Slow in the Application, Fast in SSMS?
Understanding Performance Mysteries

An SQL text by Erland Sommarskog, SQL Server MVP. Last revision: 2019-10-26.

1. Introduction

When I read various forums about SQL Server, I frequently see questions from deeply mystified posters. They have identified a slow query or slow stored procedure in their application. They take the SQL batch from the application and run it in SQL Server Management Studio (SSMS) to analyse it, only to find that the response is instantaneous. At this point they are inclined to think that SQL Server is all about magic. A similar mystery is when a developer has extracted a query in his stored procedure to run it stand-alone only to find that it runs much faster – or much slower – than inside the procedure.

No, SQL Server is not about magic. But if you don't have a good understanding of how SQL Server compiles queries and maintains its plan cache, it may seem so. Furthermore, there are some unfortunate combinations of different defaults in different environments. In this article, I will try to straighten out why you get this seemingly inconsistent behaviour. I explain how SQL Server compiles a stored procedure, what *parameter sniffing* is and why it is part of the equation in the vast majority of these confusing situations. I explain how SQL Server uses the cache, and why there may be multiple entries for a procedure in the cache. Once you have come this far, you will understand how come the query runs so much faster in SSMS.

To understand how to address that performance problem in your application, you need to read on. I first make a short break from the theme of parameter sniffing to discuss a few situations where there are other reasons for the difference in performance. This is followed by two chapters how to deal with performance problems where parameter sniffing is involved. The first is about gathering information. In the second chapter I discuss some scenarios – both real-world situations and more generic ones – and possible solutions. Next comes a chapter where I discuss how dynamic SQL is compiled and interacts with the plan cache and why there are more reasons you may experience differences in performance between SSMS and the application with dynamic SQL. In the last chapter I look at how you can use the Query Store that was introduced in SQL 2016 for your troubleshooting. At the end there is a section with links to Microsoft white papers and similar documents in this area.

Table of Contents

1.1 Presumptions

The essence of this article applies to all versions of SQL Server, but the focus is on SQL 2005 and later versions. The article includes several queries to inspect the plan cache; these queries run only on SQL 2005 and later. SQL 2000 and earlier versions had far less instrumentation in this regard. Beware that to run these queries you need to have the server-level permission VIEW SERVER STATE.

For the examples in this article, I use the **Northwind** sample database. This database shipped with SQL 2000. For later versions of SQL Server you can download it from Microsoft's web site.

This is not a beginner-level article, but I assume that the reader has a working experience of SQL programming. You don't need to have any prior experience of performance tuning, but it certainly helps if you have looked a little at query plans and if you have some basic knowledge of indexes. I will not explain the basics in depth, as my focus is a little beyond that point. This article will not teach you everything about performance tuning, but at least it will be a start.

Caveat: In some places, I give links to the online version of Books Online. Beware that the URL may lead to Books Online for a different version of SQL Server than you are using. In the top left there is a drop-down where you can select a different SQL Server version. Or more precisely, as of this writing there is – they tend to redesign the pages every now and then, and it is difficult for me to keep up.

2. How SQL Server Compiles a Stored Procedure

In this chapter we will look at how SQL Server compiles a stored procedure and uses the plan cache. If your application does not use stored procedures, but submits SQL statements directly, most of what I say this chapter is still applicable. But there are further complications with dynamic SQL, and since the facts about stored procedures are confusing enough I have deferred the discussion on dynamic SQL to a separate chapter.

2.1 What is a Stored Procedure?

That may seem like a silly question, but the question I am getting at is *What objects have query plans on their own?* SQL Server builds query plans for these types of objects:

- Stored procedures.
- Scalar user-defined functions.
- Multi-step table-valued functions.
- Triggers.

With a more general and stringent terminology I should talk about *modules*, but since stored procedures is by far the most widely used type of module, I prefer to talk about stored procedures to keep it simple.

For other types of objects than the four listed above, SQL Server does not build query plans. Specifically, SQL Server does not create query plans for views and inline-table functions. Queries like:

```
SELECT abc, def FROM myview
SELECT a, b, c FROM mytablefunc(9)
```

are no different from ad-hoc queries that access the tables directly. When compiling the query, SQL Server expands the view/function into the query, and the optimizer works with the expanded query text.

There is one more thing we need to understand about what constitutes a stored procedure. Say that you have two procedures, where the outer calls the inner one:

```
CREATE PROCECURE Outer_sp AS
...
EXEC Inner_sp
...
```

I would guess most people think of **Inner_sp** as being independent from **Outer_sp**, and indeed it is. The execution plan for **Outer_sp** does not include the query plan for **Inner_sp**, only the invocation of it. However, there is a very similar situation where I've noticed that posters on SQL forums often have a different mental image, to wit dynamic SQL:

```
CREATE PROCEDURE Some_sp AS
DECLARE @sql    nvarchar(MAX),
        @params nvarchar(MAX)
SELECT @sql = 'SELECT ...'
...
EXEC sp_executesql @sql, @params, @par1, ...
```

It is important to understand that this is no different from nested stored procedures. The generated SQL string is **not** part of **Some_sp**, nor does it appear anywhere in the query plan for **Some_sp**, but it has a query plan and a cache entry of its own. This applies, no matter if the dynamic SQL is executed through EXEC() or **sp_executesql**.

2.2 How SQL Server Generates the Query Plan

Overview

When you enter a stored procedure with CREATE PROCEDURE (or CREATE FUNCTION for a function or CREATE TRIGGER for a trigger), SQL Server verifies that the code is syntactically correct, and also checks that you do not refer to non-existing columns. (But if you refer to non-existing tables, it lets get you away with it, due to a misfeature known as deferred named resolution.) However, at this point SQL Server does **not** build any query plan, but merely stores the query text in the database.

It is not until a user executes the procedure, that SQL Server creates the plan. For each query, SQL Server looks at the distribution statistics it has collected about the data in the tables in the query. From this, it makes an estimate of what may be best way to execute the query. This phase is known as *optimisation*. While the procedure is compiled in one go, each query is optimised on its own, and there is no attempt to analyse the flow of execution. This has a very important ramification: the optimizer has no idea about the run-time values of variables. However, it does know what values the user specified for the parameters to the procedure.

Parameters and Variables

Consider the **Orders** table in the **Northwind** database, and these three procedures:

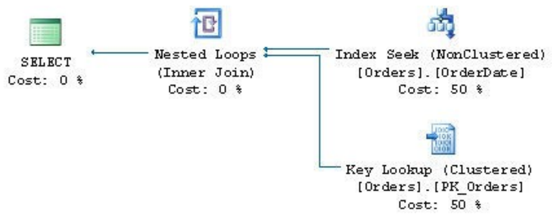
```
CREATE PROCEDURE List_orders_1 AS
   SELECT * FROM Orders WHERE OrderDate > '20000101'
go
CREATE PROCEDURE List_orders_2 @fromdate datetime AS
   SELECT * FROM Orders WHERE OrderDate > @fromdate
go
CREATE PROCEDURE List_orders_3 @fromdate datetime AS
   DECLARE @fromdate_copy datetime
   SELECT @fromdate_copy = @fromdate
   SELECT * FROM Orders WHERE OrderDate > @fromdate_copy
go
```

Note: Using SELECT * in production code is bad practice. I use it in this article to keep the examples concise.

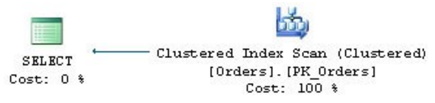
Then we execute the procedures in this way:

```
   EXEC List_orders_1
   EXEC List_orders_2 '20000101'
   EXEC List_orders_3 '20000101'
```

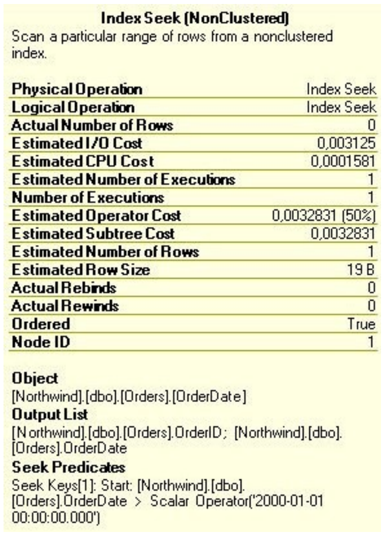
Before you run the procedures, enable *Include Actual Execution Plan* under the *Query* menu. (There is also a toolbar button and Ctrl-M is the normal keyboard shortcut.) If you look at the query plans for the procedures, you will see the first two procedures have identical plans:



That is, SQL Server seeks the index on **OrderDate**, and uses a key lookup to get the other data. The plan for the third execution is different:



In this case, SQL Server scans the table. (Keep in mind that in a clustered index the leaf pages contain the data, so a clustered index scan and a table scan is essentially the same the thing.) Why this difference? To understand why the optimizer makes certain decisions, it is always a good idea to look at what estimates it is working with. If you hover with the mouse over the two Seek operators and the Scan operator, you will see the pop-ups similar to those below.

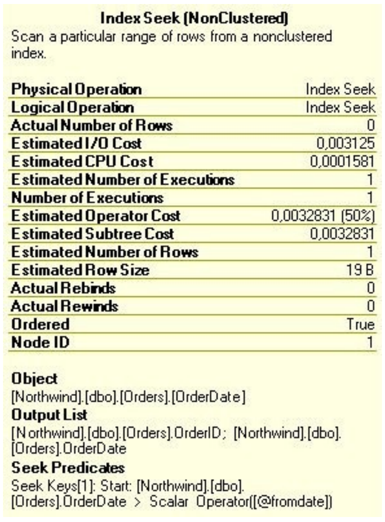


Index Seek (NonClustered)
Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Number of Rows	0
Estimated I/O Cost	0,003125
Estimated CPU Cost	0,0001581
Estimated Number of Executions	1
Number of Executions	1
Estimated Operator Cost	0,0032831 (50%)
Estimated Subtree Cost	0,0032831
Estimated Number of Rows	1
Estimated Row Size	19 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	1

Object
[Northwind].[dbo].[Orders].[OrderDate]
Output List
[Northwind].[dbo].[Orders].OrderID; [Northwind].[dbo].[Orders].OrderDate
Seek Predicates
Seek Keys[1]: Start: [Northwind].[dbo].[Orders].OrderDate > Scalar Operator('2000-01-01 00:00:00.000')

List_orders_1

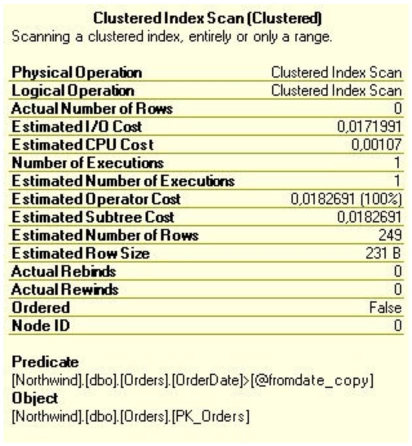


Index Seek (NonClustered)
Scan a particular range of rows from a nonclustered index.

Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Number of Rows	0
Estimated I/O Cost	0,003125
Estimated CPU Cost	0,0001581
Estimated Number of Executions	1
Number of Executions	1
Estimated Operator Cost	0,0032831 (50%)
Estimated Subtree Cost	0,0032831
Estimated Number of Rows	1
Estimated Row Size	19 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	1

Object
[Northwind].[dbo].[Orders].[OrderDate]
Output List
[Northwind].[dbo].[Orders].OrderID; [Northwind].[dbo].[Orders].OrderDate
Seek Predicates
Seek Keys[1]: Start: [Northwind].[dbo].[Orders].OrderDate > Scalar Operator([@fromdate])

List_orders_2



Clustered Index Scan (Clustered)
Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Number of Rows	0
Estimated I/O Cost	0,0171991
Estimated CPU Cost	0,00107
Number of Executions	1
Estimated Number of Executions	1
Estimated Operator Cost	0,0182691 (100%)
Estimated Subtree Cost	0,0182691
Estimated Number of Rows	249
Estimated Row Size	231 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0

Predicate
[Northwind].[dbo].[Orders].[OrderDate]>[@fromdate_copy]
Object
[Northwind].[dbo].[Orders].[PK_Orders]

List_orders_3

Note: the exact appearance depends on which version of SSMS and SQL Server you are using. The samples above were captured with SSMS 2008 running against SQL 2008. If you use a later version of SSMS, you may see the items in a somewhat different order, and there are also more items listed. None of those extra items are relevant for the example, which is why I have preferred to keep my original screen shots.

The interesting element is *Estimated Number of Rows*. For the first two procedures, SQL Server estimates that one row will be returned, but for **List_orders_3**, the estimate is 249 rows. This difference in estimates explains the different choice of plans. Index Seek + Key Lookup is a good strategy to return a smaller amount of rows from a table. But if more rows match the seek criteria, the cost increases, and there is a increased likelihood that SQL Server will need to access the same data page more than once. In the extreme case where all rows are returned, a table scan is much more efficient than seek and lookup. With a scan, SQL Server has to read every data page exactly once, whereas with seek + key lookup, every page will be visited once for each row on the page. The **Orders** table in **Northwind** has 830 rows, and when SQL Server estimates that as many as 249 rows will be returned, it (rightly) concludes that the scan is the best choice.

Where Do These Estimates Come From?

Now we know why the optimizer arrives at different execution plans: because the estimates are different. But that only leads to the next question: why are the estimates different? That is the key topic of this article.

In the first procedure, the date is a constant, which means that the SQL Server only needs to consider exactly this case. It interrogates the statistics for the **Orders** table, which indicates that there are no rows with an **OrderDate** in the third millennium. (All orders in the **Northwind** database are from 1996 to 1998.) Since statistics are statistics, SQL Server cannot be sure that the query will return no rows at all, why it makes an estimate of one single row.

In the case of **List_orders_2**, the query is against a variable, or more precisely a parameter. When performing the optimisation, SQL Server knows that the procedure was invoked with the value 2000-01-01. Since it does not any perform flow analysis, it can't say for sure whether the parameter will have this value when the query is executed. Nevertheless, it uses the input value to come up with an estimate, which is the same as for **List_orders_1**: one single row. This strategy of looking at the values of the input parameters when optimising a stored procedure is known as *parameter sniffing*.

In the last procedure, it's all different. The input value is copied to a local variable, but when SQL Server builds the plan, it has no understanding of this and says to itself *I don't know what the value of this variable will be*. Because of this, it applies a standard assumption, which for an inequality operation such as > is a 30 % hit-rate. 30 % of 830 is indeed 249.

Here is a variation of the theme:

```
CREATE PROCEDURE List_orders_4 @fromdate datetime = NULL AS
   IF @fromdate IS NULL
      SELECT @fromdate = '19900101'
   SELECT * FROM Orders WHERE OrderDate > @fromdate
```

In this procedure, the parameter is optional, and if the user does not fill in the parameter, all orders are listed. Say that the user invokes the procedure as:

```
EXEC List_orders_4
```

The execution plan is identical to the plan for **List_orders_1** and **List_orders_2**. That is, Index Seek + Key Lookup, despite that all orders are returned. If you look at the pop-up for the Index Seek operator, you will see that it is identical to the pop-up for **List_orders_2** but in one regard, the actual number of rows. When compiling the procedure, SQL Server does not know that the value of **@fromdate** changes, but compiles the procedure under the assumption that **@fromdate** has the value NULL. Since all comparisons with NULL yield UNKNOWN, the query cannot return any rows at all, if **@fromdate** still has this value at run-time. If SQL Server would take the input value as the final truth, it could construct a plan with only a Constant Scan that does not access the table at all (run the query `SELECT * FROM Orders WHERE OrderDate > NULL` to see an example of this). But SQL Server must generate a plan which returns the correct result no matter what value **@fromdate** has at run-time. On the other hand, there is no obligation to build a plan which is the best for all values. Thus, since the assumption is that no rows will be returned, SQL Server settles for the Index Seek. (The estimate is still that one row will be returned. This is because SQL Server never uses an estimate of 0 rows.)

This is an example of when parameter sniffing backfires, and in this particular case it may be better to write the procedure in this way:

```
CREATE PROCEDURE List_orders_5 @fromdate datetime = NULL AS
   DECLARE @fromdate_copy datetime
   SELECT @fromdate_copy  = coalesce(@fromdate, '19900101')
   SELECT * FROM Orders WHERE OrderDate > @fromdate_copy
```

With **List_orders_5** you always get a Clustered Index Scan.

Key Points

In this section, we have learned three very important things:

- A constant is a constant, and when a query includes a constant, SQL Server can use the value of the constant with full trust, and even take such shortcuts to not access a table at all, if it can infer from constraints that no rows will be returned.
- For a parameter, SQL Server does not know the run-time value, but it "sniffs" the input value when compiling the query.
- For a local variable, SQL Server has no idea at all of the run-time value, and applies standard assumptions. (Which the assumptions are depends on the operator and what can be deduced from the presence of unique indexes.)

And there is a corollary of this: if you take out a query from a stored procedure and replace variables and parameters with constants, you now have quite a different query. More about this later.

2.3 Putting the Query Plan into the Cache

If SQL Server would compile a stored procedure – that is, optimise and build a query plan – every time the procedure is executed, there is a big risk that SQL Server would crumble from all the CPU resources it would take. I immediately need to qualify this, because it is not true for all systems. In a big data warehouse where a handful of business analysts runs complicated queries that take a minute on average to execute, there would be no damage if there was compilation every time – rather it could be beneficial. But in an OLTP database where plenty of users run stored procedures with short and simple queries, this concern is very much for real.

For this reason, SQL Server caches the query plan for a stored procedure, so when the next user runs the procedure, the compilation phase can be skipped, and execution can commence directly. The plan will stay in the cache, until some event forces the plan out of the cache. Examples of such events are:

- SQL Server's buffer cache is fully utilised, and SQL Server needs to age out buffers that have not been used for some time from the cache. The buffer cache includes table data as well as query plans.
- Someone runs ALTER PROCEDURE on the procedure.
- Someone runs **sp_recompile** on the procedure.
- Someone runs the command DBCC FREEPROCCACHE which clears the entire plan cache.
- SQL Server is restarted. Since the cache is memory-only, the cache is not preserved over restarts.
- Changing of certain configuration parameters (with **sp_configure** or through the Server Properties pages in SSMS) evicts the entire plan cache.

If such an event occurs, a new query plan will be created the next time the procedure is executed. SQL Server will anew "sniff" the input parameters, and if the parameter values are different this time, the new query plan may be different from the previous plan.

There are other events that do not cause the entire procedure plan to be evicted from the cache, but which trigger recompilation of one or more individual statements in the procedure. The recompilation occurs the next time the statement is executed. This applies even if the event occurred after the procedure started executing. Here are examples of such events:

- Changing the definition of a table that appears in the statement.
- Dropping or adding an index for a table appearing in the statement. This includes rebuilding an index with ALTER INDEX or DBCC DBREINDEX.
- New or updated statistics for a table in the statement. Statistics can be created and updated by SQL Server automatically. The DBA can also create and update statistics with the commands CREATE STATISTICS and UPDATE STATISTICS. However, changed statistics do not always cause recompilation. The basic rule is that there should have been a change in the data for recompilation to be triggered. See this blog post from Kimberly Tripp for more details.
- Someone runs **sp_recompile** on a table referred to in the statement.

Note: In SQL Server 2000, there is no statement recompilation, but the entire procedure is always recompiled.

These lists are by no means exhaustive, but you should observe one thing which is **not** there: executing the procedure with different values for the input parameters from the original execution. That is, if the second invocation of **List_orders_2** is:

```
EXEC List_orders_2 '19900101'
```

The execution will still use the index on **OrderDate**, despite the query now retrieves all orders. This leads to a very important observation: the parameter values of the first execution of the procedure have a huge impact for subsequent executions. If this first set of values for some reason is atypical, the cached plan may not be optimal for future executions. This is why parameter sniffing is such a big deal.

Note: for a complete list of what can cause plans to be flushed or statements to be recompiled, see the white paper on Plan Caching listed in the *Further Reading* section.

2.4 Different Plans for Different Settings

There is a plan for the procedure in the cache. That means that everyone can use it, or? No, in this section we will learn that there can be multiple plans for the same procedure in the cache. To understand this, let's consider this contrived example:

```
CREATE PROCEDURE List_orders_6 AS
   SELECT *
   FROM   Orders
   WHERE  OrderDate > '12/01/1998'
go
SET DATEFORMAT dmy
go
EXEC List_orders_6
go
SET DATEFORMAT mdy
go
EXEC List_orders_6
go
```

If you run this, you will notice that the first execution returns many orders, whereas the second execution returns no orders. And if you look at the execution plans, you will see that they are different as well. For the first execution, the plan is a Clustered Index Scan (which is the best choice with so many rows returned), whereas the second execution plan uses Index Seek with Key Lookup (which is the best when no rows are returned).

How could this happen? Did SET DATEFORMAT cause recompilation? No, that would not be smart. In this example, the executions come one after each other, but they could just as well be submitted in parallel by different users with different settings for the date format. Keep in mind that the entry for a stored procedure in the plan cache is not tied to a certain session or user, but it is global to all connected users.

Instead the answer is that SQL Server creates a second cache entry for the second execution of the procedure. We can see this if we peek into the plan cache with this query:

```
SELECT qs.plan_handle, a.attrlist
FROM   sys.dm_exec_query_stats qs
CROSS  APPLY sys.dm_exec_sql_text(qs.sql_handle) est
CROSS  APPLY (SELECT epa.attribute + '=' + convert(nvarchar(127), epa.value) + '   '
              FROM   sys.dm_exec_plan_attributes(qs.plan_handle) epa
              WHERE  epa.is_cache_key = 1
              ORDER  BY epa.attribute
              FOR    XML PATH('')) AS a(attrlist)
WHERE  est.objectid = object_id ('dbo.List_orders_6')
  AND  est.dbid     = db_id('Northwind')
```

Reminder: You need the server-level permission VIEW SERVER STATE to run queries against the plan cache.

