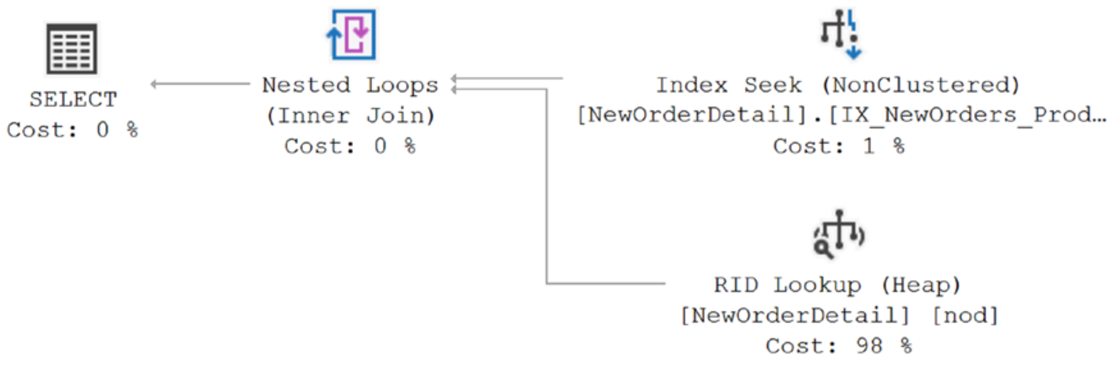


***Figure 18-5.*** *Execution plan prior to data changes*

---

**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.

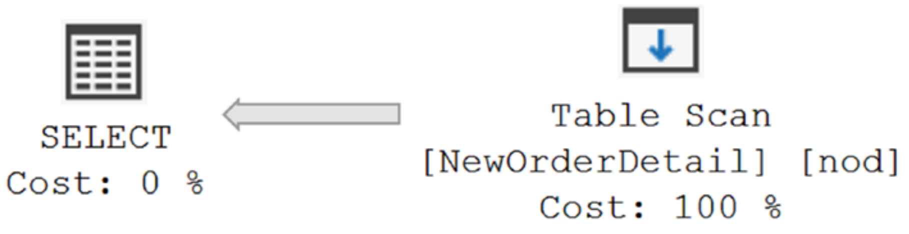
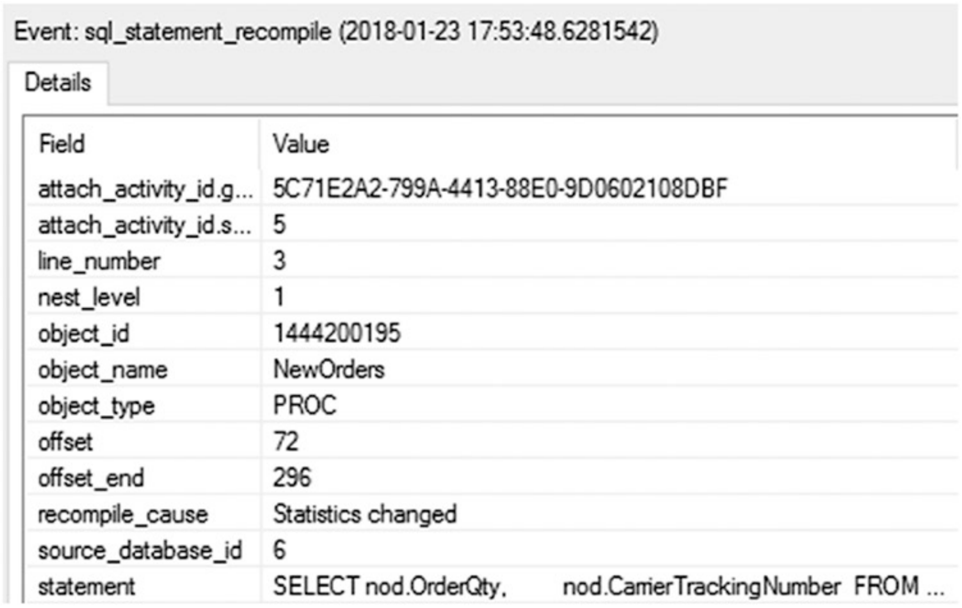


***Figure 18-6.*** *Effect of statistics change on the execution plan*

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.

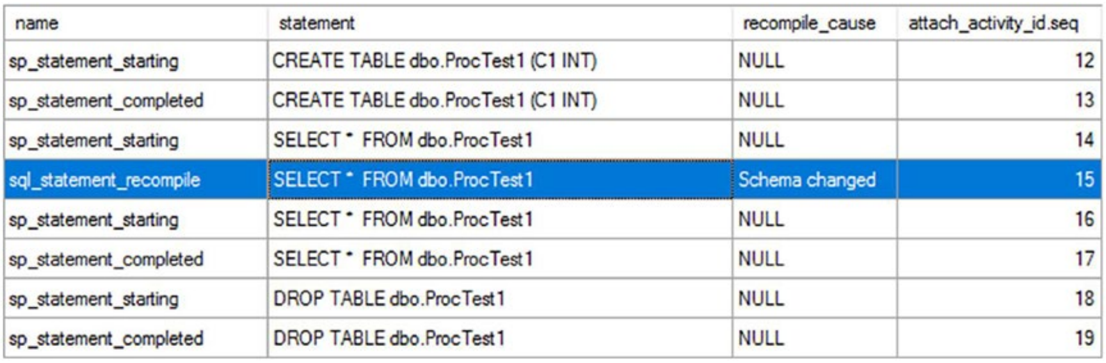
Both a regular table and a local temporary table can be created within a stored procedure to hold intermediate result sets. The recompilation of the statement because of deferred object resolution behaves differently for a regular table when compared to a local temporary table, as explained in the following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

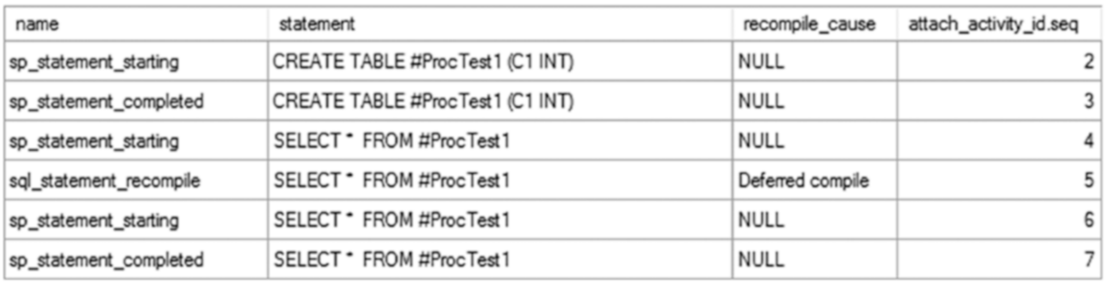
## Recompilation Because of a Local Temporary Table

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.

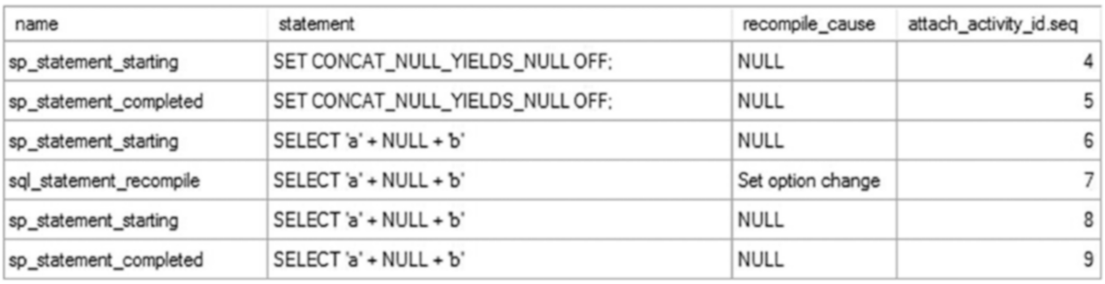
---

## 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

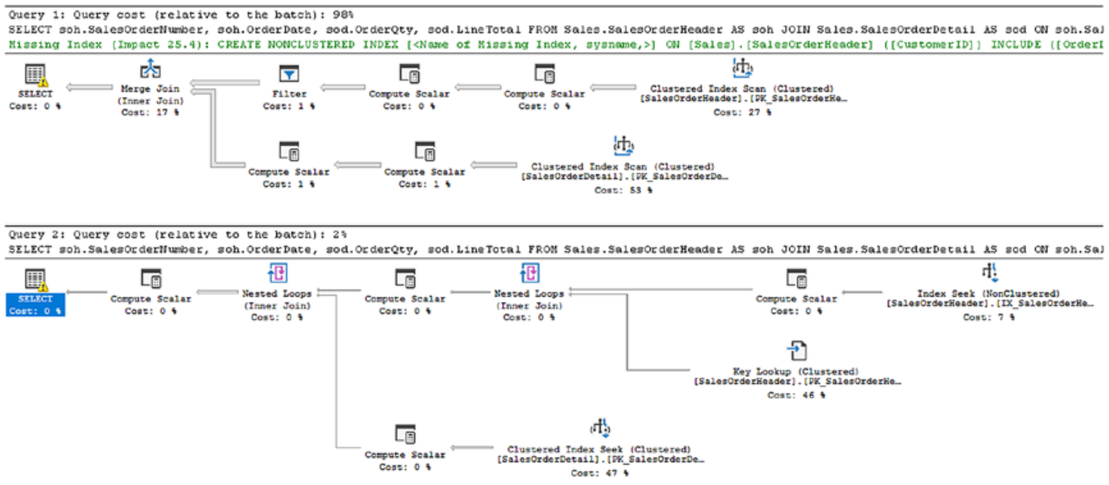
Sometimes the plan requirements of a stored procedure will vary as the parameter values to the stored procedure change. In such a case, reusing the plan with different parameter values may degrade the performance of the stored procedure. You can avoid this by using the RECOMPILE clause with the CREATE 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.



```
Query 1: Query cost (relative to the batch): 98%
SELECT soh.SalesOrderNumber, soh.OrderDate, sod.OrderQty, sod.LineTotal FROM Sales.SalesOrderHeader AS soh JOIN Sales.SalesOrderDetail AS sod ON soh.Sal
Missing Index (Impact 25.4): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Sales].[SalesOrderHeader] ([CustomerID]) INCLUDE ([OrderI
```

```
Query 2: Query cost (relative to the batch): 2%
SELECT soh.SalesOrderNumber, soh.OrderDate, sod.OrderQty, sod.LineTotal FROM Sales.SalesOrderHeader AS soh JOIN Sales.SalesOrderDetail AS sod ON soh.Sal
```

