Figure 17-4 is not the same execution plan as that shown in Figure 17-2. The number of reads drops slightly, but the execution time stays roughly the same. The second execution, using London as the value for the parameter, results in the following I/O and execution times:

```
Reads:1084
Duration:97.7ms
```

This time the reads are radically higher, up to what they were when using the local variable, and the execution time was increased. The plan created in the first execution of the procedure with the parameter London results in a plan best suited to retrieve the 434 rows that match those criteria in the database. Then the next execution of the procedure using the parameter value Mentor did well enough using the same plan generated by the first execution. When the order is reversed, a new execution plan was created for the value Mentor that did not work at all well for the value London.

In these examples, I've actually cheated just a little. If you were to look at the distribution of the data in the statistics in question, you'd find that the average number of rows returned is around 34, while London's 434 is an outlier. The slightly better performance you saw when the procedure was compiled for London reflects the fact that a different plan was needed. However, the performance for values like Mentor was slightly reduced with the plan for London. Yet, the improved plan for Mentor was absolutely disastrous for a value like London. Now comes the hard part.

You have to determine which of your plans is correct for your system's load. One plan is slightly worse for the average values, while another plan is better for average values but seriously hurts the outliers. The question is, is it better to have somewhat slower performance for all possible data sets and support the outliers' better performance or let the outliers suffer in order to support a larger cross section of the data because it may be called more frequently? You'll have to figure this out on your own system.

Identifying Bad Parameter Sniffing

Bad parameter sniffing will generally be an intermittent problem. You'll sometimes get one plan that works well enough and no one complains, and you'll sometimes get another, and suddenly the phone is ringing off the hook with complaints about the speed of the system. Therefore, the problem is difficult to track down. The trick is in identifying that you are getting two (or sometimes more) execution plans for a given parameterized

query. When you start getting these intermittent changes in performance, you must capture the query plans involved. One method for doing this would be pull the estimated plans directly out of cache using the sys.dm_exec_query_plan DMO like this:

```
SELECT deps.execution_count,
       deps.total_elapsed_time,
       deps.total_logical_reads,
       deps.total_logical_writes,
       deqp.query_plan
FROM sys.dm_exec_procedure_stats AS deps
    CROSS APPLY sys.dm_exec_query_plan(deps.plan_handle) AS deqp
WHERE deps.object_id = OBJECT_ID('AdventureWorks2012.dbo.AddressByCity');
```

This query is using the sys.dm_exec_procedure_stats DMO to retrieve information about the procedure in the cache and the query plan.

If you have enabled the Query Store, another approach would be to retrieve the plans from there:

```
SELECT SUM(qsrs.count_executions) AS ExecutionCount,
       AVG(qsrs.avg_duration) AS AvgDuration,
       AVG(qsrs.avg_logical_io_reads) AS AvgReads,
       AVG(qsrs.avg_logical_io_writes) AS AvgWrites,
       CAST(qsp.query_plan AS XML) AS Query_Plan,
       qsp.query_id,
       qsp.plan_id
FROM sys.query_store_query AS qsq
    JOIN sys.query_store_plan AS qsp
        ON qsp.query_id = qsq.query_id
    JOIN sys.query_store_runtime_stats AS qsrs
        ON qsrs.plan_id = qsp.plan_id
WHERE qsq.object_id = OBJECT_ID('dbo.AddressByCity')
GROUP BY qsp.query_plan,
         qsp.query_id,
         qsp.plan_id;
```

This query, unlike the other, can return more than one execution plan.

The results from either query when run within SSMS will include a column for `query_plan` that is clickable. Clicking it will open a graphical plan even though what is retrieved is XML. If you're dealing with a single plan from cache, right-click the plan itself and select Save Execution Plan As from the context menu. You can then keep this plan to compare it to a later plan. If you're operating out of the Query Store, you'll have multiple plans available in a bad parameter sniffing situation.

What you're going to look at is in the properties of the first operator, in this case the `SELECT` operator. There you'll find the Parameter List item that will show the values that were used when the plan was compiled by the optimizer, as shown in Figure 17-5.

