```
    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.

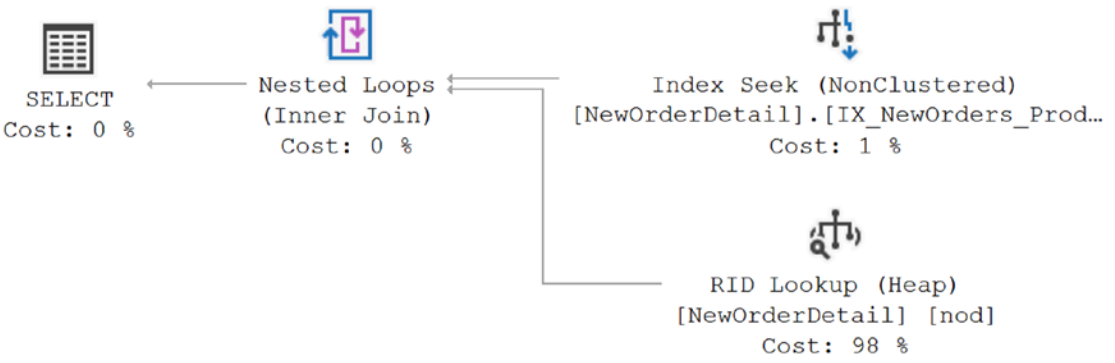


Figure 18-5. *Execution plan prior to data changes*

Note Please ensure that the setting for the graphical execution plan is OFF; otherwise, the output of `STATISTICS XML` won't display.

While reexecuting the stored procedure, SQL Server automatically detects that the statistics on the index have changed. This causes a recompilation of the `SELECT` statement within the procedure, with the optimizer determining a better processing strategy, before executing the `SELECT` statement within the stored procedure, as you can see in Figure 18-6.

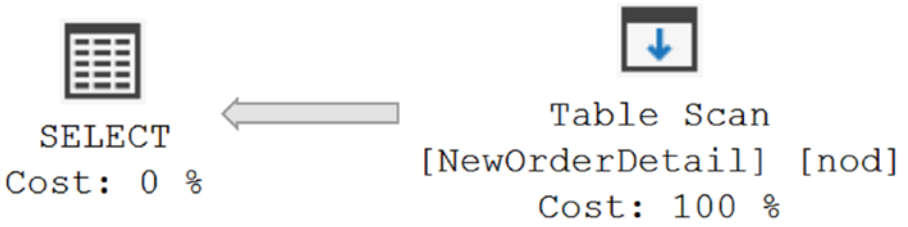
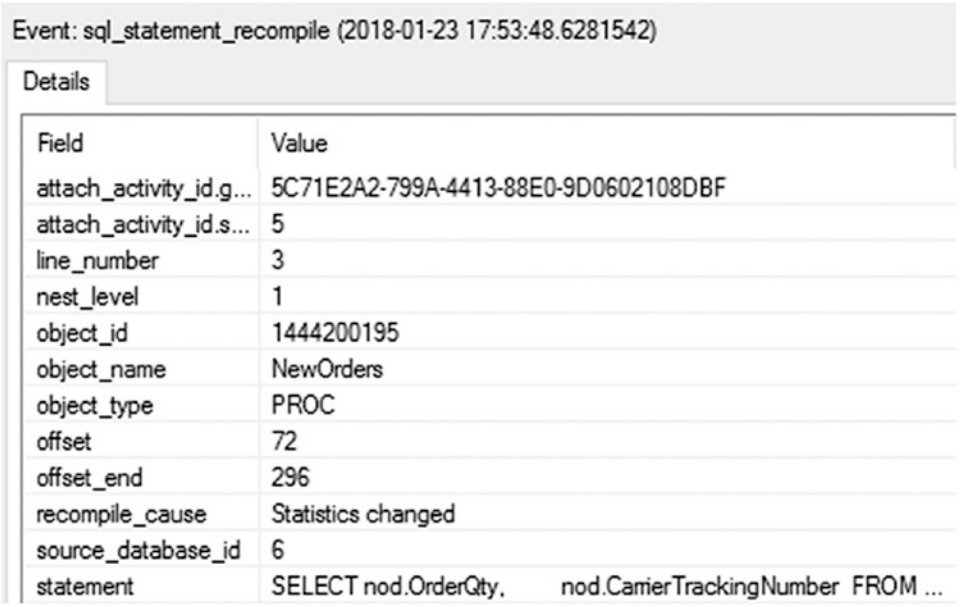


Figure 18-6. *Effect of statistics change on the execution plan*

Figure 18-7 shows the corresponding Extended Events output.


Event: sql_statement_recompile (2018-01-23 17:53:48.6281542)

Details

Field	Value
attach_activity_id.g...	5C71E2A2-799A-4413-88E0-9D0602108DBF
attach_activity_id.s...	5
line_number	3
nest_level	1
object_id	1444200195
object_name	NewOrders
object_type	PROC
offset	72
offset_end	296
recompile_cause	Statistics changed
source_database_id	6
statement	SELECT nod.OrderQty, nod.CarrierTrackingNumber FROM ...

Figure 18-7. *Effect of statistics change on the stored procedure recompilation*

In Figure 18-7, you can see that to execute the `SELECT` statement during the second execution of the stored procedure, a recompilation was required. From the value of `recompile_cause` (`Statistics Changed`), you can understand that the recompilation was because of the statistics change. As part of creating the new plan, the statistics are automatically updated, as indicated by the `Auto Stats` event, which occurred after the call for a recompile of the statement. You can also verify the automatic update of the statistics using the `DBCC SHOW_STATISTICS` statement or `sys.dm_db_stats_properties`, as explained in Chapter 13.

Deferred Object Resolution

Queries often dynamically create and subsequently access database objects. When such a query is executed for the first time, the first execution plan won't contain the information about the objects to be created during runtime. Thus, in the first execution plan, the processing strategy for those objects is deferred until the runtime of the query. When a DML statement (within the query) referring to one of those objects is executed, the query is recompiled to generate a new plan containing the processing strategy for the object.

Both a regular table and a local temporary table can be created within a stored procedure to hold intermediate result sets. The recompilation of the statement because of deferred object resolution behaves differently for a regular table when compared to a local temporary table, as explained in the following section.

Recompilation Because of a Regular Table

To understand the query recompilation issue by creating a regular table within the stored procedure, consider the following example:

```
CREATE OR ALTER PROC dbo.TestProc
AS
CREATE TABLE dbo.ProcTest1 (C1 INT); --Ensure table doesn't exist
SELECT *
FROM dbo.ProcTest1; --Causes recompilation
DROP TABLE dbo.ProcTest1;
GO

EXEC dbo.TestProc; --First execution
EXEC dbo.TestProc; --Second execution
```

When the stored procedure is executed for the first time, an execution plan is generated before the actual execution of the stored procedure. If the table created within the stored procedure doesn't exist (as expected in the preceding code) before the stored procedure is created, then the plan won't contain the processing strategy for the `SELECT` statement referring to the table. Thus, to execute the `SELECT` statement, the statement needs to be recompiled, as shown in Figure 18-8.

name	statement	recompile_cause	attach_activity_id.seq
sp_statement_starting	CREATE TABLE dbo.ProcTest1 (C1 INT)	NULL	12
sp_statement_completed	CREATE TABLE dbo.ProcTest1 (C1 INT)	NULL	13
sp_statement_starting	SELECT * FROM dbo.ProcTest1	NULL	14
sql_statement_recompile	SELECT * FROM dbo.ProcTest1	Schema changed	15
sp_statement_starting	SELECT * FROM dbo.ProcTest1	NULL	16
sp_statement_completed	SELECT * FROM dbo.ProcTest1	NULL	17
sp_statement_starting	DROP TABLE dbo.ProcTest1	NULL	18
sp_statement_completed	DROP TABLE dbo.ProcTest1	NULL	19

Figure 18-8. *Extended Events output showing a stored procedure recompilation because of a regular table*

You can see that the `SELECT` statement is recompiled when it's executed the second time. Dropping the table within the stored procedure during the first execution doesn't drop the query plan saved in the plan cache. During the subsequent execution of the stored procedure, the existing plan includes the processing strategy for the table. However, because of the re-creation of the table within the stored procedure, SQL Server considers it a change to the table schema. Therefore, SQL Server recompiles the statement within the stored procedure before executing the `SELECT` statement during the subsequent execution of the rest of the stored procedure. The value of the `recompile_clause` for the corresponding `sql_statement_recompile` event reflects the cause of the recompilation.

Recompilation Because of a Local Temporary Table

Most of the time in the stored procedure you create local temporary tables instead of regular tables. To understand how differently the local temporary tables affect stored procedure recompilation, modify the preceding example by just replacing the regular table with a local temporary table.

