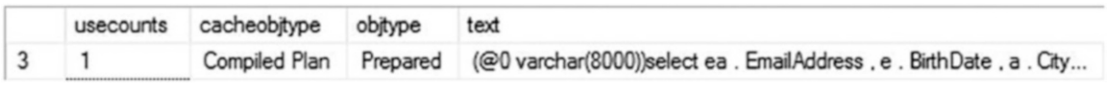
The output from sys.dm_exec_cached_plans changes so that the output looks different, as shown in Figure 16-13.



| | usecounts | cacheobjtype | objtype | text |
|---|---|---|---|---|
| 3 | 1 | Compiled Plan | Prepared | (@0 varchar(8000))select ea . EmailAddress , e . BirthDate , a . City... |

***Figure 16-13.** Forced parameterization changes the plan*

Now a prepared plan is visible in the third row. However, only a single parameter was supplied, `@0 varchar(8000)`. If you get the full text of the prepared plan out of `sys.dm_exec_querytext` and format it, it looks like this:

```
(@0 varchar(8000))
SELECT  ea.EmailAddress,
        e.BirthDate,
        a.City
FROM    Person.Person AS p
JOIN    HumanResources.Employee AS e
        ON p.BusinessEntityID = e.BusinessEntityID
JOIN    Person.BusinessEntityAddress AS bea
        ON e.BusinessEntityID = bea.BusinessEntityID
JOIN    Person.Address AS a
        ON bea.AddressID = a.AddressID
JOIN    Person.StateProvince AS sp
        ON a.StateProvinceID = sp.StateProvinceID
JOIN    Person.EmailAddress AS ea
        ON p.BusinessEntityID = ea.BusinessEntityID
WHERE   ea.EmailAddress LIKE 'david%'
        AND sp.StateProvinceCode = @0
```

Because of its restrictions, forced parameterization was unable to substitute anything for the string `'david%'`, but it was able to for the string `'WA'`. Worth noting is that the variable was declared as a full 8,000-length `VARCHAR` instead of the three-character `NCHAR` like the actual column in the `Person.StateProvince` table. Even though the parameter value here might be different than the actual column value in the database, this will not lead to the loss of index use. The implicit data conversion for string length, such as from `VARCHAR(8000)` to `VARCHAR(8)`, won't cause problems.

Before you start using forced parameterization, the following list of restrictions may give you information to help you decide whether forced parameterization will work in your database. (This is a partial list; for the complete list, please consult Books Online.)

- `INSERT ... EXECUTE` queries
- Statements inside procedures, triggers, and user-defined functions since they already have execution plans
- Client-side prepared statements (you'll find more detail on these later in this chapter)
- Queries with the query hint `RECOMPILE`
- Pattern and escape clause arguments used in a `LIKE` statement (as shown earlier)

This gives you an idea of the types of restrictions placed on forced parameterization. Forced parameterization is going to be potentially helpful only if you are suffering from large amounts of compiles and recompiles because of ad hoc queries. Any other load won't benefit from the use of forced parameterization.

Before continuing, change the database back to `SIMPLE PARAMETERIZATION`.

```
ALTER DATABASE AdventureWorks2017 SET PARAMETERIZATION SIMPLE;
```

One other topic around parameterization that is worth mentioning is how Azure SQL Database deals with the issue. If a query is being recompiled regularly but always getting the same execution plan, you may see a tuning recommendation in Azure suggesting that you turn on `FORCED PARAMETERIZATION`. It's an aspect of the automated tuning recommendations that I'll cover in detail in Chapter 25.

## Plan Reusability of a Prepared Workload

Defining queries as a prepared workload allows the variable parts of the queries to be explicitly parameterized. This enables SQL Server to generate a query plan that is not tied to the variable parts of the query, and it keeps the variable parts separate in an execution context. As you saw in the previous section, SQL Server supports three techniques to submit a prepared workload.

- Stored procedures
- `sp_executesql`
- Prepare/execute model

In the sections that follow, I cover each of these techniques in more depth and point out where it's possible for parameterized execution plans to cause problems.

## Stored Procedures

Using stored procedures is a standard technique for improving the effectiveness of plan caching. When the stored procedure is compiled at execution time (this is different for native compiled procedures, which are covered in Chapter 24), a plan is generated for each of the SQL statements within the stored procedure. The execution plan generated for the stored procedure can be reused whenever the stored procedure is reexecuted with different parameter values.

In addition to checking `sys.dm_exec_cached_plans`, you can track the execution plan caching for stored procedures using the Extended Events tool. Extended Events provides the events listed in Table 16-2 to track the plan caching for stored procedures.

***Table 16-2.*** *Events to Analyze Plan Caching for the Stored Procedures Event Class*

| Event | Description |
|---|---|
| `sp_cache_hit` | The plan is found in the cache. |
| `sp_cache_miss` | The plan is not found in the cache. |
| `sp_cache_insert` | The event fires when a plan is added to cache. |
| `sp_cache_remove` | The event fires when a plan gets removed from cache. |

To track the stored procedure plan caching using trace events, you can use these events along with the other stored procedure events. To understand how stored procedures can improve plan caching, reexamine the procedure created earlier called `BasicSalesInfo`. The procedure is repeated here for clarity:

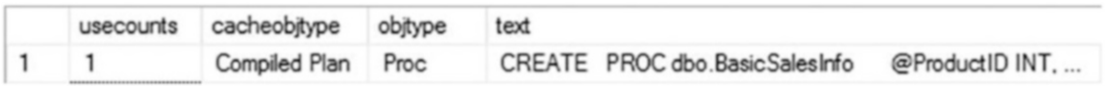
```
CREATE OR ALTER PROC dbo.BasicSalesInfo
    @ProductID INT,
    @CustomerID INT
```

```
AS
SELECT soh.SalesOrderNumber,
       soh.OrderDate,
       sod.OrderQty,
       sod.LineTotal
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE soh.CustomerID = @CustomerID
      AND sod.ProductID = @ProductID;
```

To retrieve a result set for `soh.Customerld = 29690` and `sod.ProductId=711`, you can execute the stored procedure like this:

```
EXEC dbo.BasicSalesInfo @CustomerID = 29690, @ProductID = 711;
```

Figure 16-14 shows the output of `sys.dm_exec_cached_plans`.

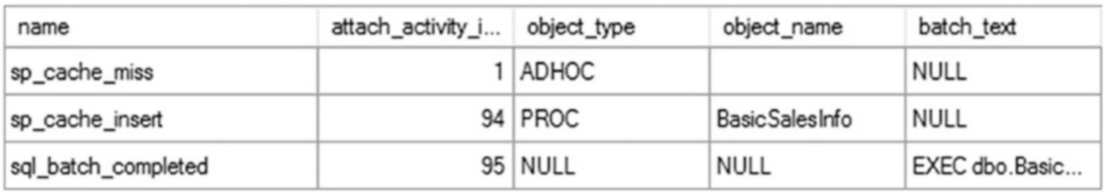


| | usecounts | cacheobjtype | objtype | text |
|---|---|---|---|---|
| 1 | 1 | Compiled Plan | Proc | CREATE PROC dbo.BasicSalesInfo @ProductID INT, ... |

***Figure 16-14.*** *sys.dm_exec_cached_plans output showing stored procedure plan caching*

From Figure 16-14, you can see that a compiled plan of type `Proc` is generated and cached for the stored procedure. The `usecounts` value of the executable plan is 1 since the stored procedure is executed only once.

Figure 16-15 shows the Extended Events output for this stored procedure execution.



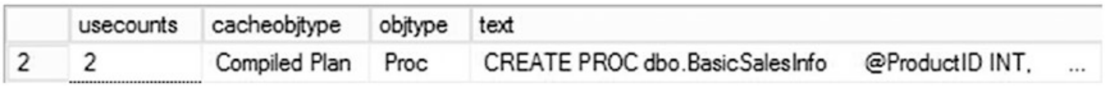
| name | attach_activity_i... | object_type | object_name | batch_text |
|---|---|---|---|---|
| sp_cache_miss | 1 | ADHOC | | NULL |
| sp_cache_insert | 94 | PROC | BasicSalesInfo | NULL |
| sql_batch_completed | 95 | NULL | NULL | EXEC dbo.Basic... |

***Figure 16-15.*** *Extended Events output showing that the stored procedure plan isn't easily found in the cache*

From the Extended Events output, you can see that the plan for the stored procedure is not found in the cache. When the stored procedure is executed the first time, SQL Server looks in the plan cache and fails to find any cache entry for the procedure `BasicSalesInfo`, causing an `sp_cache_miss` event. On not finding a cached plan, SQL Server makes arrangements to compile the stored procedure. Subsequently, SQL Server generates and saves the plan and proceeds with the execution of the stored procedure. You can see this in the `sp_cache_insert` event.

If this stored procedure is reexecuted to retrieve a result set for `@Productld = 777`, then the existing plan is reused, as shown in the `sys.dm_exec_cached_plans` output in Figure 16-16.