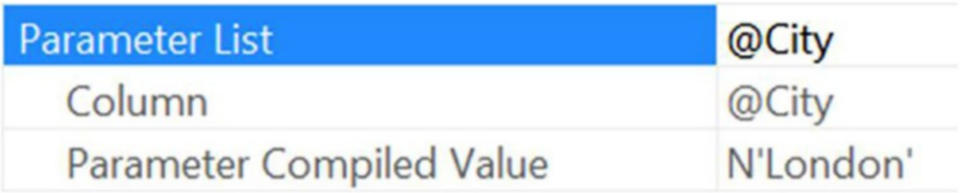


Figure 17-5. *Parameter values used to compile the query plan*

You can then use this value to look at your statistics to understand why you're seeing a plan that is different from what you expected. In this case, if I run the following query, I can check out the histogram to see where values like `London` would likely be stored and how many rows I can expect:

```
DBCC SHOW_STATISTICS('Person.Address','_WA_Sys_00000004_164452B1');
```

Figure 17-6 shows the applicable part of the histogram.

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
85	Lavender Bay	5	85	3	1.666667
86	Lebanon	0	111	0	1
87	Leeds	0	55	0	1
88	Lemon Grove	32	109	2	16
89	Les Ulis	0	94	0	1
90	Lille	32	56	2	16
91	Lincoln Acres	0	102	0	1
92	London	32	434	2	16
93	Long Beach	0	97	0	1
94	Los Angeles	2	93	2	1
95	Lynnwood	2	101	1	2
96	Malabar	32	81	2	16

Figure 17-6. *Part of the histogram showing how many rows you can expect*

You can see that the value of `London` returns a lot more rows than any of the average rows displayed in `AVG_RANGE_ROWS`, and it's higher than many of the other steps `RANG_HI_KEY` counts that are stored in `EQ_ROWS`. In short, the value for `London` is skewed from the rest of the data. That's why the plan there is different from others.

You'll have to go through the same sort of evaluation of the statistics and compile-time parameter values to understand where bad parameter sniffing is coming from.

But, if you have a parameterized query that is suffering from bad parameter sniffing, you can take control in several different ways to attempt to reduce the problem.

Mitigating Bad Parameter Sniffing

Once you've identified that you're experiencing bad parameter sniffing in one case, you don't just have to suffer with it. You can do something about it, but you have to make a decision. You have several choices for mitigating the behavior of bad parameter sniffing.

- You can force a recompile of the plan at the time of execution by running `sp_recompile` against the procedure prior to executing.
- Another way to force the recompile is to use `EXEC <procedure name> WITH RECOMPILE`.

- Yet another mechanism for forcing recompiles on each execution would be to create the procedure using `WITH RECOMPILE` as part of the procedure definition.
- You can also use `OPTION (RECOMPILE)` on individual statements to have only those statements instead of the entire procedure recompile. This is frequently the best approach if you're going to force recompiles. Just know that this is a trade-off between execution time and compile time. You could see serious issues if this query is called frequently and recompiled every time.
- You can reassign input parameters to local variables. This popular fix forces the optimizer to make a best guess at the values likely to be used by looking at the statistics of the data being referenced, which can and does eliminate the values being taken into account. This is the old way of doing it and has been replaced by using `OPTIMIZE FOR UNKNOWN`. This method also suffers from the possibility of variable sniffing during recompiles.
- You can use a query hint, `OPTIMIZE FOR`, when you create the procedure and supply it with known good parameters that will generate a plan that works well for most of your queries. You can specify a value that generates a specific plan, or you can specify `UNKNOWN` to get a generic plan based on the average of the statistics.
- You can use a plan guide, which is a mechanism to get a query to behave a certain way without making modifications to the procedure. This will be covered in detail in Chapter 18.
- You can use plan forcing if you have the Query Store enabled to choose the preferred plan. This is an elegant solution since it doesn't require any code changes to implement.
- You can disable parameter sniffing for the server by setting trace flag 4136 to on. Understand that this beneficial behavior will be turned off for the entire server, not just one problematic query. This is potentially a highly dangerous choice to make for your system.

- You can now disable parameter sniffing at the database level using `DATABASE SCOPED CONFIGURATION` to turn off parameter sniffing at the database level. This is a much safer operation than using the trace flag as outlined earlier. It is still potentially problematic since most databases are benefiting from parameter sniffing.
- If you have a particular query pattern that leads to bad parameter sniffing, you can isolate the functionality by setting up two, or more, different procedures using a wrapper procedure to determine which to call. This can help you use multiple different approaches at the same time. You can also address this issue using dynamic string execution; just be cautious of SQL injection.

