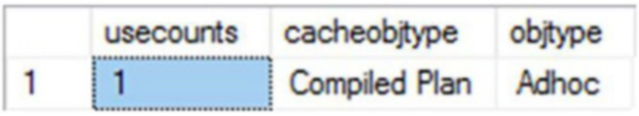
The execution plan generated for this ad hoc query is based on the exact text of the query, which includes comments, case, trailing spaces, and hard returns. You'll have to use the exact text to pull the information out of `sys.dm_exec_cached_plans`.

```
SELECT    c.usecounts
    ,c.cacheobjtype
    ,c.objtype
FROM      sys.dm_exec_cached_plans c
    CROSS APPLY sys.dm_exec_sql_text(c.plan_handle) t
WHERE     t.text =  '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 = 29690
        AND sod.ProductID = 711;';
```

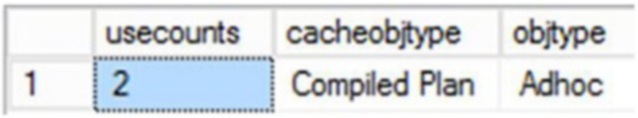
Figure 16-1 shows the output of `sys.dm_exec_cached_plans`.



	usecounts	cacheobjtype	objtype
1	1	Compiled Plan	Adhoc

Figure 16-1. *sys.dm_exec_cached_plans output*

You can see from Figure 16-1 that a compiled plan is generated and saved in the plan cache for the preceding ad hoc query. To find the specific query, I used the query itself in the `WHERE` clause. You can see that this plan has been used once up until now (`usecounts = 1`). If this ad hoc query is reexecuted, SQL Server reuses the existing executable plan from the plan cache, as shown in Figure 16-2.



	usecounts	cacheobjtype	objtype
1	2	Compiled Plan	Adhoc

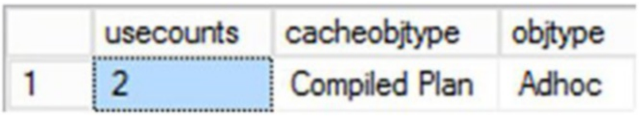
Figure 16-2. *Reusing the executable plan from the plan cache*

In Figure 16-2, you can see that the usecounts value for the preceding query's executable plan has increased to 2, confirming that the existing plan for this query has been reused. If this query is executed repeatedly, the existing plan will be reused every time.

Since the plan generated for the preceding query includes the filter criterion value, the reusability of the plan is limited to the use of the same filter criterion value. Reexecute the query, but change son.CustomerlD to 29500.

```
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 = 29500
      AND sod.ProductID = 711;
```

The existing plan can't be reused, and if sys.dm_exec_cached_plans is rerun as is, you'll see that the execution count hasn't increased (Figure 16-3).



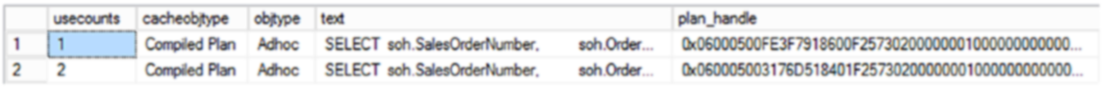
	usecounts	cacheobjtype	objtype
1	2	Compiled Plan	Adhoc

Figure 16-3. sys.dm_exec_cached_plans shows that the existing plan is not reused

Instead, I'll adjust the query against sys.dm_exec_cached_plans.

```
SELECT  c.usecounts,
        c.cacheobjtype,
        c.objtype,
        t.text,
        c.plan_handle
FROM    sys.dm_exec_cached_plans c
CROSS APPLY sys.dm_exec_sql_text(c.plan_handle) t
WHERE   t.text LIKE 'SELECT  soh.SalesOrderNumber,
        soh.OrderDate,
        sod.OrderQty,
        sod.LineTotal
FROM    Sales.SalesOrderHeader AS soh
JOIN    Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID%';
```

You can see the output from this query in Figure 16-4.



	usecounts	cacheobjtype	objtype	text		plan_handle
1	1	Compiled Plan	Adhoc	SELECT soh.SalesOrderNumber,	soh.Order...	0x06000500FE3F7918600F257302000000010000000000...
2	2	Compiled Plan	Adhoc	SELECT soh.SalesOrderNumber,	soh.Order...	0x060005003176D518401F257302000000010000000000...