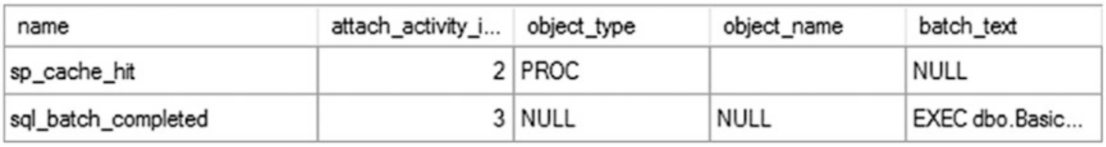
```
EXEC dbo.BasicSalesInfo @CustomerID = 29690, @ProductID = 777;
```



| | usecounts | cacheobjtype | objtype | text |
|---|---|---|---|---|
| 2 | 2 | Compiled Plan | Proc | CREATE PROC dbo.BasicSalesInfo @ProductID INT, ... |

***Figure 16-16.*** *sys.dm_exec_cached_plans output showing reuse of the stored procedure plan*

You can also confirm the reuse of the execution plan from the Extended Events output, as shown in Figure 16-17.



| name | attach_activity_i... | object_type | object_name | batch_text |
|---|---|---|---|---|
| sp_cache_hit | 2 | PROC | | NULL |
| sql_batch_completed | 3 | NULL | NULL | EXEC dbo.Basic... |

***Figure 16-17.*** *Profiler trace output showing reuse of the stored procedure plan*

From the Extended Events output, you can see that the existing plan is found in the plan cache. On searching the cache, SQL Server finds the executable plan for the stored procedure `BasicSalesInfo` causing an `sp_cache_hit` event. Once the existing execution plan is found, SQL reuses the plan to execute the stored procedure. One interesting note is that there is an `sp_cache_miss` event just prior to the `sp_cache_hit`, which is for the SQL batch calling the procedure. Because of the change to the parameter value, that statement was not found in the cache, but the procedure's execution plan was. This apparently "extra" cache miss event can cause confusion.

These other aspects of stored procedures are worth considering:

- Stored procedures are compiled on first execution.
- Stored procedures have other performance benefits, such as reducing network traffic.
- Stored procedures have additional benefits, such as the isolation of the data.

## Stored Procedures Are Compiled on First Execution

The execution plan of a stored procedure is generated when it is executed the first time. When the stored procedure is created, it is only parsed and saved in the database. No normalization and optimization processes are performed during the stored procedure creation. This allows a stored procedure to be created before creating all the objects accessed by the stored procedure. For example, you can create the following stored procedure, even when table `NotHere` referred to in the stored procedure does not exist:

```
CREATE OR ALTER PROCEDURE dbo.MyNewProc
AS
SELECT MyID
FROM dbo.NotHere; --Table dbo.NotHere doesn't exist
```

The stored procedure will be created successfully since the normalization process to bind the referred object to the query tree (generated by the command parser during the stored procedure execution) is not performed during the stored procedure creation. The stored procedure will report the error when it is first executed (if table `NotHere` is not created by then) since the stored procedure is compiled the first time it is executed.

## Other Performance Benefits of Stored Procedures

Besides improving the performance through execution plan reusability, stored procedures provide the following performance benefits:

- *Business logic is close to the data*: The parts of the business logic that perform extensive operations on data stored in the database should be put in stored procedures since SQL Server’s engine is extremely powerful for relational and set theory operations.

- *Network traffic is reduced*: The database application, across the network, sends just the name of the stored procedure and the parameter values. Only the processed result set is returned to the application. The intermediate data doesn't need to be passed back and forth between the application and the database.
- *The application is isolated from data structure changes*: If all critical data access is made through stored procedures, then when the database schema changes, the stored procedures can be re-created without affecting the application code that accesses the data through the stored procedures. In fact, the application accessing the database need not even be stopped.
- *There is a single point of administration*: All the business logic implemented in stored procedures is maintained as part of the database and can be managed centrally on the database itself. Of course, this benefit is highly relative, depending on whom you ask. To get a different opinion, ask a non-DBA!
- *Security can be increased*: User privileges on database tables can be restricted and can be allowed only through the standard business logic implemented in the stored procedure. For example, if you want user `UserOne` to be restricted from physically deleting rows from table `RestrictedAccess` and to be allowed to mark only the rows virtually deleted through stored procedure `MarkDeleted` by setting the rows' status as `'Deleted'`, then you can execute the `DENY` and `GRANT` commands as follows:

```
DROP TABLE IF EXISTS dbo.RestrictedAccess;
GO
CREATE TABLE dbo.RestrictedAccess (ID INT,
                                   Status VARCHAR(7));
INSERT INTO dbo.RestrictedAccess
VALUES (1, 'New');
GO
IF (SELECT OBJECT_ID('dbo.MarkDeleted')) IS NOT NULL
    DROP PROCEDURE dbo.MarkDeleted;
```

```
GO
CREATE PROCEDURE dbo.MarkDeleted @ID INT
AS
UPDATE dbo.RestrictedAccess
SET Status = 'Deleted'
WHERE ID = @ID;
GO

--Prevent user u1 from deleting rows
DENY DELETE ON dbo.RestrictedAccess TO  UserOne;

--Allow user u1 to mark a row as 'deleted'
GRANT EXECUTE ON dbo.MarkDeleted TO UserOne;
```

This assumes the existence of user `UserOne`. Note that if the query within the stored procedure `MarkDeleted` is built dynamically as a string (`@sql`) as follows, then granting permission to the stored procedure won't grant any permission to the query since the dynamic query isn't treated as part of the stored procedure:

```
CREATE OR ALTER PROCEDURE dbo.MarkDeleted @ID INT
AS
DECLARE @SQL NVARCHAR(MAX);

SET @SQL = 'UPDATE  dbo.RestrictedAccess
SET     Status = "Deleted"
WHERE   ID = ' + @ID;

EXEC sys.sp_executesql @SQL;
GO

GRANT EXECUTE ON dbo.MarkDeleted TO UserOne;
```

Consequently, user `UserOne` won't be able to mark the row as `'Deleted'` using the stored procedure `MarkDeleted`. (I cover the aspects of using a dynamic query in the stored procedure in the next chapter.) However, if that user had explicit privileges or a role membership that granted that execution, this wouldn't work.

Since stored procedures are saved as database objects, they add deployment and management overhead to the database administration. Many times, you may need to execute just one or a few queries from the application. If these singleton queries are executed frequently, you should aim to reuse their execution plans to improve performance. But creating stored procedures for these individual singleton queries adds a large number of stored procedures to the database, increasing the database administrative overhead significantly. To avoid the maintenance overhead of using stored procedures and yet derive the benefit of plan reuse, submit the singleton queries as a prepared workload using the `sp_executesql` system stored procedure.

## sp_executesql

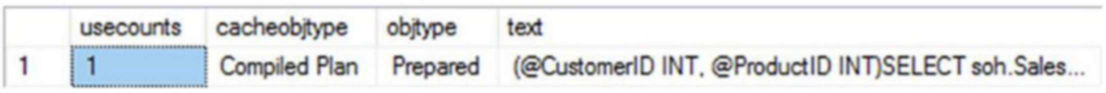
`sp_executesql` is a system stored procedure that provides a mechanism to submit one or more queries as a prepared workload. It allows the variable parts of the query to be explicitly parameterized, and it can therefore provide execution plan reusability as effective as a stored procedure. The `SELECT` statement from `BasicSalesInfo` can be submitted through `sp_ executesql` as follows:

```
DECLARE @query NVARCHAR(MAX),
        @paramlist NVARCHAR(MAX);

SET @query
    = N'SELECT soh.SalesOrderNumber,
       soh.OrderDate,
       sod.OrderQty,
       sod.LineTotal
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE soh.CustomerID = @CustomerID
      AND sod.ProductID = @ProductID';

SET @paramlist = N'@CustomerID INT, @ProductID INT';

EXEC sp_executesql @query,
                   @paramlist,
                   @CustomerID = 29690,
                   @ProductID = 711;
```

Note that the strings passed to the sp_executesql stored procedure are declared as NVARCHAR and that they are built with a prefix of N. This is required since sp_executesql uses Unicode strings as the input parameters.

The output of sys.dm_exec_cached_plans is shown next (see Figure 16-18):

```
SELECT c.usecounts,
       c.cacheobjtype,
       c.objtype,
       t.text
FROM sys.dm_exec_cached_plans AS c
    CROSS APPLY sys.dm_exec_sql_text(c.plan_handle) AS t
WHERE text LIKE '(@CustomerID%';
```



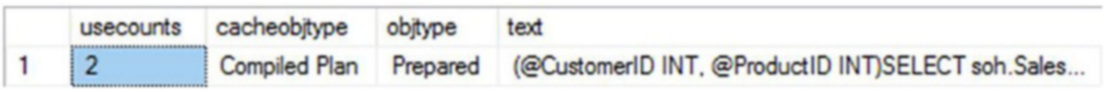
| | usecounts | cacheobjtype | objtype | text |
|---|---|---|---|---|
| 1 | 1 | Compiled Plan | Prepared | (@CustomerID INT, @ProductID INT)SELECT soh.Sales... |

***Figure 16-18.*** *sys.dm_exec_cached_plans output showing a parameterized plan generated using sp_executesql*

In Figure 16-18, you can see that the plan is generated for the parameterized part of the query submitted through sp_executesql. Since the plan is not tied to the variable part of the query, the existing execution plan can be reused if this query is resubmitted with a different value for one of the parameters (d.ProductID=777), as follows:

```
EXEC sp_executesql @query,@paramlist,@CustomerID = 29690,@ProductID = 777;
```

Figure 16-19 shows the output of sys.dm_exec_cached_plans.