Each of these possible solutions comes with trade-offs that must be taken into account. If you decide to just recompile the query each time it's called, you'll have to pay the price for the additional CPU needed to recompile the query. This goes against the whole idea of trying to get plan reuse by using parameterized queries, but it could be the best solution in your circumstances. Reassigning your parameters to local variables is something of an old-school approach; the code can look quite silly.

```
CREATE OR ALTER PROC dbo.AddressByCity @City NVARCHAR(30)
AS
DECLARE @LocalCity NVARCHAR(30) = @City;

SELECT a.AddressID,
       a.AddressLine1,
       AddressLine2,
       a.City,
       sp.Name AS StateProvinceName,
       a.PostalCode
FROM Person.Address AS a
    JOIN Person.StateProvince AS sp
        ON a.StateProvinceID = sp.StateProvinceID
WHERE a.City = @LocalCity;
```

Using this approach, the optimizer makes its cardinality estimates based on the density of the columns in question, not using the histogram. But it looks odd in a query. In fact, if you take this approach, I strongly suggest adding a comment in front of the variable declaration so it's clear why you're doing this. Here's an example:

```
-- This allows the query to bypass bad parameter sniffing
```

But, with this approach you're now subject to the possibility of variable sniffing, so it's not really recommended unless you're on a SQL Server instance that is older than 2008. From SQL Server 2008 and onward, you're better off using the `OPTIMIZE FOR UNKOWN` query hint to achieve the same result without the problems of variable sniffing possibly being introduced.

You can use the `OPTIMIZE FOR` query hint and pass a specific value. So, for example, if you wanted to be sure that the plan that was generated by the value `Mentor` is always used, you can do this to the query:

```
CREATE OR ALTER PROC dbo.AddressByCity @City NVARCHAR(30)
AS
SELECT a.AddressID,
       a.AddressLine1,
       AddressLine2,
       a.City,
       sp.Name AS StateProvinceName,
       a.PostalCode
FROM Person.Address AS a
    JOIN Person.StateProvince AS sp
        ON a.StateProvinceID = sp.StateProvinceID
WHERE a.City = @City
OPTION (OPTIMIZE FOR (@City = 'Mentor'));
```

Now the optimizer will ignore any values passed to `@City` and will always use the value of `Mentor`. You can even see this in action if you modify the query as shown, which will remove the query from cache, and then you execute it using the parameter value of `London`. This will generate a new plan in the cache. If you open that plan and look at the `SELECT` properties, you'll see evidence of the hint in Figure 17-7.

Parameter List	@City
Column	@City
Parameter Compiled Value	N'Mentor'
Parameter Runtime Value	N'London'

Figure 17-7. *Runtime and compile-time values differ*

As you can see, the optimizer did exactly as you specified and used the value `Mentor` to compile the plan even though you can also see that you executed the query using the value `London`. The problem with this approach is that data changes over time and what might have been an optimal plan for your data at point is no longer. If you choose to use the `OPTIMIZE FOR` hint, you need to plan to regularly reassess it.

If you choose to disable parameter sniffing entirely by using the trace flag or the `DATABASE SCOPED CONFIGURATION`, understand that it turns it off on the entire server or database. Since, most of the time, parameter sniffing is absolutely helping you, you had best be sure that you're receiving no benefits from it and the only hope of dealing with it is to turn off sniffing. This doesn't require even a server reboot, so it's immediate. The plans generated will be based on the averages of the statistics available, so the plans can be seriously suboptimal depending on your data. Before doing this, explore the possibility of using the `RECOMPILE` hint on your most problematic queries. You're more likely to get better plans that way even though you won't get plan reuse.

The simplest approach to dealing with parameter sniffing has to be the use of plan forcing through the Query Store, assuming you're in a situation where one particular plan is the most useful. You can use the reports in the GUI, or you can retrieve information directly from the system views.

```
SELECT CAST(qsp.query_plan AS XML) AS query_plan,
       qsp.plan_id,
       qsq.query_id
FROM sys.query_store_plan AS qsp
    JOIN sys.query_store_query AS qsq
        ON qsq.query_id = qsp.query_id
WHERE qsq.object_id = OBJECT_ID('dbo.AddressByCity');
```

You have all you need to determine which execution plan will best suit the needs of your system. Once you have it determined, it's a simple matter to force the plan choice on the optimizer. To see this in action, let's force the plan that is better suited to the value `Mentor`. Assuming you've been running with the Query Store enabled, you should be able to retrieve the data using the previous query and pick that plan. If not, enable the Query Store (see Chapter 11 for the details) and then run both queries, taking the time to clear the plan from the cache between the executions using the previous scripts.