The DMV (Dynamic Management View) **sys.dm_exec_query_stats** has one entry for each query currently in the plan cache. If a procedure has multiple statements, there is one row per statement. Of interest here is **sql_handle** and **plan_handle**. I use **sql_handle** to determine which procedure the cache entry relates to (later we will see examples where we also retrieve the query text) so that we can filter out all other entries in the cache. Most often you use **plan_handle** to retrieve the query plan itself, and we will see an example of this later, but in this query I access a DMV that returns the attributes of the query plan. More specifically, I return the attributes that are *cache keys*. When there is more than one entry in the cache for the same procedure, the entries have at least one difference in the cache keys. A cache key is a run-time setting, which for one reason or another calls for a different query plan. Most of these settings are controlled with a SET command, but not all.

The query above returns two rows, indicating that there are two entries for the procedure in the cache. The output may look like this:

```
plan_handle                       attrlist
--------------------------------- -------------------------------------------------
0x0500070064EFCA5DB8A0A90500... compat_level=100   date_first=7   date_format=1
                                  set_options=4347   user_id=1
0x0500070064EFCA5DB8A0A80500... compat_level=100   date_first=7   date_format=2
                                  set_options=4347   user_id=1
```

To save space, I have abbreviated the plan handles and deleted many of the values in the **attrlist** column. I have also folded that column into two lines. If you run the query yourself, you can see the complete list of cache keys, and they are quite a few of them. If you look up the topic for **sys.dm_exec_plan_attributes** in Books Online, you will see description for many of the plan attributes, but you will also note that far from all cache keys are documented. In this article, I will not dive into all cache keys, not even the documented ones, but focus only on the most important ones.

As I said, the example is contrived, but it gives a good illustration to why the query plans must be different: different date formats may yield different results. A somewhat more normal example is this:

```
EXEC sp_recompile List_orders_2
go
SET DATEFORMAT dmy
go
EXEC List_orders_2 '12/01/1998'
go
SET DATEFORMAT mdy
go
EXEC List_orders_2 '12/01/1998'
go
```

(The initial **sp_recompile** is to make sure that the plan from the previous example is flushed.) This example yields the same results and the same plans as with **List_orders_6** above. That is, the two query plans use the actual parameter value when the respective plan is built. The first query uses 12 Jan 1998, and the second 1 Dec 1998.

A very important cache key is **set_options**. This is a bit mask that gives the setting of a number of SET options that can be ON or OFF. If you look further in the topic of **sys.dm_exec_plan_attributes**, you find a listing that details which SET option each bit describes. (You will also see that there are a few more items that are not controlled by the SET command.) Thus, if two connections have any of these options set differently, the connections will use different cache entries for the same procedure – and therefore they could be using different query plans, with possibly big difference in performance.

One way to translate the **set_options** attribute is to run this query:

```
SELECT convert(binary(4), 4347)
```

This tells us that the hex value for 4347 is 0x10FB. Then we can look in Books Online and follow the table to find out that the following SET options are in force: ANSI_PADDING, Parallel Plan, CONCAT_NULL_YIELDS_NULL, ANSI_WARNINGS, ANSI_NULLS, QUOTED_IDENTIFIER, ANSI_NULL_DFLT_ON and ARITHABORT.

You can also use this table-valued function that I have written and run:

```
SELECT Set_option FROM setoptions (4347) ORDER BY Set_option
```

Note: You may be wondering what *Parallel Plan* is doing here, not the least since the plan in the example is not parallel. When SQL Server builds a parallel plan for a query, it may later also build a non-parallel plan if the CPU load in the server is such that it is not defensible to run a parallel plan. It seems that for a plan that is always serial that the bit for parallel plan is nevertheless set in **set_options**.

To simplify the discussion, we can say that each of these SET options – ANSI_PADDING, ANSI_NULLS etc – is a cache key on its own. The fact that they are added together in a singular numeric value is just a matter of packaging.

2.5 The Default Settings

About all of the SET ON/OFF options that are cache keys exist because of legacy reasons. Originally, in the dim and distant past, SQL Server included a number of behaviours that violated the ANSI standard for SQL. With SQL Server 6.5, Microsoft introduced all these SET options (save for ARITHABORT, which was in the product already in 4.x), to permit users to use SQL Server in an ANSI-compliant way. In SQL 6.5, you had to use the SET options explicitly to get ANSI compliance, but with SQL 7, Microsoft changed the defaults for clients that used the new versions of the ODBC and OLE DB APIs. The SET options still remained to provide backwards compatibility for older clients.

Note: In case you are curious what effect these SET options have, I refer you to Books Online. Some of them are fairly straight-forward to explain, whereas others are just too confusing. To understand this article, you only need to understand that they exist, and what impact they have on the plan cache.

Alas, Microsoft did not change the defaults with full consistency, and even today the defaults depend on how you connect, as detailed in the table below.

	Applications using ADO .Net, ODBC or OLE DB	**SSMS**	**SQLCMD, OSQL, BCP, SQL Server Agent**	**ISQL, DB-Library**
ANSI_NULL_DFLT_ON	ON	ON	ON	**OFF**
ANSI_NULLS	ON	ON	ON	**OFF**
ANSI_PADDING	ON	ON	ON	**OFF**
ANSI_WARNINGS	ON	ON	ON	**OFF**
CONACT_NULLS_YIELD_NULL	ON	ON	ON	**OFF**
QUOTED_IDENTIFIER	ON	ON	**OFF**	**OFF**
ARITHABORT	**OFF**	ON	**OFF**	**OFF**

You might see where this is getting at. Your application connects with ARITHABORT OFF, but when you run the query in SSMS, ARITHABORT is ON and thus you will not reuse the cache entry that the application uses, but SQL Server will compile the procedure anew, sniffing your current parameter values, and you may get a different plan than from the application. So there you have a likely answer to the initial question of this article. There are a few more possibilities that we will look into in the next chapter, but by far the most common reason for *slow in the application, fast in SSMS* is parameter sniffing and the different defaults for ARITHABORT. (If that was all you wanted to know, you can stop reading. If you want to fix your performance problem – hang on! And, no, putting SET ARITHABORT ON in the procedure is **not** the solution.)

Besides the SET command and the defaults above, ALTER DATABASE permits you to say that a certain SET option always should be ON by default in a database and thus override the default set by the API. However, while the syntax may indicate so, you cannot specify than an option should be OFF this way. Also, beware that if you test these options from Management Studio, they may not seem to work, since SSMS submits explicit SET commands, overriding any default. There is also a server-level setting for the same purpose, the configuration option **user options** which is a bit mask. You can set the individual bits in the mask from the *Connection* pages of the *Server Properties* in Management Studio. Overall, I recommend against controlling the defaults this way, as in my opinion they mainly serve to increase the confusion.

It is not always the run-time setting of an option that applies. When you create a procedure, view, table etc, the settings for ANSI_NULLS and QUOTED_IDENTIFIER, are saved with the object. That is, if you run this:

```
SET ANSI_NULLS, QUOTED_IDENTIFIER OFF
go
CREATE PROCEDURE stupid @x int AS
IF @x = NULL PRINT "@x is NULL"
go
SET ANSI_NULLS, QUOTED_IDENTIFIER ON
go
EXEC stupid NULL
```

It will print

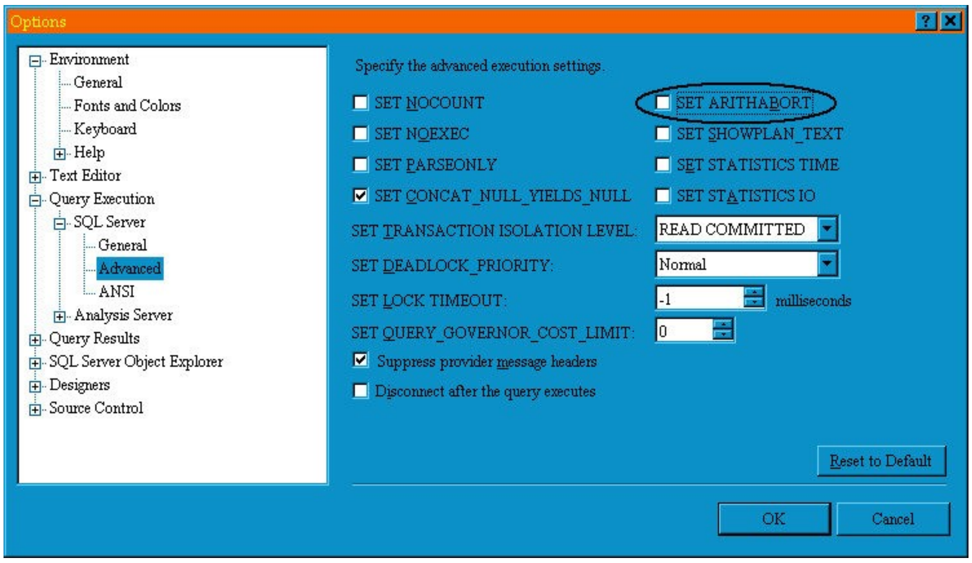
```
@x is NULL
```

(When QUOTED_IDENTIFIER is OFF, double quote(`"`) is a string delimiter on equal basis with single quote(`'`). When the setting is ON, double quotes delimit identifiers in the same way that square brackets (`[]`) do and the PRINT statement would yield a compilation error.)

In addition, the setting for ANSI_PADDING is saved per table column where it is applicable (The data types **varchar** and **varbinary**).

All these options and different defaults are certainly confusing, but here are some pieces of advice. First, remember that the first six of these seven options exist only to supply backwards compatibility, so there is little reason why you should ever have any of them OFF. Yes, there are situations when some of them may seem to buy a little more convenience if they are OFF, but don't fall for that temptation. One complication here, though, is that the SQL Server tools spew out SET commands for some of these options when you script objects. Thankfully, they mainly produce SET ON commands that are harmless. (But when you script a table, scripts may still in have a SET ANSI_PADDING OFF at the end. You can control this under Tools->Options->Scripting where you can set *Script ANSI_PADDING commands* to False, which I recommend.)

Next, when it comes to ARITHABORT, you should know that in SQL 2005 and later versions, this setting has **zero** impact as long as ANSI_WARNINGS is ON. (To be precise: it has no impact as long as the compatibility level is 90 or higher.) Thus, there is no reason to turn it on for the sake of the matter. And when it comes to SQL Server Management Studio, you might want do yourself a favour, and open this dialog and uncheck SET ARITHABORT as highlighted:

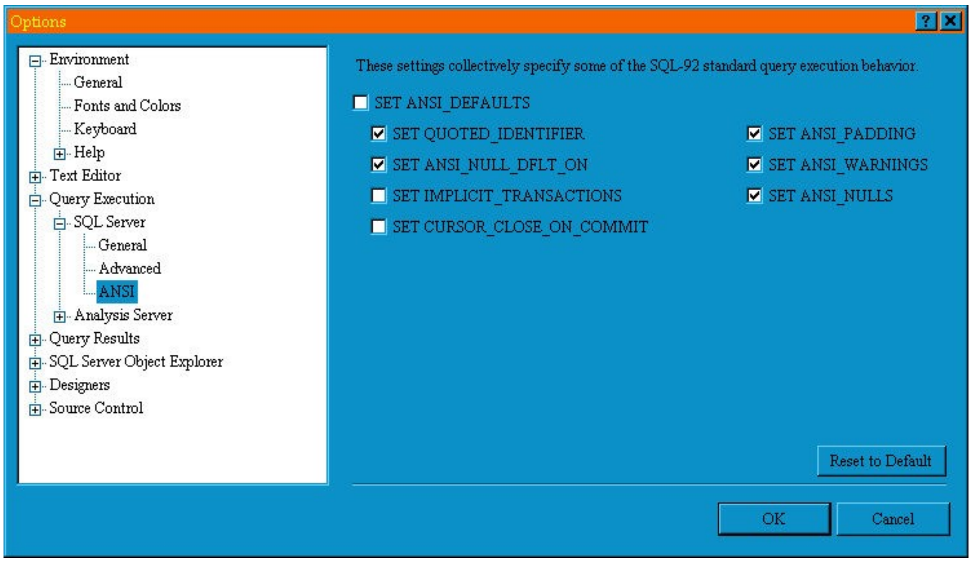


Specify the advanced execution settings.

☐ SET NOCOUNT
☐ SET NOEXEC
☐ SET PARSEONLY
☑ SET CONCAT_NULL_YIELDS_NULL
☐ SET ARITHABORT
☐ SET SHOWPLAN_TEXT
☐ SET STATISTICS TIME
☐ SET STATISTICS IO
SET TRANSACTION ISOLATION LEVEL: READ COMMITTED
SET DEADLOCK_PRIORITY: Normal
SET LOCK TIMEOUT: -1 milliseconds
SET QUERY_GOVERNOR_COST_LIMIT: 0
☑ Suppress provider message headers
☐ Disconnect after the query executes

This will change your default setting for ARITHABORT when you connect with SSMS. It will not help you to make your application to run faster, but you will at least not have to be perplexed by getting different performance in SQL Server Management Studio.

For reference, below is how the ANSI page should look like. A very strong recommendation: never change anything on this page!



These settings collectively specify some of the SQL-92 standard query execution behavior.

☐ SET ANSI_DEFAULTS
☑ SET QUOTED_IDENTIFIER
☑ SET ANSI_NULL_DFLT_ON
☐ SET IMPLICIT_TRANSACTIONS
☐ SET CURSOR_CLOSE_ON_COMMIT
☑ SET ANSI_PADDING
☑ SET ANSI_WARNINGS
☑ SET ANSI_NULLS

When it comes to SQLCMD and OSQL, make the habit to always use the `-I` option, which causes these tools to run with QUOTED_IDENTIFIER ON. The corresponding option for BCP is `-q`. (To confuse, `-q` has one more effect for BCP which I discuss in my article *Using the Bulk-Load Tools in SQL Server*.) It's a little more difficult in Agent, since there is no way to change the default for Agent – at least I have not found any. Then again, if you only run stored procedures from your job steps, this is not an issue, since the saved setting for stored procedures takes precedence. But if you would run loose batches of SQL from Agent jobs, you could face the problem with different query plans in the job and SSMS because of the different defaults for QUOTED_IDENTIFER. For such jobs, you should always include the command SET QUOTED_IDENTIFIER ON as the first command in the job step.

We have already looked at SET DATEFORMAT, and there are two more options in that group: LANGUAGE and DATEFIRST. The default language is configured per user, and there is a server-wide configuration option which controls what is the default language for new users. The default language controls the default for the other two. Since they are cache keys, this means that two users with different default languages will have different cache entries, and may thus have different query plans.

My recommendation is that you should try to avoid being dependent on language and date settings in SQL Server altogether. For instance, in as far as you use date literals at all, use a format that is always interpreted the same, such as YYYYMMDD. (For more details about date formats, see the article *The ultimate guide to the datetime datatypes* by SQL Server MVP Tibor Karaszi.) If you want to produce localised output from a stored procedure depending on the user's preferred language, it may be better to roll your own than rely on the language setting in SQL Server.

Note: I said above that ARITHABORT has no impact at all as long as ANSI_WARNINGS is on. Yet, the topic for SET ARITHABORT ON has this passage: *You should always set ARITHABORT to ON in your logon sessions. Setting ARITHABORT to OFF can negatively impact query optimization leading to performance issues*. I have checked older versions of this topic, and the passage first appeared in the SQL 2012 version of the topic. Since I don't know the optimizer internals, I can't say for sure the passage is nonsense, but it certainly does not make sense to me. Thus, I stand to what I said above: the setting has zero impact on compatibility level 90 and higher. I you feel compelled to follow the recommendation, please bear in mind that if you start to submit an extra batch for every connection you make, you will add an extra network roundtrip, which can be a performance issue in itself.

2.6 The Effects of Statement Recompile

To get a complete picture how SQL Server builds the query plan, we need to study what happens when individual statements are recompiled. Above, I mentioned a few situations where it can happen, but at that point I did not go into details.

The procedure below is certainly contrived, but it serves well to demonstrate what happens.

```
   CREATE PROCEDURE List_orders_7 @fromdate datetime,
                                  @ix       bit AS
```

```
      SELECT @fromdate = dateadd(YEAR, 2, @fromdate)
      SELECT * FROM  Orders WHERE OrderDate > @fromdate
      IF @ix = 1 CREATE INDEX test ON Orders(ShipVia)
      SELECT * FROM  Orders WHERE OrderDate > @fromdate
   go
   EXEC List_orders_7 '19980101', 1
```

When you run this and look at the actual execution plan, you will see that the plan for the first SELECT is a Clustered Index Scan, which agrees with what we have learnt this far. SQL Server sniffs the value 1998-01-01 and estimates that the query will return 267 rows which is too many to read with Index Seek + Key Lookup. What SQL Server does not know is that the value of **@fromdate** changes before the queries are executed. Nevertheless, the plan for the second, identical, query is precisely Index Seek + Key Lookup and the estimate is that one row is returned. This is because the CREATE INDEX statement sets a mark that the schema of the **Orders** table has changed, which triggers a recompile of the second SELECT statement. When recompiling the statement, SQL Server sniffs the value of the parameter which is current at this point, and thus finds the better plan.

Run the procedure again, but with different parameters (note that the date is two years earlier in time):

```
   EXEC List_orders_7 '19960101', 0
```

The plans are the same as in the first execution, which is a little more exciting than it may seem at first glance. On this second execution, the first query is recompiled because of the added index, but this time the scan is the "correct" plan, since we retrieve about one third of the orders. However, since the second query is not recompiled now, the second query runs with the Index Seek from the previous execution, although now it is not an efficient plan.

Before you continue, clean up:

```
   DROP INDEX test ON Orders
   DROP PROCEDURE List_orders_7
```

As I said, this example is contrived. I made it that way, because I wanted a compact example that is simple to run. In a real-life situation, you may have a procedure that uses the same parameter in two queries against different tables. The DBA creates a new index on one of the tables, which causes the query against that table to be recompiled, whereas the other query is not. The key takeaway here is that the plans for two statements in a procedure may have been compiled for different "sniffed" parameter values.

When we have seen this, it seems logical that this could be extended to local variables as well. But this is not the case:

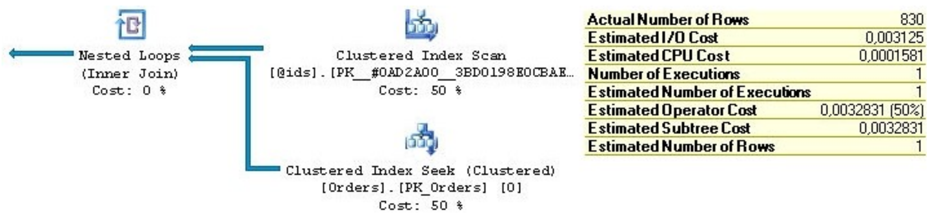
```
   CREATE PROCEDURE List_orders_8 AS
      DECLARE @fromdate datetime
      SELECT @fromdate = '20000101'
      SELECT * FROM  Orders WHERE OrderDate > @fromdate
      CREATE INDEX test ON Orders(ShipVia)
      SELECT * FROM  Orders WHERE OrderDate > @fromdate
      DROP INDEX test ON Orders
   go
   EXEC List_orders_8
   go
   DROP PROCEDURE List_orders_8
```

In this example, we get a Clustered Index Scan for both SELECT statements, despite that the second SELECT is recompiled during execution and the value of **@fromdate** is known at this point.

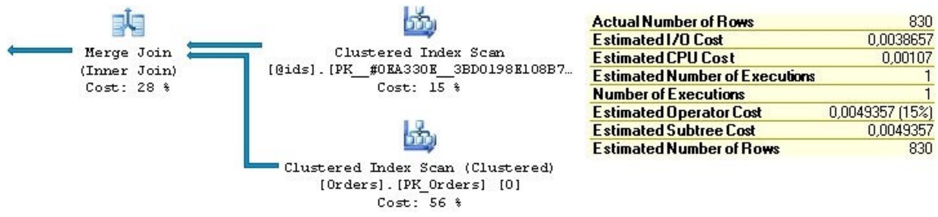
However, there is one exception, and that is table variables. Normally SQL Server estimates that a table variable has a single row, but when there are recompiles in sway, the estimate may be different:

```
   CREATE PROCEDURE List_orders_9 AS
      DECLARE @ids TABLE (a int NOT NULL PRIMARY KEY)
      INSERT @ids (a)
         SELECT OrderID FROM Orders
      SELECT COUNT(*)
      FROM   Orders O
      WHERE  EXISTS (SELECT *
                     FROM   @ids i
                     WHERE  O.OrderID = i.a)
      CREATE INDEX test ON Orders(ShipVia)
      SELECT COUNT(*)
      FROM   Orders O
      WHERE  EXISTS (SELECT *
                     FROM   @ids i
                     WHERE  O.OrderID = i.a)
      DROP INDEX test ON Orders
   go
   EXEC List_orders_9
   go
   DROP PROCEDURE List_orders_9
```

When you run this, you will get in total four execution plans. The two of interest are the second and fourth plans that come from the two identical SELECT COUNT(*) queries. I have included the interesting parts of the plans here, together with the pop-up for the Clustered Index Scan operator over the table variable.



Actual Number of Rows	830
Estimated I/O Cost	0,003125
Estimated CPU Cost	0,0001581
Number of Executions	1
Estimated Number of Executions	1
Estimated Operator Cost	0,0032831 (50%)
Estimated Subtree Cost	0,0032831
Estimated Number of Rows	1



Actual Number of Rows	830
Estimated I/O Cost	0,0038657
Estimated CPU Cost	0,00107
Estimated Number of Executions	1
Number of Executions	1
Estimated Operator Cost	0,0049357 (15%)
Estimated Subtree Cost	0,0049357
Estimated Number of Rows	830

In the first plan, there is a Nested Loops Join operator together with a Clustered Index Seek on the **Orders** table, which is congruent with the estimate of the number of rows in the table variable: one single row, the standard assumption. In the second query, the join is carried out with a Merge Join together with a table scan of **Orders**. As you can see, the estimate for the table variable is now 830 rows, because when recompiling a query, SQL Server "sniffs" the cardinality of the table variable, even if it is not a parameter.

And with some amount of bad luck this can cause issues similar to those with parameter sniffing. I once ran into a situation where a key procedure in our system suddenly was slow, and I tracked it down to a statement with a table variable where the estimated number of rows was 41. Unfortunately, when this procedure runs in the daily processing it's normally called on one-by-one basis, so one row was a much better estimate in that case.

