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.

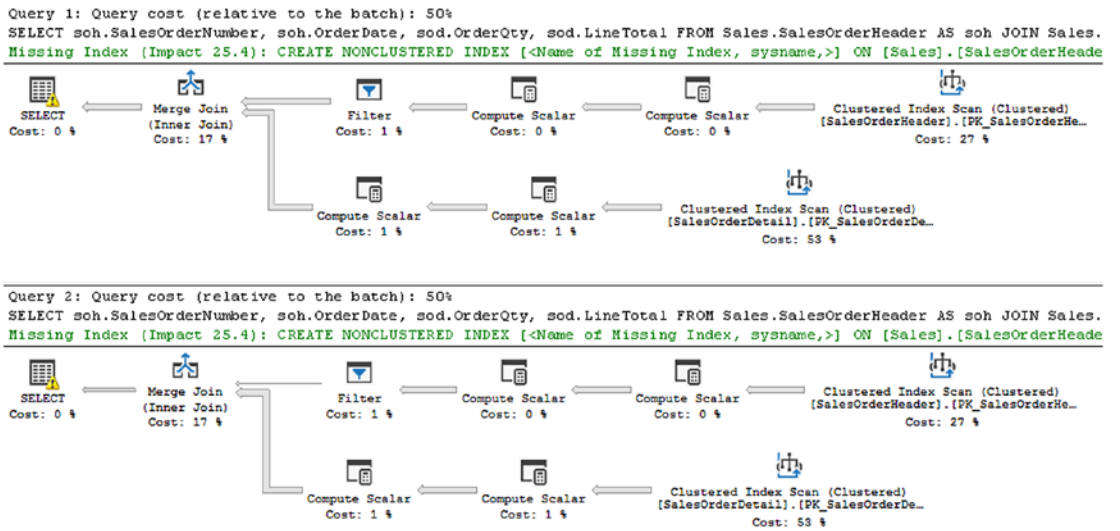


***Figure 18-15.*** *WITH RECOMPILE doesn't change identical execution plans*

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.