After you've completed that, you have to use the values for `query_id` and `plan_id` along with the `sys.sp_query_store_force_plan` function.

```
EXEC sys.sp_query_store_force_plan 1545, 1602;
```

The result is not immediately apparent. However, if we rerun the stored procedure passing it a value of `London`, we will see the plan in Figure 17-8.

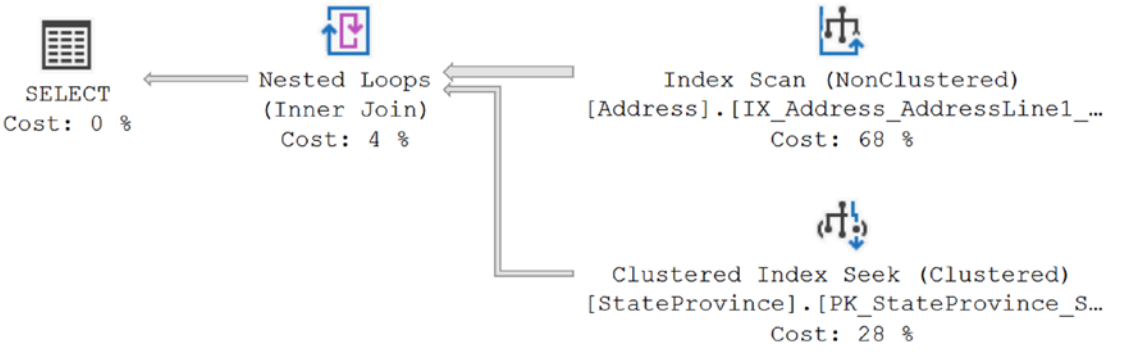


Figure 17-8. *A forced execution plan*

You can try removing the plan from cache and rerunning it for the value of `London`. However, nothing you do at this point will bring back that execution plan because the optimizer is now forcing the plan. You can monitor plan forcing using Extended Events. You can also query the Query Store views to see which plans are forced. Finally, the plan itself stores a little bit of information to let you know that it is a forced plan. Looking at the first operator, in this case the `SELECT` operator, you can see the properties in Figure 17-9.

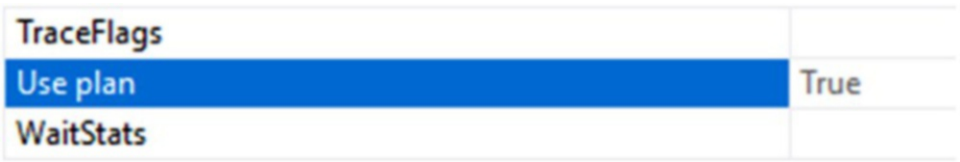


Figure 17-9. *The Use plan property showing a forced execution plan*

This is the one indication that you can see within the execution plan that it has been forced. There's no indication of the source, so you'll have to look to the reports within SSMS or query the tables to track down the information yourself. There is a dedicated report shown in Figure 17-10.