Note: what I said above is true up to SQL 2017. If you run **List_orders_9** on SQL 2019 or later, you will find that the first SELECT COUNT(*) also runs with a Merge Join and has a correct estimate of 830 rows. Starting with SQL 2019, SQL Server defers compilation of a statement referfing to a table variable until execution reaches that statement. In this way, the statement will be compiled with the actual cardinality of the table instead of a blind assumption of one row. This is intended to improve performance, and certainly it should in many cases. But as I noted above, there are situations where this can backfire.

Speaking of table variables, you may be curious about table-valued *parameters*, introduced in SQL 2008. They are handled very similar to table variables, but since the parameter is populated before the procedure is invoked, it is now a common situation that SQL Server will use an estimate of more than one row. Here is an example:

```
CREATE TYPE temptype AS TABLE (a int NOT NULL PRIMARY KEY)
go
CREATE PROCEDURE List_orders_10 @ids temptype READONLY AS
   SELECT COUNT(*)
   FROM   Orders O
   WHERE  EXISTS (SELECT *
                  FROM   @ids i
                  WHERE  O.OrderID = i.a)
go
DECLARE @ids temptype
INSERT @ids (a)
   SELECT OrderID FROM Orders
EXEC List_orders_10 @ids
go
DROP PROCEDURE List_orders_10
DROP TYPE temptype
```

The query plan for this procedure is the same as for the second SELECT query in **List_orders_9**, that is Merge Join + Clustered Index Scan of **Orders**, since SQL Server sees the 830 rows in **@ids** when the query is compiled.

2.7 The Story So Far

In this chapter, we have looked at how SQL Server compiles a stored procedure and what significance the actual parameter values have for compilation. We have seen that SQL Server puts the plan for the procedure into cache, so that the plan can be reused later. We have also seen that there can be more than one entry for the same stored procedure in the cache. We have seen that there is a large number of different cache keys, so potentially there can be very many plans for a single stored procedure. But we have also learnt that many of the SET options that are cache keys are legacy options that you should never change.

In practice, the most important SET option is ARITHABORT, because the default for this option is different in an application and in SQL Server Management Studio. This explains why you can spot a slow query in your application, and then run it at good speed in SSMS. The application uses a plan which was compiled for a different set of sniffed parameter values than the actual values, whereas when you run the query in SSMS, it is likely that there is no plan for ARITHABORT ON in the cache, so SQL Server will build a plan that fits with your current parameter values.

You have also understood that you can verify that this is the case by running this command in your query window:

```
SET ARITHABORT OFF
```

and with great likelihood, you will now get the slow behaviour of the application also in SSMS. If this happens, you know that you have a performance problem related to parameter sniffing. What you may not know yet is how to address this performance problem, and in the following chapters I will discuss possible solutions, before I return to the theme of compilation, this time for ad-hoc queries, a.k.a. dynamic SQL.

Note: There are always these funny variations. The application I mainly work with actually issues SET ARITHABORT ON when it connects, so we should never see this confusing behaviour in SSMS. Except that we do. Some part of the application also issues the command SET NO_BROWSETABLE ON on connection. I have never been able to understand the impact of this undocumented SET command, but I seem to recall that it is related to early versions of "classic" ADO. And, yes, this setting is a cache key.

3. It's Not Always Parameter Sniffing...

Before we delve into how to address performance problems related to parameter sniffing, which is quite a broad topic, I would first like to give some coverage to a couple of cases where parameter sniffing is not involved, but where you nevertheless may experience different performance in the application and SSMS.

3.1 Replacing Variables and Parameters

I have already touched at this, but it is worth expanding on a bit.

Occasionally, I see people in the forums telling me that their stored procedure is slow, but when they run the same query outside the procedure it's fast. After a few posts in the thread, the truth is revealed: the query they are struggling with refer to variables, be that local variables or parameters. To troubleshoot the query on its own, they have replaced the variables with constants. But as we have seen, the resulting stand-alone query is quite different, and SQL Server can make more accurate estimates with constants instead of variables, and therefore it arrives at a better plan. Furthermore, SQL Server does not have to consider that the constant may have a different value next time the query is run.

A similar mistake is to make the parameters into variables. Say that you have:

```
CREATE PROCEDURE some_sp @par1 int AS
   ...
   -- Some query that refers to @par1
```

You want to troubleshoot this query on its own, so you do:

```
DECLARE @par1 int
SELECT @par1 = 4711
-- query goes here
```

From what you have learnt here, you know that this is very different from when **@par1** really is a parameter. SQL Server has no idea about the value for **@par1** when you declare it as a local variable and will make standard assumptions.

But if you have a 1000-line stored procedure, and one query is slow, how do you run it stand-alone with great fidelity, so that you have the same presumptions as in the stored procedure?

One way to tackle this is to embed the query in **sp_executesql**:

```
EXEC sp_executesql N'-- Some query that refers to @par1', N'@par1 int', 4711
```

You will need to double any single quotes in the query to be able to put it in a character literal. If the query refers to local variables, you should assign them in the block of dynamic SQL and not pass them as parameters so that you have the same presumptions as in the stored procedure.

Another option is to create dummy procedure with the problematic statement; this saves from doubling any quotes. To avoid litter in the database, you could create a temporary stored procedure:

```
CREATE PROCEDURE #test @par1 int AS
   -- query goes here.
```

As with dynamic SQL, make sure that local variables are locally declared also in your dummy. I will need to add the caveat I have not investigated whether SQL Server have special tweaks or limitations when optimising temporary stored procedures. Not that I see why there should be any, but I have been burnt before...

3.2 Blocking

You should not forget that one possible reason that the procedure ran slow in the application was simply a matter of blocking. When you tested the query three hours later in SSMS, the blocker had completed its work. If you find that no matter how you run the procedure in SSMS, with or without ARITHABORT, the procedure is always fast, blocking is starting to seem a likely explanation. Next time you are alarmed that the procedure is slow, you should start your investigation with some blocking analysis. That is a topic which is completely outside the scope for this article, but for a good tool to investigate locking, see my beta_lockinfo.

3.3 Database Settings

Let's say that you run a query like this in two different databases:

```
SELECT ...
FROM   dbo.sometable
JOIN   dbo.someothertable ON ...
JOIN   dbo.yetanothertable ON ...
WHERE  ...
```

You find that the query performs very differently in the two databases. There can be a whole lot of reasons for these differences. Maybe the data sizes are entirely different in one or more of the tables, or the data is distributed differently. It could be that the statistics are different, even if the data is identical. It could also be that one database has an index that the other database has not.

Say that the query instead goes:

```
SELECT ...
FROM   thatdb.dbo.sometable
JOIN   thatdb.dbo.someothertable ON ...
JOIN   thatdb.dbo.yetanothertable ON ...
WHERE ...
```

That is, the query refers to the tables in three-part notation. And yet you find that the query runs faster in SSMS than in the application or vice versa. But when you get the idea to change the database in SSMS to be the same as in the application, you get slow performance in SSMS as well. What is going on? It can certainly not be anything of what I discssused above, since the tables are the same, no matter the database you issue the query from.

The answer is that it *may* be parameter sniffing, because if you look at the output of the query I introduced in the section *Different Plans for Different Settings*, you will see that **dbid** is one of the cache keys. That is, if you run the same query against the same tables from two different databases, you get different cache entries, and thus there can be different plans. And, as we have learnt, one of the possible reasons for this is parameter sniffing. But it could also be that the settings for the two databases are different:

Thus, f this occurs to you, run this query:

```
SELECT * FROM sys.databases WHERE name IN ('slowdb', 'fastdb')
```

(Obviously, you should replace *slowdb* and *fastdb* with the names of the actual databases you ran from.) Compare the rows, and make note of the differences. Far from all the columns you see affect the plan choice. The by far the most important one is the compatibility level. When Microsoft makes enhancements to the optimiser in a new version of SQL Server, they only make these enhancements available in the latest compatibility level, because although they are intended to be enhancements, there will always be queries that will be negatively affected by the change and run a lot slower.

If you are on SQL 2016 or later, you also need to look in **sys.database_scoped_configurations** and compare the settings between the two databases. (This view does not hold the database id or the name, but each database has its own version.) Quite a few of these settings affects the work of the optimizer, which could lead to different plans.

What you would do once you have identified the difference depends on the situation. But generally, I recommend against changing away from the defaults. (Starting with SQL 2017, **sys.database_scoped_configurations** has a column **is_value_default** which is 1, if the current setting is the default setting.)

For isntance, if you find that the slow plan comes from a database with a lower compatibility level, consider changing the compatitiblity level for that database. But if the slow plan occurs with the database with the higher compatibility level, you should not change the setting, but rather work with the query and available indexes to see what can be done. In my experience, when a query regresses when going to a newer version of the optimiser, there is usually something that is problematic, and you were only lucky that the query ran fast on the earlier version.

There is one quite obvious exception to the rule of sticking with the defaults: if you find that in the fast database, the database-scoped configuration QUERY_OPTIMIZER_HOTFIXES is set to 1, you should consider to enable this setting for the other database as well, as this will give you access to optimiser fixes released after the original release of the SQL Server version you are using.

Just to make it clear: the database settings do not only apply to queries with tables in three-part notation, but they can also explain why the same query or stored procedure with tables in one- or two-part notation gets different plans and performance in seemingly similar databases.

3.4 Indexed Views and Indexed Computed Columns

This is a problem which is far more common on SQL 2000 than on later versions. Or to be precise, it applies to databases which are in compatibility level 80, which is why this can occur also on SQL 2005 and SQL 2008. (Later versions do not support this compat level.)

As long as the database is compatibility level 90 or later, for SQL Server 2005 to consider indexed views and indexes on computed columns (as well as filtered indexes added in SQL 2008) when compiling a query, these settings must be ON: QUOTED_IDENTIFIER, ANSI_NULLS, ANSI_WARNINGS, ANSI_PADDING, CONCAT_NULL_YIELDS_NULL. Furthermore, NUMERIC_ROUNDABORT must be OFF. But in compat level 80, there is one more option that must be ON, and, yes, you guessed it: ARITHABORT. (The reason for this is that in compatibility level 80 there is one type of error – domain errors, e.g. `sqrt(-1)` – that is covered by ARITHABORT, but not by ANSI_WARNINGS.)

Thus, with compatibility level 80, an application using the default SET options will not be able to take benefit of an index on a computed column or an indexed view, even if that would be the optimal query plan. But when you run the query in SSMS, performance will be a lot better, even if no parameter sniffing is involved, because there ARITHABORT is ON by default.

There is a second phenomenon you can run into with indexed computed columns and indexed views which also is related to SQL 2000, but because of legacy can persist into even recent versions. You have a stored procedure that is slow. You take out the statement and run it on its own, and now it's lightning fast, even if you package it nice and cleanly in a temporary stored procedure as above. Why? Run this statement:

```
SELECT objectproperty(object_id('your_sp'), 'IsQuotedIdentOn'),
       objectproperty(object_id('your_sp'), 'IsAnsiNullsOn')
```

Most likely it will return 0 it at least one of the two columns. As I noted previously, these two settings, QUOTED_IDENTIFIER and ANSI_NULLS are saved with the procedure, and thus the saved settings apply when the procedure runs, not the settings of the connection.

But why would these options be OFF? I would guess that this mainly happens with databases originating from SQL 2000 or earlier versions. To wit, in SQL 2000, you could create stored procedures from Enterprise Manager, and Enterprise Manager always submitted SET QUOTED_IDENTIFIER OFF and SET ANSI_NULLS OFF prior to creating the objects and this was not configurable. The database may have moved on to newer versions of SQL Server, but when you script the procedure from SSMS, SSMS will add SET OFF commands to retain the old settings. There is all reason to consider to change the SET commands while you are at it!

If only QUOTED_IDENTIFIER is OFF, but ANSI_NULLS is ON for a procedure, one can suspect that it has been created through SQLCMD which as we have seen run with QUOTED_IDENTIFIER OFF by default. (Always run these tools with the `-I` option to override.)

To find all bad modules in a database, you can use this SELECT:

```
SELECT o.name
FROM   sys.sql_modules m
JOIN   sys.objects o ON m.object_id = o.object_id
WHERE  (m.uses_quoted_identifier = 0 or
        m.uses_ansi_nulls = 0)
  AND  o.type NOT IN ('R', 'D')
```

3.5 An Issue with Linked Servers

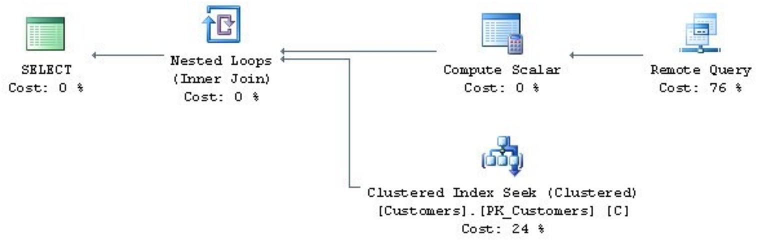
This section concerns an issue with linked servers which mainly occurs when the *remote* server is earlier than SQL 2012 SP1, but it can appear with later versions as well under some circumstances.

Consider this query:

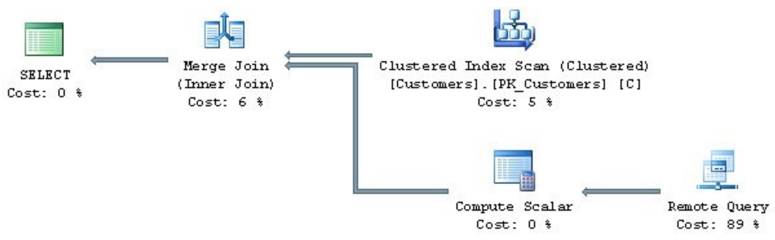
```
SELECT C.*
FROM   SQL_2008.Northwind.dbo.Orders O
JOIN   Customers C ON O.CustomerID = C.CustomerID
WHERE  O.OrderID > 20000
```

I ran this query twice, logged in as two different users. The first user is **sysadmin** on both servers, whereas the second user is a plain user with only SELECT permissions. To ensure that I would get different cache entries, I used different settings for ARITHABORT.

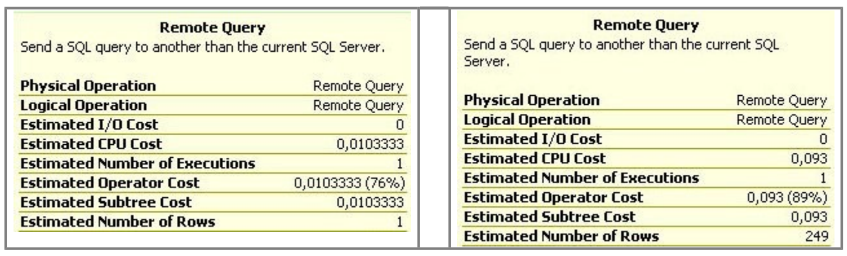
When I ran the query as **sysadmin** I got this plan:



When I ran the query as the plain user, the plan was different:



How come the plans are different? It's certainly not parameter sniffing because there are no parameters. As always when a query plan has an unexpected shape or operator, it is a good idea to look at the estimates. Here are the pop-ups for the Remote Query operators in the two plans, with the pop-up for the first plan to the left:



Remote Query
Send a SQL query to another than the current SQL Server.

Physical Operation	Remote Query
Logical Operation	Remote Query
Estimated I/O Cost	0
Estimated CPU Cost	0,0103333
Estimated Number of Executions	1
Estimated Operator Cost	0,0103333 (76%)
Estimated Subtree Cost	0,0103333
Estimated Number of Rows	1

Remote Query
Send a SQL query to another than the current SQL Server.

Physical Operation	Remote Query
Logical Operation	Remote Query
Estimated I/O Cost	0
Estimated CPU Cost	0,093
Estimated Number of Executions	1
Estimated Operator Cost	0,093 (89%)
Estimated Subtree Cost	0,093
Estimated Number of Rows	249

You can see that the estimates are different. When I ran as **sysadmin**, the estimate was 1 row, which is a correct number, since there are no orders in **Northwind** where the order ID exceeds 20000. (Recall that the optimizer never assumes zero rows from statistics.) But when I ran the query as a plain user, the estimate was 249 rows. We recognize this particular number as 30 % of 830 orders, or the estimate for an inequality operation when the optimizer has no information. Previously, this was due to an unknown variable value, but in this case there is no variable that can be unknown. No, it is the statistics themselves that are missing.

As long as a query accesses tables in the local server only, the optimizer can always access the statistics for all tables in the query. This happens internally in SQL Server and there are no extra checks to see whether the user has permission to see the statistics. But this is different with tables on linked servers. When SQL Server accesses a linked server, the optimizer needs to retrieve the statistics on the same connection that is used to retrieve the data, and it needs to use T-SQL commands which is why permissions come into play. And, unless login mapping has been set up, those are the permissions of the user running the query.

By using Profiler or Extended Events you can see that the optimizer retrieves the statistics in two steps. First it calls the procedure **sp_table_statistics2_rowset** which returns information about which column statistics there are, as well as the cardinality and density information of the columns. In the second step, it runs DBCC SHOW_STATISTICS to get the full distribution statistics. (We will look closer at this command later in this article.)

Note: to be accurate, the optimizer does not talk to the linked server at all, but it interacts with the OLE DB provider for the remote data source and requests the provider to return the information the optimizer needs.

For **sp_table_statistics2_rowset** to run successfully the user must have the permission VIEW DEFINITION on the table. This permission is implied if the user has SELECT permission on the table (without which the user would not be able to run the query at all). So, unless the user has been explicitly denied VIEW DEFINITION, this procedure is not so much a concern.

DBCC SHOW_STATISTICS is a different matter. For a long time, running this command required membership in the server role **sysadmin** or in one of the database roles **db_owner** or **db_ddladmin**. This was changed in SQL 2012 SP1, so starting with this version, only SELECT permission is needed.

And this is why I got different results. As you can tell from the server name, my linked server was an instance running SQL 2008. Thus, when I was connected as a user that was **sysadmin** on the remote instance, I got the full distribution statistics which indicated that there are no rows with order ID > 20000, and the estimate was one row. But when running as the plain user, DBCC SHOW_STATISTICS failed with a permission error. This error was not propagated, but instead the optimizer accepted that there were no statistics and used default assumptions. Since it did get cardinality information from **sp_table_statistics2_rowset**, it learnt that the remote table has 830 rows, whence the estimate of 249 rows.

From what I said above, this should not be an issue if the linked server runs SQL 2012 SP1 or later, but there can still be obstacles. If the user does not have SELECT permission on all columns in the table, there may be statistics the optimizer is not able to retrieve. Furthermore, according to Books Online, there is a trace flag (9485) which permits the DBA to prevent SELECT permission to be sufficient for running DBCC SHOW_STATISTICS. And more importantly, if row-level security (a feature added in SQL 2016) has been set up for the table, only having SELECT permission is not sufficient as this could permit users to see data they should not have access to. That is, to run DBCC SHOW_STATISTICS on a table with row-level filtering enabled, you need membership in **sysadmin**, **db_owner** or **db_ddladmin**. (Interesting enough, one would expect the same to apply if the table has Dynamic Data Masking enabled, but that does seem to be the case.)

Thus, when you encounter a performance problem where a query that accesses a linked server is slow in the application, but it runs fast when you test it from SSMS (where you presumably are connected as a power user), you should always investigate the permissions on the remote database. (Keep in mind that the access to the linked server may not be overt in the query, but could be hidden in a view.)

If you determine that permissions on the remote database is the problem, what actions could you take? Granting users more permissions on the remote server is of course an easy way out, but absolutely not recommendable from a security perspective. Another alternative is to set up login-mapping so that users log on the remote server with a proxy user with sufficient powers. Again, this is highly questionable from a security perspective.

Rather you would need to tweak the query. For instance, you can rewrite the query with OPENQUERY to force evaluation on the remote server. This can be particularly useful, if the query includes several remote tables, since for the query that runs on the remote server, the remote optimizer has full access to the statistics on that server. (But it can also backfire, because the local optimizer now gets even less statistics information from the remote server.) You can also use the full battery of hints and plan guides to get the plan you want.

I would also recommend that you ask yourself (and the people around you): is that linked-server access needed? Maybe the databases could be on the same server? Could data be replicated? Some other solution? Personally, linked servers is something I try to avoid as much as possible. Linked servers often mean hassle, in my experience.

Before I close this section, I like to repeat: what matters is the permissions on the remote server, not the local server where the query is issued. I also like to point out that I have given a blind eye to what may happen with other remote data sources such as Oracle, MySQL or Access as they have different permission systems of which I'm entirely ignorant. You may or may not see similar issues when running queries against such linked servers.

3.6 Could it Be MARS?

I got this mail from a reader:

> *We recently found a SQL Server 2016 select query against CCIs that was slow in the application, and fast in SSMS. Application execution time was 3 to 4 times that of SSMS – looking a bit deeper it was seen that the ratio of CPU time was similar. This held true whether the query was executed in parallel as typical or in serial by forcing MAXDOP 1. An unusual facet was that Query Store was accounting the executions against the same row in sys.query_store_query and sys.query_store_plan, whether from the app or from SSMS.*
>
> *The application was using a connection string enabling MARS, even though it didn't need to. Removing that clause from the connection string resulted in SSMS and the application experiencing the same CPU time and elapsed time.*
>
> *We'll keep looking to determine the mechanics of the difference. I suspect it may be due to locking behavior difference – perhaps MARS disables lock escalation.*

Note: MARS = Multiple Active Result Sets. If you set this property on a connection string, you can run multiple queries on the same connection in an interleaved fashion. It's mainly intended to permit you to submit UPDATE statements as you are iterating through a result set.

The observations from Query Store make it quite clear to me that the execution plan was the same in both cases. Thus, it cannot be a matter of parameter sniffing, different SET options or similar. Interesting enough, some time later after I had added this section to the article, I got a mail from a second reader who had experienced the same thing. That is, access to a clustered columnstore index was significantly slower with MARS enabled.

This kept me puzzled for a while, and there was too little information to make it possible for me to make an attempt to reproduce the issue. But then I got a third mail on this theme, and Nick Smith was kind to provide a simple example query:

```
SELECT TOP 10000 SiteId FROM MyTable
```

That is, it is simply a matter of a query that returns many rows. Note that in this case there was no columnstore index involved, but just a plain-vanilla table. I built a small C# program on this theme and measured execution time with and without MARS. At first I could not discern any difference at all, but I was running against my local instance. Once I targeted my database in Windows Azure it was a difference of a factor ten. When I connected to a server on the other side of town the difference was a factor of three or four.