Figure 16-4. *sys.dm_exec_cached_plans showing that the existing plan can't be reused*

From the `sys.dm_exec_cached_plans` output in Figure 16-4, you can see that the previous plan for the query hasn't been reused; the corresponding `usecounts` value remained at the old value of 2. Instead of reusing the existing plan, a new plan is generated for the query and is saved in the plan cache with a new `plan_handle`. If this ad hoc query is reexecuted repeatedly with different filter criterion values, a new execution plan will be generated every time. The inefficient reuse of the execution plan for this ad hoc query increases the load on the CPU by consuming additional CPU cycles to regenerate the plan.

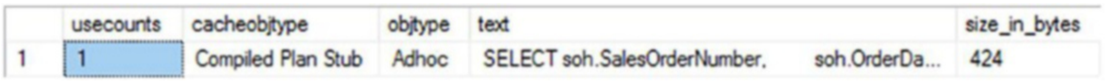
To summarize, ad hoc plan caching uses statement-level caching and is limited to an exact textual match. If an ad hoc query is not complex, SQL Server can implicitly parameterize the query to increase plan reusability by using a feature called *simple parameterization.* The definition of a query for simple parameterization is limited to quite basic cases such as ad hoc queries with only one table. As shown in the previous example, most queries requiring a join operation cannot be autoparameterized.

Optimize for an Ad Hoc Workload

If your server is going to primarily support ad hoc queries, it is possible to achieve a small degree of performance improvement. One server option is called `optimize for ad hoc workloads`. Enabling this for the server changes the way the engine deals with ad hoc queries. Instead of saving a full compiled plan for the query the first time it's called, a compiled plan stub is stored. The stub does not have a full execution plan associated, saving the storage space required for it and the time saving it to the cache. This option can be enabled without rebooting the server.

```
EXEC sp_configure 'show advanced option', '1';
GO
RECONFIGURE
GO
EXEC sp_configure 'optimize for ad hoc workloads', 1;
GO
RECONFIGURE;
```

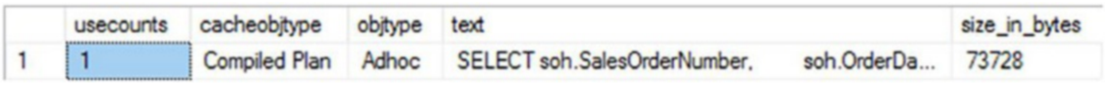
After changing the option, flush the cache and then rerun the ad hoc query. Modify the query against `sys.dm_exec_cached_plans` so that you include the `size_in_bytes` column; then run it to see the results in Figure 16-5.



	usecounts	cacheobjtype	objtype	text	size_in_bytes
1	1	Compiled Plan Stub	Adhoc	SELECT soh.SalesOrderNumber, soh.OrderDa...	424

***Figure 16-5.** sys.dm_exec_cached_plans showing a compiled plan stub*

Figure 16-5 shows in the `cacheobjtype` column that the new object in the cache is a compiled plan stub. Stubs can be created for lots more queries with less impact on the server than full compiled plans. But the next time an ad hoc query is executed, a fully compiled plan is created. To see this in action, run the query one more time and check the results in `sys.dm_exec_cachedplans`, as shown in Figure 16-6.



	usecounts	cacheobjtype	objtype	text	size_in_bytes
1	1	Compiled Plan	Adhoc	SELECT soh.SalesOrderNumber, soh.OrderDa...	73728

***Figure 16-6.** The compiled plan stub has become a compiled plan*

Check the `cacheobjtype` value. It has changed from Compiled Plan Stub to Compiled Plan. Finally, to see the real difference between a stub and a full plan, check the `sizeinbytes` column in Figure 16-5 and Figure 16-6. The size changed from 424 in the stub to 73728 in the full plan. This shows precisely the savings available when working with lots of ad hoc queries. Before proceeding, be sure to disable `optimize for ad hoc workloads`.