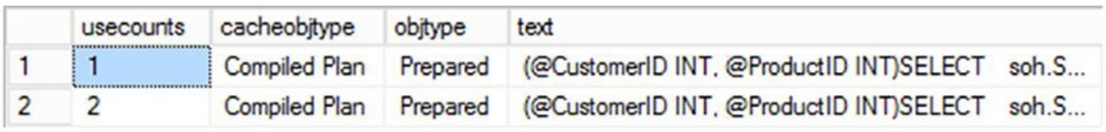
| | usecounts | cacheobjtype | objtype | text |
|---|---|---|---|---|
| 1 | 2 | Compiled Plan | Prepared | (@CustomerID INT, @ProductID INT)SELECT soh.Sales... |

***Figure 16-19.*** *sys.dm_exec_cached_plans output showing reuse of the parameterized plan generated using sp_executesql*

From Figure 16-19, you can see that the existing plan is reused (usecounts is 2 on the plan on line 2) when the query is resubmitted with a different variable value. If this query is resubmitted many times with different values for the variable part, the existing execution plan can be reused without regenerating new execution plans.

The query for which the plan is created (the text column) matches the exact textual string of the parameterized query submitted through sp_executesql. Therefore, if the same query is submitted from different parts of the application, ensure that the same textual string is used in all places. For example, if the same query is resubmitted with a minor modification in the query string (say in lowercase instead of uppercase letters), then the existing plan is not reused, and instead a new plan is created, as shown in the sys.dm_exec_cached_plans output in Figure 16-20.

```
SET @query = N'SELECT    soh.SalesOrderNumber ,soh.OrderDate ,sod.OrderQty
,sod.LineTotal FROM       Sales.SalesOrderHeader AS soh JOIN Sales.
SalesOrderDetail AS sod ON soh.SalesOrderID = sod.SalesOrderID where
soh.CustomerID = @CustomerID AND sod.ProductID = @ProductID' ;
```



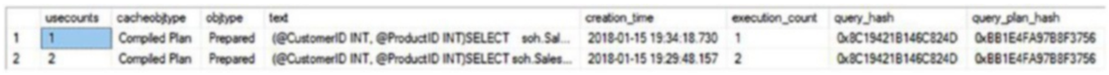
| | usecounts | cacheobjtype | objtype | text |
|---|---|---|---|---|
| 1 | 1 | Compiled Plan | Prepared | (@CustomerID INT, @ProductID INT)SELECT  soh.S... |
| 2 | 2 | Compiled Plan | Prepared | (@CustomerID INT, @ProductID INT)SELECT  soh.S... |

***Figure 16-20.*** *sys.dm_exec_cached_plans output showing sensitivity of the plan generated using sp_executesql*

Another way to see that there are two different plans created in the cache is to use additional dynamic management objects to see the properties of the plans in the cache.

```
SELECT  decp.usecounts,
        decp.cacheobjtype,
        decp.objtype,
        dest.text,
        deqs.creation_time,
        deqs.execution_count,
    deqs.query_hash,
    deqs.query_plan_hash
FROM    sys.dm_exec_cached_plans AS decp
CROSS APPLY sys.dm_exec_sql_text(decp.plan_handle) AS dest
JOIN    sys.dm_exec_query_stats AS deqs
        ON decp.plan_handle = deqs.plan_handle
WHERE   dest.text LIKE '(@CustomerID INT, @ProductID INT)%' ;
```

Figure 16-21 shows the results from this query.



| | usecounts | cacheobjtype | objtype | text | creation_time | execution_count | query_hash | query_plan_hash |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | Compiled Plan | Prepared | (@CustomerID INT, @ProductID INT)SELECT soh.Sal... | 2018-01-15 19:34:18.730 | 1 | 0x8C19421B146C824D | 0xBB1E4FA97B8F3756 |
| 2 | 2 | Compiled Plan | Prepared | (@CustomerID INT, @ProductID INT)SELECT soh.Sales... | 2018-01-15 19:29:48.157 | 2 | 0x8C19421B146C824D | 0xBB1E4FA97B8F3756 |

***Figure 16-21.** Additional output from sys.dm_exec_query_stats*

The output from `sys.dm_exec_query_stats` shows that the two versions of the query have different `creation_time` values. More interestingly, they have identical `query_hash` values but different `query_plan_hash` values (more on the hash values in that section later). All this shows that changing the case resulted in differing execution plans being stored in the cache.

In general, use `sp_executesql` to explicitly parameterize queries to make their execution plans reusable when the queries are resubmitted with different values for the variable parts. This provides the performance benefit of reusable plans without the overhead of managing any persistent object as required for stored procedures. This feature is exposed by both ODBC and OLEDB through `SQLExecDirect` and `ICommandWithParameters`, respectively. Like .NET developers or users of ADO.NET (ADO 2.7 or newer), you can submit the preceding `SELECT` statement using ADO `Command` and `Parameters`. If you set the ADO `Command Prepared` property to `FALSE` and use ADO `Command ('SELECT * FROM "Order Details" d, Orders o WHERE d.OrderID=o.OrderID and d.ProductID=?')` with ADO `Parameters`, ADO.NET will send the `SELECT` statement using `sp_executesql`. Most object-to-relational mapping tools, such as nHibernate or Entity Framework, also have mechanisms to allow for preparing statements and using parameters.

Finally, if you do have to build queries through strings like we did earlier, be sure to use parameters. When you pass in parameters, using any method, ensure that you're using strongly typed parameters and using those parameters as parameters within your T-SQL statements. All this will help to avoid SQL injection attacks.

Along with the parameters, `sp_executesql` sends the entire query string across the network every time the query is reexecuted. You can avoid this by using the prepare/execute model of ODBC and OLEDB (or OLEDB .NET).

### Prepare/Execute Model

ODBC and OLEDB provide a prepare/execute model to submit queries as a prepared workload. Like `sp_executesql`, this model allows the variable parts of the queries to be parameterized explicitly. The prepare phase allows SQL Server to generate the execution plan for the query and return a handle of the execution plan to the application. This execution plan handle is used by the execute phase to execute the query with different parameter values. This model can be used only to submit queries through ODBC or OLEDB, and it can't be used within SQL Server itself—queries within stored procedures can't be executed using this model.

The SQL Server ODBC driver provides the `SQLPrepare` and `SQLExecute` APIs to support the prepare/execute model. The SQL Server OLEDB provider exposes this model through the `ICommandPrepare` interface. The OLEDB .NET provider of ADO.NET behaves similarly.

**Note** For a detailed description of how to use the prepare/execute model in a database application, please refer to the MSDN article "SqlCommand.Prepare Method" (`http://bit.ly/2DBzN4b`).

## Query Plan Hash and Query Hash

With SQL Server 2008, new functionality around execution plans and the cache was introduced called the *query plan hash* and the *query hash*. These are binary objects using an algorithm against the query or the query plan to generate the binary hash value. These are useful for a common practice in developing known as *copy and paste*. You will find that common patterns and practices will be repeated throughout your code. Under the best circumstances, this is a good thing because you will see the best types of queries, joins, set-based operations, and so on, copied from one procedure to another as needed. But sometimes, you will see the worst possible practices repeated over and over again in your code. This is where the query hash and the query plan hash come into play to help you out.

You can retrieve the query plan hash and the query hash from `sys.dm_exec_query_stats` or `sys.dm_exec_requests`. You can also get the hash values from the Query Store. Although this is a mechanism for identifying queries and their plans, the hash values are not unique. Dissimilar plans can arrive at the same hash, so you can't rely on this as an alternate primary key.