It seems quite clear to me that it is a matter of network latency. I assume that the interleaved nature of MARS introduces chattiness on the wire, so the slower and longer the network connection is, the more is that chattiness going to affect you.

Thus, if you find that a query that returns a lot of data runs slow in the application and a lot faster in SSMS, there is all reason to see whether the application specifies `MultipleActiveResultSets=true` in the connection string. If it does, ask yourself if you need it, and if not take it out.

Could MARS make an application go slower for some other reason? I see no reason to believe so, but it is still a little telling that my first two correspondents mentiond columnstore indexes. Also, the network latency does not really explain the difference in CPU mentioned in the quote above. So, if you encounter a situation where you conclude that MARS slows things and that there is no network latency, I would be very interested in hearing from you.

4. Getting Information to Solve Parameter Sniffing Problems

We have learnt how it may come that you have a stored procedure that runs slow in the application, and yet the very same call runs fast when you try it in SQL Server Management Studio: Because of different settings of ARITHABORT you get different cache entries, and since SQL Server employs parameter sniffing, you may get different execution plans.

While the secret behind the mystery now has been unveiled, the main problem still remains: how do you address the performance problem? From what you read this far, you already know of a quick fix. If you have never seen the problem before and/or the situation is urgent, you can always do:

```
EXEC sp_recompile problem_sp
```

As we have seen, this will flush the procedure from the plan cache, and next time it is invoked, there will be a new query plan. And if the problem never comes back, consider case closed.

But if the problem keeps reoccurring – and unfortunately, this is the more likely outcome – you need to perform a deeper analysis, and in this situation you should **not** use **sp_recompile**, or in some other way alter the procedure. You should keep that slow plan around, so that you can examine it, and not the least find what parameter values the bad plan was built for. This is the topic for this chapter.

Note: the advice in the paragraph above does not apply if you are on SQL 2016 or later and you have enabled the Query Store for your database. In this case, you can find all information about the plans in the Query Store views, even after the plans have been flushed. In the last chapter, I present Query Store versions of the queries that follows below.

Before I go on, a small observation: above I recommended that you should change your preferences in SSMS, so that you by default connect with ARITHABORT OFF to avoid this kind of confusion. But there is actually a small disadvantage with having the same settings as the application: you may not observe that the performance problem is related to parameter sniffing. But if you make it a habit when investigating performance issue to run your problem procedure with ARITHABORT both ON and OFF, you can easily conclude whether parameter sniffing is involved.

4.1 Getting the Necessary Facts

All performance troubleshooting requires facts. If you don't have facts, you will be in the situation that Bryan Ferry describes so well in the song *Sea Breezes* from the first Roxy Music album:

We've been running round in our present state
Hoping help will come from above
But even angels there make the same mistakes

If you don't have facts, not even the angels will be able to help you. The base facts you need to troubleshoot performance issues related to parameter sniffing are:

1. Which is the slow statement?
2. What are the different query plans?
3. What parameter values did SQL Server sniff?
4. What are the table and index definitions?
5. How do the distribution statistics look like? Is it up to date?

Almost all of these points apply to about any query-tuning effort. Only the third point is unique to parameter-sniffing issues. That, and the plural in the second point: you want to look at two plans, the good plan and the bad plan. In the following sections, we will look at these points one by one.

4.2 Which is the Slow Statement?

First on the list is to find the slow statement – in most cases, the problem lies with a single statement. If the procedure has only one statement, this is trivial. Else, you can use Profiler to find out; the Duration column will tell you. Either just trace the procedure from the application, or run the procedure from Management Studio (with ARITHABORT OFF!) and filter for your own spid.

Yet another option is to use the stored procedure **sp_sqltrace**, written by Lee Tudor and which I am glad to host on my web site. **sp_sqltrace** takes an SQL batch as parameter, starts a server-side trace, runs the batch, stops the trace and then summarises the result. There are a number of input parameters to control the procedure, for instance how to sort the output. This procedure is particularly useful to determine the slow statement in a loop, if you would happen to have such in your stored procedure.

4.3 Getting the Query Plans and Parameters with Management Studio

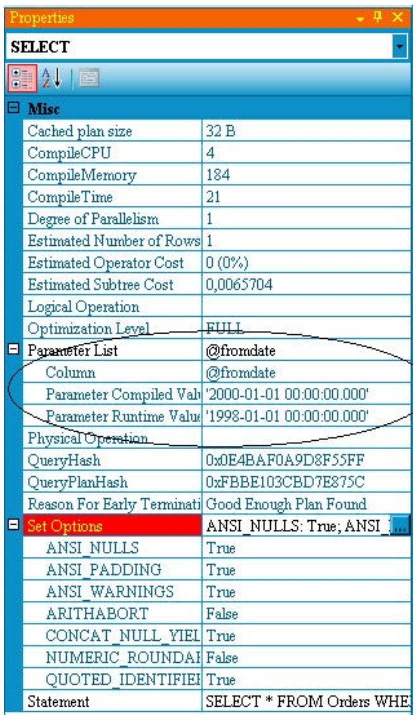
In many cases, you can easily find the query plans by running the procedure in Management Studio, after first enabling *Include Actual Execution Plan* (you find it under the Query menu). This works well, as long as the procedure does not include a multitude of queries, in which case the *Execution Plan* tab gets too littered to work with. We will look at alternative strategies in coming sections.

Typically, you would run the procedure like this:

```
SET ARITHABORT ON
go
EXEC that_very_sp 4711, 123, 1
go
SET ARITHABORT OFF
go
EXEC that_very_sp 4711, 123, 1
```

The assumption here is that the application runs with the default options, in which case the first call will give the good plan – because the plan is sniffed for the parameters you provide – and the second call will run with the bad plan which already is in the plan cache. To determine the cache keys in sway, you can use the query in the section *Different Plans for Different Settings* to see the cache-key values for the plan(s) in the cache. (If you already have tried the procedure in Management Studio, you may have two entries. The column **execution_count** in **sys.dm_exec_query_stats** can help you to discern the entries from each other; the one with the low count is probably your attempt from SSMS.)

Once you have the plans, you can easily find the sniffed parameter values. Right-click the left-most operator in the plan – the one that reads SELECT, INSERT, etc – select *Properties*, which will open a pane to the right. (That is the default position; the pane is detachable.) Here is an example how it can look like:



Properties

SELECT

Misc

Cached plan size	32 B
CompileCPU	4
CompileMemory	184
CompileTime	21
Degree of Parallelism	1
Estimated Number of Rows	1
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0,0065704
Logical Operation	
Optimization Level	FULL
Parameter List	@fromdate
Column	@fromdate
Parameter Compiled Valu	'2000-01-01 00:00:00.000'
Parameter Runtime Value	'1998-01-01 00:00:00.000'
Physical Operation	
QueryHash	0x0E4BAF0A9D8F55FF
QueryPlanHash	0xFBBE103CBD7E875C
Reason For Early Terminati	Good Enough Plan Found
Set Options	ANSI_NULLS: True; ANSI_
ANSI_NULLS	True
ANSI_PADDING	True
ANSI_WARNINGS	True
ARITHABORT	False
CONCAT_NULL_YIEL	True
NUMERIC_ROUNDAI	False
QUOTED_IDENTIFIEI	True
Statement	SELECT * FROM Orders WHE

The first *Parameter Compiled Value* is the sniffed value which is causing you trouble one way or another. If you know your application and its usage pattern, you may get an immediate revelation when you see the value. Maybe you do not, but at least you know now that there is a situation where the application calls the procedure with this possibly odd value.

Note also that you can see the settings of some of the SET options that are cache keys. However, if you are still on SQL 2005, beware that there is a bug, so that that the SET options will be always be reported as False. This is an engine bug, and thus your version of SSMS does not matter. (This bug is fixed in SQL 2008 and later.)

While Management Studio provides a very good interface to examine query plans, I still want to put in a plug for a more versatile alternative, to wit Plan Explorer, a free tool from SentryOne, a tool vendor in the SQL Server space. Plan Explorer permits you to view the plan in different ways, and it is also easier to navigate among the queries if your batch has a lot of them.

When you look at a query plan, it is far from always apparent what part of the plan that is really costly. But the thickness of the arrows is a good lead. The thicker the arrow, the more rows are passed to the next operator. And if you are looking at an actual execution plan, the thickness is based on the actual number of rows. On the other hand, I usually don't give any attention to the percentages at all. They are always estimates, and they can be way off, particular if there is a gross misestimate somewhere in the plan. And that is often the case when you have a performance problem related to parameter sniffing.

4.4 Getting the Query Plans and Parameters Directly from the Plan Cache

It is not always feasible to use SSMS to get the query plans and the sniffed parameter values. The bad query maybe runs for more minutes than your patience can accept, or the procedure includes so many statements that you get a mess in SSMS. Not the least this can be an issue if the procedure includes a loop that is executed many times, in which case not even SentryOne's Plan Explorer may be workable.

One option to get hold of the query plan and the sniffed parameters is to retrieve it directly from the plan cache. This is quite convenient with help of the query below, but there is an obvious limitation with this method: you only get the estimates. The actual number of rows and actual number of executions, two values that are very important to understand why a plan is bad, are missing.

The Query

This query will return the statements, the sniffed parameter values and the query plans for a stored procedure:

```
DECLARE @dbname    nvarchar(256),
        @procname  nvarchar(256)
SELECT @dbname = 'Northwind',
       @procname = 'dbo.List_orders_11'

; WITH basedata AS (
   SELECT qs.statement_start_offset/2 AS stmt_start,
          qs.statement_end_offset/2 AS stmt_end,
          est.encrypted AS isencrypted, est.text AS sqltext,
          epa.value AS set_options, qp.query_plan,
          charindex('<ParameterList>', qp.query_plan) + len('<ParameterList>')
             AS paramstart,
          charindex('</ParameterList>', qp.query_plan) AS paramend
   FROM   sys.dm_exec_query_stats qs
   CROSS  APPLY sys.dm_exec_sql_text(qs.sql_handle) est
   CROSS  APPLY sys.dm_exec_text_query_plan(qs.plan_handle,
                                             qs.statement_start_offset,
                                             qs.statement_end_offset) qp
   CROSS  APPLY sys.dm_exec_plan_attributes(qs.plan_handle) epa
   WHERE  est.objectid  = object_id (@procname)
     AND  est.dbid      = db_id(@dbname)
     AND  epa.attribute = 'set_options'
), next_level AS (
   SELECT stmt_start, set_options, query_plan,
          CASE WHEN isencrypted = 1 THEN '-- ENCRYPTED'
               WHEN stmt_start >= 0
               THEN substring(sqltext, stmt_start + 1,
                              CASE stmt_end
                                   WHEN 0 THEN datalength(sqltext)
                                   ELSE stmt_end - stmt_start + 1
```

```
                               END)
            END AS Statement,
            CASE WHEN paramend > paramstart
                 THEN CAST (substring(query_plan, paramstart,
                                   paramend - paramstart) AS xml)
            END AS params
     FROM   basedata
)
SELECT set_options AS [SET], n.stmt_start AS Pos, n.Statement,
       CR.c.value('@Column', 'nvarchar(128)') AS Parameter,
       CR.c.value('@ParameterCompiledValue', 'nvarchar(128)') AS [Sniffed Value],
       CAST (query_plan AS xml) AS [Query plan]
FROM   next_level n
CROSS  APPLY   n.params.nodes('ColumnReference') AS CR(c)
ORDER  BY n.set_options, n.stmt_start, Parameter
```

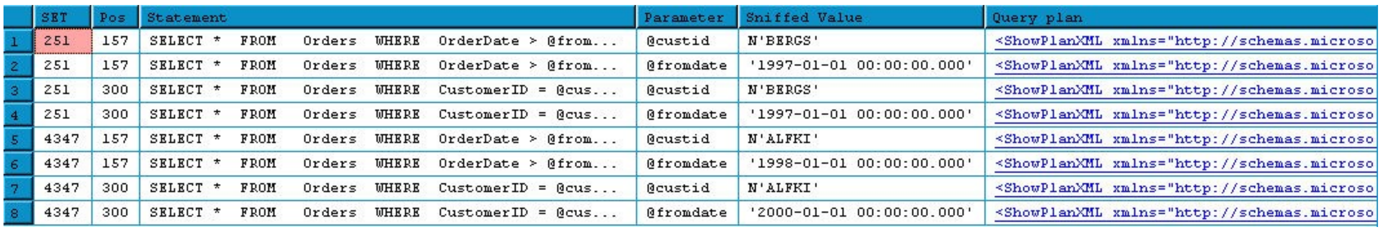
If you have never worked with these DMVs before, I appreciate if this is mainly mumbo-jumbo to you. To keep the focus on the main subject of this article, I will defer to a later section to explain this query. The only thing I like to give attention to here and now is that you specify the database and the procedure you want to work with in the beginning. You may think this would better be a stored procedure, but it is quite likely that you want to add or remove columns, depending on what you are looking for.

The Output

To see the query in action, you can use this test batch (and, yes, the examples get more and more contrived as we move on):

```
CREATE PROCEDURE List_orders_11 @fromdate datetime,
                                @custid   nchar(5) AS
SELECT @fromdate = dateadd(YEAR, 2, @fromdate)
SELECT *
FROM   Orders
WHERE  OrderDate > @fromdate
  AND  CustomerID = @custid
IF @custid = 'ALFKI' CREATE INDEX test ON Orders(ShipVia)
SELECT *
FROM   Orders
WHERE  CustomerID = @custid
  AND  OrderDate > @fromdate
IF @custid = 'ALFKI' DROP INDEX test ON Orders
go
SET ARITHABORT ON
EXEC List_orders_11 '19980101', 'ALFKI'
go
SET ARITHABORT OFF
EXEC List_orders_11 '19970101', 'BERGS'
```

When you have executed this batch, you can run the query above. When I do this I see this result in SSMS:



	SET	Pos	Statement	Parameter	Sniffed Value	Query plan
1	251	157	SELECT * FROM Orders WHERE OrderDate > @from...	@custid	N'BERGS'	<ShowPlanXML xmlns="http://schemas.microso
2	251	157	SELECT * FROM Orders WHERE OrderDate > @from...	@fromdate	'1997-01-01 00:00:00.000'	<ShowPlanXML xmlns="http://schemas.microso
3	251	300	SELECT * FROM Orders WHERE CustomerID = @cus...	@custid	N'BERGS'	<ShowPlanXML xmlns="http://schemas.microso
4	251	300	SELECT * FROM Orders WHERE CustomerID = @cus...	@fromdate	'1997-01-01 00:00:00.000'	<ShowPlanXML xmlns="http://schemas.microso
5	4347	157	SELECT * FROM Orders WHERE OrderDate > @from...	@custid	N'ALFKI'	<ShowPlanXML xmlns="http://schemas.microso
6	4347	157	SELECT * FROM Orders WHERE OrderDate > @from...	@fromdate	'1998-01-01 00:00:00.000'	<ShowPlanXML xmlns="http://schemas.microso
7	4347	300	SELECT * FROM Orders WHERE CustomerID = @cus...	@custid	N'ALFKI'	<ShowPlanXML xmlns="http://schemas.microso
8	4347	300	SELECT * FROM Orders WHERE CustomerID = @cus...	@fromdate	'2000-01-01 00:00:00.000'	<ShowPlanXML xmlns="http://schemas.microso

These are the columns:

SET – The **set_options** attribute for the plan. As I discussed earlier, this is a bit mask. In this picture, you see the two most likely values. 251 is the default settings and 4347 is the default settings + ARITHABORT ON. If you see other values, you can use the function setoptions to translate the bit mask.

Pos – This is the position for the query in the procedure, counted in characters from the start of the batch that created the procedure, including any comments preceding CREATE PROCEDURE. Not terribly useful in itself, but serves to sort the statements in the order they appear in the procedure.

Statement – The SQL statement. Note that the statements are repeated once for each parameter in the query.

Parameter – The name of the parameter. Only parameters that appear in this statement are listed. As a consequence of this, statements that do not refer to any parameters are not included in the output at all.

Sniffed Value – The compile-time value for the parameter, that is, the value that the optimizer sniffed when it built the plan. In difference to the *Properties* pane for the plan, you don't see any actual parameter value here. As I discussed previously, the sniffed value for a parameter can be different for different statements in the procedure, and you see an example of this in the picture above.

Query Plan – The query plan. You can double-click the XML document to see the graphical plan directly. (This does not always work. Typically, it does not work if your version of SSMS is lower than the SQL Server version. This feature was also broken in some versions of SSMS 2008 R2. And the feature did not exist at all in SSMS 2005.) As I noted above, this is only the estimated plan. You cannot get any actual values from the cache.

The Query Explained

This query refers to some DMVs which not all readers may be acquainted with. It also uses some query techniques that you may not be very familiar with, for instance XQuery. It would take up too much space and distract you from the main topic to dive into the query in full, so I will explain it only briefly. If the query and the explanation goes over your head, don't feel too bad about it. As long you understand the output, you can still have use for the query.

The query uses two CTEs (Common Table Expression). The first CTE, **basedata**, includes all access to DMVs. We have already seen all of them but **sys.dm_exec_text_query_plan**. There are two more columns we retrieve from **sys.dm_exec_query_stats**, to wit **statement_start_offset** and **statement_end_offset**. They delimit the statement for this row, and we pass them to **sys.dm_exec_text_query_plan** to get the plan for this statement only. (Recall that the procedure is a single cache entry with a single **plan_handle**.) **sys.dm_exec_text_query_plan** returns the column **query_plan** which contrary to what you may expect is **nvarchar(MAX)**. The reason for this is that the XML for a query plan may be so deeply nested, that it cannot be represented with SQL Server's built-in **xml** data type. The CTE returns the query plan as such, but it also extracts the positions for the part in the document where the parameter values appear.

In the next CTE, **next_level**, I go on and use the values obtained in **basedata**. The CASE expression extracts the statement from the text returned by **sys.dm_exec_sql_text**. The way to do this is fairly clunky, not the least with those long column names. Since there is little reason to modify that part of the query, I say no more but refer you to Books Online. Or just believe me when I say it works. :-) The next column in the CTE, **params**, performs the actual extraction of the parameter values from the query-plan document and converts that element to the **xml** data type.

In the final SELECT, I shred the **params** document, so that we get one row per parameter. It can certainly be argued that it is better to have all parameters on a single row, since in this case each statement will only appear once, and here is a variation of the final SELECT that uses more XML functionality to achieve the string aggregation:

```
SELECT set_options AS [SET], n.stmt_start AS Pos, n.Statement,
       (SELECT CR.c.value('@Column', 'nvarchar(128)') + ' = ' +
               CR.c.value('@ParameterCompiledValue', 'nvarchar(512)') + ' '
        FROM   n.params.nodes('ColumnReference') AS CR(c)
        FOR XML PATH(''), TYPE).value('.', 'nvarchar(MAX)'),
       CAST (query_plan AS xml) AS [Query plan]
FROM   next_level n
ORDER  BY n.set_options, n.stmt_start
```

In the final SELECT, I also convert the query-plan column to XML, but as noted above, this could fail because of limitations with the **xml** data type. If you get such an error, just comment out that column, or change CAST to TRY_CAST if you are on SQL 2012 or higher. (TRY_CAST returns NULL if the conversion fails.)

Beside the alterations I have already mentioned, there are several ways you could modify the query to retrieve information you find interesting. For instance, you could add more columns from **sys.dm_exec_query_stats** or more plan attributes. I opted to include the **set_options** attribute only, since this is the cache key which is most likely to vary. If you would like to include all statements in the procedure, including those that do not refer to any of the input parameters, just change CROSS APPLY on the next-to-last line to OUTER APPLY.

4.5 Live Query Plan

Returning to Management Studio, there is one more option to see query plans and that is a *Live Query Plan*. The situation when you would use a live query plan is when the query does not complete in reasonable time and you want more information than the estimated plan gives. With a live query plan, the actual values in the plan are updated as the query progresses. You can see both solid lines and dotted lines. I believe the theory is that solid lines represents parts of the plan that are supposed to be completed, while dotted ones are where data is still flowing. But looking at a live query plan for a big a slow query, I can't really piece it together. Certainly an operator is still working if the row count below it is increasing. Nevertheless, hovering over the operators, you can inspect the current actual values, and you can see if there are gross deviations from the estimates.

Live Query Plan is a relatively new feature. It was introduced in SSMS 16, that is the first free version of SSMS that is available as a separate download. (Which I recommend you to use, even if you are on an older version of SQL Server.) Furthermore, there are some requirements on the SQL Server side as noted below.

There are two ways you can see a live query plan in SSMS. One is to run a query from a query window and enable *Include Live Query Statistics* from the *Query* menu. This requires that you are connected to SQL 2014 or later.

If you did not think of enabling live query statistics before starting the query, or you have the query running in the application right now, a second option is to use Activity Monitor (found in Object Explorer, by right-clicking the Server node itself.) Open it and find the pane *Active Expensive Queries*. You are likely to find your slow query here. You can right-click the query, and the option *Show Execution Plan* is always there. That gives you the estimated execution plan. If the *lightweight execution profiling infrastructure* is enabled, the option *Show Live Execution Plan* is also available. This infrastructure is available if any of these are true:

- You are on SQL 2019 or later.
- You are on SQL 2017 or SQL 2016 SP1 and up, and trace flag 7412 is enabled. (It's not a bad idea to have this trace flag set as a startup parameter for SQL Server.)
- You are SQL 2017, SQL 2016 SP1, or SQL 2014 SP2 or later and there is an active Extended Events session that includes the event **query_thread_profile**.

Finally, I like to mention that you can use my **beta_lockinfo** in this situation. It will return with the actual values so far when lightweight execution profiling infrastructure is enabled.

4.6 Getting Query Plans and Parameters from a Trace

Yet another alternative to get hold of the query plans is to run a trace against the application or against your connection in SSMS. There are several Showplan events you can include in a trace. The most versatile is **Showplan XML Statistics Profile** which gives you the same information as you see in SSMS when you enable *Include Actual Execution Plan*.

However, for several reasons a trace is rarely a very good alternative. To start with, enabling query-plan information in a trace adds a lot of overhead to generate that XML information. And observe that this applies even if you narrow your filter to a single spid. The way the trace engine works, all processes still have to generate the event, so this can have severe impact on a busy server.