***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.

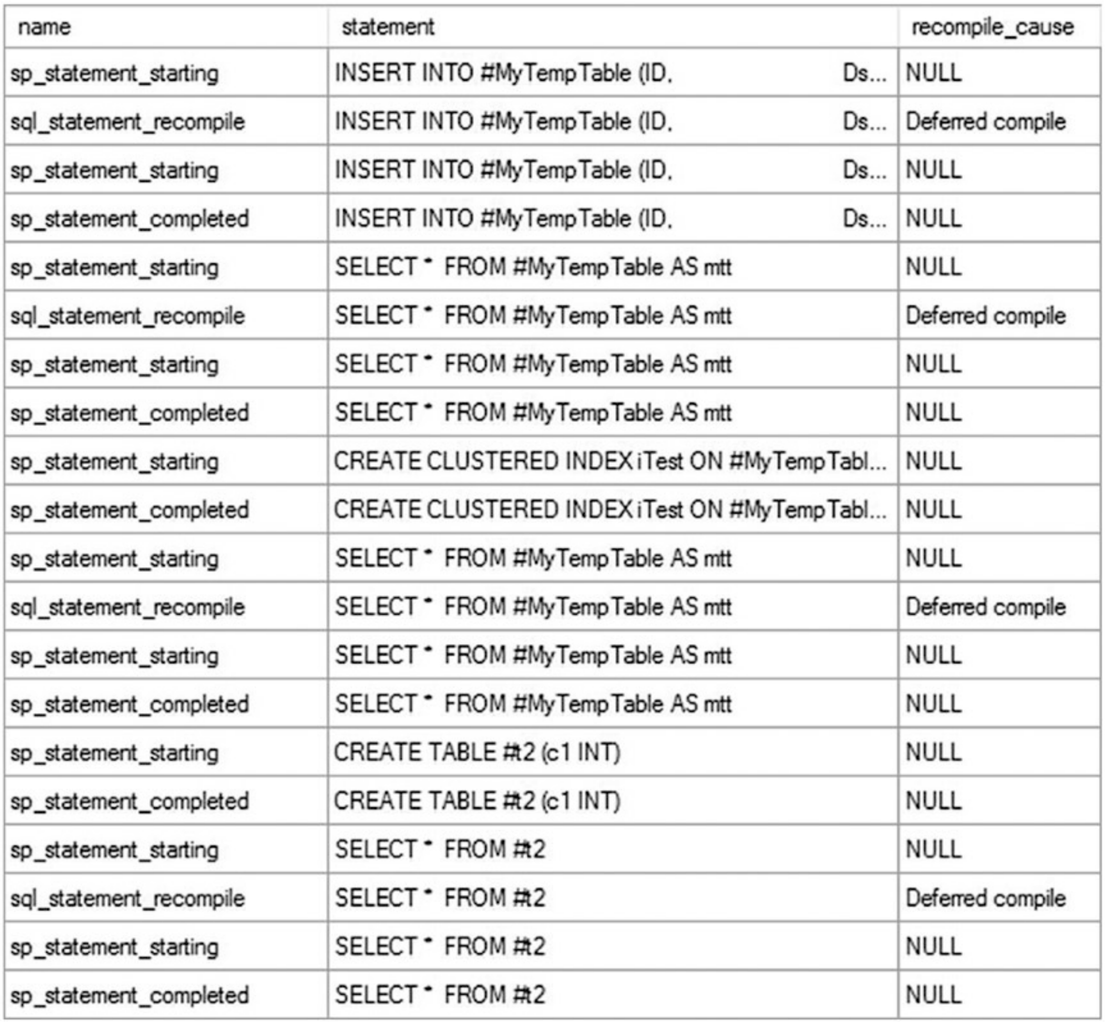
## Don't Interleave DDL and DML Statements

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.

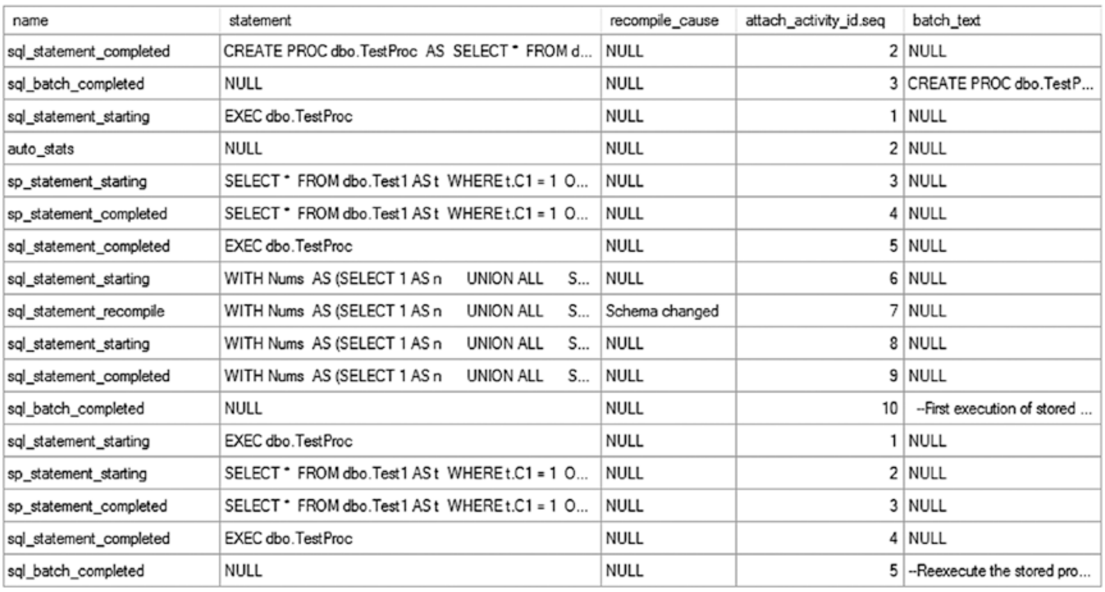
- Use the KEEPFIXED PLAN option.
- Disable the auto update statistics feature on the table.

## 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.

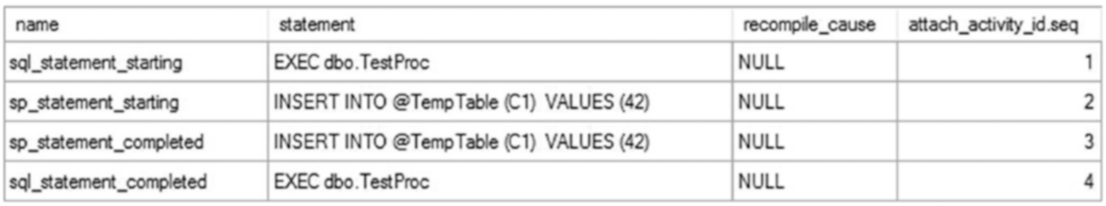
You can avoid the recompilations caused by a temporary table if you use a table variable. Since statistics are not created for table variables, the different recompilation issues associated with temporary tables are not applicable to it. For instance, consider the script used in the section tables are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

However, table variables have their limitations. The main ones are as follows:

- No DDL statement can be executed on the table variable once it is created, which means no indexes or constraints can be added to the table variable later. Constraints can be specified only as part of the table variable's `DECLARE` statement. Therefore, only one index can be created on a table variable, using the `PRIMARY KEY` or `UNIQUE` constraint.
- No statistics are created for table variables, which means they resolve as single-row tables in execution plans. This is not an issue when the table actually contains only a small quantity of data, approximately less than 100 rows. It becomes a major performance problem when the table variable contains more data since appropriate decisions regarding the right sorts of operations within an execution plan are completely dependent on statistics.

## Avoiding Changing SET Options Within a Stored Procedure

It is generally recommended that you not change the environment settings within a stored procedure and thus avoid recompilation because the `SET` options changed. For ANSI compatibility, it is recommended that you keep the following `SET` options `ON`:

- `ARITHABORT`
- `CONCAT_NULL_YIELDS_NULL`
- `QUOTED_IDENTIFIER`
- `ANSI_NULLS`
- `ANSI_PADDINC`
- `ANSI_WARNINGS`
- And `NUMERIC_ROUNDABORT` should be `OFF`.

The earlier example illustrated what happens when you do choose to modify the `SET` options within the procedure.

## Using OPTIMIZE FOR Query Hint

Although you may not always be able to reduce or eliminate recompiles, using the OPTIMIZE FOR query hint can help you get the plan you want when the recompile does occur. The OPTIMIZE FOR query hint uses parameter values supplied by you to compile the plan, regardless of the values of the parameter passed in by the calling application.

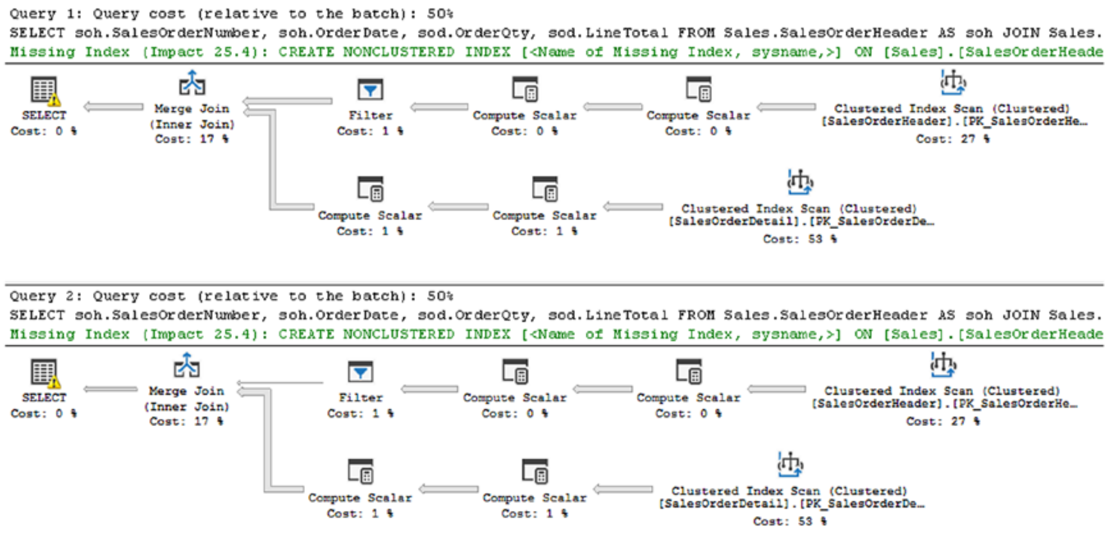
For an example, examine CustomerList from earlier in the chapter. You know that if this procedure receives certain values, it will need to create a new plan. Knowing your data, you also know two more important facts: the frequency that this query will return small data sets is exceedingly small, and when this query uses the wrong plan, performance suffers. Rather than recompiling it over and over again, modify it so that it creates the plan that works best most of the time.

```
CREATE OR ALTER PROCEDURE dbo.CustomerList @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
OPTION (OPTIMIZE FOR (@CustomerID = 1));
GO
```

When this query is executed the first time or is recompiled for any reason, it always gets the same execution plan based on the statistics of the value being passed. To test this, execute the procedure this way:

```
EXEC dbo.CustomerList
    @CustomerID = 7920
    WITH RECOMPILE;
EXEC dbo.CustomerList
    @CustomerID = 30118
    WITH RECOMPILE;
```