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

Personally, I see little downside to implementing this on just about any system. Like with all recommendations, you should test it to ensure your system isn't exceptional. However, the cost of writing the plan into memory when it's called a second time is extremely trivial to the savings in memory overall that you see by not storing plans that are only ever going to be used once. In all my testing and experience, this is a pure benefit with little downside. You can now use a database-scoped configuration setting to enable this in your Azure SQL Database too:

```
ALTER DATABASE SCOPED CONFIGURATION SET OPTIMIZE_FOR_AD_HOC_WORKLOADS = ON;
```

Simple Parameterization

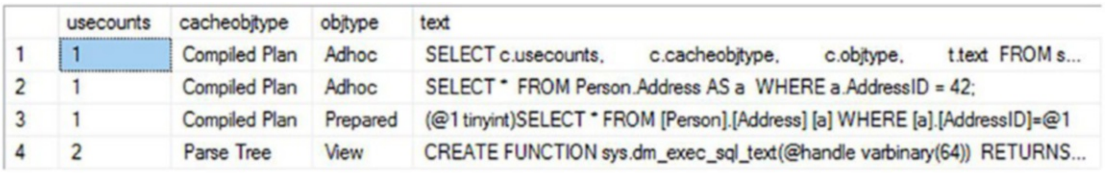
When an ad hoc query is submitted, SQL Server analyzes the query to determine which parts of the incoming text might be parameters. It looks at the variable parts of the ad hoc query to determine whether it will be safe to parameterize them automatically and use the parameters (instead of the variable parts) in the query so that the query plan can be independent of the variable values. This feature of automatically converting the variable part of a query into a parameter, even though not parameterized explicitly (using a prepared workload technique), is called *simple parameterization.*

During simple parameterization, SQL Server ensures that if the ad hoc query is converted to a parameterized template, the changes in the parameter values won't widely change the plan requirement. On determining the simple parameterization to be safe, SQL Server creates a parameterized template for the ad hoc query and saves the parameterized plan in the plan cache.

To understand the simple parameterization feature of SQL Server, consider the following query:

```
SELECT *
FROM Person.Address AS a
WHERE a.AddressID = 42;
```

When this ad hoc query is submitted, SQL Server can treat this query as it is for plan creation. However, before the query is executed, SQL Server tries to determine whether it can be safely parameterized. On determining that the variable part of the query can be parameterized without affecting the basic structure of the query, SQL Server parameterizes the query and generates a plan for the parameterized query. You can observe this from the `sys.dm_exec_cached_plans` output shown in Figure 16-7.



	usecounts	cacheobjtype	objtype	text
1	1	Compiled Plan	Adhoc	SELECT c.usecounts, c.cacheobjtype, c.objtype, t.text FROM s...
2	1	Compiled Plan	Adhoc	SELECT * FROM Person.Address AS a WHERE a.AddressID = 42;
3	1	Compiled Plan	Prepared	(@1 tinyint)SELECT * FROM [Person].[Address] [a] WHERE [a].[AddressID]=@1
4	2	Parse Tree	View	CREATE FUNCTION sys.dm_exec_sql_text(@handle varbinary(64)) RETURNS...