```
CREATE OR ALTER PROC dbo.TestProc
AS
CREATE TABLE #ProcTest1 (C1 INT); --Ensure table doesn't exist
SELECT *
FROM #ProcTest1; --Causes recompilation
DROP TABLE #ProcTest1;
```

```
GO

EXEC dbo.TestProc; --First execution
EXEC dbo.TestProc; --Second execution
```

Since a local temporary table is automatically dropped when the execution of a stored procedure finishes, it's not necessary to drop the temporary table explicitly. But, following good programming practice, you can drop the local temporary table as soon as its work is done. Figure 18-9 shows the Extended Events output for the preceding example.

name	statement	recompile_cause	attach_activity_id.seq
sp_statement_starting	CREATE TABLE #ProcTest1 (C1 INT)	NULL	2
sp_statement_completed	CREATE TABLE #ProcTest1 (C1 INT)	NULL	3
sp_statement_starting	SELECT * FROM #ProcTest1	NULL	4
sql_statement_recompile	SELECT * FROM #ProcTest1	Deferred compile	5
sp_statement_starting	SELECT * FROM #ProcTest1	NULL	6
sp_statement_completed	SELECT * FROM #ProcTest1	NULL	7

Figure 18-9. *Extended Events output showing a stored procedure recompilation because of a local temporary table*

You can see that the query is recompiled when executed for the first time. The cause of the recompilation, as indicated by the corresponding `recompile_cause` value, is the same as the cause of the recompilation on a regular table. However, note that when the stored procedure is reexecuted, it isn't recompiled, unlike the case with a regular table.

The schema of a local temporary table during subsequent execution of the stored procedure remains the same as during the previous execution. A local temporary table isn't available outside the scope of the stored procedure, so its schema can't be altered in any way between multiple executions. Thus, SQL Server safely reuses the existing plan (based on the previous instance of the local temporary table) during the subsequent execution of the stored procedure and thereby avoids the recompilation.

Note To avoid recompilation, it makes sense to hold the intermediate result sets in the stored procedure using local temporary tables, instead of using temporarily created regular tables. But, this makes sense only if you can avoid data skew, which could lead to other bad plans. In that case, the recompile might be less painful.

SET Options Changes

The execution plan of a stored procedure is dependent on the environment settings. If the environment settings are changed within a stored procedure, then SQL Server recompiles the queries on every execution. For example, consider the following code:

```
CREATE OR ALTER PROC dbo.TestProc
AS
SELECT  'a' + NULL + 'b'; --1st
SET CONCAT_NULL_YIELDS_NULL OFF;
SELECT  'a' + NULL + 'b'; --2nd
SET ANSI_NULLS OFF;
SELECT  'a' + NULL + 'b';
 --3rd
GO
EXEC dbo.TestProc; --First execution
EXEC dbo.TestProc; --Second execution
```

Changing the SET options in the stored procedure causes SQL Server to recompile the stored procedure before executing the statement after the SET statement. Thus, this stored procedure is recompiled twice: once before executing the second SELECT statement and once before executing the third SELECT statement. The Extended Events output in Figure 18-10 shows this.

name	statement	recompile_cause	attach_activity_id.seq
sp_statement_starting	SET CONCAT_NULL_YIELDS_NULL OFF;	NULL	4
sp_statement_completed	SET CONCAT_NULL_YIELDS_NULL OFF;	NULL	5
sp_statement_starting	SELECT 'a' + NULL + 'b'	NULL	6
sql_statement_recompile	SELECT 'a' + NULL + 'b'	Set option change	7
sp_statement_starting	SELECT 'a' + NULL + 'b'	NULL	8
sp_statement_completed	SELECT 'a' + NULL + 'b'	NULL	9

Figure 18-10. *Extended Events output showing a stored procedure recompilation because of a SET option change*

If the procedure were reexecuted, you wouldn't see a recompile since those are now part of the execution plans.

Since `SET NOCOUNT` doesn't change the environment settings, unlike the `SET` statements used to change the ANSI settings as shown previously, `SET NOCOUNT` doesn't cause stored procedure recompilation. I explain how to use `SET NOCOUNT` in detail in Chapter 19.

Execution Plan Aging

SQL Server manages the size of the procedure cache by maintaining the age of the execution plans in the cache, as you saw in Chapter 16. If a stored procedure is not reexecuted for a long time, the age field of the execution plan can come down to 0, and the plan can be removed from the cache because of memory pressure. When this happens and the stored procedure is reexecuted, a new plan will be generated and cached in the procedure cache. However, if there is enough memory in the system, unused plans are not removed from the cache until memory pressure increases.

Explicit Call to sp_recompile

SQL Server automatically recompiles queries when the schema changes or statistics are altered enough. It also provides the `sp_recompile` system stored procedure to manually mark entire stored procedures for recompilation. This stored procedure can be called on a table, view, stored procedure, or trigger. If it is called on a stored procedure or a trigger, the stored procedure or trigger is recompiled the next time it is executed. Calling `sp_recompile` on a table or a view marks all the stored procedures and triggers that refer to the table/view for recompilation the next time they are executed.

For example, if `sp_recompile` is called on table `Test1`, all the stored procedures and triggers that refer to table `Test1` are marked for recompilation and are recompiled the next time they are executed, like so:

```
sp_recompile  'Test1';
```

You can use `sp_recompile` to cancel the reuse of an existing plan when executing dynamic queries with `sp_executesql`. As demonstrated in the previous chapter, you should not parameterize the variable parts of a query whose range of values may require different processing strategies for the query. For instance, reconsidering the corresponding example, you know that the second execution of the query reuses the plan generated for the first execution. The example is repeated here for easy reference:

```
--clear the procedure cache
DECLARE @planhandle VARBINARY(64)
SELECT @planhandle = deqs.plan_handle
FROM sys.dm_exec_query_stats AS deqs
    CROSS APPLY sys.dm_exec_sql_text(deqs.sql_handle) AS dest
WHERE dest.text LIKE '%SELECT soh.SalesOrderNumber,%'
IF @planhandle IS NOT NULL
    DBCC FREEPROCCACHE(@planhandle);
GO

DECLARE @query NVARCHAR(MAX);
DECLARE @param 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;'
SET @param = N'@CustomerId INT';
EXEC sp_executesql @query, @param, @CustomerId = 1;
EXEC sp_executesql @query, @param, @CustomerId = 30118;
```

The second execution of the query performs an `Index Scan` operation on the `SalesOrderHeader` table to retrieve the data from the table. As explained in Chapter 8, an `Index Seek` operation may have been preferred on the `SalesOrderHeader` table for the second execution. You can achieve this by executing the `sp_recompile` system stored procedure on the `SalesOrderHeader` table as follows:

```
EXEC sp_recompile  'Sales.SalesOrderHeader'
```

Now, if the query with the second parameter value is reexecuted, the plan for the query will be recompiled as marked by the preceding `sp_recompile` statement. This allows SQL Server to generate an optimal plan for the second execution.

Well, there is a slight problem here: you will likely want to reexecute the first statement again. With the plan existing in the cache, SQL Server will reuse the plan (the `Index Scan` operation on the `SalesOrderHeader` table) for the first statement even though an `Index Seek` operation (using the index on the filter criterion column `soh.CustomerID`) would have been optimal. One way of avoiding this problem is to create a stored procedure for the query and use the `OPTION (RECOMPILE)` clause on the statement. I'll go over the various methods for controlling the recompile next.

Explicit Use of RECOMPILE

SQL Server allows stored procedures and queries to be explicitly recompiled using the `RECOMPILE` command in three ways: with the `CREATE PROCEDURE` statement, as part of the `EXECUTE` statement, and in a query hint. These methods decrease the effectiveness of plan reusability and can result in radical use of the CPU, so you should consider them only under the specific circumstances explained in the following sections.

RECOMPILE Clause with the CREATE PROCEDURE Statement

Sometimes the plan requirements of a stored procedure will vary as the parameter values to the stored procedure change. In such a case, reusing the plan with different parameter values may degrade the performance of the stored procedure. You can avoid this by using the `RECOMPILE` clause with the `CREATE PROCEDURE` statement. For example, for the query in the preceding section, you can create a stored procedure with the `RECOMPILE` clause.