Next, if you run the trace in Profiler, you are likely to find it very difficult to set up a good filter that captures what you want to see but hides all the noise. One possibility, once the trace has completed, is to save the trace to a table in the database, which permits you to find the interesting information through queries. (But don't ask Profiler to save to a table while the trace is running. The overhead for that is awful.) The plan is in the **TextData** column. Cast it to **xml** and then you can view it as I described in the previous section.

A slightly better alternative is to use Lee Tudor's **sp_sqltrace** that I mentioned earlier. It has a parameter to request that query plans should be collected, and you can opt to collect only estimated plans or actual plans. The overall performance on the server is still impacted, but at least you can find the plan you are looking for easily. However, **sp_sqltrace** will not work for you if you want to look at an application that is using multiple spids.

You can also use Extended Events to get hold of the query plan by capturing the event **query_post_execution_showplan**, but it is not any better than Trace. Even if you filter you event session for a specific spid, SQL Server activates this event for all processes, so the overall performance is affected in this case as well.

Note: Starting with SQL 2016 SP2 CU3 and SQL 2017 CU11, there is an extended event **query_plan_profile** which does not have this problem. However, this event only fires if the query has the hint QUERY_PLAN_PROFILE. Which of course your random slow statement in your application will not have. You can inject it with a plan guide, something I discuss later in this text. But it goes without saying that this is a horrendeously complicated approach, and I have not tried it myself.

4.7 Getting Table and Index Definitions

I assume that you are already acquainted with ways to find out how a table is defined, either with **sp_help** or through scripting, so I jump directly to the topic of indexes. They too can be scripted or you can use **sp_helpindex**. But scripting is bulky in my opinion, and **sp_helpindex** does not support features added in SQL 2005 or later. This query can be helpful:

```
DECLARE @tbl nvarchar(265)
SELECT @tbl = 'Orders'

SELECT o.name, i.index_id, i.name, i.type_desc,
       substring(ikey.cols, 3, len(ikey.cols)) AS key_cols,
       substring(inc.cols, 3, len(inc.cols)) AS included_cols,
       stats_date(o.object_id, i.index_id) AS stats_date,
       i.filter_definition
FROM   sys.objects o
JOIN   sys.indexes i ON i.object_id = o.object_id
CROSS  APPLY (SELECT ', ' + c.name +
                     CASE ic.is_descending_key
```

```
                             WHEN 1 THEN ' DESC'
                             ELSE ''
                        END
                 FROM   sys.index_columns ic
                 JOIN   sys.columns c ON ic.object_id = c.object_id
                                     AND ic.column_id = c.column_id
                 WHERE  ic.object_id = i.object_id
                   AND  ic.index_id  = i.index_id
                   AND  ic.is_included_column = 0
                 ORDER  BY ic.key_ordinal
                 FOR XML PATH('')) AS ikey(cols)
OUTER  APPLY (SELECT ', ' + c.name
                 FROM   sys.index_columns ic
                 JOIN   sys.columns c ON ic.object_id = c.object_id
                                     AND ic.column_id = c.column_id
                 WHERE  ic.object_id = i.object_id
                   AND  ic.index_id  = i.index_id
                   AND  ic.is_included_column = 1
                 ORDER  BY ic.index_column_id
                 FOR XML PATH('')) AS inc(cols)
WHERE  o.name = @tbl
  AND  i.type IN (1, 2)
ORDER  BY o.name, i.index_id
```

As listed, the query will not run on SQL 2005, but just remove the final column, **filter_definition**, from the result set. This column applies to filtered indexes, a feature added in SQL 2008. As for the column **stats_date**, see the next section.

The query only lists regular relational indexes, not XML indexes or spatial indexes. Problems related to searches in XML documents or spatial columns are beyond the scope for this article anyway. Nor have I mustered the energy to support columnstore indexes, but that is left as an exercise to the reader.

4.8 Finding Information About Statistics

To view all statistics for a table, you can use this query:

```
DECLARE @tbl nvarchar(265)
SELECT @tbl = 'Orders'

SELECT o.name, s.stats_id, s.name, s.auto_created, s.user_created,
       substring(scols.cols, 3, len(scols.cols)) AS stat_cols,
       stats_date(o.object_id, s.stats_id) AS stats_date,
       s.filter_definition
FROM   sys.objects o
JOIN   sys.stats s ON s.object_id = o.object_id
CROSS  APPLY (SELECT ', ' + c.name
              FROM   sys.stats_columns sc
              JOIN   sys.columns c ON sc.object_id = c.object_id
                                  AND sc.column_id = c.column_id
              WHERE  sc.object_id = s.object_id
                AND  sc.stats_id  = s.stats_id
              ORDER  BY sc.stats_column_id
              FOR XML PATH('')) AS scols(cols)
WHERE  o.name = @tbl
ORDER  BY o.name, s.stats_id
```

As with the query for indexes, the query does not run for SQL 2005 as listed, but just remove **filter_definition** from the result set. **auto_created** refers to statistics that SQL Server creates automatically when it gets the occasion, while **user_created** refers to indexes created explicitly with CREATE STATISTICS. If both are 0, the statistics exists because of an index.

The column **stats_date** returns when the statistics most recently was updated. If the date is way back in the past, the statistics may be out of date. The root cause to parameter-sniffing-related problems is usually something else than outdated statistics, but it is always a good idea to look out for this. One thing to keep in mind is that statistics for columns with monotonically increasing data – e.g. id and date columns – quickly go out of date, because queries are often for the most recently inserted data, which is always beyond the last slot in the histogram (more about histograms later).

If you believe statistics are out of date for a table, you can use this command:

```
UPDATE STATISTICS tbl
```

This will give you sampled statistics. Often this gives you statistics that are good enough, but sometimes the sampling does not work out well. In this case, it can be worth forcing a full scan of the data. This may be best done with this command:

```
UPDATE STATISTICS tbl WITH FULLSCAN, INDEX
```

By adding INDEX to the command, the FULLSCAN update is only performed for statistics for indexes. This can reduce the execution time for UPDATE STATISTICS since for non-index statistics, UPDATE STATISTICS scans the entire table for each statistics. (Whereas for the indexes, it scans the leaf level of the index which is typically a lot smaller.)

You can also update the statistics for a single index. The syntax for this is not what you may expect:

```
UPDATE STATISTICS tbl indexname WITH FULLSCAN
```

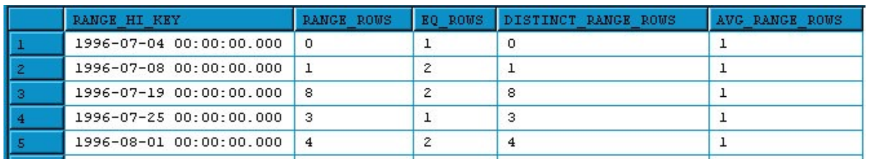
Note: there is no period between the table name and the index name, just space.

Note that after updating statistics, you may see an immediate performance improvement in the application. This does not necessarily prove that outdated statistics was the problem. Since the updated statistics causes recompilation, the parameters may be re-sniffed and you get a better plan for this reason.

To see the distribution statistics for an index use DBCC SHOW_STATISTICS. This command takes two parameters. The first is the table name, whereas the second can be the name of an index or statistics. For instance:

```
DBCC SHOW_STATISTICS (Orders, OrderDate)
```

This displays three result sets. I will not cover all of them here, but only say that the last result set is the actual histogram for the statistics. The histogram reflects the distribution that SQL Server has recorded about the data in the table. Here is how the first few lines look in the example:



	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	1996-07-04 00:00:00.000	0	1	0	1
2	1996-07-08 00:00:00.000	1	2	1	1
3	1996-07-19 00:00:00.000	8	2	8	1
4	1996-07-25 00:00:00.000	3	1	3	1
5	1996-08-01 00:00:00.000	4	2	4	1

This tells us that according to the statistics there is exactly one row with **OrderDate** = 1996-07-04. Then there is one row in the range from 1996-07-05 to 1996-07-07 and two rows with **OrderDate** = 1996-07-08. (Because RANGE_ROWS is 1 and EQ_ROWS is 2.) The histogram then continues, and there is in total 188 steps for this particular statistics. There are never more than 200 steps in a histogram. For full details on the output, please see the topic for DBCC SHOW_STATISTICS in Books Online. One of the white papers listed in the section *Further Reading* has more valuable information about statistics and the histogram.

Note: In SQL 2005 there is a bug, so if there is a step for NULL values, DBCC SHOW_STATISTICS fails to show this row. This is a display error, and the value is still there in the histogram.

You do not typically run DBCC SHOW_STATISTICS for all statistics to have the information just in case, but only when you think that the information may be useful to you. We will look at such an example in the next chapter.

For auto-created statistics the name is something like **_WA_Sys_00000003_42E1EEFE**. This is somewhat cumbersome to use, so in this case DBCC SHOW_STATISTICS permits you to specify the column name instead. Note that you cannot use the column name if there are user-created statistics on the column (through CREATE INDEX or CREATE STATISTICS). The above example appears to contradict that, since there is an index on the **OrderDate** column. But that index is also called **OrderDate**!

5. Examples of How to Fix Parameter-Sniffing Issues

It is important to understand that parameter sniffing in itself is not a problem; au contraire, it is a feature, since without it SQL Server would have to rely on blind assumptions, which in most cases would lead to less optimal query plans. But sometimes parameter sniffing works against you. We can identify three typical situations:

1. The query usage is such that parameter sniffing is entirely inappropriate. That is, a plan which is good for a certain execution may be poor for the next.
2. There is a specific pattern in the application where one group of calls is very different from the main bulk. Often this is a call the application performs on start-up or at the beginning of a new day.
3. The index structure for one or more tables is such that there is no perfect index for the query, but there are several half-good indexes, and it is haphazard which index the optimizer chooses.

It can be difficult to say beforehand which applies to your situation, and that is why you need to make a careful analysis. In the previous section I discussed what information you need, although I did not always make it clear what for. In addition, there is one more thing I did not list but which is immensely helpful: intimate knowledge about how the application works and its usage pattern.

Since there are several possible reasons why parameter sniffing could give you a headache, this means that there is no single solution that you can apply. Rather there is a host of them, depending on where the root cause lies. In the following I will give some examples of parameter-sniffing related issues and how to address them. Some are real-life examples, others are more generic in nature. Some of the examples focus more on the analysis, others take a straight look at the solution.

5.1 A Non-Solution

Before I go into the real solutions, let me first point out that adding SET ARITHABORT ON to your procedure is **not** a solution. It will seem to work when you try it. But that is only because you recreated the procedure which forced a new compilation and then the next invocation sniffed the current set of parameters. SET ARITHABORT ON is only a placebo, and not even a good one. The problem will most likely come back. It will not even help you avoid the confusion with different performance in the application and SSMS, because the overall cache entry will still have ARITHABORT OFF as its plan attribute.

So, don't put SET ARITHABORT ON in your stored procedures. Overall, I strongly discourage from you using any of the SET commands that are cache keys in your code.

5.2 Best Index Depends on Input

Consider this procedure:

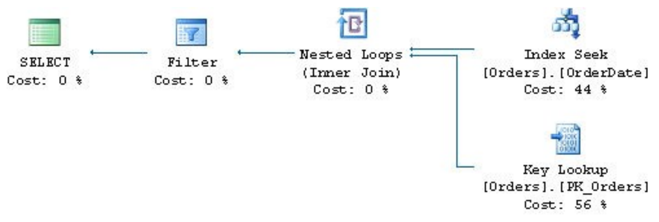
```
CREATE PROCEDURE List_orders_12 @custid   nchar(5),
                                @fromdate datetime,
                                @todate   datetime AS
   SELECT *
   FROM   Orders
   WHERE  CustomerID = @custid
     AND  OrderDate BETWEEN @fromdate AND @todate
```

There is a non-clustered index on **CustomerID** and another one on **OrderDate**. Assume that the order activity among customers varies vividly. Many customers make just a handful a orders per year. But some customers are more active, and some real big guys may place several orders per day.

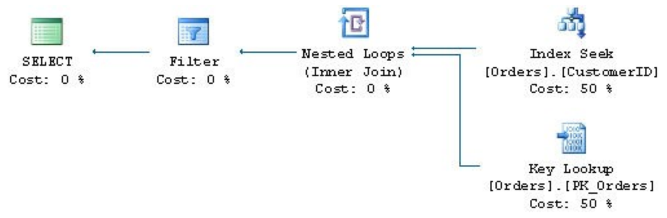
In the **Northwind** database, the most active customer is SAVEA with 31 orders, whereas CENTC has only one order. Run the below:

```
EXEC List_orders_12 'SAVEA', '19970811', '19970811'
go
sp_recompile List_orders_12
go
EXEC List_orders_12 'CENTC', '19960101', '19961231'
```

That is, for SAVEA we only look at the orders for a single day, but for CENTC we look at the orders for an entire year. As you may sense, these two invocations are best served by different indexes. Here is the plan for the first invocation:



SQL Server here uses the index for **OrderDate** which is the most selective. The plan for the second invocation is different:



Here **CustomerID** is the most selective column, and SQL Server uses the index on **CustomerID**.

One solution to address this is to force recompilation every time with the RECOMPILE query hint:

```
CREATE PROCEDURE List_orders_12 @custid   nchar(5),
                                @fromdate datetime,
                                @todate   datetime AS
SELECT *
FROM   Orders
WHERE  CustomerID = @custid
  AND  OrderDate BETWEEN @fromdate AND @todate
OPTION (RECOMPILE)
```

With this hint, SQL Server will compile the query every time and since it knows that the plan is not to be reused, it handles the values of the parameters **and** local variables as constants.

Note: This is true if you have SQL 2012 or the latest service pack of SQL 2008 and SQL 2008 R2. On SQL 2005 and early builds of SQL 2008 and SQL 2008 R2, OPTION (RECOMPILE) does not handle the variables as constants. However, in this particular example, OPTION (RECOMPILE) serves the purposes also on the earlier versions.

In many cases, forcing recompilation every time is quite alright, but there are a few situations where it is not:

1. The procedure is called with a very high frequency, and the compilation overhead hurts the system.
2. The query is very complex and the compilation time has a noticeable negative impact on the response time.

More to the point, while forcing recompilation is a solution that is almost always feasible, it is not always the best solution. As a matter of fact, the example we have looked at in this section is probably not very typical for the situation *fast in SSMS, slow in the application*. Because, if you have varying usage pattern, you will be alerted of varying performance within the application itself. So there is all reason to read on, and see if the situation you are facing fits with the other examples I present.

5.3 Dynamic Search Conditions

It is very common to have forms where users can select from a number of search conditions. For instance, they can select to see orders on a certain date, by a certain customer, for a certain product etc, including combinations of the parameters. Such procedures are sometimes implemented with a WHERE clauses that goes:

```
WHERE (CustomerID = @custid OR @custid IS NULL)
  AND (OrderDate = @orderdate OR @orderdate IS NULL)
  ...
```

As you may imagine, parameter sniffing is not beneficial for such procedures. I am not going to take up much space on this problem here, for two reasons: 1) As I've already said, problem with such procedures usually manifests itself with varying performance within the application. 2) I have a separate article devoted to this topic, entitled *Dynamic Search Conditions in T-SQL*. Short story: OPTION (RECOMPILE) often works very well here.

5.4 Reviewing Indexing

Some time ago, one of my clients contacted me because one of their customers experienced a severe performance problem with a function in their system. My client said that the same code ran well at other sites, and there had been no change recently to the application. (But you know, clients always say that, it seems.) They had been able to isolate the problematic procedure, which included a query which looked something like this:

```
SELECT DISTINCT c.*
FROM   Table_C c
JOIN   Table_B b ON c.Col1 = b.Col2
JOIN   Table_A a ON a.Col4 = b.Col1
WHERE  a.Col1 = @p1
  AND  a.Col2 = @p2
  AND  a.Col3 = @p3
```

When executed from the application, the query took 10-15 minutes. When they ran the procedure from SSMS, they found that response time was instant. That was then they called me.

An account was set up for me to make it possible to log in to the site in question. I found that all three tables were of some size, at least a million rows in each. I looked at the indexes for **Table_A**, and I found that it had some 7-8 indexes. Of interest for this query was:

- One non-clustered, non-unique index **Combo_ix** on (**Col1**, **Col2**, **Col5**, **Col4**) and maybe some more columns.
- One non-clustered, non-unique index **Col2_ix** on (**Col2**).
- One non-clustered, non-unique index **Col3_ix** on (**Col3**).

Thus, where was no index that covered all conditions in the WHERE clause.

When I ran the procedure in SSMS with default settings, the optimizer chose the first index to get data from **Table_A**. When I changed the setting of ARITHABORT to OFF to match the application, I saw a plan that used the index on **Col3**.

At this point I ran

```
DBCC SHOW_STATISTICS (Table_A, Col3_ix)
```

The output looked like something like this:

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
APRICOT	0	17	0	1
BANANA	0	123462	0	1

BLUEBERRY	0	46	0	1
CHERRY	0	92541	0	1
FIG	0	1351	0	1
KIWI	0	421121	0	1
LEMON	0	6543	0	1
LIME	0	122131	0	1
MANGO	0	95824	0	1
ORANGE	0	10410	0	1
PEAR	0	46512	0	1
PINEAPPLE	0	21102	0	1
PLUM	0	13	0	1
RASPBERRY	0	95	0	1
RHUBARB	0	7416	0	1
STRAWBERRY	0	24611	0	1

That is, there were only 17 distinct values for this column and with a very uneven distribution among these values. I also confirmed this fact by running the query:

```
SELECT Col3, COUNT(*) FROM Table_A GROUP BY Col3 ORDER BY Col3
```

I proceeded to look at the bad query plan to see what value that the optimizer had sniffed for **@p3**. I found that it was – APPLE, a value not present in the table at all! That is, the first time the procedure was executed, SQL Server had estimated that the query would return a single row (recall that it never estimates zero rows), and that the index on **Col3** would be the most efficient index to find that single row.

Now, you may ask how it could be that unfortunate that the procedure was first executed for APPLE? Was that just bad luck? Since I don't know this system, I can't tell for sure, but apparently this had happened more than once. I got the impression that this procedure was executed many times as part of some bigger operation. Say that the operation would always start with APPLE. Remember that reindexing a table triggers recompilation, and it is very common to run maintenance jobs at night to keep fragmentation in check. That is, the first call in the morning can be very decisive for your performance. (Why would the operation start with a value that is not in the database? Who knows, but maybe APPLE is an unusual condition that needs to be handled first if it exists. Or maybe it is just the alphabet.)

For this particular query, there is a whole slew of possible measures to address the performance issue.

1. OPTION (RECOMPILE)
2. Add the "optimal" index on (**Col1**, **Col2**, **Col3**) INCLUDE (**Col4**).
3. Make the index on **Col3** filtered or drop it entirely.
4. Use an index hint to force use of any of the other indexes.
5. The query hint OPTIMIZE FOR.
6. Copy **@p3** to a local variable.
7. Change the application behaviour.

We have already looked at forcing recompilation, and while it most probably would have solved the problem, it is not likely that it would have been the best solution. Below, I will look into the other options in more detail.

Add a New Index

My primary recommendation to the customer was the second on the list: add an index that matches the query perfectly. That is, an index where all columns in the WHERE clauses are index keys, with the least selective column **Col3** last among the keys. On top of that, add **Col4** as an included column, so that the query can be resolved from the new index alone, without any need to access the data pages.

However, adding a new index is not always a good solution. If you have a table which is accessed in a multitude of ways, it may not be feasible to add indexes that matches all WHERE and JOIN conditions, even less to add covering indexes for each and every query. Particularly, this applies to tables with a heavy update rate, like an **Orders** table. The more indexes you add to a table, the higher the cost to insert, update and delete rows in the table.

Change / Drop the Index on Col3

How useful is the index on **Col3** really? Since I don't know this system it is difficult to tell. But generally, indexes on columns with only a small set of values are not very selective and thus not that useful. So an option could be to drop the index on **Col3** altogether and thereby save the optimiser from this trap. Maybe the index was added at some point in time by mistake, or it was added without understanding of how the database would evolve by time. (Having worked a little more with this client, it seems that they add indexes on all FK columns as a matter of routine. Which may or may not be a good idea.)

It cannot be denied, that you have to be a very brave person to drop an index entirely. You can consult **sys.dm_db_index_usage_stats** to see how much the index is used. Just keep in mind that this DMV is cleared when SQL Server is restarted or the database is taken offline.

Since the distribution in **Col3** is so uneven, it is not unlikely that there are queries that look for these rare values specifically. Maybe FIG is "New unprocessed rows". Maybe RASPBERRY means "errors". In this case, it could be beneficial to make **Col3_ix** a *filtered index*, a feature added in SQL 2008. For instance, the index definition could read:

```
CREATE INDEX col3_ix ON Table_A(col3) WHERE col3 IN ('FIG', 'RASPBERRY', 'APPLE', 'APRICOT')
```

This has two benefits:

1. The size of the index is reduced with more than 99%.
2. The index is no longer eligible for the problem query. Recall that SQL Server must select a plan which is correct for all input values, so even if the sniffed parameter value is APPLE, SQL Server cannot use the index, because the plan would yield incorrect result for KIWI.

Force a Different Index

If you know that the index on **Col1**, **Col2** will always be the best index, but you don't want to add or drop any index, you can force the index with:

```
SELECT c.*
FROM   Table_C c
JOIN   Table_B b ON c.Col1 = b.Ccol2
JOIN   Table_A a WITH (INDEX = Combo_ix) ON a.Col4 = b.Col1
WHERE  a.Col1 = @p1
  AND  a.Col2 = @p2
  AND  a.Col3 = @p3
```

You can even say:

```
WITH (INDEX (combo_ix, col2_ix))
```

to give the optimizer the choice between the two "good" indexes.

Index hints are very popular, too popular one might say. You should think twice before you add an index hint, because tomorrow your data distribution may be different, and another index would serve the query better. A second problem is that if you later decide to rearrange the indexes, the query will fail with an error because of the missing index.

OPTIMIZE FOR

OPTIMIZE FOR is a query hint that permits you to control parameter sniffing. For the example query, it could look like this:

```
SELECT c.*
FROM   Table_C c
JOIN   Table_B b ON c.col1 = b.col2
JOIN   Table_A a ON a.col4 = b.col1
WHERE  a.col1 = @p1
  AND  a.col2 = @p2
  AND  a.col3 = @p3
OPTION (OPTIMIZE FOR (@p3 = 'KIWI'))
```

The hint tells SQL Server to ignore the input value, but instead compile the query as if the input value of **@p3** is KIWI, the most common value in **Col3**. This will surely dissuade SQL Server from using the index.

