   CROSS APPLY sys.dm_exec_sql_text(sql_handle)
   CROSS APPLY sys.dm_exec_query_plan(plan_handle)
ORDER BY total_elapsed_time/execution_count DESC;
```

This view has one row per query statement within a batch, and when a plan is removed from cache, the corresponding rows and the accumulated statistics for that statement are removed from this view. In addition to the *plan_handle, sql_handle*, and performance information, this view contains two new columns in SQL Server 2008, which can help you identify similar queries with different plans.

- **query_hash** This value is a hash of the query text and can be used to identify similar queries with the plan cache. Queries that differ only in the values of constants have the same *query_hash* value.
- **query_plan_hash** This value is a hash of the query execution plan and can be used to identify similar plans based on logical and physical operators and a subset of the operator attributes. To look for cases where you might not want to implement forced parameterization, you can search for queries that have similar *query_hash* values but different *query_plan_hash* values.

There are two main differences between *sys.dm_exec_cached_plans* and *sys.dm_exec_query_stats*. First, *sys.dm_exec_cached_plans* has one row for each batch that has been compiled and cached, whereas *sys.dm_exec_query_stats* has one row for each statement. Second, *sys.dm_exec_query_stats* contains summary information aggregating all the executions of a particular statement. The *sys.dm_exec_query_stats* returns a tremendous amount of performance information for each query, including the number of times it was executed, and the cumulative I/O, CPU, and duration information. Keep in mind that this view is updated only when a query is completed, so you might need to retrieve information multiple times if there is currently a large workload on your server.

### Cache Size Management

We've already talked about plan reuse and how SQL Server finds a plan in cache. In this section, we look at how SQL Server manages the size of plan cache and how it determines which plans to remove if there is no room left in cache. Earlier, I discussed a few situations in which plans would be removed from cache. These situations included global operations like running *DBCC FREEPROCCACHE* to clear all plans from cache, as well as changes to a single procedure, such as *ALTER PROCEDURE,* which would drop all plans for that procedure from cache. In most other situations, plans are removed from cache only when memory pressure is detected. The algorithm that SQL Server uses to determine when and how plans should be removed from cache is called the *eviction policy.* Each cache store can have its own eviction policy, but we are discussing only the policies for the Object Plan store and the SQL Plan store.

Determining which plans to evict is based on the cost of the plan, which is discussed in the next section. When eviction starts is based on memory pressure. When SQL Server detects memory pressure, zero-cost plans are removed from cache and the cost of all other plans is reduced by half. As discussed in Chapter 1, there are two types of memory pressure, and both types lead to removal of plans from cache. These two types of memory pressure are referred to as *local* and *global memory pressure*.

When discussing memory pressure, we refer to the term *visible memory.* Visible memory is the directly addressable

physical memory available to the SQL Server buffer pool. On a 32-bit SQL Server instance, the maximum value for the visible memory is either 2 GB or 3 GB, depending on whether you have the */3 GB* flag set in your boot.ini file. Memory with addresses greater than 2 GB or 3 GB is available only indirectly, through *AWE-mapped-memory.* On a 64-bit SQL Server instance, visible memory has no special meaning, as all the memory is directly addressable. In any of the discussion that follows, if we refer to visible target memory greater than 3 GB, keep in mind that is only possible on a 64-bit SQL Server instance. The term *target memory* refers to the maximum amount of memory that can be committed to the SQL Server process. Target memory refers to the physical memory committed to the buffer pool and is the lesser of the value you have configured for max server memory and the total amount of physical memory available to the operating system. So *visible target memory* is the visible portion of the target memory. Query plans can be stored only in the non-AWE-mapped memory, which is why the concept of visible memory is important. You can see a value for visible memory, specified as the number of 8-KB buffers, in the *bpool_visible* column in the *sys.dm_os_sys_info* DMV. This view also contains values for *bpool_committed* and *bpool_commit_target*.

SQL Server defines a cache store pressure limit value, which varies depending on the version you're running and the amount of visible target memory. We explain shortly how this value is used. The formula for determining the plan cache pressure limit changed in SQL Server 2005 SP2. Table 9-3 shows how to determine the plan cache pressure limit in SQL Server 2000 and 2005, and indicates the change in SP2, which reduced the pressure limit with higher amounts of memory. SQL Server 2008 RTM uses the same formulas that were added in SQL Server 2005 SP2. Be aware that these formulas may be subject to change again in future service packs.

**Table 9-3: Determining the Plan Cache Pressure Limit**

| SQL Server Version | Cache Pressure Limit |
|---|---|
| SQL Server 2005 RTM & SP1 | 75 percent of visible target memory from 0 to 8 GB + 50 percent of visible target memory from 8 GB to 64 GB + 25 percent of visible target memory > 64 GB |
| SQL Server 2005 SP2 and SP3, S QL Server 2008 RTM | 75 percent of visible target memory from 0 to 4 GB + 10 percent of visible target memory from 4 GB to 64 GB + 5 percent of visible target memory > 64 GB |
| SQL Server 2000 | 4 GB upper cap on the plan cache |

As an example, assume we are on SQL Server 2005 SP1 on a 64-bit SQL Server instance with 28 GB of target memory. The plan cache pressure limit would be 75 percent of 8 GB plus 50 percent of the target memory over 8 GB (or 50 percent of 20 GB), which is 6 GB + 10 GB, or 16 GB.

On a 64-bit SQL Server 2008 RTM instance with 28 GB of target memory, the plan cache pressure limit would be 75 percent of 4 GB plus 10 percent of the target memory over 4 GB (or 10 percent of 24 GB), which is 3 GB + 2.4 GB, or 5.4 GB.

#### Local Memory Pressure

If any single cache store grows too big, it indicates local memory pressure and SQL Server starts removing entries from only that store. This behavior prevents one store from using too much of the total system memory.

If a cache store reaches 75 percent of the plan cache pressure limit, described in Table 9-3, in single-page allocations or 50 percent of the plan cache pressure limit in multipage allocations, internal memory pressure is triggered and plans are removed from cache. For example, in the situation described previously, we computed the plan cache pressure limit to be 5.4 GB. If any cache store exceeds 75 percent of that value, or 4.05 GB in single-page allocations, internal memory pressure is triggered. If adding a particular plan to cache causes the cache store to exceed the limit, the removal of other plans from cache happens on the same thread as the one adding the new plan, which can cause the response time of the new query to be increased.

In addition to memory pressure occurring when the total amount of memory reaches a particular limit, SQL Server also indicates memory pressure when the number of plans in a store exceeds four times the hash table size for that store, regardless of the actual size of the plans. As I mentioned previously when describing the cache stores, there are either about 10,000 or 40,000 buckets in these hash tables, for 32-bit and 64-bit systems, respectively. That means memory pressure can be triggered when either the SQL Store or the Object Store has more than 40,000 or 160,000 entries. The first query shown here is one we saw earlier, and it can be used to determine the number of buckets in the hash tables for the Object Store and the SQL Store, and the second query returns the number of entries in each of those stores:

```
SELECT type as 'plan cache store', buckets_count
FROM sys.dm_os_memory_cache_hash_tables
WHERE type IN ('CACHESTORE_OBJCP', 'CACHESTORE_SQLCP');
GO
```