***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;
```

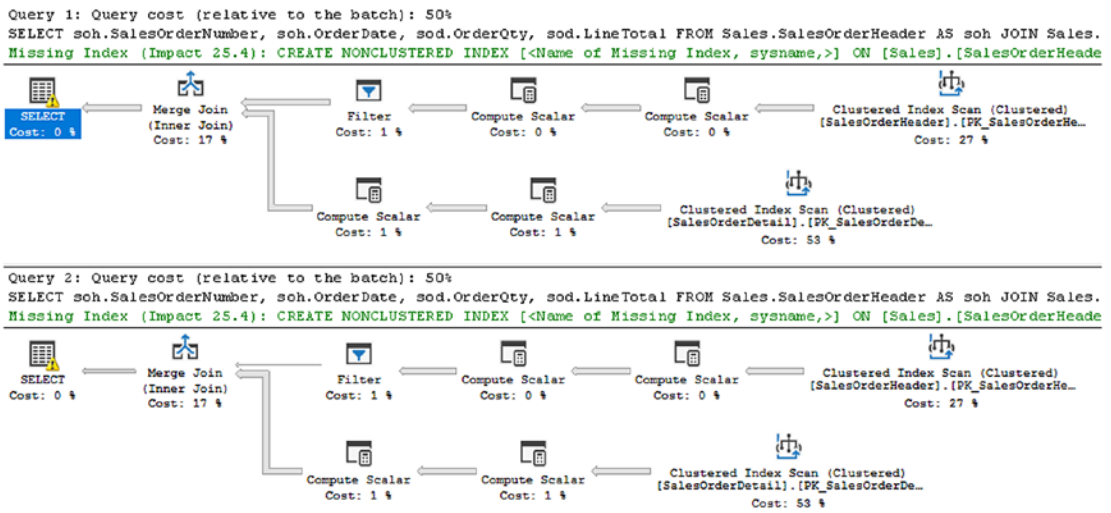


***Figure 18-17.*** *Using a plan guide to apply the OPTIMIZE FOR query hint*

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.

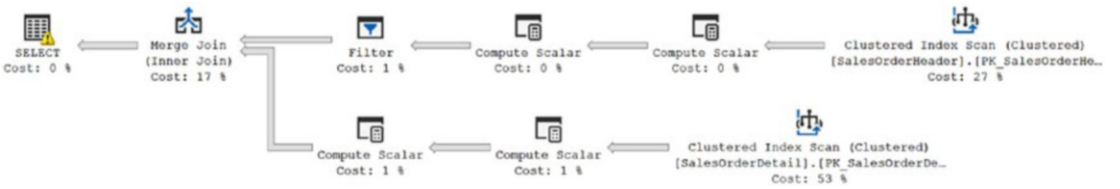


***Figure 18-19.*** *The query uses a different execution plan from the one wanted*

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.

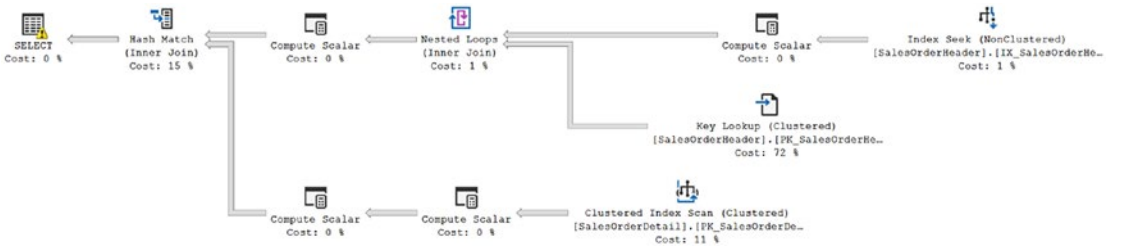


***Figure 18-20.*** *The plan guide forces a new execution plan on the same query*

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.




G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_19

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
```

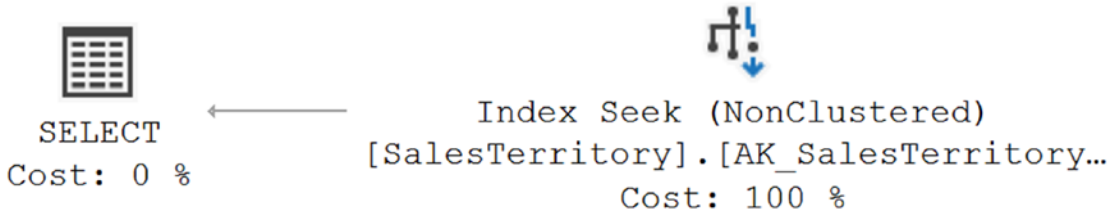


***Figure 19-1.*** *Execution plan showing the benefit of referring to a limited number of columns*

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.