Just as earlier in the chapter, this will force the procedure to be recompiled each time it is executed. Figure 18-15 shows the resulting execution plans.



```
Query 1: Query cost (relative to the batch): 50%
SELECT soh.SalesOrderNumber, soh.OrderDate, sod.OrderQty, sod.LineTotal FROM Sales.SalesOrderHeader AS soh JOIN Sales.
Missing Index (Impact 25.4): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Sales].[SalesOrderHeade

Query 2: Query cost (relative to the batch): 50%
SELECT soh.SalesOrderNumber, soh.OrderDate, sod.OrderQty, sod.LineTotal FROM Sales.SalesOrderHeader AS soh JOIN Sales.
Missing Index (Impact 25.4): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Sales].[SalesOrderHeade
```

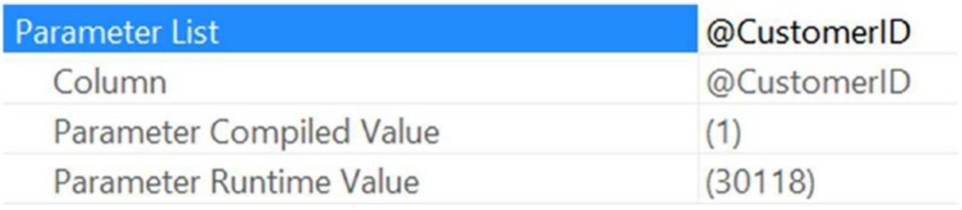
SELECT Cost: 0 %
Merge Join (Inner Join) Cost: 17 %
Filter Cost: 1 %
Compute Scalar Cost: 0 %
Compute Scalar Cost: 0 %
Clustered Index Scan (Clustered) [SalesOrderHeader].[PK_SalesOrderHe… Cost: 27 %
Compute Scalar Cost: 1 %
Compute Scalar Cost: 1 %
Clustered Index Scan (Clustered) [SalesOrderDetail].[PK_SalesOrderDe… Cost: 53 %

***Figure 18-15.*** *WITH RECOMPILE doesn't change identical execution plans*

Unlike earlier in the chapter, recompiling the procedure now doesn't result in a new execution plan. Instead, the same plan is generated, regardless of input, because the query optimizer has received instructions to use the value supplied, `@Customerld = 1`, when optimizing the query.

This doesn't really reduce the number of recompiles, but it does help you control the execution plan generated. It requires that you know your data very well. If your data changes over time, you may need to reexamine areas where the `OPTIMIZE FOR` query hint was used.

To see the hint in the execution plan, just look at the `SELECT` operator properties, as shown in Figure 18-16.



| Parameter List | @CustomerID |
|---|---|
| Column | @CustomerID |
| Parameter Compiled Value | (1) |
| Parameter Runtime Value | (30118) |

***Figure 18-16.*** *The Parameter Compiled Value matches the value supplied by the query hint*

You can see that while the query was recompiled and it was given a value of 30118, because of the hint, the compiled value used was 1 as supplied by the hint.

You can specify that the query be optimized using `OPTIMIZE FOR UNKOWN`. This is almost the opposite of the `OPTIMIZE FOR` hint. The `OPTIMIZE FOR` hint will attempt to use the histogram, while the `OPTIMIZE FOR UNKNOWN` hint will use the density vector of the statistics. What you are directing the processor to do is perform the optimization based on the average of the statistics, always, and to ignore the actual values passed when the query is optimized. You can use it in combination with `OPTIMIZE FOR <value>`. It will optimize for the value supplied on that parameter but will use statistics on all other parameters. As was discussed in the preceding chapter, these are both mechanisms for dealing with bad parameter sniffing.

## Using Plan Guides

A plan guide allows you to use query hints or other optimization techniques without having to modify the query or procedure text. This is especially useful when you have a third-party product with poorly performing procedures you need to tune but can't modify. As part of the optimization process, if a plan guide exists when a procedure is compiled or recompiled, it will use that guide to create the execution plan.

In the previous section, I showed you how using `OPTIMIZE FOR` would affect the execution plan created on a procedure. The following is the query from the original procedure, with no hints:

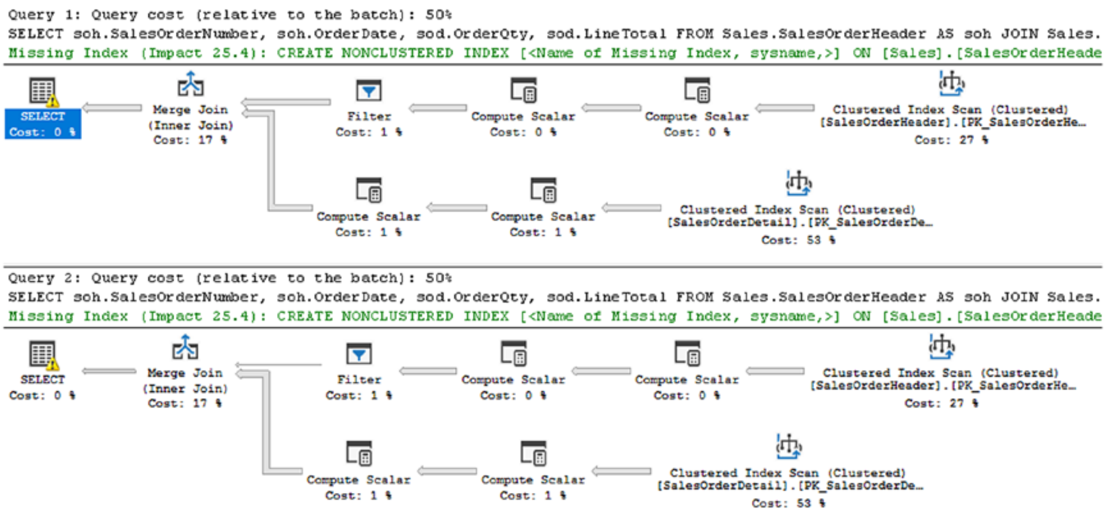
```
CREATE OR ALTER PROCEDURE dbo.CustomerList @CustomerID INT
AS
SELECT soh.SalesOrderNumber,
       soh.OrderDate,
       sod.OrderQty,
       sod.LineTotal
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE soh.CustomerID >= @CustomerID;
GO
```

Now assume for a moment that this query is part of a third-party application and you are not able to modify it to include OPTION (OPTIMIZE FOR). To provide it with the query hint, OPTIMIZE FOR, create a plan guide as follows:

```
sp_create_plan_guide @name = N'MyGuide',
                     @stmt = 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;',
                     @type = N'OBJECT',
                     @module_or_batch = N'dbo.CustomerList',
                     @params = NULL,
                     @hints = N'OPTION (OPTIMIZE FOR (@CustomerID = 1))';
```

Now, when the procedure is executed with each of the different parameters, even with the RECOMPILE being forced as shown next, the OPTIMIZE FOR hint is applied. Figure 18-17 shows the resulting execution plan.

```
EXEC dbo.CustomerList
    @CustomerID = 7920
    WITH RECOMPILE;
EXEC dbo.CustomerList
    @CustomerID = 30118
    WITH RECOMPILE;
```

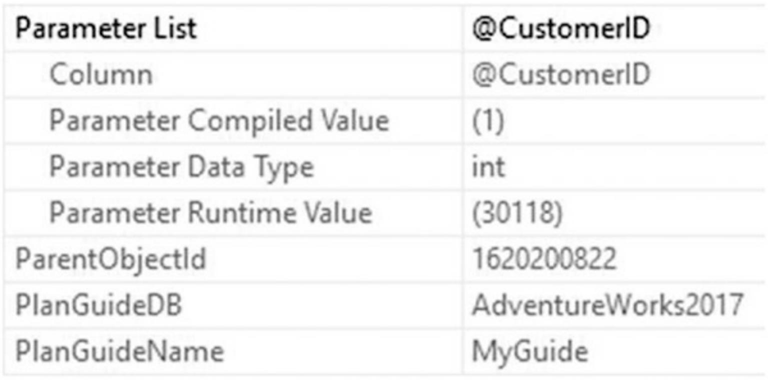


```
Query 1: Query cost (relative to the batch): 50%
SELECT soh.SalesOrderNumber, soh.OrderDate, sod.OrderQty, sod.LineTotal FROM Sales.SalesOrderHeader AS soh JOIN Sales.
Missing Index (Impact 25.4): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Sales].[SalesOrderHeade
```

```
Query 2: Query cost (relative to the batch): 50%
SELECT soh.SalesOrderNumber, soh.OrderDate, sod.OrderQty, sod.LineTotal FROM Sales.SalesOrderHeader AS soh JOIN Sales.
Missing Index (Impact 25.4): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Sales].[SalesOrderHeade
```

***Figure 18-17.*** *Using a plan guide to apply the OPTIMIZE FOR query hint*

The results are the same as when the procedure was modified, but in this case, no modification was necessary. You can see that a plan guide was applied within the execution plan by looking at the `SELECT` properties again (Figure 18-18).



| Parameter List | @CustomerID |
|---|---|
| Column | @CustomerID |
| Parameter Compiled Value | (1) |
| Parameter Data Type | int |
| Parameter Runtime Value | (30118) |
| ParentObjectId | 1620200822 |
| PlanGuideDB | AdventureWorks2017 |
| PlanGuideName | MyGuide |

***Figure 18-18.*** *SELECT operator properties show the plan guide*

Various types of plan guides exist. The previous example is an *object* plan guide, which is a guide matched to a particular object in the database, in this case `CustomerList`. You can also create plan guides for ad hoc queries that come into your system repeatedly by creating a SQL plan guide that looks for particular SQL statements.

Instead of a procedure, the following query gets passed to your system and needs an OPTIMIZE FOR query hint:

```
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 >= 1;
```

Running this query results in the execution plan you see in Figure 18-19.

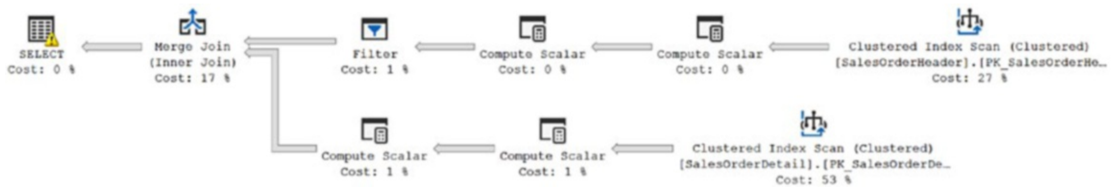


***Figure 18-19.*** *The query uses a different execution plan from the one wanted*

To get a query plan guide, you first need to know the precise format used by the query in case parameterization, forced or simple, changes the text of the query. The text has to be precise. If your first attempt at a query plan guide looked like this:

```
EXECUTE sp_create_plan_guide @name = N'MyBadSQLGuide',
                             @stmt = 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',
                             @type = N'SQL',
                             @module_or_batch = NULL,
                             @params = N'@CustomerID int',
                             @hints = N'OPTION  (TABLE
                             HINT(soh,  FORCESEEK))';
```

you'll still get the same execution plan when running the select query. This is because the query doesn't look like what was typed in for the plan guide. Several things are different, such as the spacing and the case on the JOIN statement. You can drop this bad plan guide using the T-SQL statement.

```
EXECUTE sp_control_plan_guide @operation = 'Drop',
                              @name = N'MyBadSQLGuide';
```

Inputting the correct syntax will create a new plan.

```
EXECUTE sp_create_plan_guide @name = N'MyGoodSQLGuide',
                             @stmt = 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 >= 1;',
                             @type = N'SQL',
                             @module_or_batch = NULL,
                             @params = NULL,
                             @hints = N'OPTION  (TABLE
                             HINT(soh,  FORCESEEK))';
```

Now when the query is run, a completely different plan is created, as shown in Figure 18-20.