```
CREATE OR ALTER PROCEDURE dbo.CustomerList @CustomerId INT
WITH RECOMPILE
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;
GO
```

The RECOMPILE clause prevents the caching of the stored procedure plan for every statement within the procedure. Every time the stored procedure is executed, new plans are generated. Therefore, if the stored procedure is executed with the soh.CustomerID value as 30118 or 1,

```
EXEC CustomerList
    @CustomerId = 1;
EXEC CustomerList
    @CustomerId = 30118;
```

a new plan is generated during the individual execution, as shown in Figure 18-11.

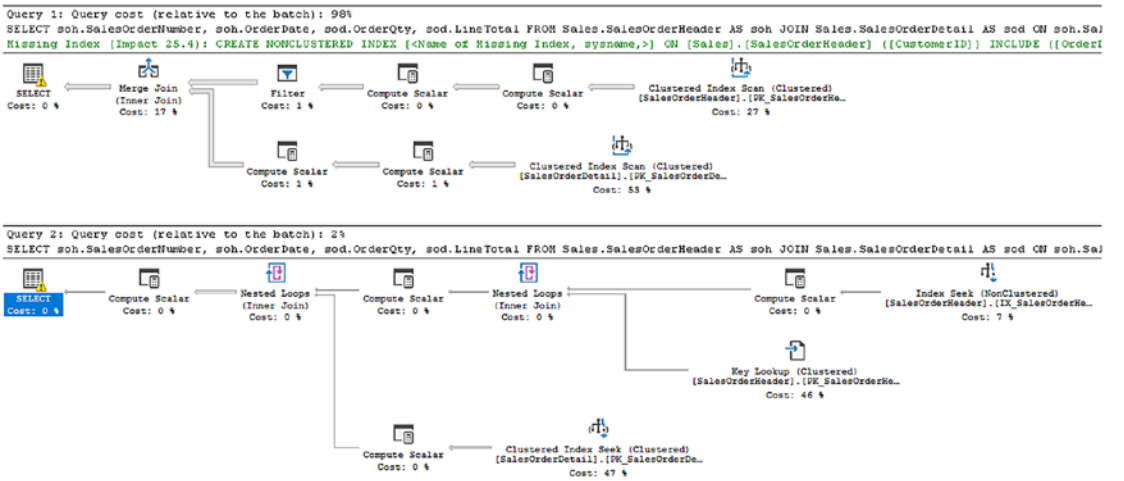


Figure 18-11. *Effect of the RECOMPILE clause used in stored procedure creation*

RECOMPILE Clause with the EXECUTE Statement

As shown previously, specific parameter values in a stored procedure may require a different plan, depending upon the nature of the values. You can take the RECOMPILE clause out of the stored procedure and use it on a case-by-case basis when you execute the stored procedure, as follows:

```
EXEC dbo.CustomerList
    @CustomerId = 1
    WITH RECOMPILE;
```

When the stored procedure is executed with the RECOMPILE clause, a new plan is generated temporarily. The new plan isn't cached, and it doesn't affect the existing plan. When the stored procedure is executed without the RECOMPILE clause, the plan is cached as usual. This provides some control over reusability of the existing plan cache rather than using the RECOMPILE clause with the CREATE PROCEDURE statement.

Since the plan for the stored procedure when executed with the RECOMPILE clause is not cached, the plan is regenerated every time the stored procedure is executed with the RECOMPILE clause. However, for better performance, instead of using RECOMPILE, you should consider creating separate stored procedures, one for each set of parameter values that requires a different plan, assuming they are easily identified and you're dealing only with a small number of possible plans.

RECOMPILE Hints to Control Individual Statements

While you can use either of the previous methods to recompile an entire procedure, this can be problematic if the procedure has multiple commands. All statements within a procedure will be recompiled using either of the previous methods. Compile time for queries can be the most expensive part of executing some queries, so recompiles should be avoided. Because of this, a more granular approach is to isolate the recompile to just the statement that needs it. This is accomplished using the RECOMPILE query hint as follows:

```
CREATE OR ALTER PROCEDURE dbo.CustomerList @CustomerId INT
AS
SELECT a.AddressLine1,
       a.AddressLine2,
       a.City,
       a.PostalCode
FROM Person.Address AS a
    JOIN Sales.SalesOrderHeader AS soh
        ON soh.ShipToAddressID = a.AddressID
WHERE soh.CustomerID = @CustomerId;