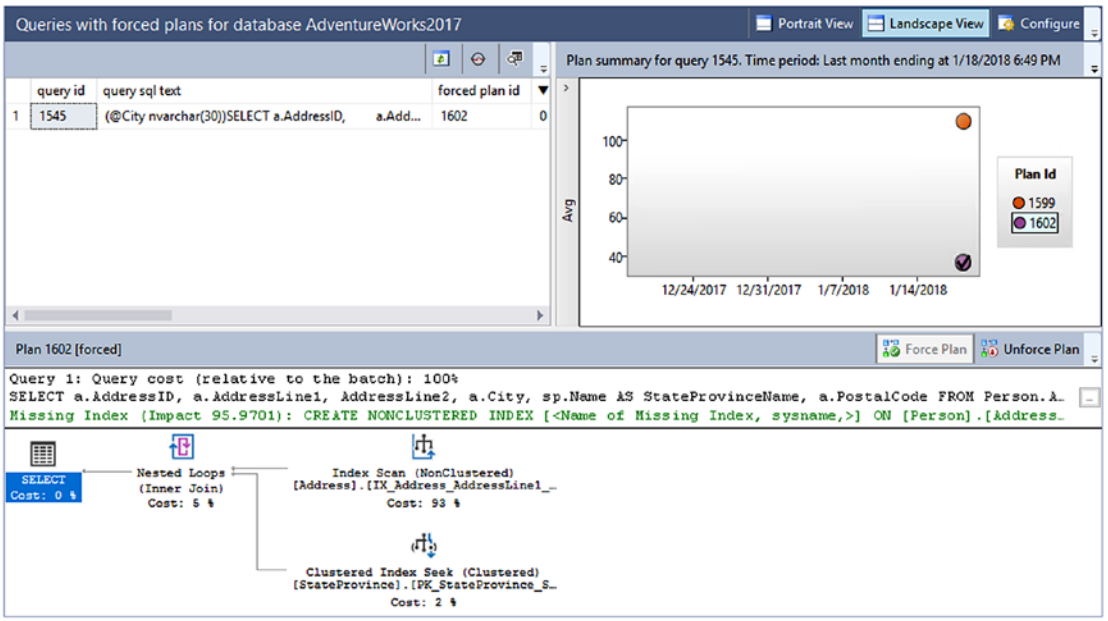


Figure 17-10. *The queries with forced plans report*

You can see that there are two different plans for the query. You can even see the checkmark on the plan, 1602 in Figure 17-10, indicating that it is a forced plan.

Before proceeding, remove the plan forcing using the GUI or the following command:

```
EXEC sys.sp_query_store_unforce_plan 1545, 1602;
```

With all these possible mitigation approaches, test carefully on your systems before you decide on an approach. Each of these approaches works, but they work in ways that may be better in one circumstance than another, so it's good to know the different methods, and you can experiment with them all depending on your situation.

Finally, remember that this is driven by statistics, so if your statistics are inaccurate or out-of-date, you're more likely to get bad parameter sniffing. Reexamining your statistics maintenance routines to ensure their efficacy is frequently the single best solution.

Summary

In this chapter, I outlined exactly what parameter sniffing is and how it benefits all your parameterized queries most of the time. That's important to keep in mind because when you run into bad parameter sniffing, it can seem like parameter sniffing is more danger than it's worth. I discussed how statistics and data distribution can create plans that are suboptimal for some of the data set even as they are optimal for other parts of the data. This is bad parameter sniffing at work. There are several ways to mitigate bad parameter sniffing, but each one is a trade-off, so examine them carefully to ensure you do what's best for your system.

In the next chapter, I'll talk about what happens to cause queries to recompile and what can be done about that.

CHAPTER 18

Query Recompilation

Stored procedures and parameterized queries improve the reusability of an execution plan by explicitly converting the variable parts of the queries into parameters. This allows execution plans to be reused when the queries are resubmitted with the same or different values for the variable parts. Since stored procedures are mostly used to implement complex business rules, a typical stored procedure contains a complex set of SQL statements, making the price of generating the execution plan of the queries within a stored procedure a bit costly. Therefore, it is usually beneficial to reuse the existing execution plan of a stored procedure instead of generating a new plan. However, sometimes the existing plan may not be optimal, or it may not provide the best processing strategy during reuse. SQL Server resolves this condition by recompiling statements within stored procedures to generate a new execution plan. This chapter covers the following topics:

- The benefits and drawbacks of recompilation
- How to identify the statements causing recompilation
- How to analyze the causes of recompilations
- Ways to avoid recompilations when necessary

Benefits and Drawbacks of Recompilation

The recompilation of queries can be both beneficial and harmful. Sometimes, it may be beneficial to consider a new processing strategy for a query instead of reusing the existing plan, especially if the data distribution in the table, and the corresponding statistics, has changed. The addition of new indexes, constraints, or modifications to existing structures within a table could also result in a recompiled query performing better. Recompiles in SQL Server and Azure SQL Database are at the statement level.

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_18

This increases the overall number of recompiles that can occur within a procedure, but it reduces the effects and overhead of recompiles in general. Statement-level recompiles reduce overhead because they recompile only an individual statement rather than all the statements within a procedure, whereas recompiles in SQL Server 2000 caused a procedure, in its entirety, to be recompiled over and over. Despite this smaller footprint for recompiles, they are generally considered to be something to be reduced and controlled as much as is practical for your situation.

The exception to the standard recompile process is when plan forcing is enabled using the Query Store. In that case, a recompile will still occur. However, the plan that gets generated will be used only if the plan that exists within the Query Store that has been marked as the forced plan is invalid. If that marked plan is invalid, the newly generated plan will be used.

To understand how the recompilation of an existing plan can sometimes be beneficial, assume you need to retrieve some information from the `Production.WorkOrder` table. The stored procedure may look like this:

```
CREATE OR ALTER PROCEDURE dbo.WorkOrder
AS
SELECT wo.WorkOrderID,
       wo.ProductID,
       wo.StockedQty
FROM Production.WorkOrder AS wo
WHERE wo.StockedQty BETWEEN 500
                    AND     700;
```

With the current indexes, the execution plan for the `SELECT` statement, which is part of the stored procedure plan, scans the index `PK_WorkOrder_WorkOrderID`, as shown in Figure 18-1.

