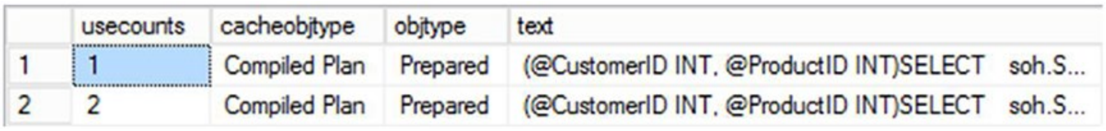
The query for which the plan is created (the text column) matches the exact textual string of the parameterized query submitted through sp_executesql. Therefore, if the same query is submitted from different parts of the application, ensure that the same textual string is used in all places. For example, 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' ;
```



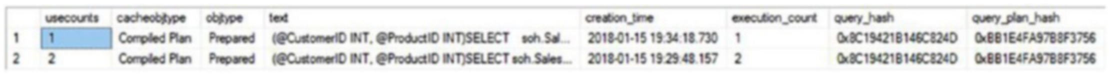
	usecounts	cacheobjtype	objtype	text
1	1	Compiled Plan	Prepared	(@CustomerID INT, @ProductID INT)SELECT soh.S...
2	2	Compiled Plan	Prepared	(@CustomerID INT, @ProductID INT)SELECT soh.S...

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.

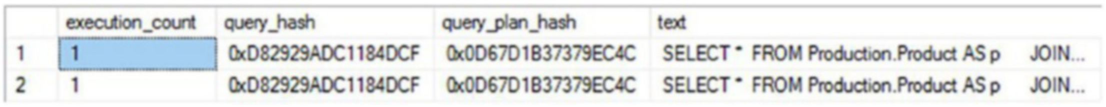
To see the hash values in action, create two queries.

```
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%';
```

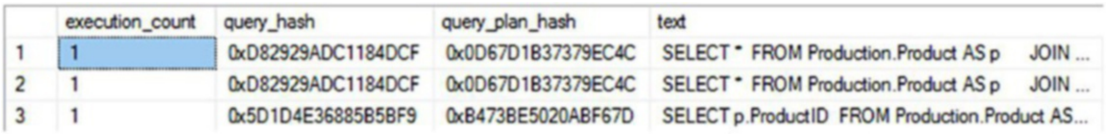


	execution_count	query_hash	query_plan_hash	text
1	1	0xD82929ADC1184DCF	0x0D67D1B37379EC4C	SELECT * FROM Production.Product AS p JOIN...
2	1	0xD82929ADC1184DCF	0x0D67D1B37379EC4C	SELECT * FROM Production.Product AS p JOIN...

***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*

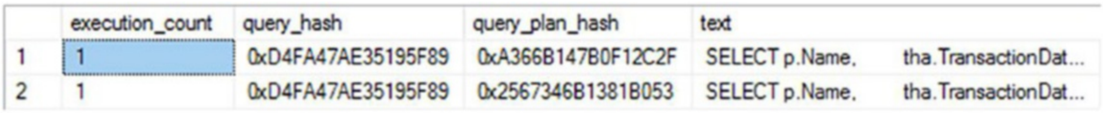
Although the basic structure of the query is the same, the change in the columns returned was enough to change the query hash value and the query plan hash value.

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;
```

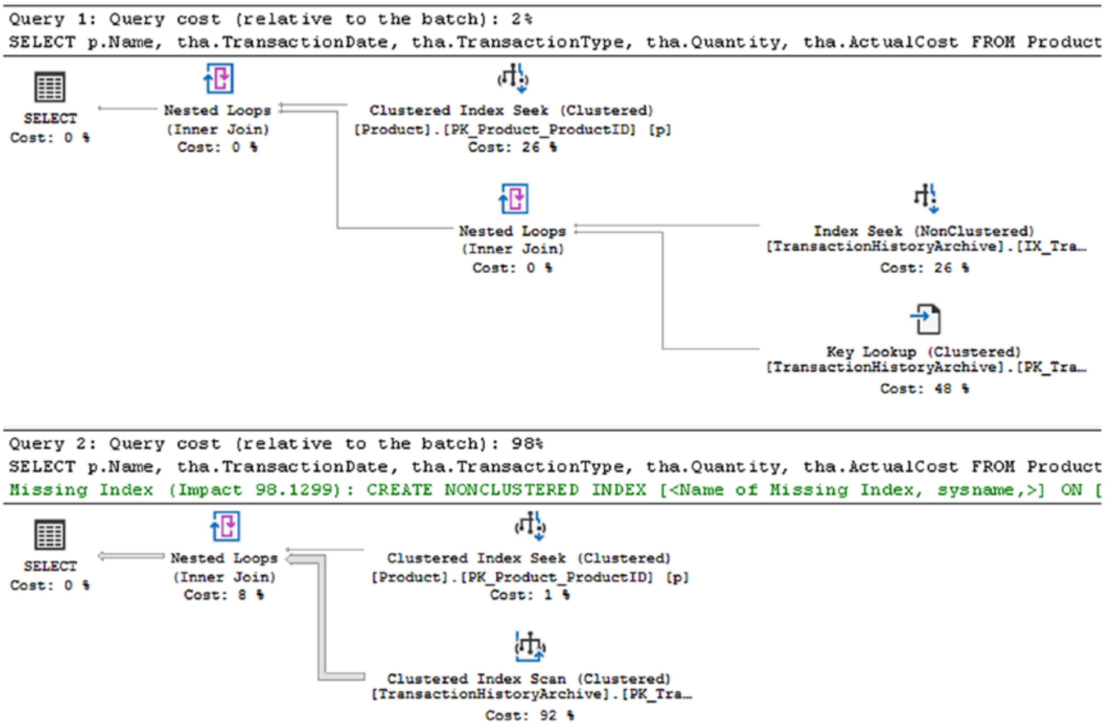
Like the original queries used earlier, these queries vary only by the values passed to the `ProductID` column. When both queries are run, you can select data from `sys.dm_exec_query_ stats` to see the hash values (Figure 16-24).