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
OPTION (RECOMPILE);

SELECT bom.BillOfMaterialsID,
       p.Name,
       sod.OrderQty
FROM Production.BillOfMaterials AS bom
    JOIN Production.Product AS p
        ON p.ProductID = bom.ProductAssemblyID
    JOIN Sales.SalesOrderDetail AS sod
        ON sod.ProductID = p.ProductID
    JOIN Sales.SalesOrderHeader AS soh
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE soh.CustomerID = @CustomerId;
GO
```

This middle query in this procedure will appear to behave the same way as the one where the `RECOMPILE` was applied to the entire procedure, but if you added multiple statements to this query, only the statement with the `OPTION (RECOMPILE)` query hint would be compiled at every execution of the procedure.

Avoiding Recompilations

Sometimes recompilation is beneficial, but at other times it is worth avoiding. If a new index is created on a column referred to in the `WHERE` or `JOIN` clause of a query, it makes sense to regenerate the execution plans of stored procedures referring to the table so they can benefit from using the index. However, if recompilation is deemed detrimental to performance, such as when it's causing blocking or using up resources such as the CPU, you can avoid it by following these implementation practices:

- Don't interleave DDL and DML statements.
- Avoid recompilation caused by statistics changes.
- Use the `KEEPFIXED PLAN` option.

- Disable the auto update statistics feature on the table.
- Use table variables.
- Avoid changing SET options within the stored procedure.
- Use the OPTIMIZE FOR query hint.
- Use plan guides.

Don't Interleave DDL and DML Statements

In stored procedures, DDL statements are often used to create local temporary tables and to change their schema (including adding indexes). Doing so can affect the validity of the existing plan and can cause recompilation when the stored procedure statements referring to the tables are executed. To understand how the use of DDL statements for local temporary tables can cause repetitive recompilation of the stored procedure, consider the following example:

```
IF (SELECT OBJECT_ID('dbo.TempTable')) IS NOT NULL
    DROP PROC dbo.TempTable
GO
CREATE PROC dbo.TempTable
AS
CREATE TABLE #MyTempTable (ID INT,
                           Dsc NVARCHAR(50))
INSERT INTO #MyTempTable (ID,
                          Dsc)
SELECT pm.ProductModelID,
       pm.Name
FROM Production.ProductModel AS pm; --Needs 1st recompilation
SELECT *
FROM #MyTempTable AS mtt;
CREATE CLUSTERED INDEX iTest ON #MyTempTable (ID);
SELECT *
FROM #MyTempTable AS mtt; --Needs 2nd recompilation
CREATE TABLE #t2 (c1 INT);
```