Figure 18-1. *Execution plan for the stored procedure*

This plan is saved in the procedure cache so that it can be reused when the stored procedure is reexecuted. But if a new index is added on the table as follows, then the existing plan won't be the most efficient processing strategy to execute the query.

```
CREATE INDEX IX_Test ON Production.WorkOrder(StockedQty,ProductID);
```

In this case, it is beneficial to spend extra CPU cycles to recompile the stored procedure so that you generate a better execution plan.

Since index `IX_Test` can serve as a covering index for the `SELECT` statement, the cost of a bookmark lookup can be avoided by using index `IX_Test` instead of scanning `PK_WorkOrder_WorkOrderID`. SQL Server automatically detects that the new plan was created and recompiles the existing plan to consider the benefit of using the new index. This results in a new execution plan for the stored procedure (when executed), as shown in Figure 18-2.

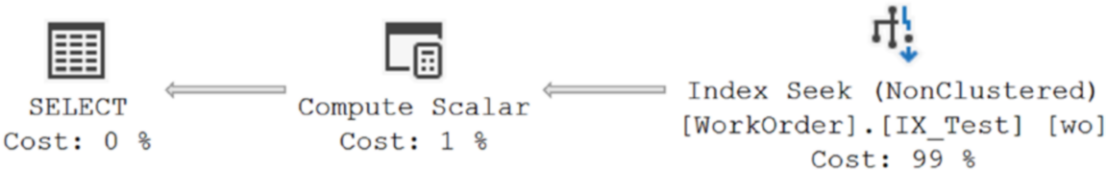


Figure 18-2. *New execution plan for the stored procedure*

SQL Server automatically detects the conditions that require a recompilation of the existing plan. SQL Server follows certain rules in determining when the existing plan needs to be recompiled. If a specific implementation of a query falls within the rules of recompilation (execution plan aged out, `SET` options changed, and so on), then the statement will be recompiled every time it meets the requirements for a recompile, and SQL Server may, or may not, generate a better execution plan. To see this in action, you'll need a different stored procedure. The following procedure returns all the rows from the `WorkOrder` table:

```
CREATE OR ALTER PROCEDURE dbo.WorkOrderAll
AS
SELECT *
FROM Production.WorkOrder AS wo;
```

Before executing this procedure, drop the index `IXTest`.

```
DROP INDEX Production.WorkOrder.IX_Test;
```

When you execute this procedure, the SELECT statement returns the complete data set (all rows and columns) from the table and is therefore best served through a table scan on the table WorkOrder. If we had a more appropriate query with a limited SELECT list, a scan of a nonclustered index could be an option. As explained in Chapter 4, the processing of the SELECT statement won't benefit from a nonclustered index on any of the columns. Therefore, ideally, creating the nonclustered index (as follows) before the execution of the stored procedure shouldn't matter.

```
EXEC dbo.WorkOrderAll;
GO
CREATE INDEX IX_Test ON Production.WorkOrder(StockedQty,ProductID);
GO
EXEC dbo.WorkOrderAll; --After creation of index IX_Test
```

But the stored procedure execution after the index creation faces recompilation, as shown in the corresponding extended event output in Figure 18-3.

name	statement	recompile_cause	attach_activity_id.seq	batch_text
sql_statement_recompile	SELECT * FROM Production.W...	Schema changed	1	NULL
sql_batch_completed	NULL	NULL	2	EXEC dbo.WorkOrderAll; --...

Figure 18-3. *Nonbeneficial recompilation of the stored procedure*

The sql_statement_recompile event was used to trace the statement recompiles. There is no longer a separate procedure recompile event as there was in the older trace events.

In this case, the recompilation is of no real benefit to the stored procedure. But unfortunately, it falls within the conditions that cause SQL Server to recompile the stored procedure on every execution in which the schema has been changed. This can make plan caching for the stored procedure ineffective and wastes CPU cycles in regenerating the same plan on this execution. Therefore, it is important to be aware of the conditions that cause the recompilation of queries and to make every effort to avoid those conditions when implementing stored procedures and parameterized queries that are targeted for plan reuse. I will discuss these conditions next, after identifying which statements cause SQL Server to recompile the statement in each respective case.