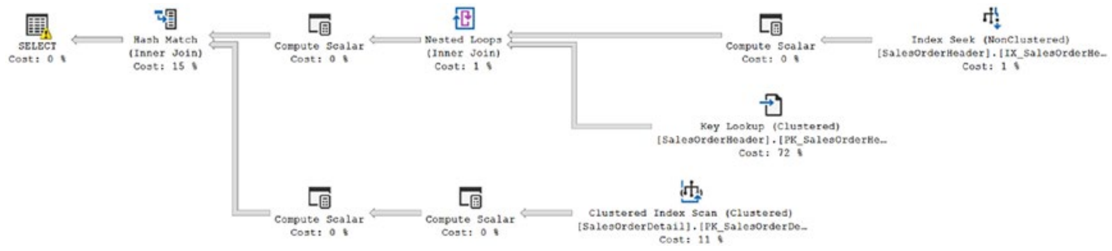


***Figure 18-20.*** *The plan guide forces a new execution plan on the same query*

One other option exists when you have a plan in the cache that you think performs the way you want. You can capture that plan into a plan guide to ensure that the next time the query is run, the same plan is executed. You accomplish this by running `sp_create_plan_guide_from_handle`.

To test it, first clear the procedure cache (assuming we're not running on a production instance) so you can control exactly which query plan is used.

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

With the procedure cache clear and the existing plan guide, `MyGoodSOQLGuide`, in place, rerun the query. It will use the plan guide to arrive at the execution plan displayed in Figure 18-18. To see whether this plan can be kept, first drop the plan guide that is forcing the `Index Seek` operation.

```
EXECUTE sp_control_plan_guide @operation = 'Drop',
                              @name = N'MyGoodSQLGuide';
```

If you were to rerun the query now, it would revert to its original plan. However, right now in the plan cache, you have the plan displayed in Figure 18-18. To keep it, run the following script:

```
DECLARE @plan_handle VARBINARY(64),
        @start_offset INT;

SELECT @plan_handle = deqs.plan_handle,
       @start_offset = deqs.statement_start_offset
FROM sys.dm_exec_query_stats AS deqs
    CROSS APPLY sys.dm_exec_sql_text(sql_handle)
    CROSS APPLY sys.dm_exec_text_query_plan(deqs.plan_handle,
                                            deqs.statement_start_offset,
                                         deqs.statement_end_offset) AS qp
WHERE text LIKE N'SELECT soh.SalesOrderNumber%'

EXECUTE sp_create_plan_guide_from_handle @name = N'ForcedPlanGuide',
                                         @plan_handle = @plan_handle,
                                         @statement_start_offset = @start_
offset;
GO
```

This creates a plan guide based on the execution plan as it currently exists in the cache. To be sure this works, clear the cache again. That way, the query has to generate a new plan. Rerun the query, and observe the execution plan. It will be the same as that displayed in Figure 18-19 because of the plan guide created using `sp_create_plan_guide_from_handle`.

Plan guides are useful mechanisms for controlling the behavior of SQL queries and stored procedures, but you should use them only when you have a thorough understanding of the execution plan, the data, and the structure of your system.

## Use Query Store to Force a Plan

Just as with the `OPTIMIZE FOR` and plan guides, forcing a plan through the Query Store won't actually reduce the number of recompiles on the system. It will however allow you to better control the results of those recompiles. Similar to how plan guides work, as your data changes over time, you may need to reassess the plans you have forced (see Chapter 11).

## Summary

As you learned in this chapter, query recompilation can both benefit and hurt performance. Recompilations that generate better plans improve the performance of the stored procedure. However, recompilations that regenerate the same plan consume extra CPU cycles without any improvement in processing strategy. Therefore, you should look closely at recompilations to determine their usefulness. You can use Extended Events to identify which stored procedure statement caused the recompilation, and you can determine the cause from the `recompile_clause` data column value in the Extended Events output. Once you determine the cause of the recompilation, you can apply different techniques to avoid the unnecessary recompilations.

Up until now, you have seen how to benefit from proper indexing and plan caching. However, the performance benefit of these techniques depends on the way the queries are designed. The cost-based optimizer of SQL Server takes care of many of the query design issues. However, you should adopt a number of best practices while designing queries. In the next chapter, I will cover some of the common query design issues that affect performance.

CHAPTER 19

# Query Design Analysis

A database schema may include a number of performance-enhancement features such as indexes, statistics, and stored procedures. But none of these features guarantees good performance if your queries are written badly in the first place. The SQL queries may not be able to use the available indexes effectively. The structure of the SQL queries may add avoidable overhead to the query cost. Queries may be attempting to deal with data in a row-by-row fashion (or to quote Jeff Moden, Row By Agonizing Row, which is abbreviated to RBAR and pronounced "reebar") instead of in logical sets. To improve the performance of a database application, it is important to understand the cost associated with varying ways of writing a query.

In this chapter, I cover the following topics:

- Aspects of query design that affect performance
- How query designs use indexes effectively
- The role of optimizer hints on query performance
- The role of database constraints on query performance

## Query Design Recommendations

When you need to run a query, you can often use many different approaches to get the same data. In many cases, the optimizer generates the same plan, irrespective of the structure of the query. However, in some situations the query structure won't allow the optimizer to select the best possible processing strategy. It is important that you are aware that this can happen and, should it occur, what you can do to avoid it.

In general, keep the following recommendations in mind to ensure the best performance:

- Operate on small result sets.
- Use indexes effectively.
- Minimize the use of optimizer hints.
- Use domain and referential integrity.
- Avoid resource-intensive queries.
- Reduce the number of network round-trips.
- Reduce the transaction cost. (I'll cover the last three in the next chapter.)

Careful testing is essential to identify the query form that provides the best performance in a specific database environment. You should be conversant with writing and comparing different SQL query forms so you can evaluate the query form that provides the best performance in a given environment. You'll also want to be able to automate your testing.

## Operating on Small Result Sets

To improve the performance of a query, limit the amount of data it operates on, including both columns and rows. Operating on a small result set reduces the amount of resources consumed by a query and increases the effectiveness of indexes. Two of the rules you should follow to limit the data set's size are as follows:

- Limit the number of columns in the select list to what is actually needed.
- Use highly selective `WHERE` clauses to limit the rows returned.

It's important to note that you will be asked to return tens of thousands of rows to an OLTP system. Just because someone tells you those are the business requirements doesn't mean they are right. Human beings don't process tens of thousands of rows. Few human beings are capable of processing thousands of rows. Be prepared to push back on these requests and be able to justify your reasons. Also, one of the reasons you'll frequently hear and have to be ready to push back is "just in case we need it in the future."

## Limit the Number of Columns in select_list

Use a minimum set of columns in the select list of a SELECT statement. Don't use columns that are not required in the output result set. For instance, don't use SELECT * to return all columns. SELECT * statements render covered indexes ineffective since it is usually impractical to include all columns in an index. For example, consider the following query:

```
SELECT Name,
       TerritoryID
FROM Sales.SalesTerritory AS st
WHERE st.Name = 'Australia';
```

A covering index on the Name column (and through the clustered key, ProductID) serves the query quickly through the index itself, without accessing the clustered index. When you have an Extended Events session switched on, you get the following number of logical reads and execution time, as well as the corresponding execution plan (shown in Figure 19-1):

```
Reads: 2
Duration: 920 mcs
```

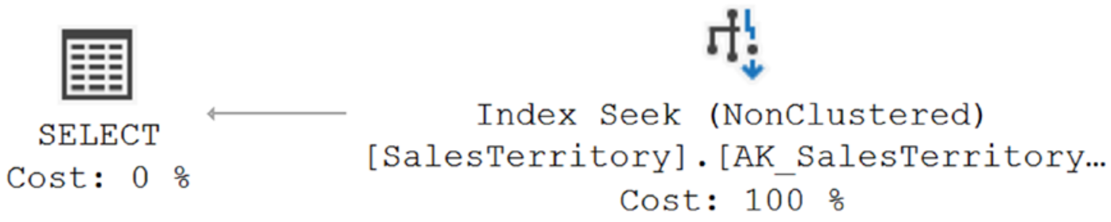

SELECT
Cost: 0 %
Index Seek (NonClustered)
[SalesTerritory].[AK_SalesTerritory…
Cost: 100 %

***Figure 19-1.*** *Execution plan showing the benefit of referring to a limited number of columns*

If this query is modified to include all columns in the select list as follows, then the previous covering index becomes ineffective because all the columns required by this query are not included in that index:

```
SELECT *
FROM Sales.SalesTerritory AS st
WHERE st.Name = 'Australia';
```

Subsequently, the base table (or the clustered index) containing all the columns has to be accessed, as shown next. The number of logical reads and the execution time have both increased.

```
Table 'SalesTerritory'. Scan count 0, logical reads 4
CPU time = 0 ms,   elapsed time = 6.4 ms
```

The fewer the columns in the select list, the better the potential for improved query performance. And remember, the query we've been looking at is a simple query returning a single, small row of data, and it has doubled the number of reads and resulted in six times the execution time. Selecting more columns than are strictly needed also increases data transfer across the network, further degrading performance. Figure 19-2 shows the execution plan.