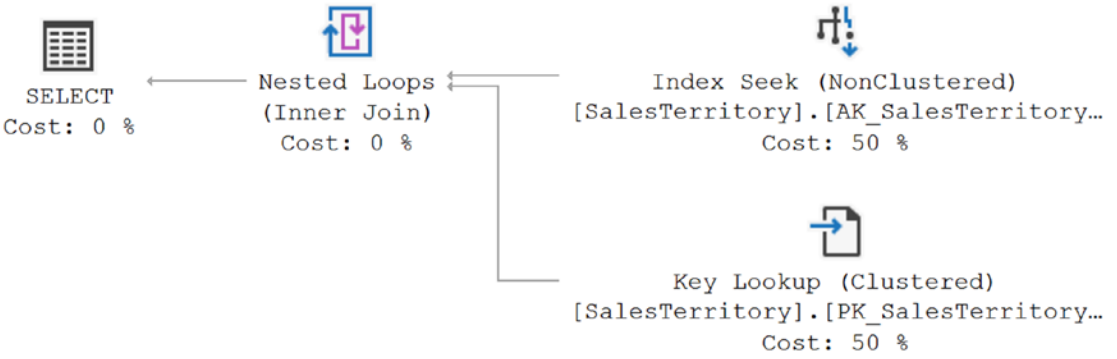


***Figure 19-2.*** *Execution plan illustrating the added cost of referring to too many columns*

## 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.

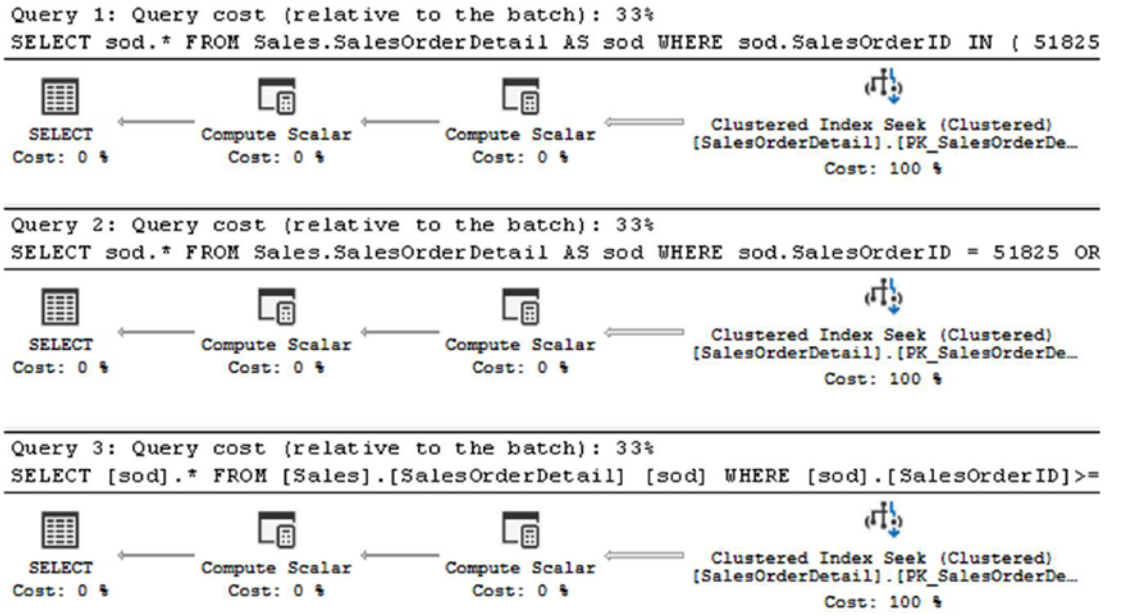


***Figure 19-3.*** *Execution plan for a simple SELECT statement using a BETWEEN clause*

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

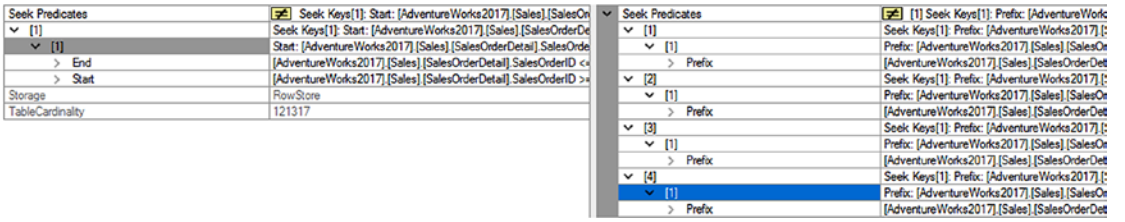


***Figure 19-4.*** *Execution plan details for a BETWEEN condition (left) and an IN condition (right)*

As shown in Figure 19-4, SQL Server resolved the `IN` condition containing four values into four `OR` conditions. Accordingly, the clustered index (`PKSalesTerritoryTerritoryId`) is accessed four times (`Scan count 4`) to retrieve rows for the four `IN` and `OR` conditions, as shown in the following corresponding `STATISTICS IO` 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