Identifying the Statement Causing Recompilation

SQL Server can recompile individual statements within a procedure or the entire procedure. Thus, to find the cause of recompilation, it's important to identify the SQL statement that can't reuse the existing plan.

You can use Extended Events sessions to track statement recompilation. You can also use the same events to identify the stored procedure statement that caused the recompilation. These are the relevant events you can use:

- `sql_batch_completed` and/or `rpc_completed`
- `sql_statement_recompile`
- `sql_batch_starting` and/or `rpc_starting`
- `sql_statement_completed` and/or `sp_statement_completed` *(Optional)*
- `sql_statement_starting` and/or `sp_statement_completed` *(Optional)*

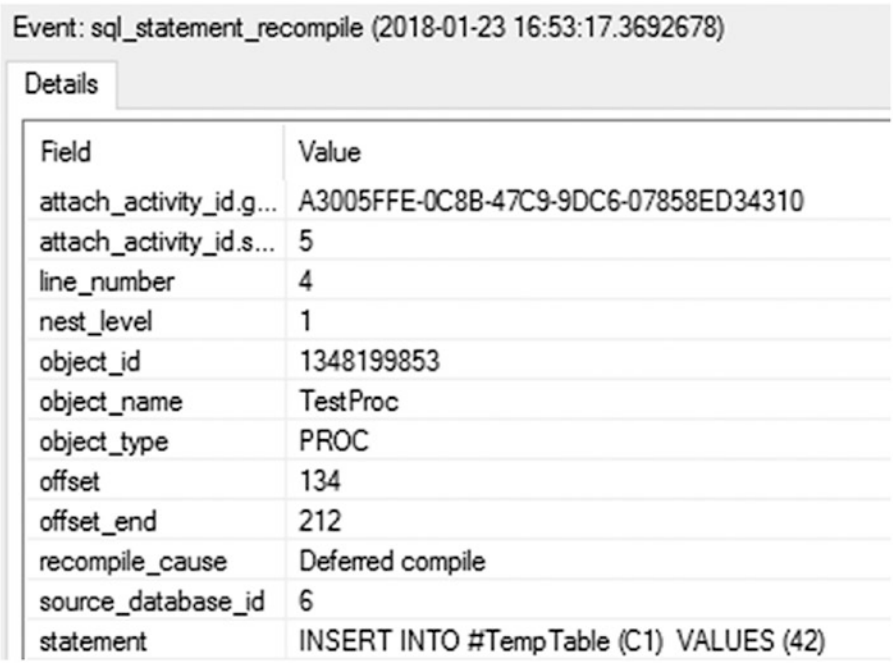
Note SQL Server 2008 supported Extended Events, but the `rpc_completed` and `rpc_starting` events didn't return the correct information. For older queries, you may have to substitute `module_end` and `module_starting`.

Consider the following simple stored procedure:

```
CREATE OR ALTER PROC dbo.TestProc
AS
CREATE TABLE #TempTable (C1 INT);
INSERT INTO #TempTable (C1)
VALUES (42);
-- data change causes recompile
GO
```

On executing this stored procedure the first time, you get the Extended Events output shown in Figure 18-4.

```
EXEC dbo.TestProc;
```


Event: sql_statement_recompile (2018-01-23 16:53:17.3692678)

Details

Field	Value
attach_activity_id.g...	A3005FFE-0C8B-47C9-9DC6-07858ED34310
attach_activity_id.s...	5
line_number	4
nest_level	1
object_id	1348199853
object_name	TestProc
object_type	PROC
offset	134
offset_end	212
recompile_cause	Deferred compile
source_database_id	6
statement	INSERT INTO #TempTable (C1) VALUES (42)

Figure 18-4. *Extended Events output showing an sql_statement_recompile event from recompilation*

In Figure 18-4, you can see that you have a recompilation event (`sql_statement_recompile`), indicating that a statement inside the stored procedure went through recompilation. When a stored procedure is executed for the first time, SQL Server compiles the stored procedure and generates an execution plan for all the statements within it, as explained in the previous chapter.

By the way, you might see other statements if you're using Extended Events to follow along. Just filter or group by your database ID to make it easier to see the events you're interested in. It's always a good idea to put filters on your Extended Events sessions.

Since execution plans are maintained in volatile memory only, they get dropped when SQL Server is restarted. On the next execution of the stored procedure, after the server restart, SQL Server once again compiles the stored procedure and generates the execution plan. These compilations aren't treated as a stored procedure recompilation since a plan didn't exist in the cache for reuse. An `sql_statement_recompile` event indicates that a plan was already there but couldn't be reused.