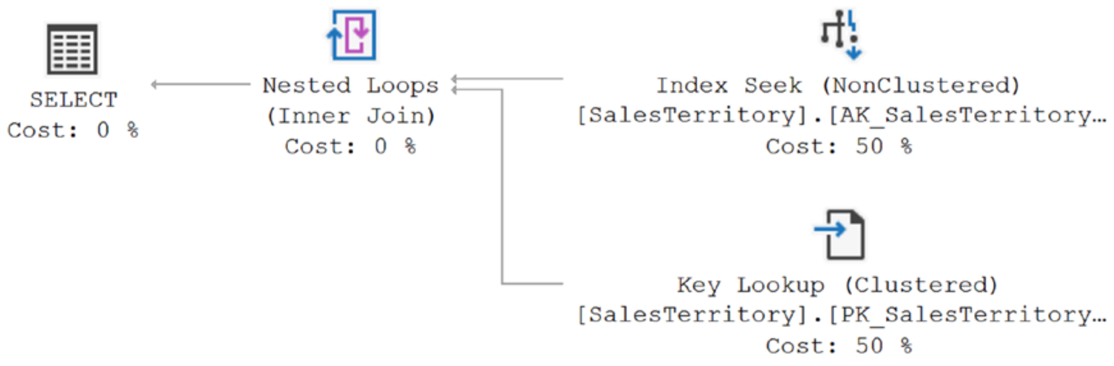

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[SalesTerritory].[AK_SalesTerritory…
Cost: 50 %
Key Lookup (Clustered)
[SalesTerritory].[PK_SalesTerritory…
Cost: 50 %

***Figure 19-2.*** *Execution plan illustrating the added cost of referring to too many columns*

## Use Highly Selective WHERE Clauses

As explained in Chapter 8, the selectivity of a column referred to in the `WHERE`, `ON`, and `HAVING` clauses governs the use of an index on the column. A request for a large number of rows from a table may not benefit from using an index, either because it can't use an index at all or, in the case of a nonclustered index, because of the overhead cost of the lookup operation. To ensure the use of indexes, the columns referred to in the `WHERE` clause should be highly selective.

Most of the time, an end user concentrates on a limited number of rows at a time. Therefore, you should design database applications to request data incrementally as the user navigates through the data. For applications that rely on a large amount of data for data analysis or reporting, consider using data analysis solutions such as Analysis Services or PowerPivot. If the analysis is around aggregation and involves a larger amount of data, put the columnstore index to use. Remember, returning huge result sets is costly, and this data is unlikely to be used in its entirety. The only common exception to this is when working with data scientists who frequently have to retrieve all data from a given data set as the first step of their operations. In this case alone, you may need to find other methods for improving performance such as a secondary server, improved hardware, or other mechanisms. However, work with them to ensure they move the data only once, not repeatedly.

## Using Indexes Effectively

It is extremely important to have effective indexes on database tables to improve performance. However, it is equally important to ensure that the queries are designed properly to use these indexes effectively. These are some of the query design rules you should follow to improve the use of indexes:

- Avoid nonsargable search conditions.
- Avoid arithmetic operators on the WHERE clause column.
- Avoid functions on the WHERE clause column.

I cover each of these rules in detail in the following sections.

### Avoid Nonsargable Search Conditions

A *sargable* predicate in a query is one in which an index can be used. The word is a contraction of "Search ARGument ABLE." The optimizer's ability to benefit from an index depends on the selectivity of the search condition, which in turn depends on the selectivity of the column(s) referred to in the WHERE, ON, and HAVING clauses, all of which are referred to the statistics on the index. The search predicate used on the columns in the WHERE clause determines whether an index operation on the column can be performed.

**Note** The use of indexes and other functions around the filtering clauses are primarily concerned with WHERE, ON, and HAVING. To make things a little easier to read (and write), I'm going to just use WHERE in a lot of cases in which ON and HAVING should be included. Unless otherwise noted, just include them mentally if you don't see them.

The sargable search conditions listed in Table 19-1 generally allow the optimizer to use an index on the columns referred to in the WHERE clause. The sargable search conditions generally allow SQL Server to seek to a row in the index and retrieve the row (or the adjacent range of rows while the search condition remains true).

***Table 19-1.*** *Common Sargable and Nonsargable Search Conditions*

| Type | Search Conditions |
|---|---|
| Sargable | Inclusion conditions =, >, >=, <, <=, and BETWEEN, and some LIKE conditions such as LIKE '<literal>%' |
| Nonsargable | Exclusion conditions <>, !=, !>, !<, NOT EXISTS, NOT IN, and NOT LIKE IN, OR, and some LIKE conditions such as LIKE '%<literal>' |

On the other hand, the *nonsargable* search conditions listed in Table 19-1 generally prevent the optimizer from using an index on the columns referred to in the WHERE clause. The exclusion search conditions generally don't allow SQL Server to perform Index Seek operations as supported by the sargable search conditions. For example, the != condition requires scanning all the rows to identify the matching rows.

Try to implement workarounds for these nonsargable search conditions to improve performance. In some cases, it may be possible to rewrite a query to avoid a nonsargable search condition. For example, in some cases an OR condition can be replaced by two (or more) UNION ALL queries, with multiple seek operations outperforming a single scan. You can also consider replacing an IN/OR search condition with a BETWEEN condition, as described in the following section. The trick is to experiment with the different mechanisms to see whether one will work better in a given situation than another. No single method within SQL Server is horrible, and no single method is perfect. Everything has a time and a place where you will need to use a given function. Be flexible and experiment when you're attempting to improve performance.

## BETWEEN vs. IN/OR

Consider the following query, which uses the search condition IN:

```
SELECT sod.*
FROM Sales.SalesOrderDetail AS sod
WHERE sod.SalesOrderID IN ( 51825, 51826, 51827, 51828 );
```

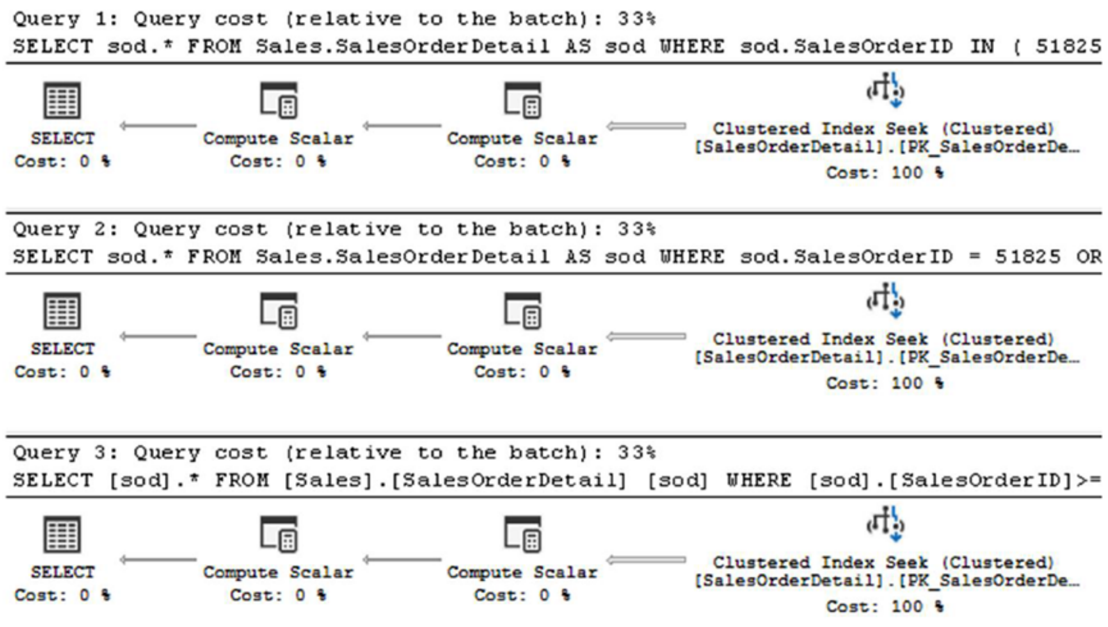
Another way to write the same query is to use the OR command.

```
SELECT sod.*
FROM Sales.SalesOrderDetail AS sod
WHERE sod.SalesOrderID = 51825
      OR sod.SalesOrderID = 51826
      OR sod.SalesOrderID = 51827
      OR sod.SalesOrderID = 51828;
```

You can replace either of these search condition in this query with a BETWEEN clause as follows:

```
SELECT sod.*
FROM Sales.SalesOrderDetail AS sod
WHERE sod.SalesOrderID BETWEEN 51825
                       AND     51828;
```

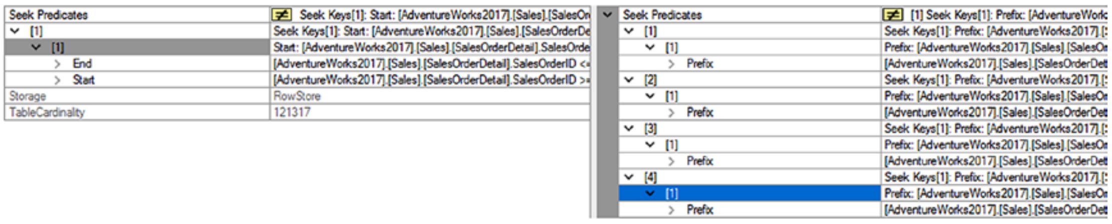
All three queries return the same results. On the face of it, the execution plan of all three queries appear to be the same, as shown in Figure 19-3.



```
Query 1: Query cost (relative to the batch): 33%
SELECT sod.* FROM Sales.SalesOrderDetail AS sod WHERE sod.SalesOrderID IN ( 51825

SELECT      Compute Scalar    Compute Scalar    Clustered Index Seek (Clustered)
Cost: 0 %   Cost: 0 %         Cost: 0 %         [SalesOrderDetail].[PK_SalesOrderDe…
                                                Cost: 100 %

Query 2: Query cost (relative to the batch): 33%
SELECT sod.* FROM Sales.SalesOrderDetail AS sod WHERE sod.SalesOrderID = 51825 OR

SELECT      Compute Scalar    Compute Scalar    Clustered Index Seek (Clustered)
Cost: 0 %   Cost: 0 %         Cost: 0 %         [SalesOrderDetail].[PK_SalesOrderDe…
                                                Cost: 100 %

Query 3: Query cost (relative to the batch): 33%
SELECT [sod].* FROM [Sales].[SalesOrderDetail] [sod] WHERE [sod].[SalesOrderID]>=

SELECT      Compute Scalar    Compute Scalar    Clustered Index Seek (Clustered)
Cost: 0 %   Cost: 0 %         Cost: 0 %         [SalesOrderDetail].[PK_SalesOrderDe…
                                                Cost: 100 %
```

***Figure 19-3.*** *Execution plan for a simple SELECT statement using a BETWEEN clause*

However, a closer look at the execution plans reveals the difference in their data-retrieval mechanism, as shown in Figure 19-4.



| Seek Predicates | Seek Keys[1]: Start: [AdventureWorks2017].[Sales].[SalesOr |
|---|---|
| [1] | Seek Keys[1]: Start: [AdventureWorks2017].[Sales].[SalesOrderDe |
| [1] | Start: [AdventureWorks2017].[Sales].[SalesOrderDetail].SalesOrde |
| End | [AdventureWorks2017].[Sales].[SalesOrderDetail].SalesOrderID <= |
| Start | [AdventureWorks2017].[Sales].[SalesOrderDetail].SalesOrderID >= |
| Storage | RowStore |
| TableCardinality | 121317 |

| Seek Predicates | [1] Seek Keys[1]: Prefix: [AdventureWork |
|---|---|
| [1] | Seek Keys[1]: Prefix: [AdventureWorks2017].[ |
| [1] | Prefix: [AdventureWorks2017].[Sales].[SalesOr |
| Prefix | [AdventureWorks2017].[Sales].[SalesOrderDet |
| [2] | Seek Keys[1]: Prefix: [AdventureWorks2017].[ |
| [1] | Prefix: [AdventureWorks2017].[Sales].[SalesOr |
| Prefix | [AdventureWorks2017].[Sales].[SalesOrderDet |
| [3] | Seek Keys[1]: Prefix: [AdventureWorks2017].[ |
| [1] | Prefix: [AdventureWorks2017].[Sales].[SalesOr |
| Prefix | [AdventureWorks2017].[Sales].[SalesOrderDet |
| [4] | Seek Keys[1]: Prefix: [AdventureWorks2017].[ |
| [1] | Prefix: [AdventureWorks2017].[Sales].[SalesOr |
| Prefix | [AdventureWorks2017].[Sales].[SalesOrderDet |

***Figure 19-4.*** *Execution plan details for a BETWEEN condition (left) and an IN condition (right)*

As shown in Figure 19-4, SQL Server resolved the `IN` condition containing four values into four `OR` conditions. Accordingly, the clustered index (`PKSalesTerritoryTerritoryld`) is accessed four times (`Scan count 4`) to retrieve rows for the four `IN` and `OR` conditions, as shown in the following corresponding `STATISTICS 10` output. On the other hand, the `BETWEEN` condition is resolved into a pair of `>=` and `<=` conditions, as shown in Figure 19-4. SQL Server accesses the clustered index only once

(Scan count 1) from the first matching row until the match condition is true, as shown in the following corresponding STATISTICS 10 and QUERY TIME output.