A second option, available in SQL 2008 and later, is:

```
OPTION (OPTIMIZE FOR (@p3 UNKNOWN))
```

Rather than hard-coding any particular value, we can tell SQL Server to make a blind assumption to completely kill parameter sniffing for **@p3**.

It is worth adding that you can use OPTIMIZE FOR also with locally declared variables, and not only with parameters.

Copy @p3 to a Local Variable

Rather than using **@p3** directly in the query, you can copy it to a local variable, and then use the variable in the query. This has the same effect as OPTIMIZE FOR UNKNOWN. And it works on any version of SQL Server.

Change the Application

Yet another option is to change the processing in the application so that it starts with one of the more common values. It's not really a solution I recommend, because it creates an extra dependency between the application and the database. There is risk that the application two years later is rewritten to accommodate new requirements that PEACH rows must be handled, and this handling is added first...

Then again, there may be a general flaw in the process. Is the application asking for values for one fruit at a time, when it should be getting values for all fruits at once? I helped this client with another query. We had a query-tuning session in a conference room, and I was never able to get really good performance from the query we were working with. But towards the end of the day, the responsible developer said that he knew how to continue, and his solution was to rearchitect the entire process the problematic query was part of.

Summing it Up

As you have seen in this example, there are a lot of options to choose from – too many, you may think. But performance tuning often requires you to have a well-stuffed bag of tricks, because different problems call for different solutions.

5.5 The Case of the Application Cache

In the system I work with most of my time, there is a kind of application cache that keeps data in a main-memory database, let's call it MemDb. It is used for several purposes, but the main purpose is to serve as a cache from which the customer's web server can retrieve data rather than querying the database. Typically, in the morning there is a total refresh of the data. When data in a database table is updated, there is a signalling mechanism which triggers MemDb to run a stored procedure to get the delta. In order to find what has changed, MemDb relies on **timestamp** columns. (A **timestamp** column holds an 8-byte value that is unique to the database and which grows monotonically. It is automatically updated when the row is inserted or updated. This data type is also known as **rowversion**.) A typical stored procedure to retrieve changes looks like this:

```
CREATE PROCEDURE memdb_get_updated_customers @tstamp timestamp AS
  SELECT CustomerID, CustomerName, Address, ..., tstamp
  FROM   Customers
  WHERE  tstamp > @tstamp
```

When MemDb calls these procedures, it passes the highest value in the **tstamp** column for the table in question from the previous call. When the procedure is invoked for a refresh, the main-memory database passes 0x as the parameter to get all rows.

Side note: The actual scheme is more complicated than the above; I have simplified it to focus on what is important for this article. If you are considering something similar, be warned that as shown, this example has lots of issues with concurrent updates, not the least if you use any form of snapshot isolation. SQL 2008 introduced Change Tracking which is a more solid solution, particularly designed for this purpose. I also like to add that MemDb was not my idea, nor was I involved in the design of it.

At one occasion when I monitored the performance for a customer that just had went live with our system, I noticed that the MemDb procedures had quite long execution times. Since they run very often to read deltas, they have to be very quick. After all, one idea with MemDb is to take load off from the database – not add to it.

For a query like the above to be fast there has to be an index on **tstamp**, but will this index be used? From what I said above, the first thing in the morning, MemDb would run:

```
EXEC memdb_get_updated_customers 0x
```

Then a little later, it would run something like:

```
EXEC memdb_get_updated_customers 0x000000000003E806
```

It is not uncommon that during the night that query plans fall out of the cache because of nightly batches that consume a lot of memory. Or there is a maintenance job to rebuild indexes which triggers recompiles. So typically, when the morning refresh runs, there is not any plan in the cache, and the value sniffed is 0x. Given this value, will the optimizer use the index on **tstamp**? Yes, if it is the clustered index. But since a **timestamp** column is updated every time a row is updated, it is not a very good choice for the clustered index, and all our indexes on **timestamp** columns are non-clustered. (And for tables with a high update frequency, also a non-clustered index on a **timestamp** column may be questionable.) Thus, since the optimizer sees that the parameter indicates that all rows in the table will be retrieved, it settles for a table scan. This plan is put into cache, and subsequent calls also scan the table, even if they are only looking for the most recent rows. And it was this poor performance that I saw.

When you have a situation like this, there are, just like in the previous example, several ways to skin the cat. But before we look at the possibilities, we need to make one important observation: there is no query plan that is good for both cases. We want to use the index to read the deltas, but when making the refresh we want the scan. Thus, only solutions that can yield different plans for the two cases need to apply.

OPTION (RECOMPILE)

While this solution meets the requirement, it is not a good solution. This means that every time we retrieve a delta, there is compilation. And while the above procedure is simple, some of the MemDb procedures are decently complex with a non-trivial cost for compilation.

EXECUTE WITH RECOMPILE

Rather than requesting recompilation inside the procedure, it is better to do it in the special case: the refresh. The refresh only happens once a day normally. And furthermore, the refresh is fairly costly in itself, so the cost of recompilation is marginal in this case. That is, when MemDb wants to make a refresh, it should call the procedure in this way:

```
EXECUTE memdb_get_updated_customers WITH RECOMPILE
```

As I noted previously, it is likely that there is no plan in the cache early in the morning, so you may ask what's the point? The point is that when you use WITH RECOMPILE with EXECUTE, the plan is *not* put into cache. Thus, the refresh can run with the scan, but the plan in the cache will be built from the first delta retrieval. (And if the plan for reading the delta still is in the cache, that plan will remain in the cache.)

For the particular problem in our system, this was the solution I mandated. An extra advantage for us was that there was a single code path in MemDb that had to be changed.

There is however a slight problem with this solution. Normally, when you call a stored procedure from a client, you use a special command type where you specify only the name of the procedure. (For instance, **CommandType.StoredProcedure** in ADO .NET.) The client API then makes an RPC (remote procedure call) to SQL Server, and there is never an EXECUTE statement as such. Very few client APIs seem to expose any method to specify WITH RECOMPILE when you call a procedure through RPC. The workaround is to send the full EXECUTE command, using **CommandType.Text** or similar, for instance:

```
cmd.CommandType = CommandType.Text;
cmd.Text = "EXECUTE memdb_get_updated_customers @tstamp WITH RECOMPILE";
```

Note: You should pass the parameters through the **Parameters** collection or corresponding mechanism in your API; don't inline the values in the string with the EXEC command, as this could open the door for SQL injection.

Using a Wrapper Procedure

If you have a situation where you realise that EXECUTE WITH RECOMPILE is the best solution, but it is not feasible to change the client, you can introduce a wrapper procedure. In our example the original procedure would be renamed to **memdb_get_updated_customers_inner**, and then you would write a wrapper that goes:

```
CREATE PROCEDURE memdb_get_updated_customers @tstamp timestamp AS
   IF @tstamp = 0x
      EXECUTE memdb_get_updated_customers_inner @tstamp WITH RECOMPILE
   ELSE
      EXECUTE memdb_get_updated_customers_inner @tstamp
```

In many cases this can be a plain and simple solution, particularly if you have a small number of such procedures. (We have many.)

Different Code Paths

Another approach would be to have different code paths for the two cases:

```
CREATE PROCEDURE memdb_get_updated_customers @tstamp timestamp AS
   IF @tstamp = 0x
   BEGIN
      SELECT CustomerID, CustomerName, Address, ..., tstamp
      FROM   Customers
   END
   ELSE
   BEGIN
      SELECT CustomerID, CustomerName, Address, ..., tstamp
      FROM   Customers WITH (INDEX = timestamp_ix)
      WHERE  tstamp > @tstamp
   END
```

Note that it is important to force the index in the ELSE branch, or else this branch will scan the table if the first call to the procedure is for **@tstamp** = 0x because of parameter sniffing. If your procedure is more complex and includes joins to other tables, it is not likely that this strategy will work out, even if you force the index on **tstamp**. The estimates for the joins would be grossly incorrect and the query plans would still be optimized to get all data, and not the delta.

Different Procedures

In the situation we had, there was one procedure that was particularly difficult. It retrieved account transactions (this is a system for stock trading). The refresh would not retrieve all transactions in the database, but only from the N most recent days, where N was a system parameter read from the database. In this database N was 20. The procedure did not read from a single table, but there were umpteen other tables in the query. Rather than having a **timestamp** parameter, the parameter was an id. This works since transactions are never updated, so MemDb only needs to look for new transactions.

I found that in this case EXECUTE WITH RECOMPILE alone would not save the show, because there were several other problems in the procedure. I found no choice but to have two procedures, one for the refresh and one for the delta. I replaced the original procedure with a wrapper:

```
CREATE PROCEDURE memdb_get_transactions @transid int AS
   IF coalesce(@transid, 0) = 0
      EXECUTE memdb_get_transactions_refresh
   ELSE
   BEGIN
      DECLARE @maxtransid int
      SELECT @maxtransid = MAX(transid) FROM transactions
      EXECUTE memdb_get_transactions_delta @transid, @maxtransid
   END
```

I had to add **@maxtransid** as a parameter to the delta procedure, because with an open condition like `WHERE transid > @transid`, SQL Server would tend to miss-estimate the number of rows it had to read. Making the interval closed addressed that issue, and by passing **@maxtransid** as a parameter, SQL Server could sniff the value.

From a maintainability perspective, this is by no means a pleasant step to take, not the least when the logic is as complicated as in this case. If you have to do this, it is important that you litter the code with comments to tell people that if they change one procedure, they need to change the other one as well.

Note: a few years later, a colleague rewrote **memdb_get_transactions_refresh** to become a second wrapper. As I mentioned, it reads some system parameters to determine which transactions to read. To get the refresh to perform decently, he found that he had to pass the values of these system parameters, as well as the interval of transactions ids to read as parameters to an inner procedure to get full benefit of parameter sniffing. (We were still on SQL 2000 at the time. In SQL 2005 or later OPTION (RECOMPILE) would have achieved the same thing.)

5.6 Fixing Bad SQL

There may also be situations where the root cause to the problem is simply poorly written SQL. As just one example, consider this query:

```
SELECT ...
FROM   Orders
WHERE  (CustomerID = @custid OR @custid IS NULL)
  AND  (EmployeeID = @empid  OR @empid IS NULL)
  AND  convert(varchar, OrderDate, 101) = convert(varchar, @orderdate, 101)
```

The idea is that users here are able to search orders for a given date, and optionally they can also specify other search parameters. The programmer here considers that **OrderDate** may include a time portion, and factors it out by using a format code to **convert()** that produces a string without time, and to be really sure he performs the same operation on the input parameter. (In **Northwind**, **OrderDate** is always without a time portion, but for the sake of the example, I'm assuming that this is not the case.)

For this query, the index on **OrderDate** would be the obvious choice, but the way the query is written, SQL Server cannot seek that index, because **OrderDate** is entangled in an expression. This is sometimes referred to as the expression not being *sargable*, where *sarg* is short for *search argument*, that is, something that can be used as a seek predicate in a query. (Personally, I don't like the term "sargable", but since you may see people drop it from time to time, I figured that I should mention this piece of jargon.)

Because the index on **OrderDate** has been disqualified, the optimizer may settle for any of the other indexes, depending on the input for the first parameter, causing poor performance for other types of searches. The remedy is to rewrite the query, for instance like:

```
SELECT ...
FROM   Orders
WHERE  (CustomerID = @custid OR @custid IS NULL)
  AND  (EmployeeID = @empid  OR @empid IS NULL)
  AND  OrderDate >= @orderdate
  AND  OrderDate < dateadd(DAY, 1, @orderdate)
```

(It seems reasonable to assume that an input parameter which is supposed to be a date does not have any time portion, so there is little reason to litter the code with any extra conversion.)

Poorly written SQL often manifests itself in general performance problems – that is, the query always runs slow – and maybe not so often in situations where parameter sniffing matters. Nevertheless, when you battle your parameter-sniffing problem, there is all reason to investigate whether the query could be written in a way that avoids the dependency of the input parameter for a good plan. There are many ways to write bad SQL, and this is not the place the list them all. Hiding columns in an expression is just one example, albeit a common one. Sometimes the expression is hidden because it is due to implicit conversion, for instance when a string column which is supposed to contain numbers is compared to an integer value. Examine the plan, and when an index is not used as you would expect, go back and look at the query again.

6. Dynamic SQL

Hitherto we have only looked at stored procedures, and with stored procedures the reason for different performance in the application and in SSMS is most often due to different settings for SET ARITHABORT. If you have an application that does not use stored procedures, but generates the queries in the client or in the middle layer, there are a few more reasons why you may get a new cache entry when you run the query in SSMS and maybe also a different plan than in the application. In this chapter, we will look at these possibilities. We will also look at some solutions for addressing parameter-sniffing problems that mainly are applicable to dynamic SQL.

6.1 What Is Dynamic SQL?

Dynamic SQL is any SQL which is not part of a stored procedure (or any other type of module). This includes:

- SQL statements executed with EXEC() and **sp_executesql**.
- SQL statements sent directly from the client.
- SQL statements submitted from modules written in the SQLCLR.

Dynamic SQL comes in two flavours, unparameterised and parameterised. In unparameterised SQL, the programmer composes the SQL string by concatenating the language elements with the parameter values. For instance, in T-SQL:

```
SELECT @sql = 'SELECT mycol FROM tbl WHERE keycol = ' + convert(varchar, @value)
EXEC(@sql)
```

Or in C#:

```
cmd.CommandText = "SELECT mycol FROM tbl WHERE keycol = " + value.ToString();
```

Unparameterised SQL is **bad** for several reasons, please see my article *The Curse and Blessings of Dynamic SQL* for a discussion on good and practice with dynamic SQL.

In parameterised SQL, you pass parameters much like in a stored procedure. In T-SQL:

```
EXEC sp_executesql N'SELECT mycol FROM dbo.tbl WHERE keycol = @value',
                   N'@value int', @value = @value
```

Or in C#:

```
cmd.CommandText = "SELECT mycol FROM dbo.tbl WHERE keycol = @value";
cmd.Parameters.Add("@value", SqlDbType.Int);
cmd.Parameters["@value"].Value = value;
```

The C# code results in a call to **sp_executesql** which looks exactly like the T-SQL example above.

For more details on **sp_executesql**, pleases see my article *The Curse and Blessings of Dynamic SQL*.

6.2 The Query Text is the Hash Key

Query plans for dynamic SQL are put into the plan cache, just like plans for stored procedures. (If you hear someone telling you something else, that person is either just confused or relying on very old information. Up to SQL Server 6.5, SQL Server did not cache plans for dynamic SQL.) As with stored procedures, plans for dynamic SQL may be flushed from the cache for a number of reasons, and individual statements may be recompiled. Furthermore, there may be more than one plan for the same query text because of differences in SET options.

Adding to that, there are a few more factors that can introduce puzzling behaviour with parameter sniffing that do not apply to stored procedures which we will look at in the next few sections.

When SQL Server looks up a stored procedure in the cache, it uses the name of the procedure. But that is not possible with a batch of dynamic SQL, as there is no name. Instead, SQL Server computes a hash from the query text and uses this hash as a key in the plan cache. And here is something very important: this hash value is computed without any normalisation whatsoever of the batch text. Comments are not stripped. White space is not trimmed or collapsed. Case is not forced to upper or lowercase, even if the database has a case-insensitive collation. The hash is computed from the text exactly as submitted, and any small difference will yield a different hash and a different cache entry. (There is one exception, that I will return to when we discuss auto-parameterisation.)

Run this with *Include Actual Execution Plan* enabled:

```
EXEC sp_executesql N'SELECT * FROM Orders WHERE OrderDate > @orderdate',
                   N'@orderdate datetime', '20000101'
EXEC sp_executesql N'SELECT * FROM Orders WHERE OrderDate > @orderdate',
                   N'@orderdate datetime', '19980101'
EXEC sp_executesql N'select * from Orders where OrderDate > @orderdate',
                   N'@orderdate datetime', '19980101'
```

You will find that the first two calls use the same plan, Index Seek + Key Lookup, whereas the third query uses a Clustered Index Scan. That is, the second call reuses the plan created for the first call. But in the third call, the SQL keywords are in lowercase. Therefore, there is no cache hit, and a new plan is created. Just to enforce this fact, here is a second example with the same result.

```
DBCC FREEPROCCACHE
go
EXEC sp_executesql N'SELECT * FROM Orders WHERE OrderDate > @orderdate',
                   N'@orderdate datetime', '20000101'
EXEC sp_executesql N'SELECT * FROM Orders WHERE OrderDate > @orderdate',
                   N'@orderdate datetime', '19980101'
EXEC sp_executesql N'SELECT * FROM Orders WHERE OrderDate > @orderdate ',
                   N'@orderdate datetime', '19980101'
```

The difference is that there is a single trailing space in the third statement.

There is one more thing to observe. If you run this query:

```
SELECT '<' + est.text + '>'
FROM   sys.dm_exec_query_stats qs
CROSS  APPLY sys.dm_exec_sql_text(qs.sql_handle) est
WHERE  est.text LIKE '%Orders%'
```

You will see this output:

```
<(@orderdate datetime)SELECT * FROM Orders WHERE OrderDate > @orderdate>
<(@orderdate datetime)SELECT * FROM Orders WHERE OrderDate > @orderdate >
```

(The angle brackets serve to highlight the trailing space.) Notice here that the parameter list is also part of the query text, and it too is included in the query text that is hashed. Run this:

```
EXEC sp_executesql N'SELECT * FROM Orders WHERE OrderDate > @orderdate',
                   N'@orderdate  datetime', '19980101'
```

That is, the same query again, but with one more space in the parameter list. If you now run the DMV query again, you will see that it now returns three rows.

This detail about the parameter list may seem like a thing of more trivial nature, but it has an important implication if you are writing client code. Consider this snippet of C#:

```
cmd.CommandText = "SELECT * FROM dbo.Orders WHERE CustomerID = @c";
cmd.Parameters.Add("@c", SqlDbType.NVarChar).Value = TextBox.Value;
```

Note that the parameter is added without a specific length for the parameter value. Say that the value in the text box is ALFKI. This results in the following call to SQL Server:

```
exec sp_executesql N'SELECT * FROM Orders WHERE CustomerID = @c',
                   N'@c nvarchar(5)',@c=N'ALFKI'
```

Note that the parameter is declared as **nvarchar(5)**. That is, the length of the parameter is taken from the length of the actual value passed. This means that if you run the same batch multiple times with different parameter values of different length, you will get multiple cache entries. Not only do you litter the cache, but because of parameter sniffing, you can haphazardly get different plans for different parameter lengths. This can result in confusing behaviour where the same operation is slow for one value, but fast for another, although you would expect the same performance for these values.

There is a simple remedy for this particular problem: always specify the length explicitly:

```
cmd.Parameters.Add("@c", SqlDbType.NVarChar, 5).Value = TextBox.Value;
```

Here I used 5, since the **CustomerID** column is **nchar(5)**. If you don't want to create a dependency to the data model, you can specify the length as something higher, for instance 4000 (which is the maximum for a regular **nvarchar**).

6.3 The Significance of the Default Schema

Another difference to stored procedures is less obvious, and is best shown with an example. Run this and look at the execution plans:

```
DBCC FREEPROCCACHE
go
CREATE SCHEMA Schema2
go
CREATE USER User1 WITHOUT LOGIN WITH DEFAULT_SCHEMA = dbo
CREATE USER User2 WITHOUT LOGIN WITH DEFAULT_SCHEMA = Schema2
GRANT SELECT ON Orders TO User1, User2
GRANT SHOWPLAN TO User1, User2
go
EXEC sp_executesql N'SELECT * FROM Orders WHERE OrderDate > @orderdate',
                   N'@orderdate datetime', '20000101'
go
EXECUTE AS USER = 'User1'
EXEC sp_executesql N'SELECT * FROM Orders WHERE OrderDate > @orderdate',
                   N'@orderdate datetime', '19980101'
REVERT
go
EXECUTE AS USER = 'User2'
```

```
EXEC sp_executesql N'SELECT * FROM Orders WHERE OrderDate > @orderdate',
                   N'@orderdate datetime', '19980101'
REVERT
go
DROP USER User1
DROP USER User2
DROP SCHEMA Schema2
go
```

The script first clears the plan cache and then creates a schema and two database users and grants them the permissions needed to run the queries. We then run the same query three times, but with different parameter values. The first time we pass a date which is beyond the range in **Northwind**, so the optimizer settles for a plan with Index Seek + Key Lookup. The second and third time we run the query, we pass a different date, for which a Clustered Index Scan is a better choice. But since there is a cached plan, we expect to get that plan, and that is also what happens in the second execution. However, in the third execution, we find to our surprise that the plan is the CI Scan. What is going on? Why did we not get the plan in the cache?

The key here is that we run the query as three different users. The first time we run the query as ourselves (presumably we are **dbo**), but for the other two executions we impersonate the two newly created users. (If you are not acquainted with impersonation, look up the topic for EXECUTE AS in Books Online; I also cover it in my article *Packaging Permissions in Stored Procedures*.) The users are login-less, but that is only because we don't need any logins for this example. What is important is that they have different default schemas. User1 has **dbo** as its default schema, but for User2 the default schema is **Schema2**. Why does this matter?

Keep in mind that when SQL Server looks up an object, it first looks in the default schema of the user, and if the object is not found, it looks in the **dbo** schema. For **dbo** and User1, the query is unambiguous, since **dbo** is its default schema and this is the schema for the **Orders** table. But for User2 this is different. Currently there is only **dbo.Orders**, but what if **Schema2.Orders** is added later? Per the rules, User2 should now get data from that table and not from **dbo.Orders**. But if User2 would use the same cache entry as **dbo** and User1, that would not happen. Therefore, User2 needs a cache entry of its own. If **Schema2.Orders** is added, that cache entry can be invalidated without affecting other users.