```
SELECT type, count(*) total_entries
FROM sys.dm_os_memory_cache_entries
WHERE type IN ('CACHESTORE_SQLCP', 'CACHESTORE_OBJCP')
GROUP BY type;
GO
```

Prior to SQL Server 2008, internal memory pressure was rarely triggered due to the number of entries in the hash tables but was almost always initiated by the size of the plans in the cache store. However, in SQL Server 2008, if you have enabled Optimize for Ad Hoc Workloads, the actual entries in the SQL cache store may be quite small (each *Compiled Plan Stub* is about 300 bytes) so the number of entries can grow to exceed the limit before the size of the store gets too large. If Optimize for Ad Hoc Workloads is not on, the size of the entries in cache is much larger, with a minimum size of 24 KB for each plan. To see the size of all the plans in a cache store, you need to examine *sys.dm_exec_cached_plans*, as shown here:

```
SELECT objtype, count(*) AS 'number of plans',
      SUM(size_in_bytes)/(1024.0 * 1024.0 * 1024.0)
            AS size_in_gb_single_use_plans
FROM sys.dm_exec_cached_plans
GROUP BY objtype;
```

Remember that the adhoc and prepared plans are both stored in the SQL cache store, so to monitor the size of that store, you have to add those two values together.

#### Global Memory Pressure

Global memory pressure applies to memory used by all the cache stores together, and can be either external or internal. External global pressure occurs when the operating system determines that the SQL Server process needs to reduce its physical memory consumption because of competing needs from other processes on the server. All cache stores are reduced in size when this occurs.

Internal global memory pressure can occur when virtual address space is low. Internal global memory pressure can also occur when the memory broker predicts that all cache stores combined will use more than 80 percent of the plan cache pressure limit. Again, all cache stores will have entries removed when this occurs.

As mentioned, when SQL Server detects memory pressure, all zero-cost plans are removed from cache and the cost of all other plans is reduced by half. Any particular cycle updates the cost of at most 16 entries for every cache store. When an updated entry has a zero-cost value, it can be removed. There is no mechanism to free entries that are currently in use. However, unused dependent objects for an in-use compiled plan can be removed. Dependent objects include the executable plans and cursors, and up to half of the memory for these objects can be removed when memory pressure exists. Remember that dependent objects are inexpensive to re-create, especially compared to compiled plans.

> **More Info** For more information on memory management and memory pressure, see Chapter 1.

### Costing of Cache Entries

The decision of what plans to evict from cache is based on their cost. For adhoc plans, the cost is considered to be zero, but it is increased by 1 every time the plan is reused. For other types of plans, the cost is a measure of the resources required to produce the plan. When one of these plans is reused, the cost is reset to the original cost. For non-adhoc queries, the cost is measured in units called *ticks,* with a maximum of 31. The cost is based on three factors: I/O, context switches, and memory. Each has its own maximum within the 31-tick total:

- I/O: each I/O costs 1 tick, with a maximum of 19
- Context switches: 1 tick each, with a maximum of 8
- Memory: 1 tick per 16 pages, with a maximum of 4

When not under memory pressure, costs are not decreased until the total size of all plans cached reaches 50 percent of the buffer pool size. At that point, the next plan access decrement the cost in ticks of all plans by 1. Once memory pressure is encountered, then SQL Server starts a dedicated resource monitor thread to decrement the cost of either plan objects in one particular cache (for local pressure) or all plan cache objects (for global pressure).

The *sys.dm_os_memory_cache_entries* DMV can show you the current and original cost of any cache entry, as well as

the components that make up that cost:

```
SELECT text, objtype, refcounts, usecounts, size_in_bytes,
   disk_ios_count, context_switches_count,
   pages_allocated_count, original_cost, current_cost
FROM sys.dm_exec_cached_plans p
   CROSS APPLY sys.dm_exec_sql_text(plan_handle)
   JOIN sys.dm_os_memory_cache_entries e
   ON p.memory_object_address = e.memory_object_address
WHERE cacheobjtype = 'Compiled Plan'
   AND type in ('CACHESTORE_SQLCP', 'CACHESTORE_OBJCP')
ORDER BY objtype desc, usecounts DESC;
```

Note that we can find the specific entry in *sys.dm_os_memory_cache_entries* that corresponds to a particular plan in *sys.dm_exec_cached_plans* by joining on the *memory_object_address* column.

### Objects in Plan Cache: The Big Picture

In addition to the DMVs and DMFs discussed so far, there is another metadata object called *syscacheobjects* that is really just a pseudotable. Prior to SQL Server 2005, there were no Dynamic Management Objects, but we did have about half a dozen of these pseudotables, including *sysprocesses* and *syslockinfo,* which took no space on disk and were materialized only when someone executed a query to access them, in a similar manner to the way that Dynamic Management Objects work. These objects are still available in SQL Server 2008. In SQL Server 2000, the pseudotables are available only in the *master* database, or by using a full object qualification when referencing them. In SQL Server 2008, you can access *syscacheobjects* from any database using only the *sys* schema as a qualification, so we refer to the object using its schema. Table 9-4 lists some of the more useful columns in the *sys.syscacheobjects* object.

**Table 9-4: Useful Columns in the sys.syscacheobjects View**

| Column Name | Description |
|---|---|
| *bucketid* | The bucket ID for this plan in an internal hash table; the bucket ID helps SQL Server locate the plan more quickly. Two rows with the same bucket ID refer to the same object (for example, the same procedure or trigger). |
| *cacheobjtype* | Type of object in cache: *Compiled Plan*, *Parse Tree,* and so on. |
| *objtype* | Type of object: *Adhoc*, *Prepared, Proc,* and so on. |
| *objid* | One of the main keys used for looking up an object in cache. This is the object ID stored in *sysobjects* for database objects (procedures, views, triggers, and so on). For cache objects, such as *Adhoc* or *Prepared*, *objid* is an internally generated value. |
| *dbid* | Database ID in which the cache object was compiled. |
| *uid* | The creator of the plan (for adhoc query plans and prepared plans). |
| *refcounts* | Number of other cache objects that reference this cache object. |
| *usecounts* | Number of times this cache object has been used since its creation. |
| *pagesused* | Number of memory pages consumed by the cache object. |
| *setopts* | SET option settings that affect a compiled plan. Changes to values in this column indicate that users have modified SET options. |
| *langid* | Language ID of the connection that created the cache object. |
| *dateformat* | Date format of the connection that created the cache object. |
| *sql* | Module definition or first 3,900 characters of the batch submitted. |

In SQL Server 2000, the *syscacheobjects* pseudotable also includes entries for executable plans. That is, the *cacheobjtype* column could have a value of *Executable Plan.* In SQL Server 2008, because executable plans are considered dependent objects and are stored completely separately from the compiled plans, they are no longer available through the *sys. syscacheobjects* view. To access the executable plans, you need to select directly from the *sys.dm_exec_cached_plan_dependent_objects* function, and pass in a *plan_handle* as a parameter.

As an alternative to the *sys.syscacheobjects* view, which is a compatibility view and is not guaranteed to exist in future versions, you can create your own view that retrieves the same information from the SQL Server Dynamic Management Objects. The script creates a view called *sp_cacheobjects* in the *master* database. Remember that any objects with a

name starting with *sp_*, created in the *master* database, can be accessed from any database without having to qualify the object name fully. Besides being able to access the *sp_cacheobjects* view from anywhere, another benefit of creating your own object is that you can customize it. For example, it would be relatively straightforward to do one more OUTER APPLY, to join this view with the *sys.dm_exec_query_plan* function, to get the XML plan for each of the plans in cache.