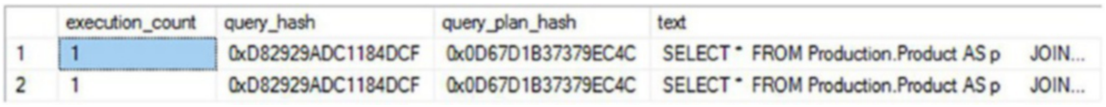
To see the hash values in action, create two queries.

```
SELECT *
FROM Production.Product AS p
    JOIN Production.ProductSubcategory AS ps
        ON p.ProductSubcategoryID = ps.ProductSubcategoryID
    JOIN Production.ProductCategory AS pc
        ON ps.ProductCategoryID = pc.ProductCategoryID
WHERE pc.Name = 'Bikes'
      AND ps.Name = 'Touring Bikes';
```

```
SELECT *
FROM Production.Product AS p
    JOIN Production.ProductSubcategory AS ps
        ON p.ProductSubcategoryID = ps.ProductSubcategoryID
    JOIN Production.ProductCategory AS pc
        ON ps.ProductCategoryID = pc.ProductCategoryID
where pc.Name = 'Bikes'
      and ps.Name = 'Road Bikes';
```

Note that the only substantial difference between the two queries is that `ProductSubcategory.Name` is different, with `Touring Bikes` in one and `Road Bikes` in the other. However, also note that the `WHERE` and `AND` keywords in the second query are lowercase. After you execute each of these queries, you can see the results of these format changes from `sys.dm_exec_query_stats` in Figure 16-22 from the following query:

```
SELECT deqs.execution_count,
       deqs.query_hash,
       deqs.query_plan_hash,
       dest.text
FROM sys.dm_exec_query_stats AS deqs
    CROSS APPLY sys.dm_exec_sql_text(deqs.plan_handle) AS dest
WHERE dest.text LIKE 'SELECT *
FROM Production.Product AS p%';
```

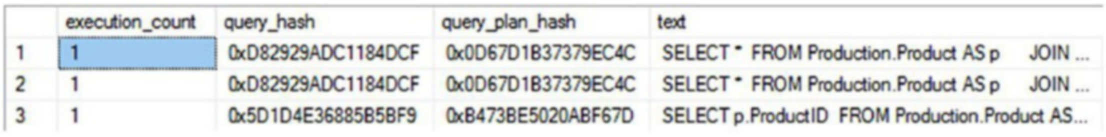


| | execution_count | query_hash | query_plan_hash | text |
|---|---|---|---|---|
| 1 | 1 | 0xD82929ADC1184DCF | 0x0D67D1B37379EC4C | SELECT * FROM Production.Product AS p JOIN... |
| 2 | 1 | 0xD82929ADC1184DCF | 0x0D67D1B37379EC4C | SELECT * FROM Production.Product AS p JOIN... |

***Figure 16-22.*** *sys.dm_exec_query_stats showing the plan hash values*

Two different plans were created because these are not parameterized queries; they are too complex to be considered for simple parameterization, and forced parameterization is off. These two plans have identical hash values because they varied only in terms of the values passed. The differences in case did not matter to the query hash or the query plan hash value. If, however, you changed the SELECT criteria, then the values would be retrieved from sys.dm_exec_query_stats, as shown in Figure 16-23, and the query would have changes.

```
SELECT   p.ProductID
FROM     Production.Product AS p
JOIN     Production.ProductSubcategory AS ps
         ON p.ProductSubcategoryID = ps.ProductSubcategoryID
JOIN     Production.ProductCategory AS pc
         ON ps.ProductCategoryID = pc.ProductCategoryID
WHERE    pc.[Name] = 'Bikes'
         AND ps.[Name] = 'Touring Bikes';
```



| | execution_count | query_hash | query_plan_hash | text |
|---|---|---|---|---|
| 1 | 1 | 0xD82929ADC1184DCF | 0x0D67D1B37379EC4C | SELECT * FROM Production.Product AS p JOIN ... |
| 2 | 1 | 0xD82929ADC1184DCF | 0x0D67D1B37379EC4C | SELECT * FROM Production.Product AS p JOIN ... |
| 3 | 1 | 0x5D1D4E36885B5BF9 | 0xB473BE5020ABF67D | SELECT p.ProductID FROM Production.Product AS... |

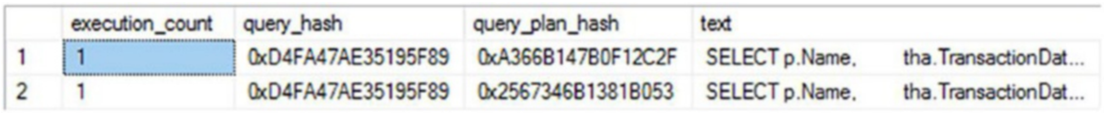
***Figure 16-23.*** *sys.dm_exec_query_stats showing a different hash*

Although the basic structure of the query is the same, the change in the columns returned was enough to change the query hash value and the query plan hash value.

Because differences in data distribution and indexes can cause the same query to come up with two different plans, the query_hash can be the same, and the query_plan_hash can be different. To illustrate this, execute two new queries.

```
SELECT p.Name,
       tha.TransactionDate,
       tha.TransactionType,
       tha.Quantity,
       tha.ActualCost
FROM Production.TransactionHistoryArchive AS tha
    JOIN Production.Product AS p
        ON tha.ProductID = p.ProductID
WHERE p.ProductID = 461;

SELECT p.Name,
       tha.TransactionDate,
       tha.TransactionType,
       tha.Quantity,
       tha.ActualCost
FROM Production.TransactionHistoryArchive AS tha
    JOIN Production.Product AS p
        ON tha.ProductID = p.ProductID
WHERE p.ProductID = 712;
```

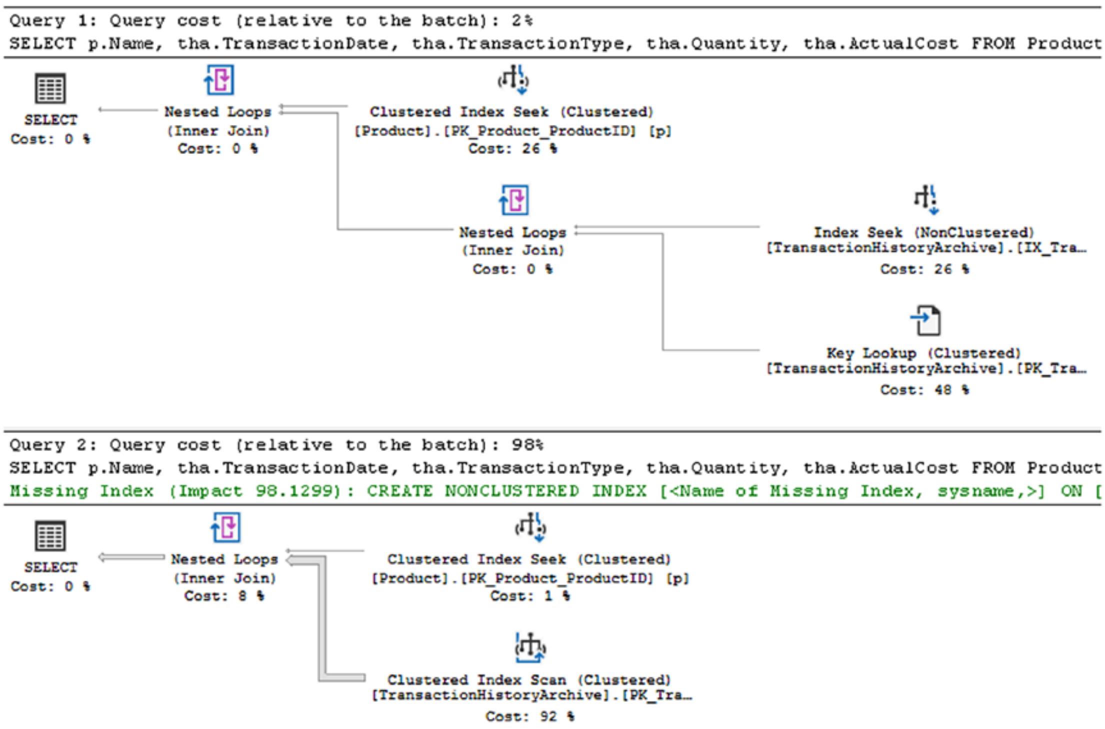
Like the original queries used earlier, these queries vary only by the values passed to the ProductID column. When both queries are run, you can select data from sys.dm_exec_query_ stats to see the hash values (Figure 16-24).



| | execution_count | query_hash | query_plan_hash | text | |
|---|---|---|---|---|---|
| 1 | 1 | 0xD4FA47AE35195F89 | 0xA366B147B0F12C2F | SELECT p.Name, | tha.TransactionDat... |
| 2 | 1 | 0xD4FA47AE35195F89 | 0x2567346B1381B053 | SELECT p.Name, | tha.TransactionDat... |

***Figure 16-24.*** *Differences in the query_plan_hash*

You can see the queryhash values are identical, but the query_plan_hash values are different. This is because the execution plans created, based on the statistics for the values passed in, are radically different, as you can see in Figure 16-25.



```
Query 1: Query cost (relative to the batch): 2%
SELECT p.Name, tha.TransactionDate, tha.TransactionType, tha.Quantity, tha.ActualCost FROM Product
```

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Clustered Index Seek (Clustered)
[Product].[PK_Product_ProductID] [p]
Cost: 26 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[TransactionHistoryArchive].[IX_Tra...
Cost: 26 %
Key Lookup (Clustered)
[TransactionHistoryArchive].[PK_Tra...
Cost: 48 %

```
Query 2: Query cost (relative to the batch): 98%
SELECT p.Name, tha.TransactionDate, tha.TransactionType, tha.Quantity, tha.ActualCost FROM Product
Missing Index (Impact 98.1299): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [
```

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 8 %
Clustered Index Seek (Clustered)
[Product].[PK_Product_ProductID] [p]
Cost: 1 %
Clustered Index Scan (Clustered)
[TransactionHistoryArchive].[PK_Tra...
Cost: 92 %

***Figure 16-25.*** *Different parameters result in radically different plans*

The query plan hash and the query hash values can be useful tools for tracking down common issues between disparate queries, but as you've seen, they're not going to retrieve an accurate set of information in every possibility. They do add yet another useful tool in identifying other places where query performance could be poor. They can also be used to track execution plans over time. You can capture the `query_plan_hash` for a query after deploying it to production and then watch it over time to see whether it changes because of data changes. With this you can also keep track of aggregated query stats by plan, referencing `sys.dm_exec_querystats`, although remember that the aggregated data is reset when the server is restarted or the plan cache is cleared in any way. However, that same information within the Query Store is persisted through backups, server restarts, clearing the plan cache, etc. Keep these tools in mind while tuning your queries.