We can see this is the plan attributes. Here is a variation of the query we ran for stored procedures:

```
SELECT qs.plan_handle, a.attrlist
FROM   sys.dm_exec_query_stats qs
CROSS  APPLY sys.dm_exec_sql_text(qs.sql_handle) est
CROSS  APPLY (SELECT epa.attribute + '=' + convert(nvarchar(127), epa.value) + '   '
              FROM   sys.dm_exec_plan_attributes(qs.plan_handle) epa
              WHERE  epa.is_cache_key = 1
              ORDER  BY epa.attribute
              FOR    XML PATH('')) AS a(attrlist)
WHERE  a.attrlist LIKE '%dbid=' + ltrim(str(db_id())) + ' %'
  --  AND  est.text LIKE '%WHERE OrderDate > @orderdate%'
   AND  est.text NOT LIKE '%sys.dm_exec_plan_attributes%'
```

There are three differences to the query for stored procedures:

1. The filtering for the database is different, since the **dbid** column in **sys.dm_exec_sql_text** is not populated for dynamic SQL, but instead we need to take this value from **sys.dm_exec_plan_attributes**. In order to avoid having to call this function twice in the query, the filtering is done in this somewhat klunky way.
2. As there is no procedure name to match on, we have to use part of the query text.
3. We need an extra condition to filter out the query against **sys.dm_exec_plan_attributes** itself.

When I ran this query, I saw this (partial) list of attributes:

```
date_first=7  date_format=1  dbid=6  objectid=158662399  set_options=251  user_id=5
date_first=7  date_format=1  dbid=6  objectid=158662399  set_options=251  user_id=1
```

First look at **objectid**. As you see, this value is identical for the two entries. This object id is the hash value that I described above. Next, look at the attribute which is distinctive: **user_id**. The name as such is a misnomer; the value is the default schema for the users using this plan. The **dbo** schema always has **schema_id** = 1. In my **Northwind** database, **Schema2** got **schema_id** = 5 when I ran the query, but you may see a different value.

Now run this query:

```
EXEC sp_executesql N'SELECT * FROM dbo.Orders WHERE OrderDate > @orderdate',
                   N'@orderdate datetime', '20000101'
```

And then run the query against **sys.dm_exec_plan_attributes** again. A third row in the output appears:

```
date_first=7  date_format=1  dbid=6  objectid=549443125  set_options=251  user_id=-2
```

objectid is different from above, since the query text is different. **user_id** is now -2. What does this mean? If you look closer at the query, you see that we now specify the schema explicitly when we access **Orders**. That is, the query is now unambiguous, and all users can use this cache entry. And that is exactly what **user_id** = -2 means: there are no ambiguous object references in the query. The corollary of this is that it is very much best practice to always use two-part notation in dynamic SQL, no matter whether you create the dynamic SQL in a client program or in a stored procedure.

You may think "we don't use schemas in our app, so this is not an issue for us", but not so fast! When you use CREATE USER, the default schema for the new user is indeed **dbo**, unless you specify something else. However, if the DBA is of the old school, he may create users with any of the legacy procedures **sp_adduser** or **sp_grantdbaccess**, and they work differently. They do not only create a user, but they also create a schema with the same name as the user and set this schema as the default schema for the newly created user and make this user the owner of the schema. Does it sound corny? Yes, but up to SQL 2000, schema and users were unified in SQL Server. Since you may not have control over how users are created, you should not rely on users having **dbo** as their default schema.

Finally, you may wonder why this issue does not apply to the caching of plans for stored procedures. The answer is that in a stored procedure, the name resolution is always performed from the procedure owner, not the current user. That is, in a procedure owned by **dbo**, **Orders** can only refer to **dbo.Orders**, never to any **Orders** table in some other schema. (But keep in mind that this applies only to the direct query text of the stored procedure. It does not apply to dynamic SQL invoked with EXEC() or **sp_executesql** inside the procedure.)

6.4 Auto-parameterisation

I said that there is one exception to the rule that SQL Server does not normalise the query text before computing the hash. That exception is auto-parameterisation, a mechanism by which SQL Server replaces constants in an unparameterised query and makes it a parameterised query. The purpose of auto-parameterisation is to reduce the menace of poorly written applications that inline parameter values into SQL strings instead of conforming to best practice and using parameterised statements.

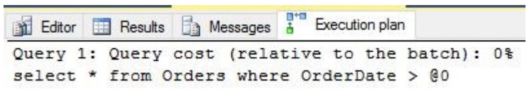
There are two models, for auto-parameterisation: simple and forced. The model is controlled by a database setting, and simple parameterisation is the default. With this model, SQL Server auto-parameterises only SQL statements of very low complexity. Furthermore, SQL Server only employs simple auto-parameterisation when there is a single possible plan; that is, simple parameterisation is designed to not cause parameter-sniffing issues. With forced parameterisation, SQL Server replaces all constants in a query with parameters, even if multiple plans are possible. That is, when forced parameterisation is in play, you can face issues with parameter sniffing.

Note: Forced parameterisation only applies to dynamic SQL, never to queries in stored procedures. Furthermore, forced parameterisation is not applied to queries that refer to variables. There are a few more situations when SQL Server does not apply forced parameterisation, as listed in this topic in Books Online.

In the following, we shall look at how you can see whether you may have a parameter sniffing problem due to auto-parameterisation. The script below sets **Northwind** into forced parameterisation and then runs two queries against the **Orders** table. One that returns no rows, and one which returns the majority of the rows.

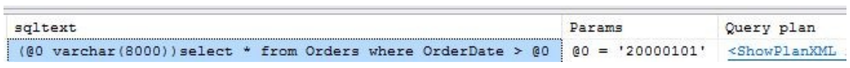
```
ALTER DATABASE Northwind SET PARAMETERIZATION FORCED
DBCC FREEPROCCACHE
go
SELECT * FROM Orders WHERE OrderDate > '20000101'
go
SELECT * FROM Orders WHERE OrderDate > '19970101'
go
; WITH basedata AS (
   SELECT est.text AS sqltext, qp.query_plan,
          charindex('<ParameterList>', qp.query_plan) + len('<ParameterList>')
             AS paramstart,
          charindex('</ParameterList>', qp.query_plan) AS paramend
   FROM   sys.dm_exec_query_stats qs
   CROSS  APPLY sys.dm_exec_sql_text(qs.sql_handle) est
   CROSS  APPLY sys.dm_exec_text_query_plan(qs.plan_handle,
                                            qs.statement_start_offset,
                                            qs.statement_end_offset) qp
   WHERE  est.text NOT LIKE '%exec_query_stats%'
), next_level AS (
   SELECT sqltext, query_plan,
          CASE WHEN paramend > paramstart
               THEN CAST (substring(query_plan, paramstart,
                                    paramend - paramstart) AS xml)
          END AS params
   FROM   basedata
)
SELECT sqltext,
       (SELECT CR.c.value('@Column', 'nvarchar(128)') + ' = ' +
               CR.c.value('@ParameterCompiledValue', 'nvarchar(512)') + ' '
        FROM    n.params.nodes('ColumnReference') AS CR(c)
        FOR XML PATH(''), TYPE).value('.', 'nvarchar(MAX)') AS Params,
       CAST (query_plan AS xml) AS [Query plan]
FROM   next_level n
go
```

If you look at the Execution Plan output in SSMS, you see this on the top:



```
Query 1: Query cost (relative to the batch): 0%
select * from Orders where OrderDate > @0
```

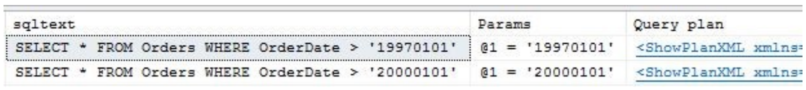
That is, the date has been replaced by a parameter, **@0**. And if you look at the execution plans, you see that both queries runs with Index Seek + Key Lookup, despite that the second query is best implemented as a Clustered Index Scan since it accesses the better part of the table. The output from the diagnostic query is like this:



sqltext	Params	Query plan
(@0 varchar(8000))select * from Orders where OrderDate > @0	@0 = '20000101'	<ShowPlanXML

That is, despite that we ran two statements with different query text, there is only one cache entry where the plan is determined from the date value in the first SELECT statement.

Now change FORCED in the script to read SIMPLE and run the script again. In the top of the execution plan, you see the same as above (except that **@0** is now **@1** for some reason). However, this time the two queries have different execution plans, and the output from the last query now looks like this:



sqltext	Params	Query plan
SELECT * FROM Orders WHERE OrderDate > '19970101'	@1 = '19970101'	<ShowPlanXML xmlns=
SELECT * FROM Orders WHERE OrderDate > '20000101'	@1 = '20000101'	<ShowPlanXML xmlns=

What seems to happen with simple parameterisation is that SQL Server attempts to auto-parameterise the query, but when it compiles the query it finds that there is more than one possible plan, and backs out of auto-parameterisation. However, some traces of the auto-parameterisation attempt remains, and this could lure you if you only look at the query text in the Showplan output. You need to get the text from **sys.dm_exec_sql_text** as in the query above, to see the actual text in the plan cache to be sure that auto-parameterisation is actually in play.

Note: When you run the above for simple parameterisation, you may see NULL in the columns **Params** and **Query plan** when you the run script the first time. If you rerun it, you will see values on the second run. This is due to the configuration parameter **optimize for ad hoc workloads**. When this parameter is 1, SQL Server only caches a shell query on the first execution of an unparameterised string. Only if the same string is executed a second time, the plan is cached. The default for this setting is 0, but 1 is the recommended setting, as it mitigates the effect of poorly written applications that do not parameterise statements. There are no known downsides with setting the option to 1.

6.5 Running Application Queries in SSMS

As you understand from the previous sections, there are a few more pitfalls when you want to troubleshoot a application query from SSMS which may cause you to get a different cache entry, and potentially a different query plan.

As with stored procedures, you need to keep ARITHABORT and other SET options in mind. But you also need to make sure that you have exactly the same query text, and that your default schema agrees with the user running the application.

The latter is the easiest to deal with. In most cases this will do:

```
EXECUTE AS USER = 'appuser'
go
-- Run SQL here
go
REVERT
```

appuser is here the database user that the application uses – either a private user for the application, or the user name of the actual person using the database. Observe that this fails if the query accesses resources outside the current database. In this case you can use EXECUTE AS LOGIN instead. Note that this requires a server-level permission.

Retrieving the exact SQL text can be more difficult. The best is use a trace to capture the SQL; you can run the trace either in Profiler or as a server-side trace. If the

SQL statement is unparameterised, you need to be careful that you copy the full text, and then select exactly that text in SSMS. That is, do not drop or add any leading or trailing blanks. Don't add line breaks to make the query more readable, and don't delete any comments. Keep it exactly as the application runs it. You can use the query against **sys.dm_exec_plan_attributes** in this article to verify that you did not add a second entry to the cache.

Another alternative is to get the text from **sys.dm_exec_query_stats** and **sys.dm_exec_sql_text**. Here is a query you can use:

```
SELECT '<' + est.text + '>'
FROM   sys.dm_exec_query_stats qs
CROSS  APPLY sys.dm_exec_sql_text(qs.sql_handle) est
WHERE  est.text LIKE '%some significant SQL text here%'
```

When copying the text from SSMS, it is important to make sure that line breaks are retained. If you run the query in Text mode this is not a problem, but if you are in Grid mode, you need to be careful as different versions of SSMS handles this differently. SSMS 2005 and 2008 replaces line breaks with spaces, while SSMS 2012 and 2014 do not. Starting with SSMS 2016 there is an option to control this, with replacing line breaks being the default. The angle brackets in the output serve as delimiters, so that you can select the exact query text including any leading or trailing blanks.

Parameterised SQL is easier to deal with, since the SQL statement is packaged in a character literal. That is, if you see this in Profiler

```
EXEC sp_executesql N'SELECT * FROM Orders WHERE OrderDate > @orderdate',
                   N'@orderdate datetime', '20000101'
```

It's no issue if you happen to change it to, say:

```
EXEC sp_executesql
N'SELECT * FROM Orders WHERE OrderDate > @orderdate',
N'@orderdate datetime', '20000101'
```

What is important is that you must not change anything of what is between the quotes of the two first parameters to **sp_executesql**, since this is what the hash is computed from. (That is, the parameter list is included in the hash computation.)

If you don't have any server-level permission – ALTER TRACE to run a trace, or VIEW SERVER STATE to query **sys.dm_exec_query_stats** and **sys.dm_exec_sql_text** – it's starting to get difficult. (Unless you are on SQL 2016 and you have enabled Query Store for the database, which I will cover in the next chapter.) If the dynamic SQL is produced in a stored procedure which you can edit, you can add a PRINT statement to get the text. (Actually, stored procedures that produce dynamic SQL should always have a **@debug** parameter and include the line: `IF @debug = 1 PRINT @sql`.) You still have to be careful to get the exact text, and not add or drop any spaces. If there is a parameter list, you need to make sure that you copy the parameter list exactly as well. If the SQL is produced by the application, getting the SQL statement may be possible if you can run the application in a debugger, but the exact text of the parameter list may be difficult to get at. The best bet may be try the application against a database on an SQL Server instance where you have all the required permissions, for instance on your local workstation and get the query text this way.

6.6 Plan Guides and Plan Freezing

Sometimes you may find that you want to modify a query to resolve a performance issue by adding a hint of some sort. For a stored procedure, it is not unlikely that you can edit the procedure and deploy the change fairly immediately. But if the query is generated inside an application, it may be more difficult. The entire executable has to be built and maybe be deployed to all user machines. It is also likely that there is a mandatory QA step involved. And if it is a third-party application, changing the code is out of the question.

However, SQL Server provides a solution for these situations, to wit *plan guides*. There are two ways to set up plan guides, a general way, and a shortcut which is also known as *plan freezing*. General plan guides were introduced in SQL 2005, whereas plan freezing was added in SQL 2008.

Note: this feature is not available on the low-end editions of SQL Server – Express, Web and Workgroup Edition.

Here is an example of a setting up a plan guide. This particular example runs on SQL 2008 and up:

```
DBCC FREEPROCCACHE
go
EXEC sp_executesql N'SELECT * FROM dbo.Orders WHERE OrderDate > @orderdate',
                   N'@orderdate datetime', @orderdate = '19960101'
go
EXEC sp_create_plan_guide
     @name = N'MyGuide',
     @stmt = N'SELECT * FROM dbo.Orders WHERE OrderDate > @orderdate',
     @type = N'SQL',
     @module_or_batch = NULL,
     @params = N'@orderdate datetime',
     @hints =  N'OPTION (TABLE HINT (dbo.Orders , INDEX (OrderDate)))'
go
EXEC sp_executesql N'SELECT * FROM dbo.Orders WHERE OrderDate > @orderdate',
                   N'@orderdate datetime', @orderdate = '19980101'
go
EXEC sp_control_plan_guide N'DROP', N'MyGuide'
```

In this example, I create a plan to ensure that a query against **OrderDate** always uses an Index Seek (maybe because I expect all queries to be for the last few days or so). I specify a name for the guide. Next, I specify the exact statement for which the guide applies. As when you run a query in SSMS to investigate it, you need to make sure that you do not add or lose any leading or trailing space, nor must you make any other alteration. The parameter **@type** specifies that the guide is for dynamic SQL and not for a stored procedure. Had the SELECT statement been part of a larger batch, I would have to specify the text of that batch in the parameter **@module_or_batch**, again exactly as submitted from the application. When I specify NULL for **@module_or_batch**, **@stmt** is assumed to be the entire batch text. **@params** is the parameter list for the batch, and again there must be an exact match character-by-character with what the application submits.

Finally, **@hints** is where the fun is. In this example I specify that the query should always use the index on **OrderDate**, no matter the sniffed value for **@orderdate**. This particular query hint, OPTION (TABLE HINT) is not available in SQL 2005, which is why the example does not run on that version.

In the script, the initial DBCC FREEPROCCACHE is only there to give us a clean slate. Also, for the purpose of the demo, I run the query with a parameter value that gives the "bad" plan, the Clustered Index Scan. Once the plan guide has been entered, it takes effect immediately. That is, any current entry for the query is evicted from the cache.

On SQL 2008 and later, you can specify the parameters to **sp_create_plan_guide** in any order as long as you name them, and you can leave out the N before the string literals. However, SQL 2005 is far less forgiving. The parameters must be entered in the given order, even if you name them, and you must specify the N before all the string literals.

In this example I used a plan guide to force an index, but you can use other hints as well, including the USE PLAN hint that permits you to specify the complete query plan to use. Certainly not for the faint of heart!

...although that is exactly the idea with plan freezing. Say that you have a query that sways between two plans, one good and one bad, due to parameter sniffing, and there is not really any civilised way to get the bad plan out of the equation. Rather than battling with the complex parameters of **sp_create_plan_guide**, you can extract a plan handle from the cache and feed it to the stored procedure **sp_create_plan_guide_from_handle** to force the plan you know is good. Here is a demo and example.

```
DBCC FREEPROCCACHE
SET ARITHABORT ON
go
EXEC sp_executesql N'SELECT * FROM dbo.Orders WHERE OrderDate > @orderdate',
                   N'@orderdate datetime', @orderdate = '19990101'
go
DECLARE @plan_handle varbinary(64),
        @rowc int

SELECT @plan_handle = plan_handle
FROM   sys.dm_exec_query_stats qs
CROSS  APPLY sys.dm_exec_sql_text(qs.sql_handle) est
WHERE  est.text LIKE '%Orders WHERE OrderDate%'
  AND  est.text NOT LIKE '%dm_exec_query_stats%'
SELECT @rowc = @@rowcount

IF @rowc = 1
   EXEC sp_create_plan_guide_from_handle 'MyFrozenPlan', @plan_handle
ELSE
   RAISERROR('%d plans found in plan cache. Cannot create plan guide', 16, 1, @rowc)
go
-- Test it out!
SET ARITHABORT OFF
go
EXEC sp_executesql N'SELECT * FROM dbo.Orders WHERE OrderDate > @orderdate',
                   N'@orderdate datetime', @orderdate = '19960101'
go
SET ARITHABORT ON
EXEC sp_control_plan_guide 'DROP', 'MyFrozenPlan'
```

For the purpose of the demo, I first clear the plan cache and set ARITHABORT to a known state. Then I run my query with a parameter that I know will give me the good plan. The next batch demonstrates how to use **sp_create_plan_guide_from_handle**. I first run a query against **sys.dm_exec_query_stats** and **sys.dm_exec_sql_text** to find the entry for my query. Next I capture **@@rowcount** into a local variable (since **@@rowcount** is set after each statement, I prefer to copy it to a local variable in a SELECT that is directly attached to the query to avoid accidents). This is a safety precaution in case I get multiple matches or no matches in the cache. If I get exactly one match, I call **sp_create_plan_guide_from_handle** which takes two parameters: the name of the plan guide and the plan handle. And that's it!

The next part tests the guide. To make sure that I don't use the same cache entry, I use a different setting of ARITHABORT. If you run the demo with display of execution plan enabled, you will see that the second execution of the query uses the same plan with Index Seek + Key Lookup as the first, although the normal plan for the given parameter would be a Clustered Index Scan. That is, the plan guide is not tied to a certain set of plan attributes.

When you use this for real, you would run the query you want the plan guide for, only if the desired plan is not in the cache already. The query against the cache will require some craftsmanship so that you get exactly one hit and the correct hit. An alternative may be to look at the matches in the query output in SSMS and copy and paste the plan handle. .

A cool thing is that you don't have to set this up on the production server, but you can experiment on a lab server. The guide is stored in **sys.plan_guides**, so once you have the guide right, you can use the content there to craft a call to **sp_create_plan_guide** that you run the production server. You can also script the plan guide through Object Explorer in SSMS.

If you have a multi-statement batch or a stored procedure, you may not want to set up a guide for the entire batch, but only for a statement. For this reason **sp_create_plan_guide_from_handle** accepts a third parameter **@statement_start_offset**, a value you can get from **sys.dm_exec_query_stats**.

One potential problem with plan guides is that since they don't appear in the query text, they can easily be overlooked and forgotten. One risk is that as the database evolves, the plan mandated by the plan guide is no longer the best one. The profile of the data could have changed, or a better index could have been added. A second risk is that the query is changed in some seemingly innocent way. But if the text changes, if so only by adding a single space, the plan guide will no longer match, and you may be back to the original poor performance, and having forgot all about the plan guide, you will have to reinvent the wheel.

For this reason, it is a good idea to add to your troubleshooting checklist that you should query **sys.plan_guides** to see if there are any plan guides at all in the database, and if there is you can further investigate whether they are relevant to the queries you are battling. If you want to see the currently active plan guides, you can also use this query to find this information in the plan cache:

```
; WITH constants AS (
    SELECT N' PlanGuideDB="' AS guide_db_attr, guide_name_attr = N'PlanGuideName="'
), basedata AS (
   SELECT est.text AS sqltext, qp.query_plan,
          charindex(c.guide_db_attr, qp.query_plan) + len(c.guide_db_attr)
                      AS guide_db_start,
          charindex(c.guide_name_attr, qp.query_plan) + len(c.guide_name_attr)
                      AS guide_name_start
   FROM   sys.dm_exec_query_stats qs
   CROSS  APPLY sys.dm_exec_sql_text(qs.sql_handle) est
   CROSS  APPLY sys.dm_exec_text_query_plan(qs.plan_handle,
                                            qs.statement_start_offset,
                                            qs.statement_end_offset) qp
   CROSS  JOIN constants c
   WHERE  est.text NOT LIKE '%exec_query_stats%'
), next_level AS (
   SELECT sqltext, query_plan,
          CASE WHEN guide_db_start > 100
               THEN substring(query_plan, guide_db_start,
                                          charindex('"', query_plan, guide_db_start + 1) -
                                                    guide_db_start)
          END AS PlanGuideDB,
          CASE WHEN guide_name_start > 100
               THEN substring(query_plan, guide_name_start,
                                          charindex('"', query_plan, guide_name_start + 1) -
                                                    guide_name_start)
          END AS PlanGuideName
   FROM   basedata
)
SELECT sqltext, PlanGuideName, PlanGuideDB,
       TRY_CAST (query_plan AS xml) AS [Query plan]
FROM   next_level n
WHERE  PlanGuideDB IS NOT NULL OR PlanGuideName IS NOT NULL
```

The query is similar in nature to the queries we have used to extract parameter values from the plan, but here we are looking for the two attributes that identify the plan guide. The normal way to extract them would be to use XQuery, but as I have remarked: there is a risk that the plan cannot be represented in the built-in **xml** data type, whence the acrobatics with **charindex** and **substring**.