```
USE master
GO
CREATE VIEW sp_cacheobjects
   (bucketid, cacheobjtype, objtype, objid, dbid, dbidexec, uid,
   refcounts, usecounts, pagesused, setopts, langid, dateformat,
   status, lasttime, maxexectime, avgexectime, lastreads,
   lastwrites, sqlbytes, sql)
AS
   SELECT pvt.bucketid,
      CONVERT(nvarchar(18), pvt.cacheobjtype) AS cacheobjtype,
      pvt.objtype,
      CONVERT(int, pvt.objectid) AS object_id,
      CONVERT(smallint, pvt.dbid) AS dbid,
      CONVERT(smallint, pvt.dbid_execute) AS execute_dbid,
      CONVERT(smallint, pvt.user_id) AS user_id,
      pvt.refcounts, pvt.usecounts,
      pvt.size_in_bytes / 8192 AS size_in_bytes,
      CONVERT(int, pvt.set_options) AS setopts,
      CONVERT(smallint, pvt.language_id) AS langid,
      CONVERT(smallint, pvt.date_format) AS date_format,
      CONVERT(int, pvt.status) AS status,
      CONVERT(bigint, 0),
      CONVERT(bigint, 0),
      CONVERT(bigint, 0),
      CONVERT(bigint, 0),
      CONVERT(bigint, 0),
      CONVERT(int, LEN(CONVERT(nvarchar(max), fgs.text)) * 2),
      CONVERT(nvarchar(3900), fgs.text)
FROM (SELECT ecp.*, epa.attribute, epa.value
  FROM sys.dm_exec_cached_plans ecp
    OUTER APPLY
      sys.dm_exec_plan_attributes(ecp.plan_handle) epa) AS ecpa
PIVOT (MAX(ecpa.value) for ecpa.attribute IN
    ("set_options", "objectid", "dbid",
    "dbid_execute", "user_id", "language_id",
    "date_format", "status")) AS pvt
OUTER APPLY sys.dm_exec_sql_text(pvt.plan_handle) fgs;
```

You might notice that several of the output columns are hardcoded to a value of 0. For the most part, these are columns for data that is no longer maintained in SQL Server 2005 or SQL Server 2008. In particular, these are columns that report on performance information for cached plans. In SQL Server 2000, this performance data was maintained for each batch. In later versions, it is maintained on a statement level and available through *sys.dm_exec_ query_ stats*. To be compatible with the *sys*.*syscacheobjects* view, the new view must return something in those column positions. If you choose to customize this view, you could choose to remove those columns.

### Multiple Plans in Cache

SQL Server tries to limit the number of plans for a query or a procedure. Because plans are reentrant, this is easy to accomplish. You should be aware of some situations that cause multiple query plans for the same procedure to be saved in cache. The most likely situation is a difference in certain SET options, as discussed previously.

One other connection issue can affect whether a plan can be reused. If an owner name must be resolved implicitly, a plan cannot be reused. For example, suppose user *sue* issues the following *SELECT* statement:

```
SELECT * FROM Orders;
```

SQL Server first tries to resolve the object by looking for an object called *Orders* in the default schema for the user *sue*, and if no such object can be found, it looks for an object called *Orders* in the *dbo* schema. If user *dan* executes the exact same query, the object can be resolved in a completely different way (to a table in the default schema of the user *dan*), so *sue* and *dan* could not share the plan generated for this query. Because there is a possible ambiguity when using the unqualified object name, the query processor does not assume that an existing plan can be reused. However, the situation

is different if *sue* issues this command:

```
SELECT * FROM dbo.Orders;
```

Now there's no ambiguity. Anyone executing this exact query always references the same object. In *the sys.syscacheobjects* view, the column *uid* indicates the user ID for the connection in which the plan was generated. For adhoc queries, only another connection with the same *user ID* value can use the same plan. The one exception is if the *user ID* value is recorded as –2 in *syscacheobjects*, which indicates that the query submitted does not depend on implicit name resolution and can be shared among different users. This is the preferred method.

**Tip** It is strongly recommended that objects are always qualified with their containing schema name, so that you never need to rely on implicit resolutions and the reuse of plan cache can be more effective.

### When to Use Stored Procedures and Other Caching Mechanisms

Keep the following guidelines in mind when you are deciding whether to use stored procedures or one of the other query mechanisms:

- **Stored procedures** These objects should be used when multiple connections are executing batches in which the parameters are known. They are also useful when you need to have control over when a block of code is to be recompiled.
- **Adhoc caching** This option is beneficial only in limited scenarios. It is not dependable enough for you to design an application expecting this behavior to correctly control reuse of appropriate plans.
- **Simple or forced parameterization** This option can be useful for applications that cannot be easily modified. However, it is preferable when you initially design your applications that you use methods that explicitly allow you to declare what your parameters and what their data types are, such as the two suggestions below.
- **The sp_executesql procedure** This procedure can be useful when the same batch might be used multiple times and when the parameters are known.
- **The prepare and execute methods** These methods are useful when multiple users are executing batches in which the parameters are known, or when a single user will definitely use the same batch multiple times.

### Troubleshooting Plan Cache Issues

To start addressing problems with plan cache usage and management, you must determine that existing problems are actually caused by plan caching issues. Performance problems caused by misuse or mismanagement of plan cache, or inappropriate recompilation, can manifest themselves as simply a decrease in throughput or an increase in query response time. Problems with caching can also show up as out-of-memory errors or connection time-out errors, which can be caused by all sorts of different conditions.

### Wait Statistics Indicating Plan Cache Problems

To determine that plan caching behavior is causing problems, one of the first things to look at is your wait statistics in SQL Server. Wait statistics are covered in more detail in Chapter 10, "Transactions and Concurrency," but here, I tell you about some of the primary wait types that can indicate problems with your plan cache.

Wait statistics are displayed when you query the *sys.dm_os_wait_stats* view. The query here lists all the resources that your SQL Server service might have to wait for, and it displays the resources with the longest waiting list:

```
SELECT *
FROM sys.dm_os_wait_stats
ORDER BY waiting_tasks_count DESC;
```

You should be aware that the values shown in this view are cumulative, so if you need to see the resources being waited on during a specific time period, you have to poll the view at the beginning and end of the period. If you see relatively large wait times for any of the following resources, or if these resources are near the top of the list returned from the previous query, you should investigate your plan cache usage:

- **CMEMTHREAD waits** This wait type indicates that there is contention on the memory object from which cache descriptors are allocated. A very high rate of insertion of entries into the plan cache can cause contention problems. Similarly, contention can also occur when entries are removed from cache and the resource monitor thread is blocked.

There is only one thread-safe memory object from which descriptors are allocated, and as we've seen, there is only a single cache store for adhoc compiled plans.

Consider the same procedure being called dozens or hundreds of times. Remember that SQL Server will cache the adhoc shell query that includes the actual parameter for each individual call to the procedure, even though there may be only one cached plan for the procedure itself. As SQL Server starts experiencing memory pressure, the work to insert the entry for each individual call to the procedure can begin to cause excessive waits resulting in a drop in throughput or even out-of-memory errors.

SQL Server 2005 SP2 added some enhancements to caching behavior to alleviate some of the flooding of cache that could occur when the same procedure or parameterized query was called repeatedly with different parameters. In all releases after SQL Server 2005 SP2, zero-cost batches that contain *SET* statements or transaction control are not cached at all. The only exception is for those batches that contain only *SET* and transaction control statements. This is not that much of a loss, as plans for batches containing *SET* statements can never be reused in any case. Also, as of SQL Server 2005 SP2, the memory object from which cache descriptors are allocated has been partitioned across all the CPUs to alleviate contention on the memory object which should reduce CMEMTHREAD waits.

- **SOS_RESERVEDMEMBLOCKLIST waits** This wait type can indicate the presence of cached plans for queries with a large number of parameters, or with a large number of values specified in an IN clause. These types of queries require that SQL Server allocate in larger units, called *multipage allocations.* You can look at the *sys.dm_os_memory_cache_counters* view to see the amount of memory allocated in the multipage units:

```
SELECT name, type, single_pages_kb, multi_pages_kb,
   single_pages_in_use_kb, multi_pages_in_use_kb
FROM sys.dm_os_memory_cache_counters
WHERE type = 'CACHESTORE_SQLCP' OR type = 'CACHESTORE_OBJCP';
```

  Clearing out plan cache with *DBCC FREEPROCCACHE* can alleviate problems caused by too many multipage allocations, at least until the queries are reexecuted and the plans are cached again. In addition, the cache management changes in SQL Server 2005 SP2 can also reduce the waits on SOS_RESERVEDMEMBLOCKLIST. You can also consider rewriting the application to use alternatives to long parameters or long IN lists. In particular, long IN lists can almost always be improved by creating a table of the values in the IN list and joining with that table.

- **RESOURCE_SEMAPHORE_QUERY_COMPILE waits** This wait type indicates that there are a large number of concurrent compilations. To prevent inefficient use of query memory, SQL Server 2008 limits the number of concurrent compile operations that need extra memory. If you notice a high value for RESOURCE_SEMAPHORE_QUERY_COMPILE waits, you can examine the entries in the plan cache through the *sys.dm_exec_cached_ plans* view, as shown here:

```
SELECT usecounts, cacheobjtype, objtype, bucketid, text
FROM sys.dm_exec_cached_plans
   CROSS APPLY sys.dm_exec_sql_text(plan_handle)
WHERE cacheobjtype = 'Compiled Plan'
ORDER BY objtype;
```

  If there are no results with the *objtype* value of *Prepared*, it means that SQL Server is not automatically parameterizing your queries. You can try altering the database to PARAMETERIZATION FORCED in this case, but this option affects the entire database, including queries that might not benefit from parameterization. To force SQL Server to parameterize just certain queries, plan guides can be used. I discuss plan guides in the next section.

Keep in mind that caching is done on a per-batch level. If you try to force parameterization using *sp_executesql* or prepare and execute, all the statements in the batch must be parameterized for the plan to be reusable. If a batch has some parameterized statements and some using constants, each execution of the batch with different constants is considered distinct, and there is no value to the parameterization in only part of the batch.

### Other Caching Issues

In addition to looking at the wait types that can indicate problems with caching, there are some other coding behaviors that can have a negative impact on plan reuse:

- **Verify parameter types, both for prepared queries and autoparameterization** With prepared queries, you actually specify the parameter data type, so it's easier to make sure you are always using the same type. When SQL Server parameterizes, it makes its own decisions as to data type. If you look at the parameterized form of your queries

of type *Prepared,* you see the data type that SQL Server assumed. We saw earlier in the chapter that a value of 12345 is assumed to be a different data type than 12, and two queries that are identical except for these specific values are never able to share the same autoparameterized plan.

If the parameter passed is numeric, SQL Server determines the data type based on the precision and scale. A value of 8.4 has a data type of *numeric* (2, 1), and 8.44 has a data type of *numeric* (3, 2). For *varchar* data type, server side parameterization is not so dependent on the length of the actual value. Take a look at these two queries in the *Northwind2* database:

```
SELECT * FROM Customers
WHERE CompanyName = 'Around the Horn';
GO
SELECT * FROM Customers
WHERE CompanyName = 'Rattlesnake Canyon Grocery';
GO
```

Both of these queries be autoparameterized to the following:

```
(@0 varchar(8000))SELECT * FROM Customers WHERE CompanyName = @0
```

- **Monitor plan cache size and data cache size** In general, as more queries are run, the amount of memory used for data page caching should increase along with the amount of memory used for plan caching. However, as we saw previously when discussing plan cache size, in SQL Server 2005 prior to SP2, the maximum limit for plan caching could grow to be up to 80 percent of the total buffer pool before memory pressure would start forcing plans to be evicted. This can result in severe performance degradation for those queries that depend on good data caching behavior. For any amount of memory greater than 4 GB, versions after SQL Server 2005 SP1 change the size limit that plan caching can grow to before memory pressure is indicated. One of the easiest places to get a comparison of the pages used for plan caching and the pages used for data caching is the performance counters. Take a look at the following counters: SQL Server: Plan Cache/Cache Pages(_Total) and SQLServer: BufferManager/Database pages.

### Handling Problems with Compilation and Recompilation

There are tools for detecting excessive compiles and recompiles. You can use either System Monitor or one of the tracing or event monitoring tools described in Chapter 2, “Change Tracking, Tracing, and Extended Events,” to detect compilations and recompilations. Keep in mind that compiling and recompiling are not the same thing. Recompiling is performed when an existing module or statement is determined to be no longer valid or no longer optimal. All recompiles are considered compiles, but not vice versa. For example, when there is no plan in cache, or when executing a procedure using the WITH RECOMPILE option or executing a procedure that was created with the WITH RECOMPILE option, SQL Server considers it a compile but not a recompile.

If these tools indicate that you have excessive compilation or recompilation, you can consider the following actions:

- If the recompile is caused by a change in a SET option, the SQL Trace text data for T-SQL statements immediately preceding the recompile event can indicate which SET option changed. It’s best to change SET options when a connection is first made and avoid changing them after you have started submitting statements on that connection, or inside a store procedure.
- Recompilation thresholds for temporary tables are lower than for normal tables, as we discussed earlier in this chapter. If the recompiles on a temporary table are caused by statistics changes, a trace has a data value in the *EventSubclass* column that indicates that statistics changed for an operation on a temporary table. You can consider changing the temporary tables to table variables, for which statistics are not maintained. Because no statistics are maintained, changes in statistics cannot induce recompilation. However, lack of statistics can result in suboptimal plans for these queries. Your own testing can determine if the benefit of table variables is worth the cost. Another alternative is to use the KEEP PLAN query hint, which sets the recompile threshold for temporary tables to be the same as for permanent tables.
- To avoid all recompilations that are caused by changes in statistics, whether on a permanent or a temporary table, you can specify the KEEPFIXED PLAN query hint. With this hint, recompilations can happen only because of correctness-related reasons, as described earlier. An example might be when a recompilation occurs if the schema of a table that is referenced by a statement changes, or if a table is marked for recompile by using the *sp_recompile* stored procedure.
- Another way to prevent recompiles caused by statistics changes is by turning off the automatic updates of statistics for indexes and columns. Note, however, that turning off the Autostatistics feature is usually not a good idea. If you do, the

Query Optimizer is no longer sensitive to data changes and is likely to come up with a suboptimal plan. This method should be considered only as a last resort after exhausting all other options.

- All T-SQL code should use two-part object names (for example, *Inventory.ProductList*) to indicate exactly what object is being referenced, which can help avoid recompilation.
- Do not use DDL within conditional constructs such as *IF* statements.
- Check to see if the stored procedure was created with the WITH RECOMPILE option. In many cases, only one or two statements within a stored procedure might benefit from recompilation on every execution, and we can use the RECOMPILE query hint for just those statements. This is much better than using the WITH RECOMPILE option for the entire procedure, which means every statement in the procedure is recompiled every time the procedure is executed.

## Plan Guides and Optimization Hints

In Chapter 8, "The Query Optimizer," we looked at many different execution plans and discussed what it meant for a query to be optimized. In this chapter, we looked at situations in which SQL Server reuses a plan when it might have been best to come up with a new one, and we've seen situations in which SQL Server does not reuse a plan even if there is a perfectly good one in cache already. One way to encourage plan reuse that has already been discussed in this chapter is to enable the PARAMETERIZATION FORCED database option. In other situations, where we just can't get the optimizer to reuse a plan, we can use optimizer hints. Optimizer hints can also be used to force SQL Server to come up with a new plan in those cases in which it might be using an existing plan. There are dozens of hints that can be used in your T-SQL code to affect the plan that SQL Server comes up with, and some of them were discussed in Chapter 8. In this section, I specifically describe only those hints that affect recompilation, as well as the mother of all hints, USE PLAN, which was added in SQL Server 2005. Finally, we discuss a SQL Server feature called *plan guides.*

### Optimization Hints

All the hints that we are telling you about in this section are referred to in *SQL Server Books Online* as Query Hints, to distinguish them from Table Hints, which are specified in the FROM clause after a table name, and Join Hints, which are specified in the JOIN clause before the word *JOIN*. However, we frequently refer to query hints as *option hints* because they are specified in a special clause called the *OPTION clause,* which is used just for specifying this type of hint. An OPTION clause, if included in a query, is always the last clause of any T-SQL statement, as you can see in the code examples in the subsequent sections.

**RECOMPILE** The RECOMPILE hint forces SQL Server to recompile a query. It is particularly useful when only a single statement within a batch needs to be recompiled. You know that SQL Server compiles your T-SQL batches as a unit, determining the execution plan for each statement in the batch, and it doesn't execute any statements until the entire batch is compiled. This means that if the batch contains a variable declaration and assignment, the assignment doesn't actually take place during the compilation phase. When the following batch is optimized, SQL Server doesn't have a specific value for the variable:

```
USE Northwind2;
DECLARE @custID nchar(10);
SET @custID = 'LAZYK';
SELECT * FROM Orders WHERE CustomerID = @custID;
```

The plan for the *SELECT* statement will show that SQL Server is scanning the entire clustered index because during optimization, SQL Server had no idea what value it was going to be searching for and couldn't use the histogram in the index statistics to get a good estimate of the number of rows. If we had replaced the variable with the constant LAZYK, SQL Server could have determined that only a very few rows would qualify and would have chosen to use the nonclustered index on *customerID*. The RECOMPILE hint can be very useful here because it tells the optimizer to come up with a new plan for the single *SELECT* statement right before that statement is executed, which is after the *SET* statement has executed:

```
USE Northwind2;
DECLARE @custID nchar(10);
SET @custID = 'LAZYK';
SELECT * FROM Orders WHERE CustomerID = @custID
OPTION (RECOMPILE);
```

**Note** A variable is not the same as a parameter, even though they are written the same way. Because a procedure is compiled only when it is being executed, SQL Server always uses a specific parameter value. Problems arise when the previously compiled plan is then used for different parameters. However, for a local variable, the value is

never known when the statements using the variable are compiled unless the RECOMPILE hint is used.

**OPTIMIZE FOR** The OPTIMIZE FOR hint tells the optimizer to optimize the query as if a particular value has been used for a variable or parameter. Execution uses the real value. Keep in mind that the OPTIMIZE FOR hint does not force a query to be recompiled. It only instructs SQL Server to assume a variable or parameter has a particular value in those cases in which SQL Server has already determined that the query needs optimization. As the OPTIMIZE FOR hint was discussed in Chapter 8, we won't say any more about it here.

**KEEP PLAN** The KEEP PLAN hint relaxes the recompile threshold for a query, particularly for queries accessing temporary tables. As we saw earlier in this chapter, a query accessing a temporary table can be recompiled when as few as six changes have been made to the table. If the query uses the KEEP PLAN hint, the recompilation threshold for temporary tables is changed to be the same as for permanent tables.

**KEEPFIXED PLAN** The KEEPFIXED PLAN hint inhibits all recompiles because of optimality issues. With this hint, queries are recompiled only when forced, or if the schema of the underlying tables is changed, as described in the section entitled "Correctness-Based Recompiles," earlier in this chapter.

**PARAMETERIZATION** The PARAMETERIZATION hint overrides the PARAMETERIZATION option for a database. If the database is set to PARAMETERIZATION FORCED, individual queries using the PARAMETERIZATION hint can avoid that and be parameterized only if they meet the strict list of conditions. Alternatively, if the database is set to PARAMETERIZATION SIMPLE, individual queries can be parameterized on a case-by-case basis. Note however that the PARAMETERIZATION hint can only be used in conjunction with plan guides, which we discuss shortly.

**USE PLAN** The USE PLAN hint was discussed in Chapter 8, as a way to force SQL Server to use a plan that you might not be able to specify using the other hints. The plan specified must be in XML format and can be obtained from a query that uses the desired plan by using the option SET SHOWPLAN_XML ON. Because USE PLAN hints contain a complete XML document in the query hint, they are best used within plan guides, which are discussed in the next section.

#### Purpose of Plan Guides

Although it is recommended in most cases that you allow the Query Optimizer to determine the best plan for each of your queries, there are times when the Query Optimizer just can't come up with the best plan and you may find that the only way to get reasonable performance is to use a hint. This is usually a straightforward change to your applications, once you have verified that the desired hint is really going to make a difference. However, in some environments, you have no control over the application code. In cases when the actual SQL queries are embedded in inaccessible vendor code or when modifying vendor code would break your licensing agreement or invalidate your support guarantees, you might not be able to simply add a hint onto the misbehaving query.

Plan guides, introduced in SQL Server 2005, provide a solution by giving you a mechanism to add hints to a query without changing the query itself. Basically, a plan guide tells the Optimizer that if it tries to optimize a query having a particular format, it should add a specified hint to the query. SQL Server supports three kinds of plan guides: SQL, Object, and Template, which we explore shortly.

Plan guides are available in the Standard, Enterprise, Evaluation, and Developer editions of SQL Server. If you detach a database containing plan guides from a supported edition and attach the database to an unsupported edition, such as Workgroup or Express, SQL Server does not use any plan guides. However the metadata containing information about plan guides is still available.

#### Types of Plan Guides

The three types of plan types can be created using the *sp_create_plan_guide* procedure. The general form of the *sp_create_plan_guide* procedure is as follows:

```
sp_create_plan_guide 'plan_guide_name', 'statement_text',
   'type_of_plan_guide', 'object_name_or_batch_text',
   'parameter_list', 'hints'
```

We discuss each of the types of plan guides, and then we look at the mechanisms for working with plan guides and the metadata that keeps track of information about them.

**Object Plan Guides** A plan guide of type *object* indicates that you are interested in a T-SQL statement appearing in the context of a SQL Server object, which can be a stored procedure, a user-defined function, or a trigger in the database in which the plan guide is created. As an example, suppose we have a stored procedure called *Sales.GetOrdersByCountry* that takes a country as a parameter, and after some error checking and other validation, it returns a set of rows for the

orders placed by customers in the specified country. Suppose further that our testing has determined that a parameter value of *US* gives us the best plan. Here is an example of a plan guide that tells SQL Server to use the OPTIMIZE FOR hint whenever the specified statement is found in the *Sales.GetOrdersByCountry* procedure:

```
EXEC sp_create_plan_guide
   @name = N'plan_US_Country',
   @stmt =
      N'SELECT SalesOrderID, OrderDate, h.CustomerID, h.TerritoryID
           FROM Sales.SalesOrderHeader AS h
           INNER JOIN Sales.Customer AS c
              ON h.CustomerID = c.CustomerID
           INNER JOIN Sales.SalesTerritory AS t
              ON c.TerritoryID = t.TerritoryID
           WHERE t.CountryRegionCode = @Country',
@type = N'OBJECT',
@module_or_batch = N'Sales.GetOrdersByCountry',
@params = NULL,
@hints = N'OPTION (OPTIMIZE FOR (@Country = N''US''))';
```

Once this plan is created in the *AdventureWorks2008* database, every time the *Sales. GetOrdersByCountry* procedure is compiled, the statement indicated in the plan is optimized as if the actual parameter passed was the string 'US'. No other statements in the procedure are affected by this plan, and if the specified query occurs outside of the *Sales.GetOrdersByCountry* procedure, the plan guide is not invoked. (The companion Web site, which contains all the code used in all the book examples, also contains a script to build the *Sales.GetOrdersByCountry* procedure.)

**SQL Plan Guides** A plan guide of type *SQL* indicates you are interested in a particular SQL statement, either as a stand-alone statement, or in a particular batch. T-SQL statements that are sent to SQL Server by CLR objects or extended stored procedures, or that are part of dynamic SQL invoked with the EXEC (*sql_string*) construct, are processed as batches on SQL Server. To use them in a plan guide, their type should be set to *SQL.* For a stand-alone statement, the *@module_or_batch* parameter to *sp_create_plan_guide* should be set to NULL*,* so that SQL Server assumes that the batch and the statement have the same value. If the statement you are interested in is in a larger batch, the entire batch text needs to be specified in the *@module_or_batch* parameter. If a batch is specified for a SQL plan guide, the text of the batch needs to be exactly the same as it appears in the application. The rules aren't quite as strict as those for adhoc query plan reuse, discussed earlier in this chapter, but they are close. Make sure you use the same case, the same whitespace, and the other characteristics that your application does.

Here is an example of a plan guide that tells SQL Server to use only one CPU (no parallelization) when a particular query is executed as a stand-alone query:

```
EXEC sp_create_plan_guide
@name = N'plan_SalesOrderHeader_DOP1',
@stmt = N'SELECT TOP 10 *
          FROM Sales.SalesOrderHeader
          ORDER BY OrderDate DESC',
@type = N'SQL',
@module_or_batch = NULL,
@params = NULL,
@hints = N'OPTION (MAXDOP 1)';
```

Once this plan is created in the *AdventureWorks2008* database, every time the specified statement is encountered in a batch by itself, it has a plan created that uses only a single CPU. If the specified query occurs as part of a larger batch, the plan guide is not invoked.

**Template Plan Guides** A plan guide of type *Template* can use only the PARAMETERIZATION FORCED or PARAMETERIZATION SIMPLE hints to override the PARAMETERIZATION database setting. Template guides are a bit trickier to work with because you have to have SQL Server construct a template of your query in the same format that it will be in once it is parameterized. This isn't hard because SQL Server supplies us with a special procedure called *sp_get_query_ template*, but to use template guides, you need to perform several prerequisite steps. If you take a look at the two plan guide examples given previously, you see that the parameter called *@params* was NULL for both OBJECT and SQL plan guides. You only specify a value for *@params* with a TEMPLATE plan guide.

To see an example of using a template guide and forcing parameterization, first clear your procedure cache, and then execute these two queries in the *AdventureWorks2008* database:

```
DBCC FREEPROCCACHE;
```

```
GO
SELECT * FROM AdventureWorks2008.Sales.SalesOrderHeader AS h
INNER JOIN AdventureWorks2008.Sales.SalesOrderDetail AS d
   ON h.SalesOrderID = d.SalesOrderID
WHERE h.SalesOrderID = 45639;
GO
SELECT * FROM AdventureWorks2008.Sales.SalesOrderHeader AS h
INNER JOIN AdventureWorks2008.Sales.SalesOrderDetail AS d
   ON h.SalesOrderID = d.SalesOrderID
WHERE h.SalesOrderID = 45640;
```

These queries are very similar, and the plans for both are identical, but because the query is considered too complex, SQL Server does not autoparameterize them. If, after executing both queries, you look at the plan cache, you see only adhoc queries. If you've created the *sp_cacheobjects* view described earlier in the chapter, you could use that; otherwise, replace *sp_cacheobjects* with *sys.syscacheobjects:*

```
SELECT objtype, dbid, usecounts, sql
FROM sp_cacheobjects
WHERE cacheobjtype = 'Compiled Plan';
```

To create a plan guide to force statements of this type to be parameterized, we first need to call the procedure *sp_get_query_template* and pass two variables as output parameters. One parameter holds the parameterized version of the query, and the other holds the parameter list and the parameter data types. The following code then *SELECT*s these two output parameters so you can see their contents. Of course, you can remove this *SELECT* from your own code. Finally, we call the *sp_create_plan_guide* procedure, which instructs the optimizer to use PARAMETERIZATION FORCED anytime it sees a query that matches this specific template. In other words, anytime a query that parameterizes to the same form as the query here, it uses the same plan already cached:

```
DECLARE @sample_statement nvarchar(max);
DECLARE @paramlist nvarchar(max);
EXEC sp_get_query_template
   N'SELECT * FROM AdventureWorks2008.Sales.SalesOrderHeader AS h
     INNER JOIN AdventureWorks2008.Sales.SalesOrderDetail AS d
        ON h.SalesOrderID = d.SalesOrderID
     WHERE h.SalesOrderID = 45639;',
   @sample_statement OUTPUT,
   @paramlist OUTPUT
SELECT @paramlist as parameters, @sample_statement as statement
EXEC sp_create_plan_guide @name = N'Template_Plan',
   @stmt = @sample_statement,
   @type = N'TEMPLATE',
   @module_or_batch = NULL,
   @params = @paramlist,
   @hints = N'OPTION(PARAMETERIZATION FORCED)';
```

After creating the plan guide, run the same two statements as shown previously, and then examine the plan cache:

```
DBCC FREEPROCCACHE;
GO
SELECT * FROM AdventureWorks2008.Sales.SalesOrderHeader AS h
INNER JOIN AdventureWorks2008.Sales.SalesOrderDetail AS d
   ON h.SalesOrderID = d.SalesOrderID
WHERE h.SalesOrderID = 45639;
GO
SELECT * FROM AdventureWorks2008.Sales.SalesOrderHeader AS h
INNER JOIN AdventureWorks2008.Sales.SalesOrderDetail AS d
   ON h.SalesOrderID = d.SalesOrderID
WHERE h.SalesOrderID = 45640;
GO
SELECT objtype, dbid, usecounts, sql
FROM sp_cacheobjects
WHERE cacheobjtype = 'Compiled Plan';
```

You should now see a prepared plan with the following parameterized form:

```
(@0 int)select * from AdventureWorks2008.Sales.SalesOrderHeader as h
   inner join AdventureWorks2008.Sales.SalesOrderDetail as d
   on h.SalesOrderID = d.SalesOrderID
   where h.SalesOrderID = @0
```

**Managing Plan Guides**

In addition to the *sp_create_plan_guide* and *sp_get_query_template* procedures, the other basic procedure for working with plan guides is *sp_control_plan_guide*. This procedure allows you to DROP, DISABLE, or ENABLE a plan guide using the following basic syntax:

```
sp_control_plan_guide '<control_option>' [, '<plan_guide_name>']
```

There are six possible *control_option* values: DISABLE, DISABLE ALL, ENABLE, ENABLE ALL, DROP, and DROP ALL. The *plan_guide_name* parameter is optional because with any of the *ALL control_option* values, no *plan_guide_name* value is supplied. Plan guides are local to a particular database, so the DISABLE ALL, ENABLE ALL, and DROP ALL values apply to all plan guides for the current database. In addition, plan guides behave like schema-bound views in a way; the stored procedures, triggers, and functions referred to in any Object plan guide in a database cannot be altered or dropped. So for our example Object plan guide, so long as the plan guide exists, the *AdventureWorks2008.Sales.GetOrdersByCountry* procedure cannot be altered or dropped. This is true whether the plan guide is disabled or enabled, and it remains true until all plan guides referencing those objects are dropped with *sp_control_plan_guide*.

The metadata view that contains information about plan guides in a particular database is *sys.plan_guides*. This view contains all the information supplied in the *sp_create_plan_guide* procedure plus additional information such as the creation date and last modification date of each plan guide. Using the information in this view, you can reconstruct the plan guide definition manually if necessary. In addition, Management Studio allows you to script your plan guide definitions from the Object Explorer tree.

**Plan Guide Considerations**

For SQL Server to determine that there is an appropriate plan guide to use, the statement text in the plan guide must match the query being compiled. This must be an exact character-for-character match, including case, whitespace, and comments, just as when SQL Server is determining whether it can reuse adhoc query plans, as we discussed earlier in the chapter. If your statement text is close, but not quite an exact match, this can lead to a situation that is very difficult to troubleshoot. When matching a SQL template, whether the definition also contains a batch that the statement must be part of, SQL Server does allow more leeway in the definition of the batch. In particular, keyword case, whitespace, and comments are ignored.

To make sure your plan guides use the exact text that is submitted by your applications, you can run a trace using SQL Server Profiler and capture the *SQL:BatchCompleted* and *RPC:Completed* events. After the relevant batch (the one you want to create a plan guide for) shows up in the top window of your Profiler output, you can right-click the event and select Extract Event Data to save the SQL Text of the batch to a text file. It is not sufficient to copy and paste from the lower window in the Profiler because the output there can introduce extra line breaks.

To verify that your plan guide was used, you can look at the XML plan for the query. If you can run the query directly, you can use the option SET SHOWPLAN_XML ON, or you can capture the showplan XML through a trace. An XML plan has two specific items, indicating that the query used a plan guide. These items are *PlanGuideDB* and *PlanGuideName*. If the plan guide was a template plan guide, the XML plan also has the items *TemplatePlanGuideDB* and *TemplatePlanGuideName*.

When a query is submitted for processing, if there are any plan guides in the database at all, SQL Server first checks to see if the statement matches a SQL plan guide or Object plan guide. The query string is hashed to make it faster to find any matching strings in the database's existing plan guides. If no matching SQL or Object plan guides are found, SQL Server then checks for a TEMPLATE plan guide. If it finds a TEMPLATE guide, it then tries to match the resulting parameterized query to a SQL plan guide. This gives you the possibility of applying additional hints to your queries using forced parameterization. Figure 9-3, copied from *SQL Server Books Online,* shows the process that SQL Server uses to check for applicable plan guides.

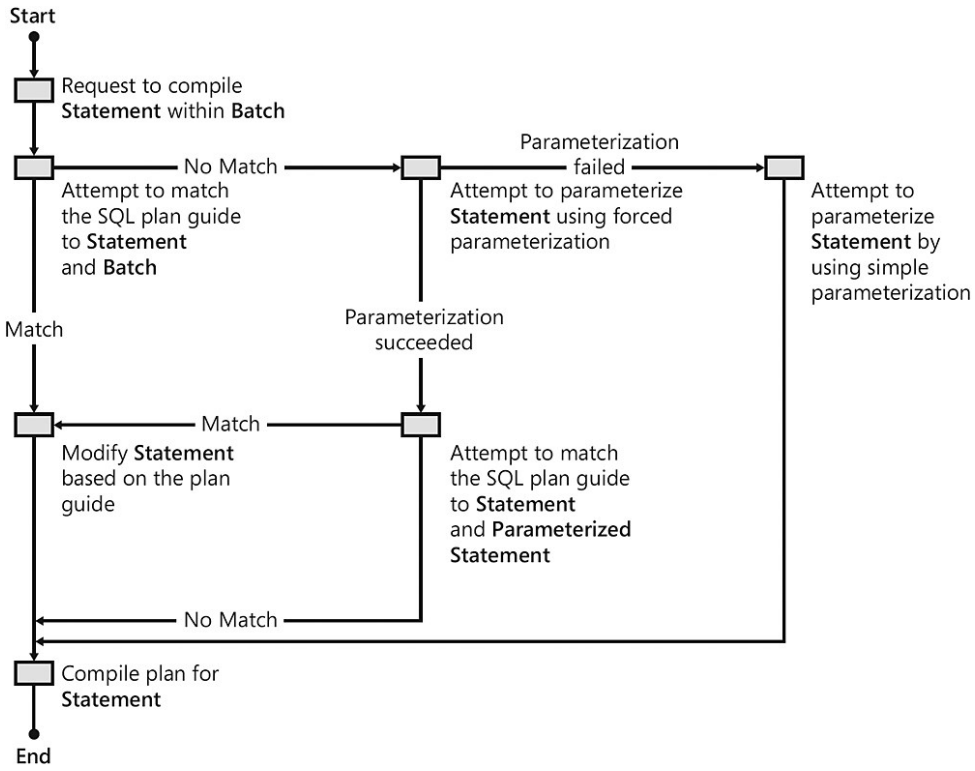


**Figure 9-3:** Checking for applicable plan guides

The key steps are the following, which follow the flowchart from the top left, take the top branch to the right, the middle branch down, and then right at the center, to the point where the statement is modified based on the plan guide and its hints:

1. For a specific statement within the batch, SQL Server tries to match the statement to a SQL-based plan guide, whose *@module_or_batch* argument matches that of the incoming batch text, including any constant literal values, and whose *@stmt* argument also matches the statement in the batch. If this kind of plan guide exists and the match succeeds, the statement text is modified to include the query hints specified in the plan guide. The statement is then compiled using the specified hints.

2. If a plan guide is not matched to the statement in step 1, SQL Server tries to parameterize the statement by using forced parameterization. In this step, parameterization can fail for any one of the following reasons :

   - The statement is already parameterized or contains local variables.
   - The PARAMETERIZATION SIMPLE database SET option is applied (the default setting), and there is no plan guide of type TEMPLATE that applies to the statement and specifies the PARAMETERIZATION FORCED query hint.
   - A plan guide of type TEMPLATE exists that applies to the statement and specifies the PARAMETERIZATION SIMPLE query hint.

Let’s look at an example that involves the distribution of data in the *SpecialOfferID* column in the *Sales.SalesOrderDetail* table in the *AdventureWorks2008* database. There are 12 different *SpecialOfferID* values, and most of them occur only a few hundred times (out of the 121317 rows in the *Sales.SalesOrderDetail*) at most, as the following script and output illustrates:

```
USE AdventureWorks2008
GO
SELECT SpecialOfferID, COUNT(*) as Total
FROM Sales.SalesOrderDetail
GROUP BY SpecialOfferID;

RESULTS:
SpecialOfferID  Total
--------------  -----------
1               115884
```

```
2               3428
3               606
4               80
5               2
7               137
8               98
9               61
11              84
13              524
14              244
16              169
```

As there are 1238 pages in the table, for most of the values, a nonclustered index on *SpecialOfferID* could be useful, so here is the code to build one:

```
CREATE INDEX Detail_SpecialOfferIndex ON Sales.SalesOrderDetail(SpecialOfferID);
```

We assume that very few queries actually search for a *SpecialOfferID* value of 1 or 2, and 99 percent of the time the queries are looking for the less popular values. We would like the Query Optimizer to autoparameterize queries that access the *Sales.SalesOrderDetail* table, specifying one particular value for *SpecialOfferID*. So we create a template plan guide to autoparameterize queries of this form:

```
SELECT * FROM Sales.SalesOrderDetail  WHERE SpecialOfferID = 4;
```

However, we want to make sure that the initial parameter that determines the plan is not one of the values that might use a Clustered Index scan, namely the values 1 or 2. So we can take the autoparameterized query produced by the *sp_get_query_template* procedure, and use it first to create a template plan guide, and then to create a SQL plan guide with the OPTIMIZE FOR hint. The hint forces SQL Server to assume a specific value of 4 every time the query needs to be reoptimized:

```
USE AdventureWorks2008;
-- Get plan template and create plan Guide
DECLARE @stmt nvarchar(max);
DECLARE @params nvarchar(max);
EXEC sp_get_query_template
   N'SELECT * FROM Sales.SalesOrderDetail WHERE SpecialOfferID = 4',
   @stmt OUTPUT,
   @params OUTPUT
--SELECT @stmt as statement -- show the value when debugging
--SELECT @params as parameters -- show the value when debugging

EXEC sp_create_plan_guide N'Template_Plan_for SpecialOfferID',
   @stmt,
   N'TEMPLATE',
   NULL,
   @params,
   N'OPTION (PARAMETERIZATION FORCED)';

EXEC sp_create_plan_guide
   @name = N'Force_Value_for_Prepared_Plan',
   @stmt = @stmt,
   @type = N'SQL',
   @module_or_batch = NULL,
   @params = @params,
   @hints = N'OPTION (OPTIMIZE FOR (@0 = 4))';
GO
```

You can verify that the plan is being autoparameterized and optimized for a value that uses a nonclustered index on SpecialOfferID by running a few tests as follows:

```
DBCC FREEPROCCACHE;
SET STATISTICS IO ON;
SELECT * FROM Sales.SalesOrderDetail
WHERE SpecialOfferID = 3;
GO
SELECT * FROM Sales.SalesOrderDetail
WHERE SpecialOfferID = 4;
GO
SELECT * FROM Sales.SalesOrderDetail
WHERE SpecialOfferID = 5;
```

```
GO
```

You should note in the STATISTICS IO output that each execution uses a different number of reads because it is finding a different number of rows through the nonclustered index. You can also verify that SQL Server is using the prepared plan by examining the STATISTICS XML output. If you set that option to ON, and run the query looking for a value of 5, you should have a node in your XML document that looks very much like this:

```
<ParameterList>
<ColumnReference Column="@0" ParameterCompiledValue="(4)"
           ParameterRuntimeValue="(5)" />
</ParameterList>
```

Plan guides are not intended to speed up query compilation time. Not only does SQL Server first have to determine if there is a plan guide that could be a potential match for the query being compiled, but the plan enforced by the plan guide has to be one that the Query Optimizer would have come up with on its own. To know that the forced plan is valid, the Query Optimizer has to go through most of the process of optimization. The benefit of plan guides is to reduce execution time for those queries in which the Query Optimizer is not coming up with the best plan on its own.

The main plan guide enhancements in SQL Server 2008 have to do with making plan guides more usable. SQL Server 2008 contains SMO and Management Studio support, including scripting of plan guides as part of scripting out a database. Once a plan guide is scripted, it can be copied to other SQL Server instances running the same queries.

**Plan Guide Validation**

One limitation of the SQL Server 2005 implementation of plan guides was that it was possible to change the physical design of a table (for example, dropping an index) in a way that could invalidate a plan guide and any queries using that plan guide would fail whenever they were executed. SQL Server 2008 can detect cases when changing the table design would break a plan guide. It can now recompile the query without the plan guide and to notify the administrator through trace events. In addition, there is a new system function that can be used to validate plan guides. This function can be used to detect physical database design changes that break existing plan guides and allow you to roll back the breaking transaction before it can break the system.

To validate all of the existing plan guides in a system, you can use the *sys.fn_validate_plan_guide* function:

```
SELECT * FROM sys.plan_guides pg
CROSS APPLY
(SELECT * FROM sys.fn_validate_plan_guide(pg.plan_guide_id)) v;
```

The function returns nothing for valid plan guides. When the guide would generate an error, it returns a row. So you can incorporate this into any schema changes in the system:

```
BEGIN TRANSACTION;
DROP INDEX t2.myindex;
IF EXISTS(
SELECT * FROM sys.plan_guides pg
CROSS APPLY
  (SELECT * FROM sys.fn_validate_plan_guide(pg.plan_guide_id)) v
)
ROLLBACK TRANSACTION
ELSE
COMMIT TRANSACTION;
```

**Freezing a Plan from Plan Cache**

SQL Server 2008 added a new stored procedure to allow you to create a plan guide automatically from a plan that has already been cached. The procedure *sp_create_plan_ guide_from_handle* requires a *plan_handle* and a plan guide name as parameters and creates a plan guide using the execution plan stored in cache for that *plan_handle* value. The capability is called *plan freezing* because it allows you to make sure that a well-performing plan is reused every time the associated query is executed. Suppose that we have found that the plan just executed for the following query performs extremely well, and we'd like to make sure that plan is the one used on subsequent executions:

```
SELECT City, StateProvinceID, PostalCode FROM Person.Address ORDER BY PostalCode DESC;
```

I can find the corresponding plan in cache by searching for a text value that matches the query:

```
SELECT plan_handle
FROM sys.dm_exec_query_stats AS qs CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st
WHERE st.text LIKE N'SELECT City,%';
```

Once I have that *plan_handle,* I can pass it as a parameter to the *sp_create_plan_guide_from_ handle* procedure as follows:

```
EXEC sp_create_plan_guide_from_handle
     @name = N'Guide1_from_XML_showplan',
     @plan_handle = 0x06000600F19B1E1FC0A14C0A000000000000000000000000
```

There are several situations in which plan guides and plan freezing can be particularly beneficial:

- You can use plan guides to provide a workaround for plan regressions after a server upgrade.
- You can disallow plan changes for critical plans in a well-performing system.
- You can troubleshoot a problematic query by freezing a good plan (assuming a good plan ever is used).
- ISVs can create known good plans for shipping with their applications.
- You can optimize on a test system and then port the plan guide to your production system.
- You can include plan guides in a cloned database.

## Summary

For all the caching mechanisms, reusing a cached plan avoids recompilation and optimization. This saves compilation time, but it means that the same plan is used regardless of the particular parameter values passed in. If the optimal plan for a given parameter value is not the same as the cached plan, the optimal execution time is not achieved. For this reason, SQL Server is very conservative about autoparameterization. When an application uses *sp_executesql*, prepare and execute, or stored procedures, the application developer is responsible for determining what should be parameterized. You should parameterize only constants whose range of values does not drastically affect the optimization choices.

In this chapter, we looked at the caching and reuse of plans generated by the Query Optimizer. SQL Server can cache and reuse plans not only from stored procedures, but also from adhoc and autoparameterized queries. Because generating query plans can be expensive, it helps to understand how and why query plans are reused and when they must be regenerated. Understanding how caching and reusing plans work helps you determine when using the cached plan can be the right choice and when you might need to make sure SQL Server comes up with a new plan to give your queries and applications the best performance.