***Figure 16-7.** sys.dm_exec_cached_plans output showing an autoparameterized plan*

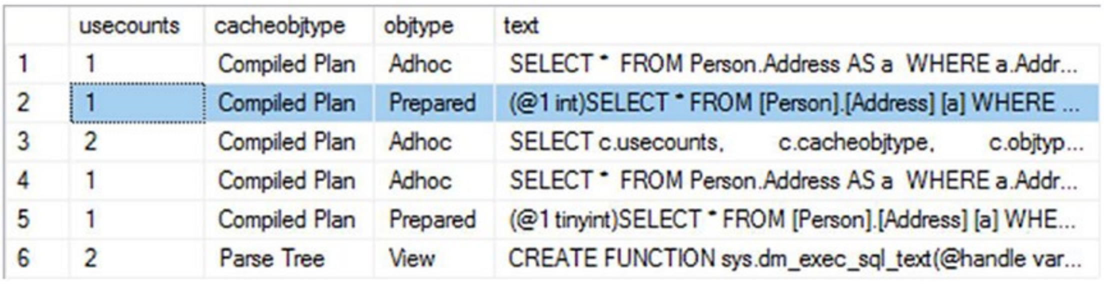
The `usecounts` of the executable plan for the parameterized query appropriately represents the number of reuses as 1. Also, note that the `objtype` for the autoparameterized executable plan is no longer `Adhoc`; it reflects the fact that the plan is for a parameterized query, Prepared.

The original ad hoc query, even though not executed, gets compiled to create the query tree required for the simple parameterization of the query. The compiled plan for the ad hoc query will be saved in the plan cache. But before creating the executable plan for the ad hoc query, SQL Server figured out that it was safe to autoparameterize and thus autoparameterized the query for further processing.

The parameter values are based on the value of the ad hoc query. Let's edit the previous query to use a different `AddressID` value.

```
SELECT *
FROM Person.Address AS a
WHERE a.AddressID = 42000;
```

If we requery `sys.dm_exec_cached_plans`, we'll see an additional plan has been added, as shown in Figure 16-8.



	usecounts	cacheobjtype	objtype	text
1	1	Compiled Plan	Adhoc	SELECT * FROM Person.Address AS a WHERE a.Addr...
2	1	Compiled Plan	Prepared	(@1 int)SELECT * FROM [Person].[Address] [a] WHERE ...
3	2	Compiled Plan	Adhoc	SELECT c.usecounts, c.cacheobjtype, c.objtyp...
4	1	Compiled Plan	Adhoc	SELECT * FROM Person.Address AS a WHERE a.Addr...
5	1	Compiled Plan	Prepared	(@1 tinyint)SELECT * FROM [Person].[Address] [a] WHE...
6	2	Parse Tree	View	CREATE FUNCTION sys.dm_exec_sql_text(@handle var...

Figure 16-8. *An additional plan with simple parameterization*

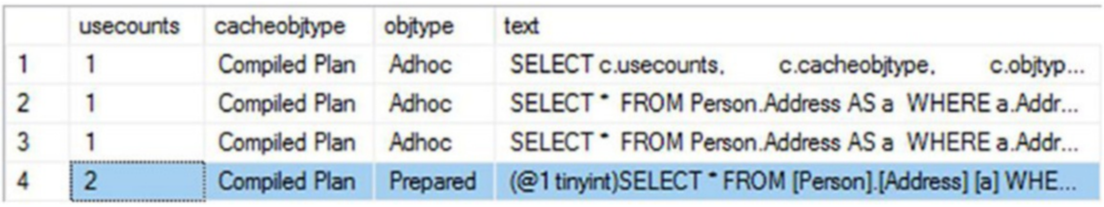
As you can see in Figure 16-8, a new plan with a parameter with a data type of `int` has been created. You can see plans for `smallint` and `bigint`. This does add some overhead to the cache but not as much as would be added by the large number of additional plans necessary for the wide variety of values. Here's the full query text from the simple parameterization:

```
(@1 int)SELECT * FROM [Person].[Address] [a] WHERE [a].[AddressID]=@1
```

Since this ad hoc query has been autoparameterized, SQL Server will reuse the existing execution plan if you reexecute the query with a different value for the variable part.

```
SELECT *
FROM Person.Address AS a
WHERE a.AddressID = 52;
```