```
SELECT *
FROM #t2;
--Needs 3rd recompilation
GO

EXEC dbo.TempTable; --First execution
```

The stored procedure has interleaved DDL and DML statements. Figure 18-12 shows the Extended Events output of this code.

name	statement	recompile_cause
sp_statement_starting	INSERT INTO #MyTempTable (ID, Ds...	NULL
sql_statement_recompile	INSERT INTO #MyTempTable (ID, Ds...	Deferred compile
sp_statement_starting	INSERT INTO #MyTempTable (ID, Ds...	NULL
sp_statement_completed	INSERT INTO #MyTempTable (ID, Ds...	NULL
sp_statement_starting	SELECT * FROM #MyTempTable AS mtt	NULL
sql_statement_recompile	SELECT * FROM #MyTempTable AS mtt	Deferred compile
sp_statement_starting	SELECT * FROM #MyTempTable AS mtt	NULL
sp_statement_completed	SELECT * FROM #MyTempTable AS mtt	NULL
sp_statement_starting	CREATE CLUSTERED INDEX iTest ON #MyTempTabl...	NULL
sp_statement_completed	CREATE CLUSTERED INDEX iTest ON #MyTempTabl...	NULL
sp_statement_starting	SELECT * FROM #MyTempTable AS mtt	NULL
sql_statement_recompile	SELECT * FROM #MyTempTable AS mtt	Deferred compile
sp_statement_starting	SELECT * FROM #MyTempTable AS mtt	NULL
sp_statement_completed	SELECT * FROM #MyTempTable AS mtt	NULL
sp_statement_starting	CREATE TABLE #t2 (c1 INT)	NULL
sp_statement_completed	CREATE TABLE #t2 (c1 INT)	NULL
sp_statement_starting	SELECT * FROM #t2	NULL
sql_statement_recompile	SELECT * FROM #t2	Deferred compile
sp_statement_starting	SELECT * FROM #t2	NULL
sp_statement_completed	SELECT * FROM #t2	NULL

Figure 18-12. *Extended Events output showing recompilation because of DDL and DML interleaving*

The statements are recompiled four times.

- The execution plan generated for a query when it is first executed doesn't contain any information about local temporary tables. Therefore, the first generated plan can never be used to access the temporary table using a DML statement.
- The second recompilation comes from the changes encountered in the data contained within the table as it gets loaded.
- The third recompilation is because of a schema change in the first temporary table (`#MyTempTable`). The creation of the index on `#MyTempTable` invalidates the existing plan, causing a recompilation when the table is accessed again. If this index had been created before the first recompilation, then the existing plan would have remained valid for the second `SELECT` statement, too. Therefore, you can avoid this recompilation by putting the `CREATE INDEX` DDL statement above all DML statements referring to the table.
- The fourth recompilation generates a plan to include the processing strategy for `#t2`. The existing plan has no information about `#t2` and therefore can't be used to access `#t2` using the third `SELECT` statement. If the `CREATE TABLE` DDL statement for `#t2` had been placed before all the DML statements that could cause a recompilation, then the first recompilation itself would have included the information on `#t2`, avoiding the third recompilation.

Avoiding Recompilations Caused by Statistics Change

In the "Analyzing Causes of Recompilation" section, you saw that a change in statistics is one of the causes of recompilation. On a simple table with uniform data distribution, recompilation because of a change of statistics may generate a plan identical to the previous plan. In such situations, recompilation can be unnecessary and should be avoided if it is too costly. But, most of the time, changes in statistics need to be reflected in the execution plan. I'm just talking about situations where you have a long recompile time or excessive recompiles hitting your CPU.

You have two techniques to avoid recompilations caused by statistics change.

- Use the KEEPFIXED PLAN option.
- Disable the auto update statistics feature on the table.

Using the KEEPFIXED PLAN Option

SQL Server provides a KEEPFIXED PLAN option to avoid recompilations because of a statistics change. To understand how you can use KEEPFIXED PLAN, consider statschanges.sql with an appropriate modification to use the KEEPFIXED PLAN option.

```
IF (SELECT OBJECT_ID('dbo.Test1')) IS NOT NULL
    DROP TABLE dbo.Test1;
GO
CREATE TABLE dbo.Test1 (C1 INT,
                        C2 CHAR(50));
INSERT INTO dbo.Test1
VALUES (1, '2');
CREATE NONCLUSTERED INDEX IndexOne ON dbo.Test1 (C1);
GO
--Create a stored procedure referencing the previous table
CREATE OR ALTER PROC dbo.TestProc
AS
SELECT *
FROM dbo.Test1 AS t
WHERE t.C1 = 1
OPTION (KEEPFIXED PLAN);
GO

--First execution of stored procedure with 1 row in the table
EXEC dbo.TestProc;
--First execution

--Add many rows to the table to cause statistics change
WITH Nums
AS (SELECT 1 AS n
    UNION ALL
```