	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
```

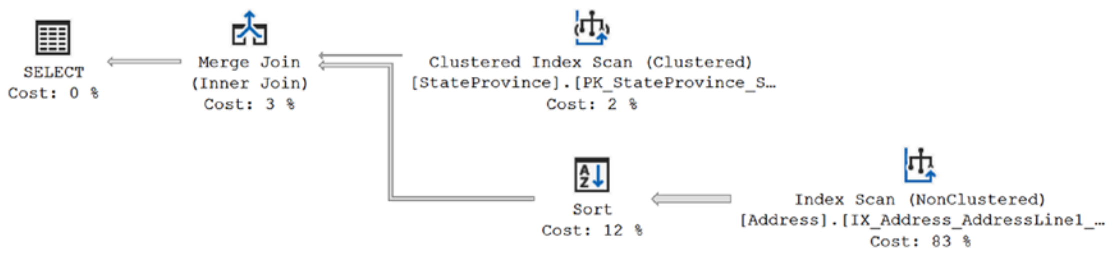

SELECT
Cost: 0 %
Merge Join
(Inner Join)
Cost: 3 %
Clustered Index Scan (Clustered)
[StateProvince].[PK_StateProvince_S…
Cost: 2 %
Sort
Cost: 12 %
Index Scan (NonClustered)
[Address].[IX_Address_AddressLine1_…
Cost: 83 %

***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.

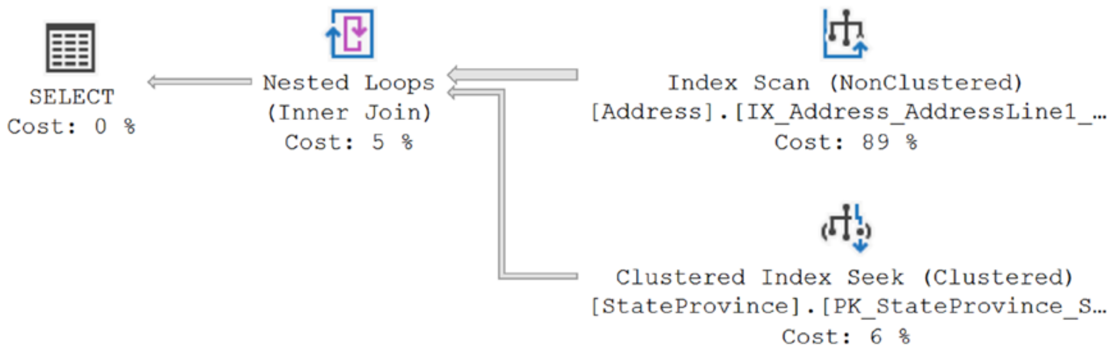

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 5 %
Index Scan (NonClustered)
[Address].[IX_Address_AddressLine1_...
Cost: 89 %
Clustered Index Seek (Clustered)
[StateProvince].[PK_StateProvince_S...
Cost: 6 %

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:

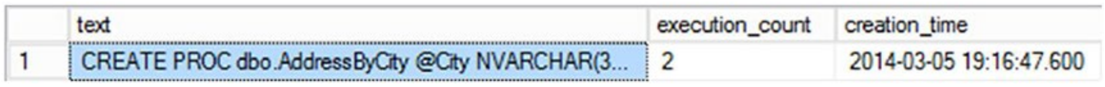
```
CREATE OR ALTER PROC dbo.AddressByCity @City NVARCHAR(30)
AS
SELECT a.AddressID,
       a.AddressLine1,
       AddressLine2,
       a.City,
       sp.Name AS StateProvinceName,
       a.PostalCode
FROM Person.Address AS a
    JOIN Person.StateProvince AS sp
        ON a.StateProvinceID = sp.StateProvinceID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

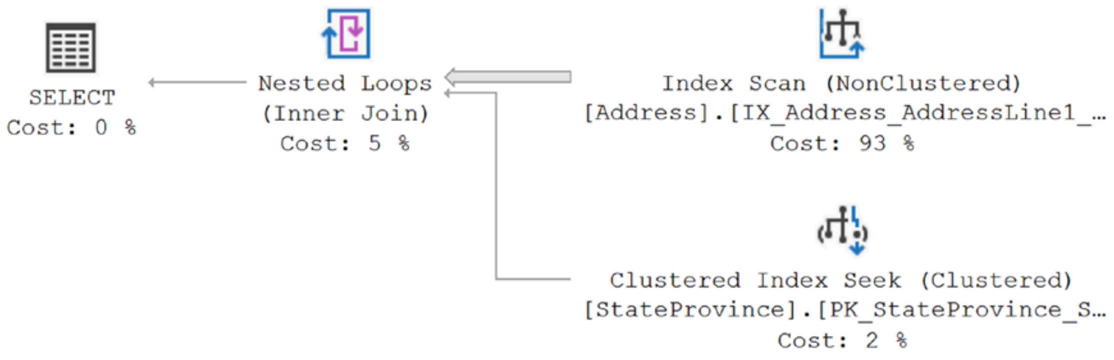

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 5 %
Index Scan (NonClustered)
[Address].[IX_Address_AddressLine1_...
Cost: 93 %
Clustered Index Seek (Clustered)
[StateProvince].[PK_StateProvince_S...
Cost: 2 %

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