## Execution Plan Cache Recommendations

The basic purpose of the plan cache is to improve performance by reusing execution plans. Thus, it is important to ensure that your execution plans actually are reusable. Since the plan reusability of ad hoc queries is inefficient, it is generally recommended that you rely on prepared workload techniques as much as possible. To ensure efficient use of the plan cache, follow these recommendations:

- Explicitly parameterize variable parts of a query.
- Use stored procedures to implement business functionality.
- Use `sp_executesql` to avoid stored procedure maintenance.
- Use the prepare/execute model to avoid resending a query string.
- Avoid ad hoc queries.
- Use `sp_executesql` over `EXECUTE` for dynamic queries.
- Parameterize variable parts of queries with care.
- Avoid modifying environment settings between connections.
- Avoid the implicit resolution of objects in queries.

Let's take a closer look at these points.

## Explicitly Parameterize Variable Parts of a Query

A query is often run several times, with the only difference between each run being that there are different values for the variable parts. Their plans can be reused, however, if the static and variable parts of the query can be separated. Although SQL Server has a simple parameterization feature and a forced parameterization feature, they have severe limitations. Always perform parameterization explicitly using the standard prepared workload techniques.

## Create Stored Procedures to Implement Business Functionality

If you have explicitly parameterized your query, then placing it in a stored procedure brings the best reusability possible. Since only the parameters need to be sent along with the stored procedure name, network traffic is reduced. Since stored procedures are reused from the cache, they can run faster than ad hoc queries.

Like anything else, it is possible to have too much of a good thing. There are business processes that belong in the database, but there are also business processes that should never be placed within the database. For example, formatting data within stored procedures is frequently better done with applications. Basically, your database and the queries around it should be focused on direct data retrieval and data storage. Any other processing should be done elsewhere.

## Code with sp_executesql to Avoid Stored Procedure Deployment

If the object deployment required for the stored procedures becomes a consideration or you are using queries generated on the client side, then use `sp_executesql` to submit the queries as prepared workloads. Unlike the stored procedure model, `sp_executesql` doesn't create any persistent objects in the database. `sp_executesql` is suited to execute a singleton query or a small batch query.

The complete business logic implemented in a stored procedure can also be submitted with `sp_executesql` as a large query string. However, as the complexity of the business logic increases, it becomes difficult to create and maintain a query string for the complete logic.

Also, using `sp_executesql` and stored procedures with appropriate parameters prevents SQL injection attacks on the server.

However, I still strongly recommend using stored procedures within your database where possible.

## Implement the Prepare/Execute Model to Avoid Resending a Query String

`sp_executesql` requires the query string to be sent across the network every time the query is reexecuted. It also requires the cost of a query string match at the server to identify the corresponding execution plan in the plan cache. In the case of an ODBC or OLEDB (or OLEDB .NET) application, you can use the prepare/execute model to avoid resending the query string during multiple executions, since only the plan handle and parameters need to be submitted. In the prepare/execute model, since a plan handle is returned to the application, the plan can be reused by other user connections; it is not limited to the user who created the plan.

## Avoid Ad Hoc Queries

Do not design new applications using ad hoc queries! The execution plan created for an ad hoc query cannot be reused when the query is resubmitted with a different value for the variable parts. Even though SQL Server has the simple parameterization and forced parameterization features to isolate the variable parts of the query, because of the strict conservativeness of SQL Server in parameterization, the feature is limited to simple queries only. For better plan reusability, submit the queries as prepared workloads.

There are systems built upon the concept of nothing but ad hoc queries. This is functional and can work within SQL Server, but, as you've seen, it carries with it large amounts of additional overhead that you'll need to plan for. Also, ad hoc queries are generally how SQL injection gets introduced to a system.

## Prefer sp_executesql Over EXECUTE for Dynamic Queries

SQL query strings generated dynamically within stored procedures or a database application should be executed using `spexecutesql` instead of the `EXECUTE` command. The `EXECUTE` command doesn't allow the variable parts of the query to be explicitly parameterized.

To understand the preceding comparison between sp_executesql and EXECUTE, consider the dynamic SQL query string used to execute the SELECT statement in adhocsproc.

```
DECLARE @n VARCHAR(3) = '776',
        @sql VARCHAR(MAX);

SET @sql
    = 'SELECT * FROM Sales.SalesOrderDetail sod  ' + 'JOIN Sales.
    SalesOrderHeader soh  '
      + 'ON sod.SalesOrderID=soh.SalesOrderID ' + 'WHERE    sod.
      ProductID="' + @n + "";

--Execute the dynamic query using EXECUTE statement
EXECUTE (@sql);
```

The EXECUTE statement submits the query along with the value of d.ProductID as an ad hoc query and thereby may or may not result in simple parameterization. Check the output yourself by looking at the cache.

```
SELECT deqs.execution_count,
       deqs.query_hash,
       deqs.query_plan_hash,
       dest.text,
       deqp.query_plan
FROM sys.dm_exec_query_stats AS deqs
    CROSS APPLY sys.dm_exec_sql_text(deqs.plan_handle) AS dest
    CROSS APPLY sys.dm_exec_query_plan(deqs.plan_handle) AS deqp
WHERE dest.text LIKE 'SELECT * FROM Sales.SalesOrderDetail sod%';
```

For improved plan cache reusability, execute the dynamic SQL string as a parameterized query using sp_executesql.

```
DECLARE @n NVARCHAR(3) = '776',
        @sql NVARCHAR(MAX),
        @paramdef NVARCHAR(6);

SET @sql
```

```
    = 'SELECT * FROM Sales.SalesOrderDetail sod  ' + 'JOIN Sales.Sales
    OrderHeader soh  '
      + 'ON sod.SalesOrderID=soh.SalesOrderID ' + 'WHERE    sod.ProductID=@1';
SET @paramdef = N'@1 INT';

--Execute the dynamic query using sp_executesql system stored procedure
EXECUTE sp_executesql @sql, @paramdef, @1 = @n;
```

Executing the query as an explicitly parameterized query using sp_executesql generates a parameterized plan for the query and thereby increases the execution plan reusability.

## Parameterize Variable Parts of Queries with Care

Be careful while converting variable parts of a query into parameters. The range of values for some variables may vary so drastically that the execution plan for a certain range of values may not be suitable for the other values. This can lead to bad parameter sniffing (covered in Chapter 17).

## Do Not Allow Implicit Resolution of Objects in Queries

SQL Server allows multiple database objects with the same name to be created under different schemas. For example, table t1 can be created using two different schemas (u1 and u2) under their individual ownership. The default owner in most systems is dbo (database owner). If user u1 executes the following query, then SQL Server first tries to find whether table t1 exists for user u1's default schema.

```
SELECT *
FROM tl
WHERE cl = 1;
```

If not, then it tries to find whether table t1 exists for the dbo user. This implicit resolution allows user u1 to create another instance of table t1 under a different schema and access it temporarily (using the same application code) without affecting other users.

On a production database, I recommend using the schema owner and avoiding implicit resolution. If not, using implicit resolution adds the following overhead on a production server:

- It requires more time to identify the objects.
- It decreases the effectiveness of plan cache reusability.

## Summary

SQL Server's cost-based query optimizer decides upon an effective execution plan not only based on the exact syntax of the query but on the cost of executing the query using different processing strategies. The cost evaluation of using different processing strategies is done in multiple optimization phases to avoid spending too much time optimizing a query. Then, the execution plans are cached to save the cost of execution plan generation when the same queries are reexecuted. To improve the reusability of cached plans, SQL Server supports different techniques for execution plan reuse when the queries are rerun with different values for the variable parts.

Using stored procedures is usually the best technique to improve execution plan reusability. SQL Server generates a parameterized execution plan for the stored procedures so that the existing plan can be reused when the stored procedure is rerun with the same or different parameter values. However, if the existing execution plan for a stored procedure is invalidated, the plan can't be reused without a recompilation, decreasing the effectiveness of plan cache reusability.

In the next chapter, I will discuss how to troubleshoot and resolve bad parameter sniffing.