Figure 16-9 shows the output of `sys.dm_exec_cached_plans`.



	usecounts	cacheobjtype	objtype	text
1	1	Compiled Plan	Adhoc	SELECT c.usecounts, c.cacheobjtype, c.objtyp...
2	1	Compiled Plan	Adhoc	SELECT * FROM Person.Address AS a WHERE a.Addr...
3	1	Compiled Plan	Adhoc	SELECT * FROM Person.Address AS a WHERE a.Addr...
4	2	Compiled Plan	Prepared	(@1 tinyint)SELECT * FROM [Person].[Address] [a] WHE...

Figure 16-9. *sys.dm_exec_cached_plans output showing reuse of the autoparameterized plan*

From Figure 16-9, you can see that although a new plan has been generated for this ad hoc query, the ad hoc one using an `Addressld` value of 52, the existing prepared plan is reused as indicated by the increase in the corresponding `usecounts` value to 2. The ad hoc query can be reexecuted repeatedly with different filter criterion values, reusing the existing execution plan—all this despite that the original text of the two queries does not match. The parameterized query for both would be the same, so it was reused.

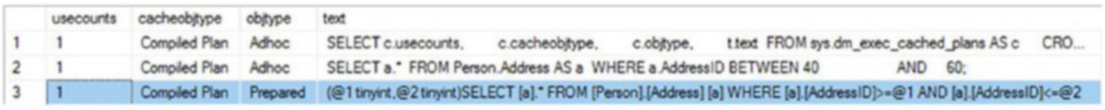
There is one more aspect to note in the parameterized query for which the execution plan is cached. In Figure 16-7, observe that the body of the parameterized query doesn't exactly match with that of the ad hoc query submitted. For instance, in the ad hoc query, there are no square brackets on any of the objects.

On realizing that the ad hoc query can be safely autoparameterized, SQL Server picks a template that can be used instead of the exact text of the query.

To understand the significance of this, consider the following query:

```
SELECT  a.*
FROM    Person.Address AS a
WHERE   a.AddressID BETWEEN 40 AND 60;
```

Figure 16-10 shows the output of `sys.dm_exec_cached_plans`.



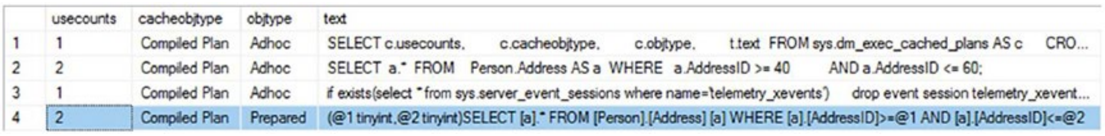
	usecounts	cacheobjtype	objtype	text
1	1	Compiled Plan	Adhoc	SELECT c.usecounts, c.cacheobjtype, c.objtype, t.text FROM sys.dm_exec_cached_plans AS c CRO...
2	1	Compiled Plan	Adhoc	SELECT a.* FROM Person.Address AS a WHERE a.AddressID BETWEEN 40 AND 60;
3	1	Compiled Plan	Prepared	(@1 tinyint,@2 tinyint)SELECT [a].* FROM [Person].[Address] [a] WHERE [a].[AddressID]>=@1 AND [a].[AddressID]<=@2

Figure 16-10. *sys.dm_exec_cached_plans output showing plan simple parameterization using a template*

From Figure 16-10, you can see that SQL Server put the query through the simplification process and substituted a pair of >= and <= operators, which are equivalent to the `BETWEEN` operator. Then the parameterization step modified the query again. That means instead of resubmitting the preceding ad hoc query using the `BETWEEN` clause, if a similar query using a pair of >= and <= is submitted, SQL Server will be able to reuse the existing execution plan. To confirm this behavior, let's modify the ad hoc query as follows:

```
SELECT  a.*
FROM    Person.Address AS a
WHERE   a.AddressID >= 40
        AND a.AddressID <= 60;
```