- With the IN condition:

```
Table  'SalesOrderDetail'.  Scan count 4,  logical reads 18
CPU time = 0 ms,    elapsed time = 140 ms.
```

- With the BETWEEN condition:

```
Table  'SalesOrderDetail'.  Scan count 1,  logical reads 6
CPU time = 0 ms,    elapsed time = 72 ms.
```

Replacing the search condition IN with BETWEEN decreases the number of logical reads for this query from 18 to 6. As just shown, although all three queries use a clustered index seek on OrderID, the optimizer locates the range of rows much faster with the BETWEEN clause than with the IN clause. The same thing happens when you look at the BETWEEN condition and the OR clause. Therefore, if there is a choice between using IN/OR and the BETWEEN search condition, always choose the BETWEEN condition because it is generally much more efficient than the IN/OR condition. In fact, you should go one step further and use the combination of >= and <= instead of the BETWEEN clause only because you're making the optimizer do a little less work.

Also worth noting is that this query violates the earlier suggestion to return only a limited set of columns rather than using SELECT *. If you look to the properties of the BETWEEN operation, it was also changed to a parameterized query with simple parameterization. That can lead to plan reuse as discussed in Chapter 18.

Not every WHERE clause that uses exclusion search conditions prevents the optimizer from using the index on the column referred to in the search condition. In many cases, the SQL Server optimizer does a wonderful job of converting the exclusion search condition to a sargable search condition. To understand this, consider the following two search conditions, which I discuss in the sections that follow:

- The LIKE condition
- The !< condition versus the >= condition

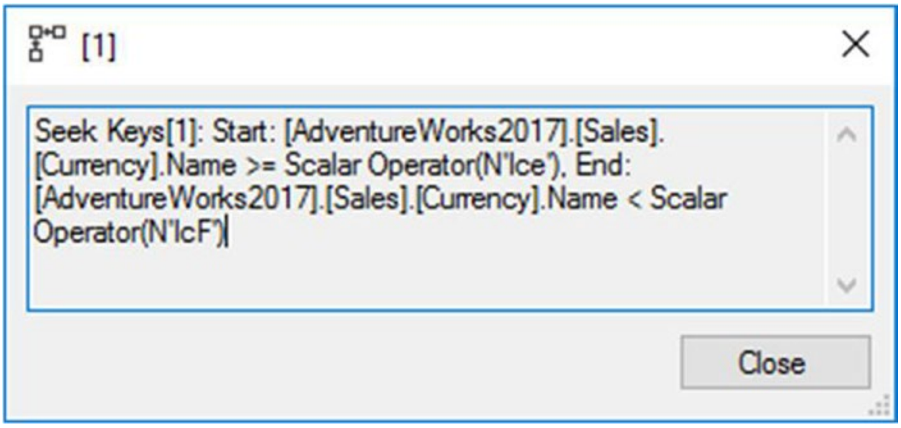
## LIKE Condition

While using the LIKE search condition, try to use one or more leading characters in the WHERE clause if possible. Using leading characters in the LIKE clause allows the optimizer to convert the LIKE condition to an index-friendly search condition. The greater the number of leading characters in the LIKE condition, the better the optimizer is able to determine an effective index. Be aware that using a wildcard character as the leading character in the LIKE condition *prevents* the optimizer from performing a SEEK (or a narrow-range scan) on the index; it relies on scanning the complete table instead.

To understand this ability of the SQL Server optimizer, consider the following SELECT statement that uses the LIKE condition with a leading character:

```
SELECT c.CurrencyCode
FROM Sales.Currency AS c
WHERE c.Name LIKE 'Ice%';
```

The SQL Server optimizer does this conversion automatically, as shown in Figure 19-5.



Seek Keys[1]: Start: [AdventureWorks2017].[Sales].[Currency].Name >= Scalar Operator(N'Ice'), End: [AdventureWorks2017].[Sales].[Currency].Name < Scalar Operator(N'IcF')

***Figure 19-5.*** *Execution plan showing automatic conversion of a LIKE clause with a trailing % sign to an indexable search condition*

As you can see, the optimizer automatically converts the LIKE condition to an equivalent pair of >= and < conditions. You can therefore rewrite this SELECT statement to replace the LIKE condition with an indexable search condition as follows:

```
SELECT c.CurrencyCode
FROM Sales.Currency AS c
WHERE c.Name >= N'Ice'
      AND c.Name < N'IcF';
```

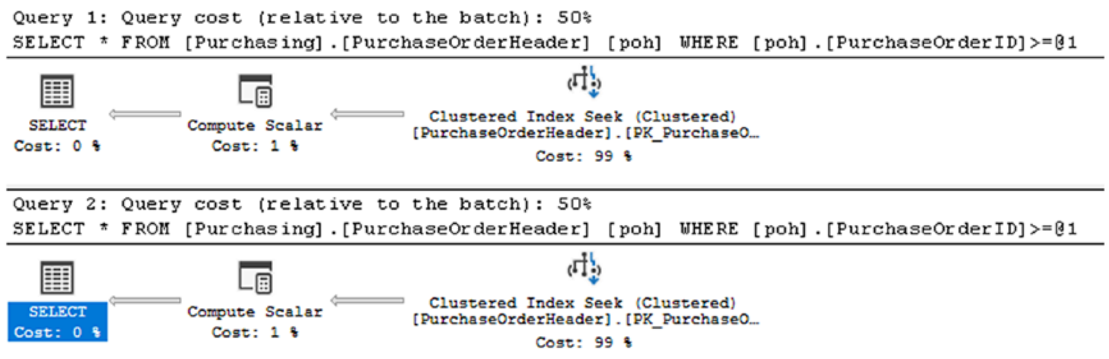
Note that, in both cases, the number of logical reads, the execution time for the query with the LIKE condition, and the manually converted sargable search condition are all the same. Thus, if you include leading characters in the LIKE clause, the SQL Server optimizer optimizes the search condition to allow the use of indexes on the column.

## !< Condition vs. >= Condition

Even though both the !< and >= search conditions retrieve the same result set, they may perform different operations internally. The >= comparison operator allows the optimizer to use an index on the column referred to in the search argument because the = part of the operator allows the optimizer to seek to a starting point in the index and access all the index rows from there onward. On the other hand, the !< operator doesn't have an = element and needs to access the column value for every row.

Or does it? As explained in Chapter 15, the SQL Server optimizer performs syntax-based optimization, before executing a query, to improve performance. This allows SQL Server to take care of the performance concern with the !< operator by converting it to >=, as shown in the execution plan in Figure 19-6 for the two following SELECT statements:

```
SELECT *
FROM Purchasing.PurchaseOrderHeader AS poh
WHERE poh.PurchaseOrderID >= 2975;
SELECT *
FROM Purchasing.PurchaseOrderHeader AS poh
WHERE poh.PurchaseOrderID !< 2975;
```



```
Query 1: Query cost (relative to the batch): 50%
SELECT * FROM [Purchasing].[PurchaseOrderHeader] [poh] WHERE [poh].[PurchaseOrderID]>=@1
```

SELECT
Cost: 0 %
Compute Scalar
Cost: 1 %
Clustered Index Seek (Clustered)
[PurchaseOrderHeader].[PK_PurchaseO...
Cost: 99 %

```
Query 2: Query cost (relative to the batch): 50%
SELECT * FROM [Purchasing].[PurchaseOrderHeader] [poh] WHERE [poh].[PurchaseOrderID]>=@1
```

SELECT
Cost: 0 %
Compute Scalar
Cost: 1 %
Clustered Index Seek (Clustered)
[PurchaseOrderHeader].[PK_PurchaseO...
Cost: 99 %

***Figure 19-6.*** *Execution plan showing automatic transformation of a nonindexable !< operator to an indexable >= operator*

As you can see, the optimizer often provides you with the flexibility of writing queries in the preferred T-SQL syntax without sacrificing performance.

Although the SQL Server optimizer can automatically optimize query syntax to improve performance in many cases, you should not rely on it to do so. It is a good practice to write efficient queries in the first place.