CHAPTER 17

# Parameter Sniffing

In the previous chapter, I discussed how to get execution plans into the cache and how to get them reused from there. It's a laudable goal and one of the many ways to improve the overall performance of the system. One of the best mechanisms for ensuring plan reuse is to parameterize the query, through either stored procedures, prepared statements, or `sp_executesql`. All these mechanisms create a parameter that is used instead of a hard-coded value when creating the plan. These parameters can be sampled, or sniffed, by the optimizer to use the values contained within when creating the execution plan. When this works well, as it does most of the time, you benefit from more accurate plans. But when it goes wrong and becomes bad parameter sniffing, you can see serious performance issues.

In this chapter, I cover the following topics:

- The helpful mechanisms behind parameter sniffing
- How parameter sniffing can turn bad
- Mechanisms for dealing with bad parameter sniffing

## Parameter Sniffing

When a parameterized query is sent to the optimizer and there is no existing plan in cache, the optimizer will perform its function to create an execution plan for manipulating the data as requested by the T-SQL statement. When this parameterized query is called, the values of the parameters are set, either through your program or through defaults in the parameter definitions. Either way, there is a value there. The optimizer knows this. So, it takes advantage of that fact and reads the value of the parameters. This is the "sniffing" aspect of the process known as *parameter sniffing*. With these values available, the optimizer will then use those specific values to look at the statistics of the data to which the parameters refer. With specific values and a set of

© Grant Fritchey 2018
G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_17

accurate statistics, you'll get a better execution plan. This beneficial process of parameter sniffing is running all the time automatically, assuming no changes to the defaults, for all your parameterized queries, regardless of where they come from.

You can also get sniffing of local variables. Before proceeding with that, though, let's delineate between a local variable and a parameter since, within a T-SQL statement, they can look the same. This example shows both a local variable and a parameter:

```
CREATE PROCEDURE dbo.ProductDetails (@ProductID INT)
AS
DECLARE @CurrentDate DATETIME = GETDATE();

SELECT p.Name,
       p.Color,
       p.DaysToManufacture,
       pm.CatalogDescription
FROM Production.Product AS p
    JOIN Production.ProductModel AS pm
        ON pm.ProductModelID = p.ProductModelID
WHERE p.ProductID = @ProductID
      AND pm.ModifiedDate < @CurrentDate;
GO
```

The parameter in the previous query is `@ProductID`. The local variable is `@CurrentDate`. The parameter is defined with the stored procedure (or the prepared statement in that case). The local variable is part of the code. It's important to differentiate these since when you get down to the `WHERE` clause, they look exactly the same.

If you get a recompile of any statement that is using local variables, those variables can be sniffed by the optimizer the same way it sniffs parameters. Just be aware of this. Other than this unique situation with the recompile, local variables are unknown quantities to the optimizer when it goes to compile a plan. Normally only parameters can be sniffed.

To see parameter sniffing in action and to show that it's useful, let's start with a different procedure.

```
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;
GO
```

After creating the procedure, run it with this parameter:

```
EXEC dbo.AddressByCity @City = N'London';
```

This will result in the following I/O and execution times as well as the query plan in Figure 17-1:

```
Reads: 219
Duration: 97.1ms
```

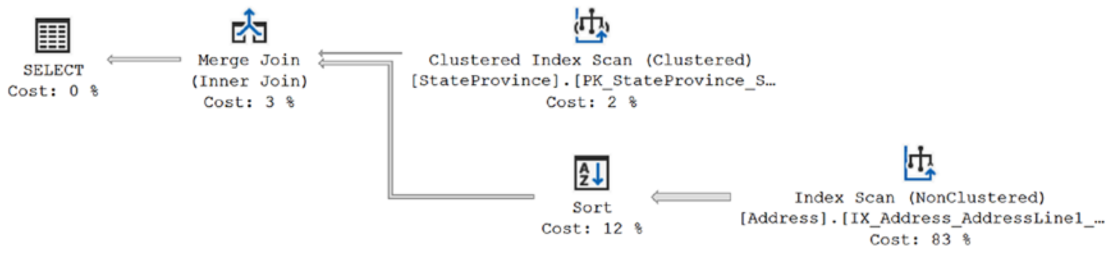

SELECT
Cost: 0 %
Merge Join
(Inner Join)
Cost: 3 %
Clustered Index Scan (Clustered)
[StateProvince].[PK_StateProvince_S…
Cost: 2 %
Sort
Cost: 12 %
Index Scan (NonClustered)
[Address].[IX_Address_AddressLine1_…
Cost: 83 %

***Figure 17-1.** Execution plan of AddressByCity*

The optimizer sniffed the value London and arrived at a plan based on the data distribution that the city of London represented within the statistics on the Address table. There may be other tuning opportunities in that query or with the indexes on the table, but the plan is optimal for the value London and the existing data structure. You can write an identical query using a local variable just like this:

```
DECLARE @City NVARCHAR(30) = N'London';

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;
```

When this query gets executed, the results of the I/O and execution times are different.

```
Reads: 1084
Duration: 127.5ms
```

The execution time has gone up, and you've moved from 219 reads total to 1084. This somewhat explained by taking a look at the new execution plan shown in Figure 17-2.

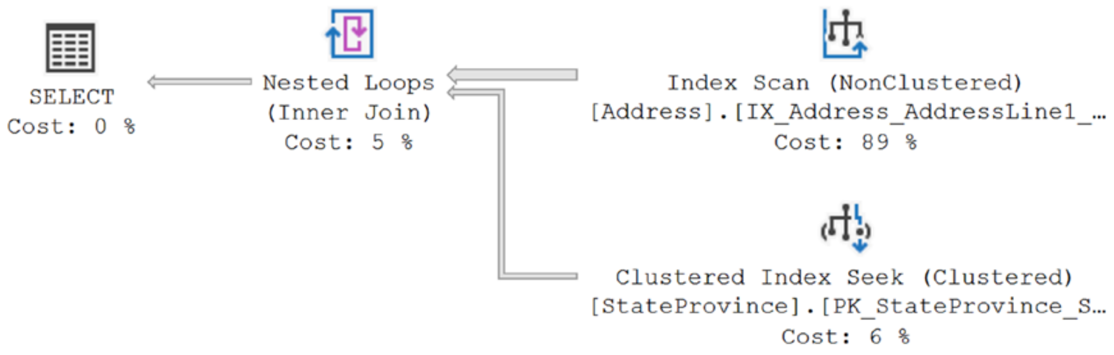

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 5 %
Index Scan (NonClustered)
[Address].[IX_Address_AddressLine1_...
Cost: 89 %
Clustered Index Seek (Clustered)
[StateProvince].[PK_StateProvince_S...
Cost: 6 %

***Figure 17-2.*** *An execution plan created using a local variable*

What has happened is that the optimizer was unable to sample, or sniff, the value for the local variable and therefore had to use an average number of rows from the statistics. You can see this by looking at the estimated number of rows in the properties of the Index Scan operator. It shows 34.113. Yet, if you look at the data returned, there are actually 434 rows for the value London. In short, if the optimizer thinks it needs to retrieve 434 rows, it creates a plan using the merge join and only 219 reads. But, if it thinks it's returning only about 34 rows, it uses the plan with a nested loop join, which, by the nature of the nested loop that seeks in the lower value once for each value in the upper set of data, results in 1,084 reads and slower performance.

That is parameter sniffing in action resulting in improved performance. Now, let's see what happens when parameter sniffing goes bad.

## Bad Parameter Sniffing

Parameter sniffing creates problems when you have issues with your statistics. The values passed in the parameter may be representative of your data and the data distribution within the statistics. In this case, you'll see a good execution plan. But what happens when the parameter passed is not representative of the rest of the data in the table? This situation can arise because your data is just distributed in a nonaverage way. For example, most values in the statistics will return only a few rows, say six, but some values will return hundreds of rows. The same thing works the other way, with a common distribution of large amounts of data and an uncommon set of small values. In this case, an execution plan is created, based on the nonrepresentative data, but it's not useful to most of the queries. This situation most frequently exposes itself through a sudden, and sometimes quite severe, drop in performance. It can even, seemingly randomly, fix itself when a recompile event allows a better representative data value to be passed in a parameter.

You can also see this occur when the statistics are out-of-date, are inaccurate because of being sampled instead of scanned (for more details on statistics in general, see Chapter 13), or even are perfectly formed and are just very jagged (odd distributions of data). Regardless, the situation creates a plan that is less than useful and stores it in cache. For example, take the following stored procedure:

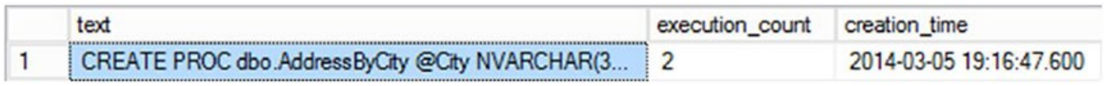
```
CREATE OR ALTER PROC dbo.AddressByCity @City NVARCHAR(30)
AS
SELECT a.AddressID,
       a.AddressLine1,
       AddressLine2,
       a.City,
       sp.Name AS StateProvinceName,
       a.PostalCode
FROM Person.Address AS a
    JOIN Person.StateProvince AS sp
        ON a.StateProvinceID = sp.StateProvinceID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

If the stored procedure created previously, dbo.AddressByCity, is run again but this time with a different parameter, then it returns with a different set of I/O and execution times but the same execution plan because it is reused from the cache.