Figure 16-11 shows the output of `sys.dm_exec_cached_plans`.



	usecounts	cacheobjtype	objtype	text
1	1	Compiled Plan	Adhoc	SELECT c.usecounts, c.cacheobjtype, c.objtype, t.text FROM sys.dm_exec_cached_plans AS c CRO...
2	2	Compiled Plan	Adhoc	SELECT a.* FROM Person.Address AS a WHERE a.AddressID >= 40 AND a.AddressID <= 60;
3	1	Compiled Plan	Adhoc	if exists(select * from sys.server_event_sessions where name='telemetry_xevents') drop event session telemetry_xevent...
4	2	Compiled Plan	Prepared	(@1 tinyint,@2 tinyint)SELECT [a].* FROM [Person].[Address] [a] WHERE [a].[AddressID]>=@1 AND [a].[AddressID]<=@2

Figure 16-11. *sys.dm_exec_cached_plans output showing reuse of the autoparameterized plan*

From Figure 16-11, you can see that the existing plan is reused, even though the query is syntactically different from the query executed earlier. The autoparameterized plan generated by SQL Server allows the existing plan to be reused not only when the query is resubmitted with different variable values but also for queries with the same template form.

Simple Parameterization Limits

SQL Server is highly conservative during simple parameterization because the cost of a bad plan can far outweigh the cost of generating a new plan. The conservative approach prevents SQL Server from creating an unsafe autoparameterized plan. Thus, simple parameterization is limited to fairly simple cases, such as ad hoc queries with only one table. An ad hoc query with a join operation between two (or more) tables (as shown in the early part of the "Plan Reusability of an Ad Hoc Workload" section) is not considered safe for simple parameterization.

In a scalable system, do not rely on simple parameterization for plan reusability. The simple parameterization feature of SQL Server makes an educated guess as to which variables and constants can be parameterized. Instead of relying on SQL Server for simple parameterization, you should actually specify it programmatically while building your application.

Forced Parameterization

If the system you're working on consists primarily of ad hoc queries, you may want to attempt to increase the number of queries that accept parameterization. You can modify a database to attempt to force, within certain restrictions, all queries to be parameterized just like in simple parameterization.

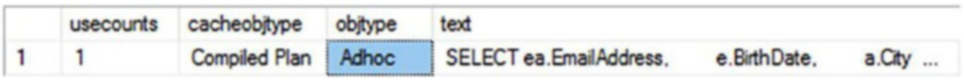
To do this, you have to change the database option PARAMETERIZATION to FORCED using ALTER DATABASE like this:

```
ALTER DATABASE AdventureWorks2017 SET PARAMETERIZATION FORCED;
```

But, if you have a query that is in any way complicated, you won't get simple parameterization.

```
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 = 'WA';
```

When you run this query, simple parameterization is not applied, as you can see in Figure 16-12.



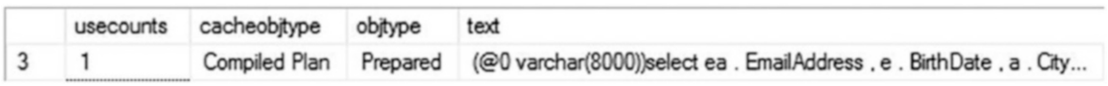
	usecounts	cacheobjtype	objtype	text
1	1	Compiled Plan	Adhoc	SELECT ea.EmailAddress, e.BirthDate, a.City ...