## Avoid Arithmetic Operators on the WHERE Clause Column

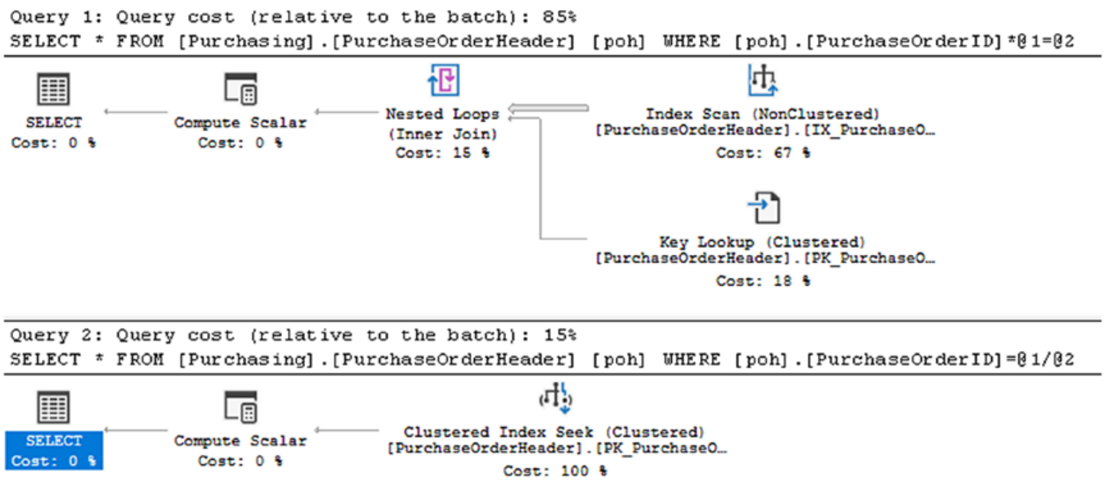
Using an arithmetic operator on a column in the `WHERE` clause can prevent the optimizer from using the statistics or the index on the column. For example, consider the following `SELECT` statement:

```
SELECT *
FROM Purchasing.PurchaseOrderHeader AS poh
WHERE poh.PurchaseOrderID * 2 = 3400;
```

A multiplication operator, `*`, has been applied on the column in the `WHERE` clause. You can avoid this on the column by rewriting the `SELECT` statement as follows:

```
SELECT *
FROM Purchasing.PurchaseOrderHeader AS poh
WHERE poh.PurchaseOrderID = 3400 / 2;
```

The table has a clustered index on the `PurchaseOrderID` column. As explained in Chapter 4, an `Index Seek` operation on this index is suitable for this query since it returns only one row. Even though both queries return the same result set, the use of the multiplication operator on the `PurchaseOrderID` column in the first query prevents the optimizer from using the index on the column, as you can see in Figure 19-7.



```
Query 1: Query cost (relative to the batch): 85%
SELECT * FROM [Purchasing].[PurchaseOrderHeader] [poh] WHERE [poh].[PurchaseOrderID]*@1=@2
```

SELECT
Cost: 0 %
Compute Scalar
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 15 %
Index Scan (NonClustered)
[PurchaseOrderHeader].[IX_PurchaseO...
Cost: 67 %
Key Lookup (Clustered)
[PurchaseOrderHeader].[PK_PurchaseO...
Cost: 18 %

```
Query 2: Query cost (relative to the batch): 15%
SELECT * FROM [Purchasing].[PurchaseOrderHeader] [poh] WHERE [poh].[PurchaseOrderID]=@1/@2
```

SELECT
Cost: 0 %
Compute Scalar
Cost: 0 %
Clustered Index Seek (Clustered)
[PurchaseOrderHeader].[PK_PurchaseO...
Cost: 100 %

***Figure 19-7.*** *Execution plan showing the detrimental effect of an arithmetic operator on a WHERE clause column*

The following are the corresponding performance metrics:

- With the * operator on the `PurchaseOrderID` column:

  Reads: 11
  Duration: 210mcs

- With no operator on the `PurchaseOrderID` column:

  Reads: 2
  Duration: 105mcs

Therefore, to use the indexes effectively and improve query performance, avoid using arithmetic operators on columns in the `WHERE` clause or `JOIN` criteria when that expression is expected to work with an index.

Worth noting in the queries shown in Figure 19-7 is that both queries were simple enough to qualify for parameterization as indicated by the `@1` and `@2` in the queries instead of the values supplied.

**Note** For small result sets, even though an index seek is usually a better data-retrieval strategy than a table scan (or a complete clustered index scan), for small tables (in which all data rows fit on one page) a table scan can be cheaper. I explain this in more detail in Chapter 8.

## Avoid Functions on the WHERE Clause Column

In the same way as arithmetic operators, functions on `WHERE` clause columns also hurt query performance—and for the same reasons. Try to avoid using functions on `WHERE` clause columns, as shown in the following two examples:

- `SUBSTRING` vs. `LIKE`
- Date part comparison
- Custom scalar user-defined function

### SUBSTRING vs. LIKE

In the following `SELECT` statement, using the `SUBSTRING` function prevents the use of the index on the `ShipPostalCode` column:

```
SELECT d.Name
FROM HumanResources.Department AS d
WHERE SUBSTRING(d.Name,
                1,
                1) = 'F';
```

Figure 19-8 illustrates this.

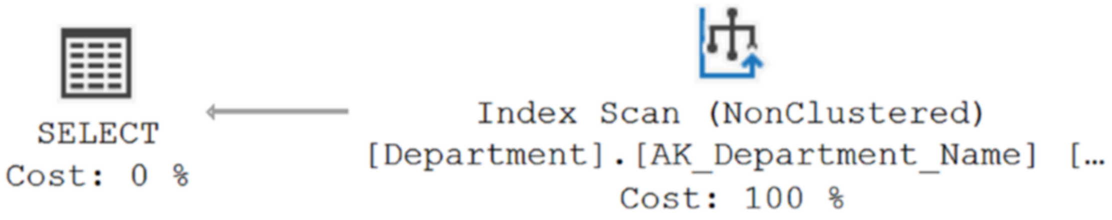

SELECT
Cost: 0 %
Index Scan (NonClustered)
[Department].[AK_Department_Name] [...
Cost: 100 %

***Figure 19-8.*** *Execution plan showing the detrimental effect of using the SUBSTRING function on a WHERE clause column*

As you can see, using the SUBSTRING function prevented the optimizer from using the index on the [Name] column. This function on the column made the optimizer use a clustered index scan. In the absence of the clustered index on the DepartmentID column, a table scan would have been performed.

You can redesign this SELECT statement to avoid the function on the column as follows:

```
SELECT d.Name
FROM HumanResources.Department AS d
WHERE d.Name LIKE 'F%';
```

This query allows the optimizer to choose the index on the [Name] column, as shown in Figure 19-9.

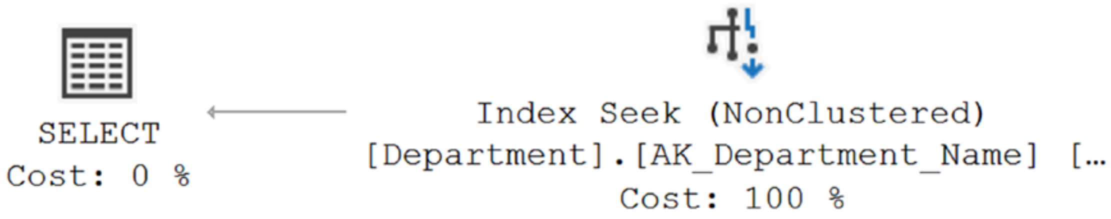

SELECT
Cost: 0 %
Index Seek (NonClustered)
[Department].[AK_Department_Name] [...
Cost: 100 %

***Figure 19-9.*** *Execution plan showing the benefit of not using the SUBSTRING function on a WHERE clause column*

## Date Part Comparison

SQL Server can store date and time data as separate fields or as a combined DATETIME field that has both. Although you may need to keep date and time data together in one field, sometimes you want only the date, which usually means you have to apply a conversion function to extract the date part from the DATETIME data type. Doing this prevents the optimizer from choosing the index on the column, as shown in the following example.

First, there needs to be a good index on the DATETIME column of one of the tables. Use Sales.SalesOrderHeader and create the following index:

```
IF EXISTS (   SELECT *
              FROM sys.indexes
              WHERE object_id = OBJECT_ID(N'[Sales].[SalesOrderHeader]')
```

```
                    AND name = N'IndexTest')
    DROP INDEX IndexTest ON Sales.SalesOrderHeader;
GO
CREATE INDEX IndexTest ON Sales.SalesOrderHeader (OrderDate);
```

To retrieve all rows from Sales.SalesOrderHeader with OrderDate in the month of April in the year 2008, you can execute the following SELECT statement:

```
SELECT soh.SalesOrderID,
       soh.OrderDate
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE DATEPART(yy,
               soh.OrderDate) = 2008
      AND DATEPART(mm,
                   soh.OrderDate) = 4;
```

Using the DATEPART function on the column OrderDate prevents the optimizer from properly using the index IndexTest on the column and instead causes a scan, as shown in Figure 19-10.

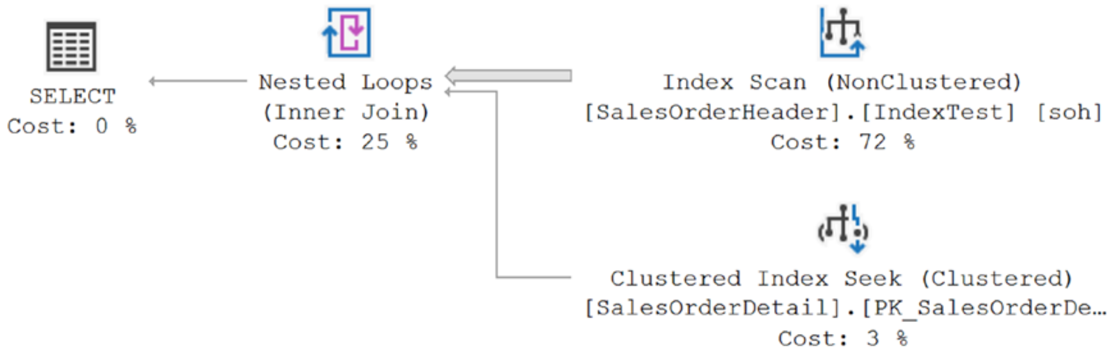

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 25 %
Index Scan (NonClustered)
[SalesOrderHeader].[IndexTest] [soh]
Cost: 72 %
Clustered Index Seek (Clustered)
[SalesOrderDetail].[PK_SalesOrderDe…
Cost: 3 %

***Figure 19-10.*** *Execution plan showing the detrimental effect of using the DATEPART function on a WHERE clause column*

These are the performance metrics:

```
Reads: 73
Duration: 2.5ms
```

The date part comparison can be done without applying the function on the DATETIME column.

```
SELECT  soh.SalesOrderID,
        soh.OrderDate
FROM    Sales.SalesOrderHeader AS soh
JOIN    Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE   soh.OrderDate >= '2008-04-01'
        AND soh.OrderDate < '2008-05-01';
```

This allows the optimizer to properly reference the index IndexTest that was created on the DATETIME column, as shown in Figure 19-11.

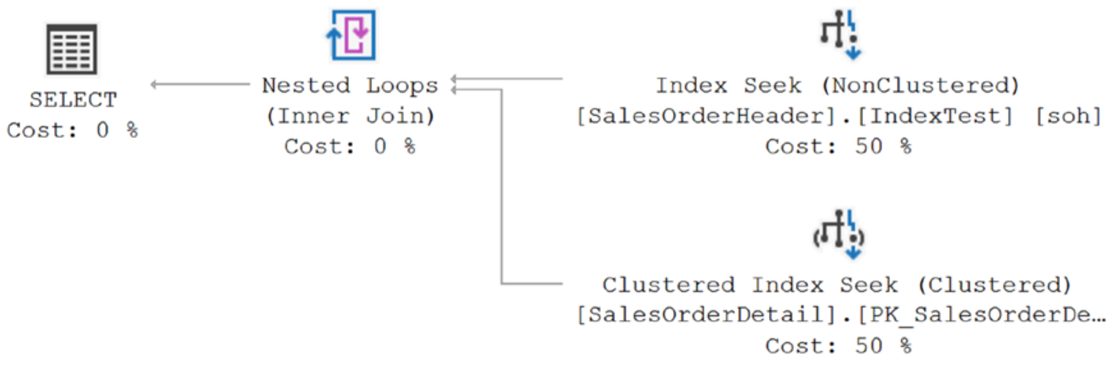

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[SalesOrderHeader].[IndexTest] [soh]
Cost: 50 %
Clustered Index Seek (Clustered)
[SalesOrderDetail].[PK_SalesOrderDe…
Cost: 50 %

***Figure 19-11.** Execution plan showing the benefit of not using the CONVERT function on a WHERE clause column*

These are the performance metrics:

```
Reads: 2
Duration: 104mcs
```

Therefore, to allow the optimizer to consider an index on a column referred to in the WHERE clause, always avoid using a function on the indexed column. This increases the effectiveness of indexes, which can improve query performance. In this instance, though, it's worth noting that the performance was minor since there's still a scan of the SalesOrderDetail table.