I cannot say that I am overly enthusiastic over plan guides given how complex they are to use. That said, they are heaven-sent when you have a third-party application of which you cannot change the code. To learn more about plan guides, see the sections on plan guides in Books Online or the white paper listed in the section *Further Reading* below.

7. Using the Query Store in SQL 2016

7.1 Introducing the Query Store

SQL Server 2016 introduced a new feature, known as the Query Store. When Query Store is enabled for a database, SQL Server saves the plans, the settings and execution statistics for queries in persisted tables. This permits you to track query execution over time and when abrupt change in performance occurs because a plan changed from one day to another, you can go and find both plans in the Query Store tables. And, if you need to address the problem quickly, you can force the old and faster plan.

This chapter does not aim at giving a full coverage of the Query Store, but only to give you queries to get the same information from the Query Store as I have previously shown for the plan cache. If you have the choice of getting plans from the Query Store and the plan cache, the Query Store is often a better alternative since data is persisted. For instance, say that users reported bad performance between nine and ten in the morning, but at the time you got to investigate the issue, performance was back to normal, because the query with the bad plan was recompiled for some reason without your intervention. In this case, you cannot find the plan in the cache to that you can investigate how the plan looked like and what parameter values it was sniffed for. On the other hand, chances are good that you can find the bad plan in the Query Store tables.

Since Query Store is on database level, this gives you an advantage if you are looking for problems in a specific database – no need to filter out other databases. But this cuts both ways; if you are looking for issues on a server-wide basis, you may prefer to query the plan cache. The fact that the Query Store is on database level, gives a second advantage if you do not have server-level permissions: you only need the permission VIEW DATABASE STATE to query the Query Store. (You have this permission, if you are a member of the **db_owner** role).

Query Store is not enabled by default for a database. To enable the Query Store for **Northwind**, run this command:

```
ALTER DATABASE Northwind SET QUERY_STORE = ON
```

There are a couple of options that permits you control the behaviour and performance of Query Store, but I will not cover these options here.

The Query Store tables are not exposed directly. Instead you query views that also includes data that has not yet been persisted to disk. (To ensure that the Query Store has a low overhead, data is flushed to persisted tables asynchronously.)

7.2 Finding Cache Keys

The first query against the plan cache we looked at is one that gives us the plan attributes that are cache keys. The Query Store version of this query is a lot more straightforward, since the plan attributes are accessed through **sys.query_context_settings** where each attribute is a column. The SET options are still a bit mask, though. Here is a query that returns the most important plan attributes:

```
SELECT q.query_id, convert(bigint, cs.set_options) as set_options,
       cs.language_id, cs.date_format, cs.date_first, cs.default_schema_id
FROM   sys.query_store_query q
JOIN   sys.query_context_settings cs
       ON q.context_settings_id = cs.context_settings_id
WHERE  q.object_id = object_id('dbo.List_orders_6')
```

The test case for this query was:

```
CREATE PROCEDURE List_orders_6 AS
   SELECT *
   FROM   Orders
   WHERE  OrderDate > '12/01/1998'
go
SET DATEFORMAT dmy
go
EXEC List_orders_6
go
SET DATEFORMAT mdy
go
EXEC List_orders_6
go
```

When I ran the query above, I got this output:

```
query_id set_options language_id date_format date_first default_schema_id
-------- ----------- ----------- ----------- ---------- -----------------
7        4347        0           2           7          -2
8        4347        0           1           7          -2
```

Note that there are different values in the column **date_format**. When it comes to the column **query_id**, this is only an internal ID for the query and you may see different values when you try the above. However, you may note that while the query text is the same, different settings result in different **query_id**s.

7.3 Finding Sniffed Parameter Values

We can use the Query Store to find which parameter values a plan was sniffed for. This query is slightly simpler than the same query against the plan cache, because we don't have to dabble with the offsets to get the query text. But the query still has to count as complex, because we need to do the same work to extract the parameter values from the plan XML:

```
; WITH basedata AS (
     SELECT q.query_text_id, q.context_settings_id, p.query_plan, qt.query_sql_text,
            last_compile_batch_offset_start / 2 AS stmt_start,
            charindex('<ParameterList>', p.query_plan) + len('<ParameterList>')
                AS paramstart,
            charindex('</ParameterList>', p.query_plan) AS paramend
     FROM   sys.query_store_query q
     JOIN   sys.query_store_plan p ON q.query_id = p.query_id
     JOIN   sys.query_store_query_text qt ON q.query_text_id = qt.query_text_id
     WHERE  q.object_id = object_id('dbo.List_orders_11')
), next_level AS (
    SELECT query_text_id, context_settings_id, query_plan, query_sql_text, stmt_start,
           CASE WHEN paramend > paramstart
                THEN TRY_CAST(substring(query_plan, paramstart,
                                        paramend - paramstart) AS xml)
           END AS params
    FROM   basedata
)
SELECT convert(bigint, cs.set_options) AS [SET], stmt_start AS Pos,
       n.query_sql_text AS Statement,
       CR.c.value('@Column', 'nvarchar(128)') AS Parameter,
       CR.c.value('@ParameterCompiledValue', 'nvarchar(128)') AS [Sniffed Value],
       TRY_CAST (query_plan AS xml) AS [Query plan]
FROM   next_level n
CROSS  APPLY  n.params.nodes('ColumnReference') AS CR(c)
JOIN   sys.query_context_settings cs ON n.context_settings_id = cs.context_settings_id
ORDER  BY [SET], Pos, Parameter
```

The query starts with **sys.query_store_query**, which is the main Query Store table that defines the query and here we can filter on the object id for the procedure we are interested in. The query text is taken from **sys.query_store_query_text**, while the query plan is in **sys.query_store_plan**. Just like in **sys.dm_exec_text_query_plan** the XML for the plan is stored in a **nvarchar(MAX)** column due to limitation in the **xml** data type. We handle the XML exactly in the same way as in the original query. In the final SELECT, I get the SET options from **sys.query_context_settings**.

The output from the query above is similar to the output to the original query earlier in the article list, but you will find that the query text also includes the parameter list.

If you want to find the parameters for a batch of dynamic SQL, you need to replace this condition in the CTE **basedata**

```
   WHERE q.object_id = object_id('dbo.List_orders_11')
```

with a condition on some suitable piece in the query text. You may also want to filter out the query against the Query Store itself. Thus, you may get something like:

```
    WHERE  qt.query_sql_text LIKE '%Orderdate >%'
      AND  qt.query_sql_text NOT LIKE '%query_store_query%'
```

If you have a query that for some reason gets recompiled frequently and occasionally gets a bad plan due to sniffing, the query above may return many rows. To reduce the noise you can use the views **sys.query_store_runtime_stats** and **sys.query_store_runtime_stats_interval** to filter the result. Say that users in the New Delhi office reported that they had bad performance between 09:00 and 10:00, their local time. You can add this condition to the CTE **basedata** above:

```
   AND  EXISTS (SELECT *
                FROM   sys.query_store_runtime_stats rs
                JOIN   sys.query_store_runtime_stats_interval rsi
                       ON rs.runtime_stats_interval_id = rsi.runtime_stats_interval_id
                WHERE  rs.plan_id = p.plan_id
                  AND  rsi.end_time   >= '20161125 09:00 +05:30'
                  AND  rsi.start_time <= '20161125 10:00 +05:30')
```

The gist here is that the Query Store saves run-time statistics for a plan by intervals. A row in **sys.query_store_runtime_stats** holds the statistics for a plan during an interval, and **sys.query_store_runtime_stats_interval** holds the duration for such an interval. You should observe that the data type of the columns **start_time** and **end_time** is **datetimeoffset**, which is why I have included the TZ offset in the queries above. Note that if you leave out the TZ offset, your values will be converted to UTC (offset +00:00) and you may not find the data you are looking for.

Instead of filtering on time, you can filter on one of the many columns in **sys.query_store_runtime_stats** to select only plans with a certain resource consumption. This is left as an exercise for the reader.

7.4 Forcing Plans with Query Store

As I mentioned, you can use the Query Store to force a plan. This is akin to plan guides, but when you force a plan through the Query Store, this does not result in an entry in **sys.plan_guides**. There is also a difference to how this works with different cache keys. This script illustrates:

```
   ALTER DATABASE Northwind SET QUERY_STORE CLEAR
   SET ARITHABORT ON
   go
   EXEC sp_executesql N'SELECT * FROM dbo.Orders WHERE OrderDate > @orderdate',
                      N'@orderdate datetime', @orderdate = '19990101'
   go
   DECLARE @query_id  bigint,
           @plan_id   bigint,
           @rowc      int

   SELECT @query_id = q.query_id, @plan_id = p.plan_id
   FROM   sys.query_store_query q
   JOIN   sys.query_store_plan p ON q.query_id = p.query_id
   JOIN   sys.query_store_query_text qt ON q.query_text_id = qt.query_text_id
   WHERE  qt.query_sql_text LIKE '%Orders WHERE OrderDate%'
     AND  qt.query_sql_text NOT LIKE '%query_store_query%'
   SELECT @rowc = @@rowcount

   IF @rowc = 1
      EXEC sp_query_store_force_plan @query_id, @plan_id
   ELSE
      RAISERROR('%d rows found in Query Store. Cannot force plan', 16, 1, @rowc)

   SET ARITHABORT OFF
   EXEC sp_executesql N'SELECT * FROM dbo.Orders WHERE OrderDate > @orderdate',
                      N'@orderdate datetime', @orderdate = '19960101'

   SET ARITHABORT ON
   EXEC sp_executesql N'SELECT * FROM dbo.Orders WHERE OrderDate > @orderdate',
                      N'@orderdate datetime', @orderdate = '19960101'

   SELECT q.query_id, p.plan_id, p.is_forced_plan,
          convert(bigint, cs.set_options) as set_options,
              qt.query_sql_text, try_cast(p.query_plan AS xml)
   FROM   sys.query_store_query q
   JOIN   sys.query_store_query_text qt ON q.query_text_id = qt.query_text_id
   JOIN   sys.query_store_plan p ON q.query_id = p.query_id
   JOIN   sys.query_context_settings cs
          ON q.context_settings_id = cs.context_settings_id
   WHERE  qt.query_sql_text LIKE '%Orders WHERE OrderDate%'
     AND  qt.query_sql_text NOT LIKE '%query_store_query%'
```

```
EXEC sp_query_store_unforce_plan @query_id, @plan_id
```

For the purpose of the demo, I clear the Query Store tables. I then run a query against the **Orders** database for which Index Seek + Key Lookup is the best plan with the given parameters. Next, I retrieve the query and plan ids for this query. The check on **@@rowcount** is only to make sure that the demo runs as expected. Following this, I force this plan with **sp_query_store_force_plan**. Next, I run the query again twice, once with ARITHABORT OFF and once with ARITHABORT ON. Note that this time, I specify parameter values so that all orders will be returned and thus the best plan is a Clustered Index Scan.

If you look at the plans for the latter two queries, you will find that the first execution with ARITHABORT OFF uses the CI Scan, that is, the forced plan is not considered, whereas the second execution indeed uses the forced plan. This is different to when we froze a plan with **sp_create_plan_guide_from_handle**; in that case the forced plan was applied also when ARITHBORT was OFF.

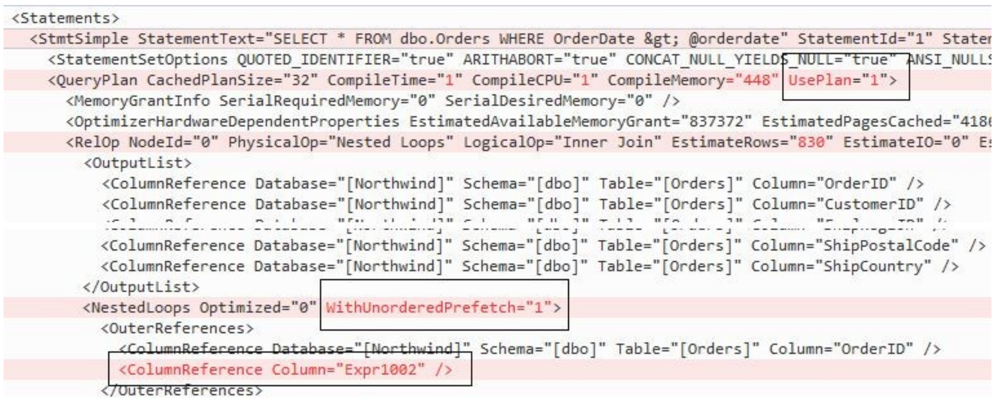
This is followed by a diagnostic query. Before we look at the output, we can note that it is very simple to find whether there are plans that has been forced through the Query Store: there is a column **is_forced_plan** in **sys.query_store_plan**. However, if we look at the output from query, it is maybe not as you would expect. (For space reasons, I don't include the last two columns):

```
query_id    plan_id   is_forced_plan set_options    count_executions
----------- --------- -------------- -------------- --------------------
4           4         0              251            1
1           1         1              4347           1
1           5         0              4347           1
```

There are three rows in the output, not one, despite that we ran the same query three times. As we learnt before, the context settings (or plan attributes) are tied to the **query_id**. Whence, the execution with **set_options** = 251 (ARITHABORT OFF) has a different **query_id**. And since we force the plan for a specific **query_id**, this explains why the Seek + Lookup was not enforced for ARITHABORT OFF.

Remains to understand why there are two entries for **query_id** = 1. We can guess that the one with **plan_id** = 5 comes from the second execution, since presumably **plan_id** is assigned sequentially from 1 after clearing the Query Store. But why is there a new plan, when we forced **plan_id** = 1? When I forced the plan, the plan fell out of cache, since the Query Store does not check whether I'm forcing the current plan or an older plan. For this reason, there was a new compilation when then the query was re-executed with ARITHABORT ON. When a plan is forced, SQL Server does not just take that plan at face value, since it may not be a usable plan. (For instance, it could refer to an index that has been dropped.) Instead, the optimizer generates plans until it has generated the forced plan – or a plan which is very similar to the original plan.

If you save the XML for the plans to disk, you can use your favourite compare tool to compare them. I used Beyond Compare, and this is an excerpt of what I saw for **plan_id** = 5:



```
<Statements>
  <StmtSimple StatementText="SELECT * FROM dbo.Orders WHERE OrderDate &gt; @orderdate" StatementId="1" Stater
    <StatementSetOptions QUOTED_IDENTIFIER="true" ARITHABORT="true" CONCAT_NULL_YIELDS_NULL="true" ANSI_NULLS
    <QueryPlan CachedPlanSize="32" CompileTime="1" CompileCPU="1" CompileMemory="448" UsePlan="1">
      <MemoryGrantInfo SerialRequiredMemory="0" SerialDesiredMemory="0" />
      <OptimizerHardwareDependentProperties EstimatedAvailableMemoryGrant="837372" EstimatedPagesCached="418
      <RelOp NodeId="0" PhysicalOp="Nested Loops" LogicalOp="Inner Join" EstimateRows="830" EstimateIO="0" E
        <OutputList>
          <ColumnReference Database="[Northwind]" Schema="[dbo]" Table="[Orders]" Column="OrderID" />
          <ColumnReference Database="[Northwind]" Schema="[dbo]" Table="[Orders]" Column="CustomerID" />
          <ColumnReference Database="[Northwind]" Schema="[dbo]" Table="[Orders]" Column="EmployeeID" />
          <ColumnReference Database="[Northwind]" Schema="[dbo]" Table="[Orders]" Column="ShipPostalCode" />
          <ColumnReference Database="[Northwind]" Schema="[dbo]" Table="[Orders]" Column="ShipCountry" />
        </OutputList>
        <NestedLoops Optimized="0" WithUnorderedPrefetch="1">
          <OuterReferences>
            <ColumnReference Database="[Northwind]" Schema="[dbo]" Table="[Orders]" Column="OrderID" />
            <ColumnReference Column="Expr1002" />
          </OuterReferences>
```

I have highlighted the pertinent differences. First note the appearance of the attribute `UsePlan="1"` which is absent from **plan_id** 1. This indicates that the plan is indeed a forced plan. (Through the Query Store, a plan guide or explicit use of the USE PLAN hint.) What is also interesting is the appearance of the attribute WithUnorderedPrefetch which is not present in the plan actually forced. That is, this plan has a different implementation of the Nested Loop Join operator from the original plan and it also has a column reference which is not in the first plan. That is, plans 1 and 5 are not exactly the same. Or more generally, when you force a plan, you may not get exactly the same plan as the one you forced, but you will get a plan – as a Microsoft developer described it to me – that is morally equivalent to the original plan.

7.5 Conclusion on Query Store

Before I leave the topic of Query Store, I should add that there is a GUI for Query Store in SQL Server Management Studio where you can investigate which are the expensive queries etc, and you can also force plans from this UI. I leave it to the reader to explore these options on your own.

If you want to learn more about the Query Store, you can read the topic *Monitoring Performance By Using the Query Store* in Books Online. For a longer more in-depth discussion, you may be interested in a series of articles on Query Store by Data Platform Enrcio van der Laar published on Simple Talk.

- -

8. Concluding Remarks

You have now learnt why a query may perform differently in the application and in SSMS. You have also seen several ways to address issues with parameter sniffing.

Did I include all reasons why there could be a performance difference between the application and SSMS? Not all, there are certainly a few more reasons. For instance, if the application runs on a remote machine, and you run SSMS directly on the server, a slow network can make quite a difference. But I believe I have captured the most likely reasons that are due to circumstances within SQL Server.

Did I include all possible reasons why parameter sniffing may give you a bad plan, and how you should address it? Probably not. There is room for more variation there, and particularly I can't know what your application is doing. But hopefully some of the methods I have presented here can help you to find a solution.

If you think that you have stumbled on something which is not covered in the article, but you think I should have included, please drop me a line at esquel@sommarskog.se. And the same applies if you see any errors, not the least spelling or grammar errors. On the other hand, if you have a question how to solve a particular problem, I recommend you to post a question to an SQL Server forum, because a lot more people will see your question. If you absolutely want me to see the question, post to Microsoft's Transact-SQL forum, and mail me the link (because there is too much traffic for me to read it all).

8.1 Further Reading

If you want learn more about query compilation, statistics etc, below are a few articles that I like to recommend. You may note that they all relate to SQL 2008 or SQL 2005. This is because I assembled the article list in that time frame. I have not investigated whether are newer versions of these articles. But even if you are on a newer version of SQL Server, the articles are still largely applicable.

Statistics Used by the Query Optimizer in Microsoft SQL Server 2008 – A white paper written by Eric Hanson and Yavor Angelov from the SQL Server team..

Plan Caching in SQL Server 2008 – A white paper written by SQL Server MVP Greg Low. Appendix A in this paper details the rules for simple parameterisation..

Troubleshooting Performance Problems in SQL Server 2008 – An extensive document, which looks at performance from several angles, not only query tuning. Written by a number of developers and CSS people from Microsoft..

Forcing Query Plans – A white paper about plan guides. This version is for SQL 2005; I have not seen any version for SQL 2008.

9. Revisions

2019-10-26
Added the section *Database Settings* to discusses a situation where you may get different performance for reasons not related to parameter sniffing.

2018-12-29
A couple of minor corrections / additions.

- In one place I suggested that ANSI_PADDING affects **nvarchar** columns. That's incorrect, it applies to **varchar** and **varbinary** only.
- In the section *The Effects of Statement Recompile*, added a note relating to that starting with SQL 2019, SQL Server now defers compilation of statements with table variables.
- Added a new section *Live Query Plan* to discuss how you can get plans with partial actual values from a running query.
- Some minor alterations of the section *Getting Query Plans and Parameters from a Trace*, include removing the old note and adding a new about a new extended event, **query_plan_profile**.

2018-08-29
Updated the section *Could it Be MARS?* after input from Nick Smith. It seems that we now have an answer why MARS can cause things run slower – it's network latency.

2017-12-05
Three minor updates:

- Added text in the section *Putting the Query Plan into the Cache* that UPDATE STATISTICS only triggers a recompile if data has changed.
- Added a note to the section *The Default Settings* to comment on a mysterious passage in the topic for SET ARITHABORT that Daniel Berber made me aware of.
- Updated the section *Could it Be MARS?* after Allen Firth reported that he had also experienced this.

2017-09-20
Added a section *Could it Be MARS?*, inspired by an observation from Lonny Niederstadt.

2017-07-09
Gordon Griffin pointed out an error in the section *Finding Information About Statistics*. I suggested that you could always use the column name as an argument to DBCC SHOW_STATISTICS, but this is only true if the column has auto-created statistics. It does not work if there is a user-created statistics on the column.

2016-12-18
A general revision of the article to reflect that it is 2016, which means that some of the text relating to SQL 2000 and SQL 2005 have been reduced or removed. The following sections have been reworked to a greater extent or added:

- *An Issue With Linked Servers*. Due to the introduction of row-level security in SQL 2016, you may again face situations where you get bad plans, because low-privileged users cannot run DBCC SHOW_STATISTICS on a table in the query.
- The chapter on dynamic SQL has had a bigger overhaul than the rest of the text with some rearrangement of the material.
  - On a suggestion from Andrew Morton, I now point out the virtue of specifying the parameter length in a call from .NET.
  - The discussion on auto-parameterisation is entirely rewritten. The previous version incorrectly suggested that you could get parameter sniffing issues with simple parameterisation, but this is very unlikely.
  - In the section on plan guides and plan freezing, I've added a query to find active plan guides in the plan cache.
- There is a new chapter on how to use the Query Store, a new feature introduced in SQL 2016, to get information to troubleshoot parameter sniffing.

2013-08-30
Corrected an error in the function setoptions that Alex Valen was kind to point out.

2013-04-14
Updated the section *An Issue with Linked Servers* to reflect that this gotcha has been removed with SQL 2012 SP1. Also added a note on SET NO_BROWSETABLE ON to the section *The Story so Far*.

2011-12-31
Added some text about SQL Server Agent which runs with QUOTED_IDENTIFIER OFF by default. Extended the section on Plan Guides to also cover plan freezing.

2011-11-27
Several of the links in the section Further Reading were broken. This has been fixed.

2011-07-02
There is now a Russian translation available. Kudos to Dima Piliugin for this work!

2011-06-25
Added a section *An Issue with Linked Servers* to discuss a special problem that may cause the situation Slow in the application, fast in SSMS.

2011-02-20
First version.

Back to my home page.