Figure 16-12. *A more complicated query doesn't get parameterized*

No prepared plans are visible in the output from sys.dm_exec_cached_plans. But if we use the previous script to set PARAMETERIZATION to FORCED, we can rerun the query after clearing the cache.

```
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
```

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:

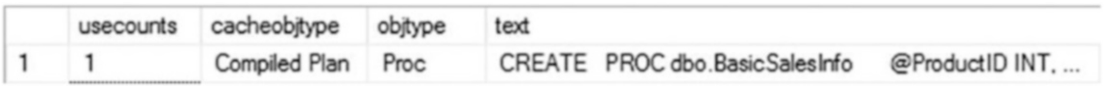
```
CREATE OR ALTER PROC dbo.BasicSalesInfo
    @ProductID INT,
    @CustomerID INT
```

```
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`.

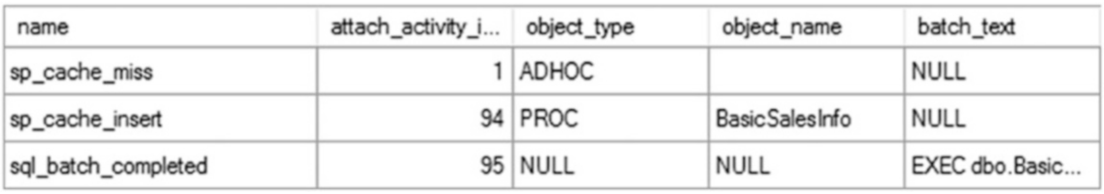


	usecounts	cacheobjtype	objtype	text
1	1	Compiled Plan	Proc	CREATE PROC dbo.BasicSalesInfo @ProductID INT, ...

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.



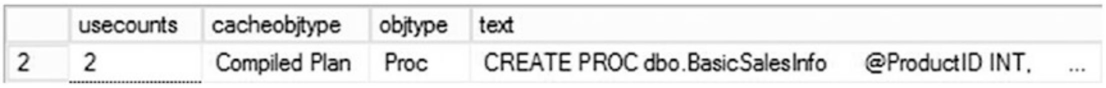
name	attach_activity_i...	object_type	object_name	batch_text
sp_cache_miss	1	ADHOC		NULL
sp_cache_insert	94	PROC	BasicSalesInfo	NULL
sql_batch_completed	95	NULL	NULL	EXEC dbo.Basic...

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.

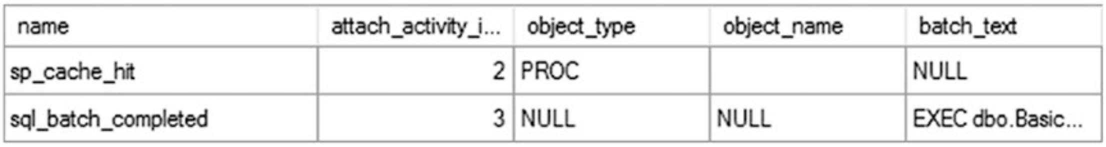
```
EXEC dbo.BasicSalesInfo @CustomerID = 29690, @ProductID = 777;
```



	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

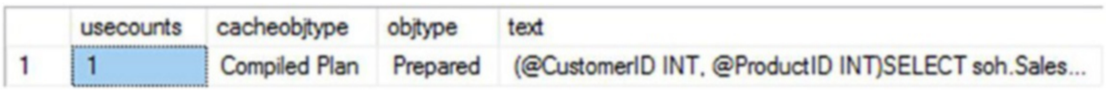
`sp_executesql` is a system stored procedure that provides a mechanism to submit one or more queries as a prepared workload. It allows the variable parts of the query to be explicitly parameterized, and it can therefore provide execution plan reusability as effective as a stored procedure. The `SELECT` 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%';
```



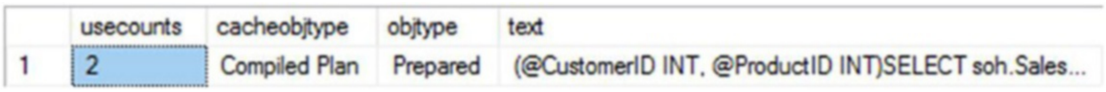
	usecounts	cacheobjtype	objtype	text
1	1	Compiled Plan	Prepared	(@CustomerID INT, @ProductID INT)SELECT soh.Sales...

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.