```
    SELECT n + 1
    FROM Nums
    WHERE n < 1000)
INSERT INTO dbo.Test1 (C1,
                       C2)
SELECT 1,
       n
FROM Nums
OPTION (MAXRECURSION 1000);
GO
--Reexecute the stored procedure with a change in statistics
EXEC dbo.TestProc; --With change in data distribution
```

Figure 18-13 shows the Extended Events output.

name	statement	recompile_cause	attach_activity_id.seq	batch_text
sql_statement_completed	CREATE PROC dbo.TestProc AS SELECT * FROM d...	NULL	2	NULL
sql_batch_completed	NULL	NULL	3	CREATE PROC dbo.TestP...
sql_statement_starting	EXEC dbo.TestProc	NULL	1	NULL
auto_stats	NULL	NULL	2	NULL
sp_statement_starting	SELECT * FROM dbo.Test1 AS t WHERE t.C1 = 1 O...	NULL	3	NULL
sp_statement_completed	SELECT * FROM dbo.Test1 AS t WHERE t.C1 = 1 O...	NULL	4	NULL
sql_statement_completed	EXEC dbo.TestProc	NULL	5	NULL
sql_statement_starting	WITH Nums AS (SELECT 1 AS n UNION ALL S...	NULL	6	NULL
sql_statement_recompile	WITH Nums AS (SELECT 1 AS n UNION ALL S...	Schema changed	7	NULL
sql_statement_starting	WITH Nums AS (SELECT 1 AS n UNION ALL S...	NULL	8	NULL
sql_statement_completed	WITH Nums AS (SELECT 1 AS n UNION ALL S...	NULL	9	NULL
sql_batch_completed	NULL	NULL	10	--First execution of stored ...
sql_statement_starting	EXEC dbo.TestProc	NULL	1	NULL
sp_statement_starting	SELECT * FROM dbo.Test1 AS t WHERE t.C1 = 1 O...	NULL	2	NULL
sp_statement_completed	SELECT * FROM dbo.Test1 AS t WHERE t.C1 = 1 O...	NULL	3	NULL
sql_statement_completed	EXEC dbo.TestProc	NULL	4	NULL
sql_batch_completed	NULL	NULL	5	--Reexecute the stored pro...

Figure 18-13. *Extended Events output showing the role of the KEEPFIXED PLAN option in reducing recompilation*

You can see that, unlike in the earlier example with changes in data, there's no `auto_stats` event (see Figure 18-7). Consequently, there's no additional recompilation. Therefore, by using the `KEEPFIXED PLAN` option, you can avoid recompilation because of a statistics change.

There is one recompile event visible in Figure 18-13, but it is the result of the data modification query, not the execution of the stored procedure as you would expect without the `KEEPFIXED PLAN` option.

Note This is a potentially dangerous choice. Before you consider using this option, ensure that any new plans that would have been generated are not superior to the existing plan and that you've exhausted all other possible solutions. In most cases, recompiling queries is preferable, though potentially costly.

Disable Auto Update Statistics on the Table

You can also avoid recompilation because of a statistics update by disabling the automatic statistics update on the relevant table. For example, you can disable the auto update statistics feature on table `Test1` as follows:

```
EXEC sp_autostats
    'dbo.Test1',
    'OFF' ;
```

If you disable this feature on the table before inserting the large number of rows that causes statistics change, you can avoid the recompilation because of a statistics change.

However, be cautious with this technique since outdated statistics can adversely affect the effectiveness of the cost-based optimizer, as discussed in Chapter 13. Also, as explained in Chapter 13, if you disable the automatic update of statistics, you should have a SQL job to manually update the statistics regularly.

Using Table Variables

One of the variable types supported by SQL Server 2014 is the table variable. You can create the table variable data type like other data types by using the `DECLARE` statement. It behaves like a local variable, and you can use it inside a stored procedure to hold intermediate result sets, as you do using a temporary table.

You can avoid the recompilations caused by a temporary table if you use a table variable. Since statistics are not created for table variables, the different recompilation issues associated with temporary tables are not applicable to it. For instance, consider

the script used in the section tables are not applicable to it. For instance, consideration issues associld ir reference:

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

EXEC dbo.TestProc; --First execution
```

Because of deferred object resolution, the stored procedure is recompiled during the first execution. You can avoid this recompilation caused by the temporary table by using the table variable as follows:

```
CREATE OR ALTER PROC dbo.TestProc
AS
DECLARE @TempTable TABLE (C1 INT);
INSERT INTO @TempTable (C1)
VALUES (42);
--Recompilation not needed
GO

EXEC dbo.TestProc; --First execution
```

Figure 18-14 shows the Extended Events output for the first execution of the stored procedure. The recompilation caused by the temporary table has been avoided by using the table variable.

name	statement	recompile_cause	attach_activity_id.seq
sql_statement_starting	EXEC dbo.TestProc	NULL	1
sp_statement_starting	INSERT INTO @TempTable (C1) VALUES (42)	NULL	2
sp_statement_completed	INSERT INTO @TempTable (C1) VALUES (42)	NULL	3
sql_statement_completed	EXEC dbo.TestProc	NULL	4

Figure 18-14. *Extended Events output showing the role of a table variable in resolving recompilation*