Be sure to drop the index created earlier.

```
DROP INDEX Sales.SalesOrderHeader.IndexTest;
```

## Custom Scalar UDF

Scalar functions are an attractive means of code reuse, especially if you need only a single value. However, while you can use them for data retrieval, it's not really their strong suit. In fact, you can see some significant performance issues depending on the UDF in question and how much data manipulation is required to satisfy its result set. To see this in action, let's start with a scalar function that retrieves the cost of a product.

```
CREATE OR ALTER FUNCTION dbo.ProductCost (@ProductID INT)
RETURNS MONEY
AS
BEGIN
        DECLARE @Cost MONEY
    SELECT TOP 1
           @Cost = pch.StandardCost
    FROM Production.ProductCostHistory AS pch
    WHERE pch.ProductID = @ProductID
    ORDER BY pch.StartDate DESC;

        IF @Cost IS NULL
                SET @Cost = 0

        RETURN @Cost
END
```

Calling the function is just a matter of making use of it within a query.

```
SELECT p.Name,
       dbo.ProductCost(p.ProductID)
FROM Production.Product AS p
WHERE p.ProductNumber LIKE 'HL%';
```

The performance from this query is about 413 microseconds and 16 reads on average. For such a simple query with a small result set, that may be fine. Figure 19-12 shows the execution plan for this.

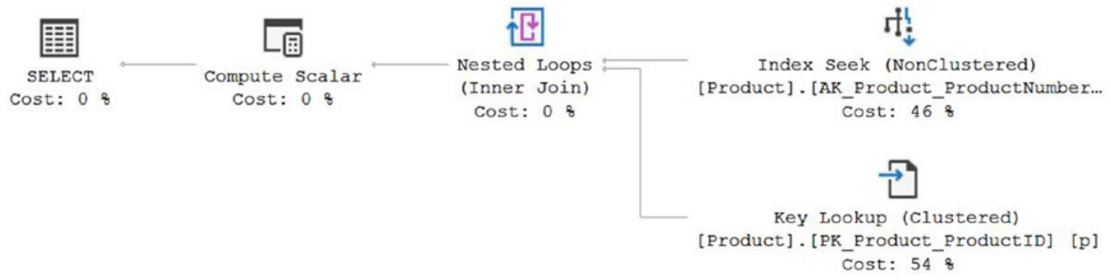
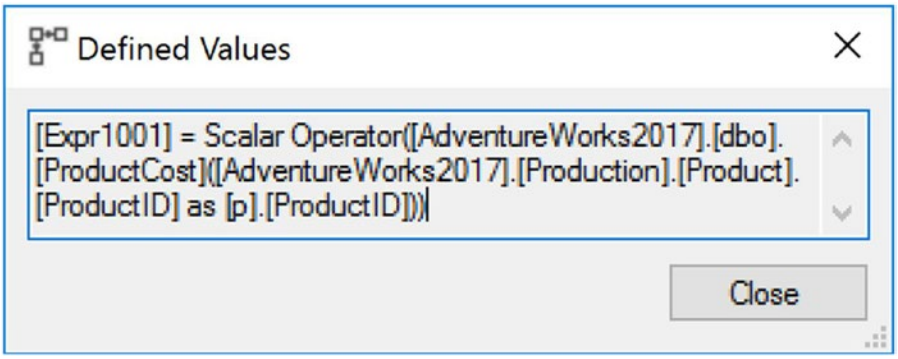


***Figure 19-12.** Execution plan with a scalar UDF*

The data is retrieved by using the index, `AK_Product_ProductNumber`, in a `Seek` operation. Because the index is not covering, a `Key Lookup` operation is used to retrieve the necessary additional data, `p.Name`. The `Compute Scalar` operator is then the scalar UDF. We can verify this by looking at the properties of the `Compute Scalar` operator in the Defined Values dialog, as shown in Figure 19-13.



[Expr1001] = Scalar Operator([AdventureWorks2017].[dbo].[ProductCost]([AdventureWorks2017].[Production].[Product].[ProductID] as [p].[ProductID]))

***Figure 19-13.** Compute Scalar operator properties showing the scalar UDF at work*

The problem is, you can't see the data access of the function. That information is hidden. You can capture it using an estimated plan, or you can query the query store to see plans for objects that are normally not immediately visible like this UDF:

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

The resulting execution plan looks like Figure 19-14.

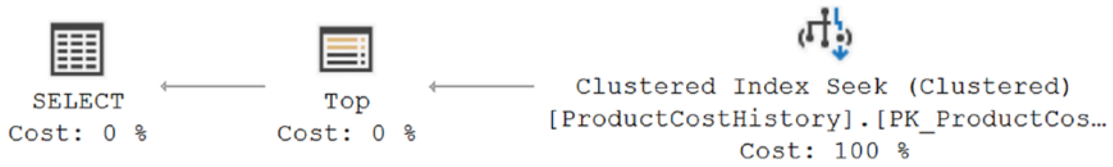
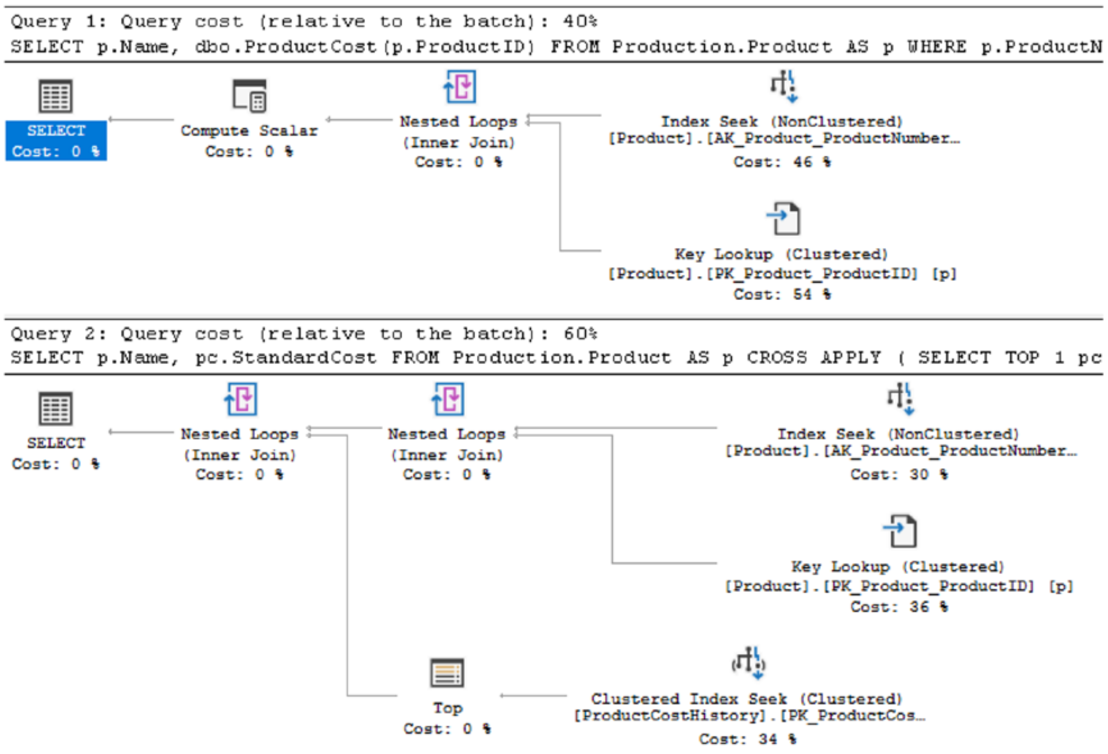

SELECT
Cost: 0 %
Top
Cost: 0 %
Clustered Index Seek (Clustered)
[ProductCostHistory].[PK_ProductCos…
Cost: 100 %

***Figure 19-14.*** *Execution plan of the scalar function*

While you can't see the work being done, as you can see from the execution plan, it is in fact, doing more work. If we were to rewrite this query as follows to eliminate the use of the function, the performance would also change:

```
SELECT p.Name,
       pc.StandardCost
FROM Production.Product AS p
    CROSS APPLY
(   SELECT TOP 1
           pch.StandardCost
    FROM Production.ProductCostHistory AS pch
    WHERE pch.ProductID = p.ProductID
    ORDER BY pch.StartDate DESC) AS pc
WHERE p.ProductNumber LIKE 'HL%';
```

Making this change results in a small increase in performance, from 413 microseconds to about 295 microseconds and a reduction in the reads from 16 to 14. While the execution plan is more complex, as shown in Figure 19-15, the overall performance is improved.



```
Query 1: Query cost (relative to the batch): 40%
SELECT p.Name, dbo.ProductCost(p.ProductID) FROM Production.Product AS p WHERE p.ProductN
```

SELECT
Cost: 0 %
Compute Scalar
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[Product].[AK_Product_ProductNumber…
Cost: 46 %
Key Lookup (Clustered)
[Product].[PK_Product_ProductID] [p]
Cost: 54 %

```
Query 2: Query cost (relative to the batch): 60%
SELECT p.Name, pc.StandardCost FROM Production.Product AS p CROSS APPLY ( SELECT TOP 1 pc
```

SELECT
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Index Seek (NonClustered)
[Product].[AK_Product_ProductNumber…
Cost: 30 %
Key Lookup (Clustered)
[Product].[PK_Product_ProductID] [p]
Cost: 36 %
Top
Cost: 0 %
Clustered Index Seek (Clustered)
[ProductCostHistory].[PK_ProductCos…
Cost: 34 %

***Figure 19-15.** An execution plan with a scalar function and one without a scalar function*

While the optimizer is suggesting that the plan with the scalar function has an estimated cost below the plan without, the actual performance metrics are almost the opposite. This is because the cost of a scalar function is fixed at a per-row cost and doesn't take into account the complexity of the process supporting it, in this case, access to a table. This ends up making the query with the compute scalar not only slower but more resource intensive.

## Minimize Optimizer Hints

SQL Server's cost-based optimizer dynamically determines the processing strategy for a query based on the current table/index structure and statistics. This dynamic behavior can be overridden using optimizer hints, taking some of the decisions away from the optimizer by instructing it to use a certain processing strategy. This makes the optimizer behavior static and doesn't allow it to dynamically update the processing strategy as the table/index structures or statistics change.