```
EXEC dbo.AddressByCity @City = N'Mentor';
Reads: 218
Duration: 2.8ms
```

The I/O is the nearly the same since the same execution plan is reused. The execution time is faster because fewer rows are being returned. You can verify that the plan was reused by taking a look at the output from sys.dm_exec_query_stats (in Figure 17-3).

```
SELECT  dest.text,
        deqs.execution_count,
        deqs.creation_time
FROM    sys.dm_exec_query_stats AS deqs
CROSS APPLY sys.dm_exec_sql_text(deqs.sql_handle) AS dest
WHERE   dest.text LIKE 'CREATE PROC dbo.AddressByCity%';
```



| | text | execution_count | creation_time |
|---|---|---|---|
| 1 | CREATE PROC dbo.AddressByCity @City NVARCHAR(3... | 2 | 2014-03-05 19:16:47.600 |

***Figure 17-3.** The output from sys.dm_exec_query_stats verifies procedure reuse*

To show how bad parameter sniffing can occur, you can reverse the order of the execution of the procedures. First flush the buffer cache by running DBCC FREEPROCCACHE, which should not be run against a production machine, unless you're careful to do what I show here, which will remove only a single execution plan from the cache:

```
DECLARE @PlanHandle VARBINARY(64);

SELECT @PlanHandle = deps.plan_handle
FROM sys.dm_exec_procedure_stats AS deps
WHERE deps.object_id = OBJECT_ID('dbo.AddressByCity');

IF @PlanHandle IS NOT NULL
BEGIN
    DBCC FREEPROCCACHE(@PlanHandle);
END
GO
```

Another option here is to only flush the plans for a given database through `ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE CACHE;`.

Now, rerun the queries in reverse order. The first query, using the parameter value `Mentor`, results in the following I/O and execution plan (Figure 17-4):

```
Reads: 218
Duration: 1.8ms
```

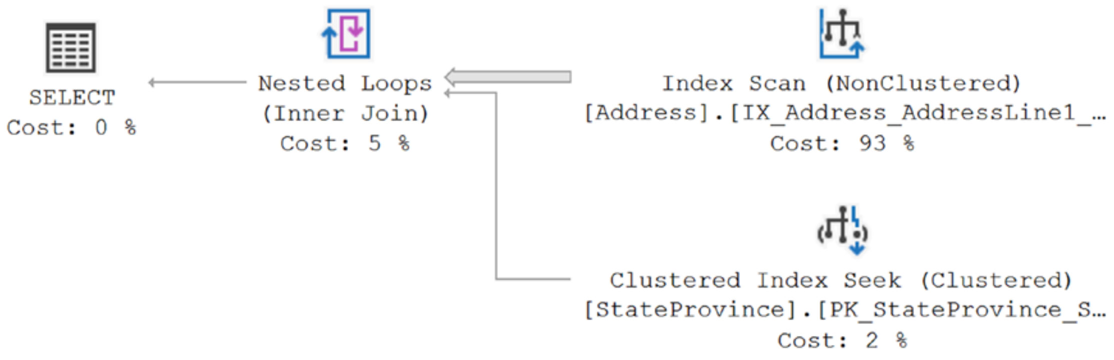

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 5 %
Index Scan (NonClustered)
[Address].[IX_Address_AddressLine1_...
Cost: 93 %
Clustered Index Seek (Clustered)
[StateProvince].[PK_StateProvince_S...
Cost: 2 %

***Figure 17-4.** The execution plan changes*

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.

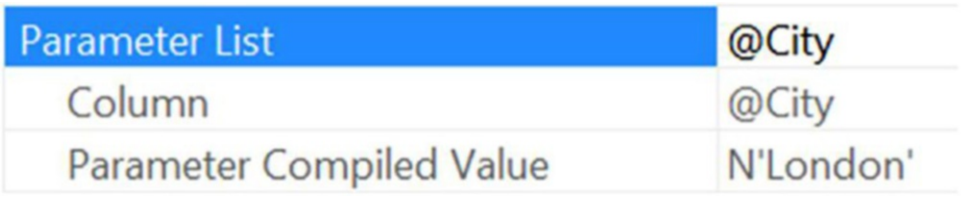
If you have enabled the Query Store, another approach would be to retrieve the plans from there:

```
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.



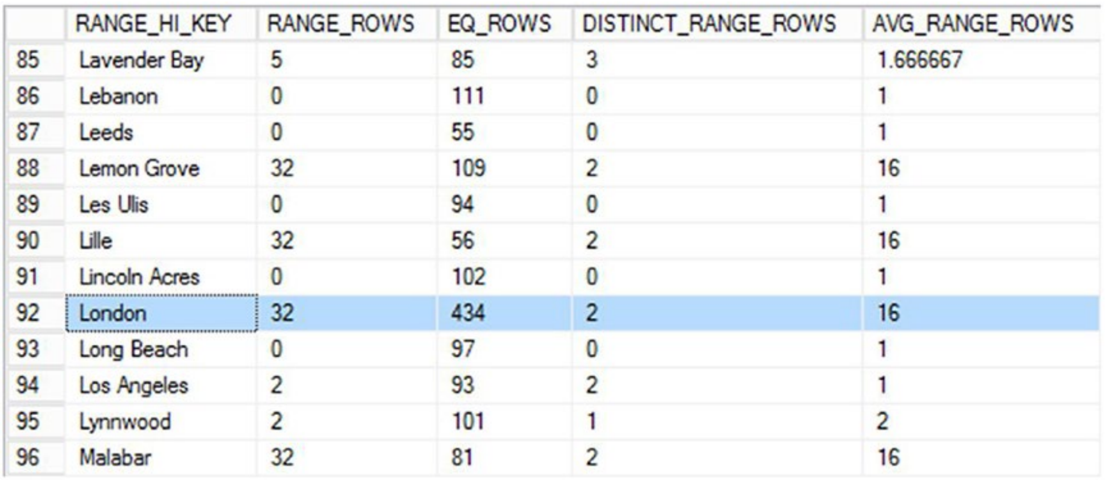
| | |
|---|---|
| Parameter List | @City |
| Column | @City |
| Parameter Compiled Value | N'London' |

***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:

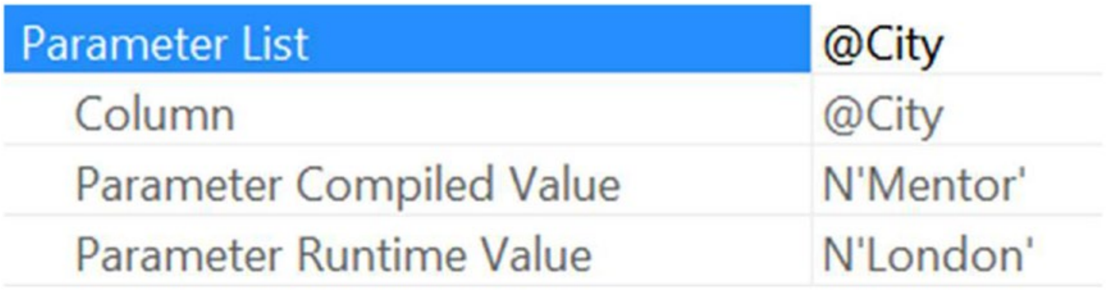
```
-- This allows the query to bypass bad parameter sniffing
```

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.

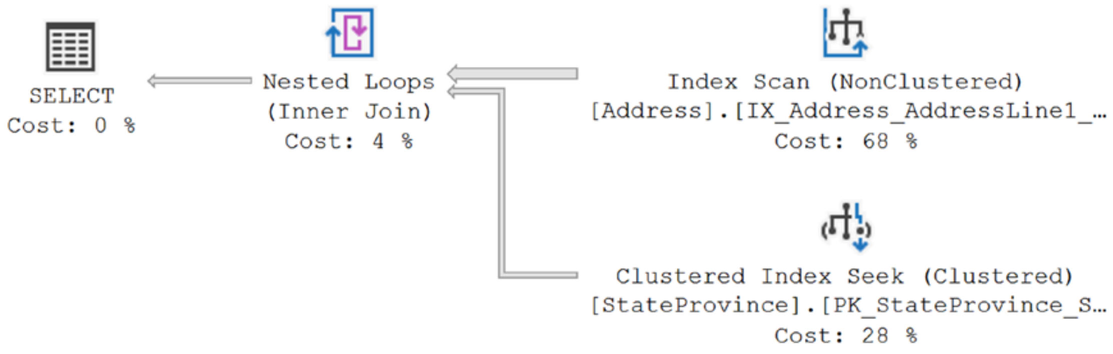
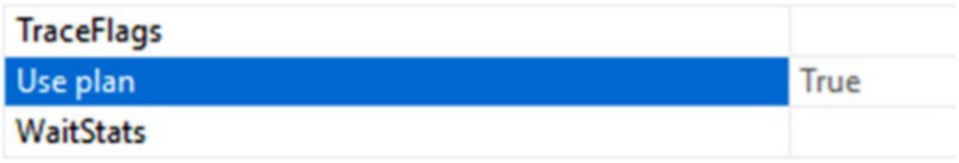