Note I discuss the significance of the `recompile_cause` data column later in the "Analyzing Causes of Recompilation" section.

To see which statement caused the recompile, look at the `statement` column within the `sql_statement_recompile` event. It shows specifically the statement being recompiled. You can also identify the stored procedure statement causing the recompilation by using any of the various statement starting events in combination with a recompile event. If you enable Causality Tracking as part of the Extended Events session, you'll get an identifier for the start of an event and then sequence numbers of other events that are part of the same chain. The Id and sequence number are the first two columns in Figure 18-4.

Note that after the statement recompilation, the stored procedure statement that caused the recompilation is started again to execute with the new plan. You can capture the statement within the event, correlate the events through sequence using the timestamps, or, best of all, use the Causality Tracking on the extended events. Any of these can be used to track down specifically which statement is causing the recompile.

Analyzing Causes of Recompilation

To improve performance, it is important that you analyze the causes of recompilation. Often, recompilation may not be necessary, and you can avoid it to improve performance. For example, every time you go through a compile or recompile process, you're using the CPU for the optimizer to get its job done. You're also moving plans in and out of memory as they go through the compile process. When a query recompiles, that query is blocked while the recompile process runs, which means frequently called queries can become major bottlenecks if they also have to go through a recompile. Knowing the different conditions that result in recompilation helps you evaluate the cause of a recompilation and determine how to avoid recompiling when it isn't necessary. Statement recompilation occurs for the following reasons:

- The schema of regular tables, temporary tables, or views referred to in the stored procedure statement have changed. Schema changes include changes to the metadata of the table or the indexes on the table.
- Bindings (such as defaults) to the columns of regular or temporary tables have changed.

- Statistics on the table indexes or columns have changed, either automatically or manually, beyond the thresholds discussed in Chapter 13.
- An object did not exist when the stored procedure was compiled, but it was created during execution. This is called *deferred object resolution*, which is the cause of the preceding recompilation.
- `SET` options have changed.
- The execution plan was aged and deallocated.
- An explicit call was made to the `sp_recompile` system stored procedure.
- There was an explicit use of the `RECOMPILE` hint.

You can see these causes in Extended Events. The cause is indicated by the `recompile_cause` data column value for the `sql_statement_recompile` event. Let's look at some of the reasons listed above for recompilation in more detail and discuss what you can do to avoid them.

Schema or Bindings Changes

When the schema or bindings to a view, regular table, or temporary table change, the existing query's execution plan becomes invalid. The query must be recompiled before executing any statement that refers to a modified object. SQL Server automatically detects this situation and recompiles the stored procedure.

Note I talk about recompilation due to schema changes in more detail in the "Benefits and Drawbacks of Recompilation" section.

Statistics Changes

SQL Server keeps track of the number of changes to the table. If the number of changes exceeds the recompilation threshold (RT) value, then SQL Server automatically updates the statistics when the table is referred to in the statement, as you saw in

Chapter 13. When the condition for the automatic update of statistics is detected, SQL Server automatically marks the statement for recompile, along with the statistics update.

The RT is determined by a formula that depends on the table being a permanent table or a temporary table (not a table variable) and how many rows are in the table. Table 18-1 shows the basic formula so that you can determine when you can expect to see a statement recompile because of data changes.

Table 18-1. *Formula for Determining Data Changes*

Type of Table	Formula
Permanent table	If number of rows (n) <= 500, then RT = 500.
	If n > 500, then RT = .2 * n or Sqrt(1000*NumberOfRows).
Temporary table	If n < 6, then RT = 6.
	If 6 <= n <= 500, then RT = 500. If n > 500, then RT = .2 * n or Sqrt(1000*NumberOfRows).

To understand how statistics changes can cause recompilation, consider the following example. The stored procedure is executed the first time with only one row in the table. Before the second execution of the stored procedure, a large number of rows are added to the table.

Note Please ensure that the AUTO_UPDATE_STATISTICS setting for the database is ON. You can determine the AUTO_UPDATE_STATISTICS setting by executing the following query:

```
SELECT  DATABASEPROPERTYEX('AdventureWorks2017', 'IsAutoUpdateStatistics');
```

```
IF EXISTS (   SELECT *
              FROM sys.objects AS o
              WHERE o.object_id = OBJECT_ID(N'dbo.NewOrderDetail')
                    AND o.type IN ( N'U' ))
```