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 4 %
Index Scan (NonClustered)
[Address].[IX_Address_AddressLine1_…
Cost: 68 %
Clustered Index Seek (Clustered)
[StateProvince].[PK_StateProvince_S…
Cost: 28 %

***Figure 17-8.*** *A forced execution plan*

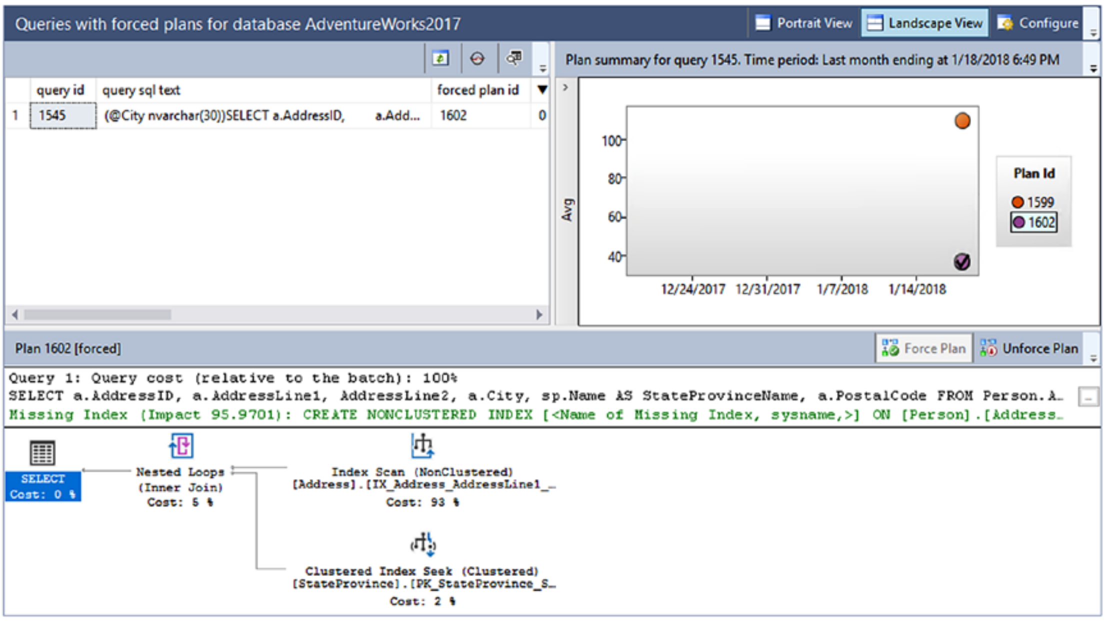
You can try removing the plan from cache and rerunning it for the value of London. However, nothing you do at this point will bring back that execution plan because the optimizer is now forcing the plan. You can monitor plan forcing using Extended Events. You can also query the Query Store views to see 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.



| | |
|---|---|
| TraceFlags | |
| Use plan | True |
| WaitStats | |

*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.



Queries with forced plans for database AdventureWorks2017

| | query id | query sql text | forced plan id | |
|---|---|---|---|---|
| 1 | 1545 | (@City nvarchar(30))SELECT a.AddressID, a.Add... | 1602 | 0 |

Plan summary for query 1545. Time period: Last month ending at 1/18/2018 6:49 PM

Plan 1602 [forced]

```
Query 1: Query cost (relative to the batch): 100%
SELECT a.AddressID, a.AddressLine1, AddressLine2, a.City, sp.Name AS StateProvinceName, a.PostalCode FROM Person.A…
Missing Index (Impact 95.9701): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Person].[Address…
```

*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.

© Grant Fritchey 2018
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_WorkOrderlD`, as shown in Figure 18-1.

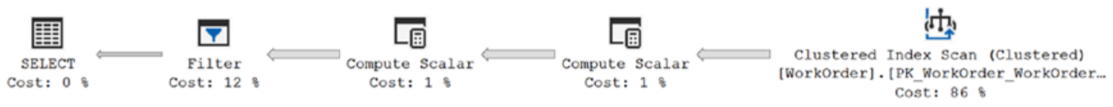


***Figure 18-1.*** *Execution plan for the stored procedure*

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.

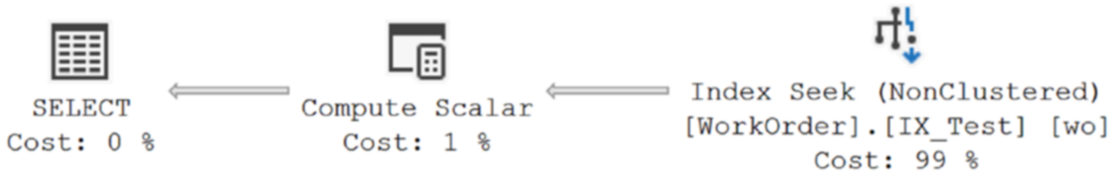


***Figure 18-2.*** *New execution plan for the stored procedure*

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;
```

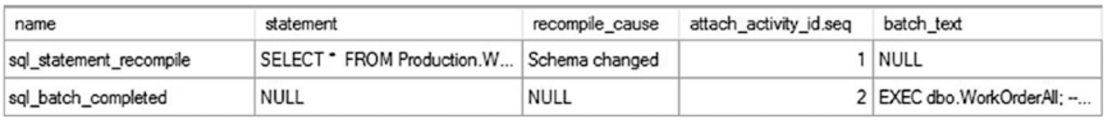
Before executing this procedure, drop the index IXTest.

```
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`.

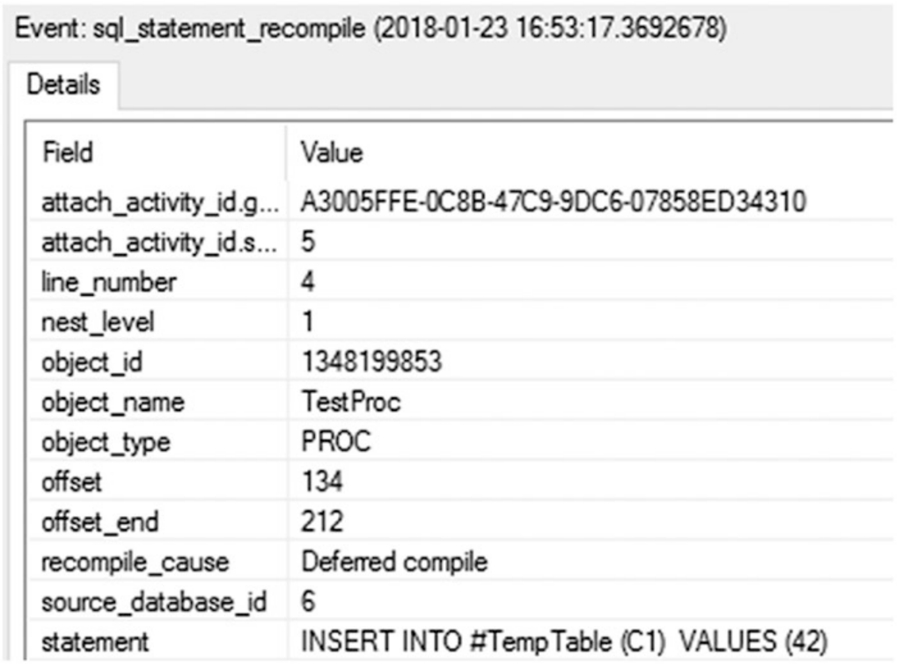
---

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.<br>If n > 500, then RT = .2 * n or Sqrt(1000*NumberOfRows). |
| Temporary table | If n < 6, then RT = 6.<br>If 6 <= n <= 500, then RT = 500.<br>If n > 500, then RT = .2 * n<br>or<br>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' ))
```

```
    DROP TABLE dbo.NewOrderDetail;
GO
SELECT *
INTO dbo.NewOrderDetail
FROM Sales.SalesOrderDetail;
GO
CREATE INDEX IX_NewOrders_ProductID ON dbo.NewOrderDetail (ProductID);
GO
CREATE OR ALTER PROCEDURE dbo.NewOrders
AS
SELECT nod.OrderQty,
       nod.CarrierTrackingNumber
FROM dbo.NewOrderDetail AS nod
WHERE nod.ProductID = 897;
GO
SET STATISTICS XML ON;
EXEC dbo.NewOrders;
SET STATISTICS XML OFF;
GO
```

Next you need to modify a number of rows before reexecuting the stored procedure.

```
UPDATE dbo.NewOrderDetail
SET ProductID = 897
WHERE ProductID BETWEEN 800
                AND     900;
GO
SET STATISTICS XML ON;
EXEC dbo.NewOrders;
SET STATISTICS XML OFF;
GO
```

The first time, SQL Server executes the SELECT statement of the stored procedure using an Index Seek operation, as shown in Figure 18-5.