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.

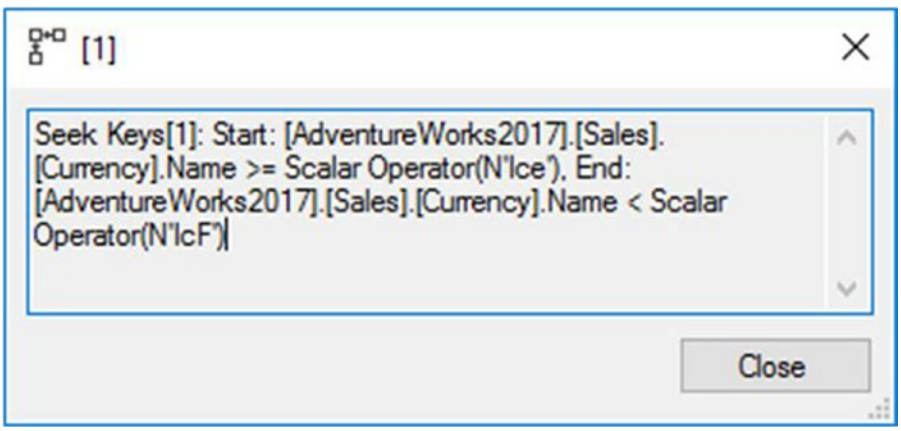

Figure 19-5. *Execution plan showing automatic conversion of a LIKE clause with a trailing % sign to an indexable search condition*

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;
```

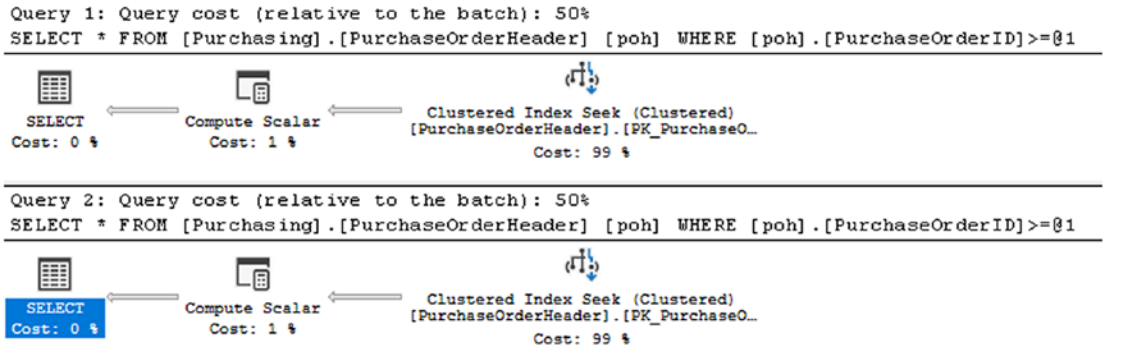

Figure 19-6. *Execution plan showing automatic transformation of a nonindexable !< operator to an indexable >= operator*

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.

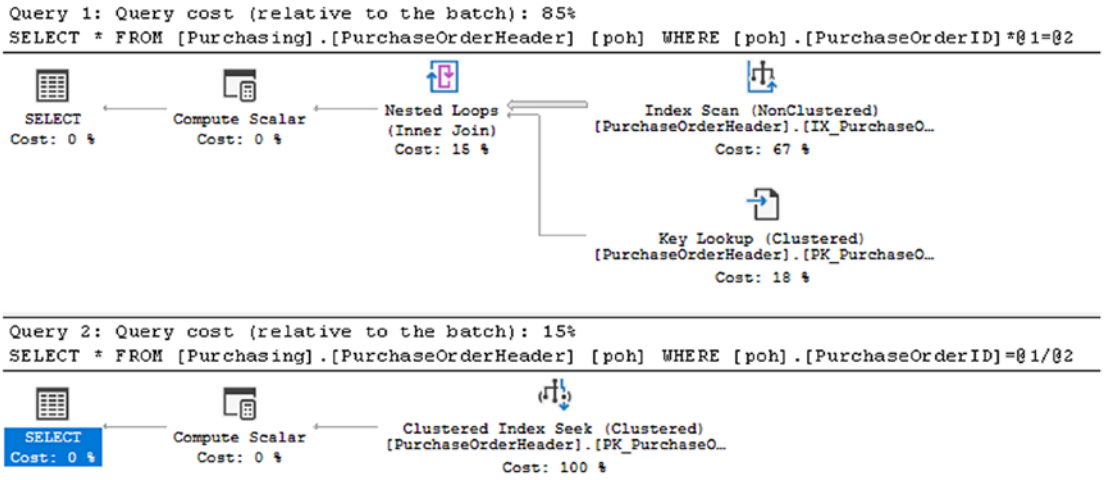

Figure 19-7. *Execution plan showing the detrimental effect of an arithmetic operator on a WHERE clause column*

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.

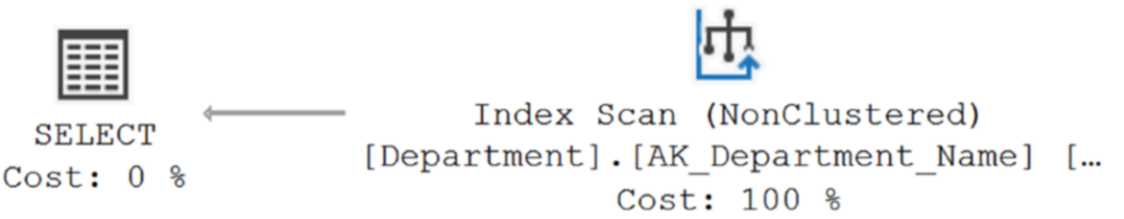

Figure 19-8. *Execution plan showing the detrimental effect of using the SUBSTRING function on a WHERE clause column*

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.

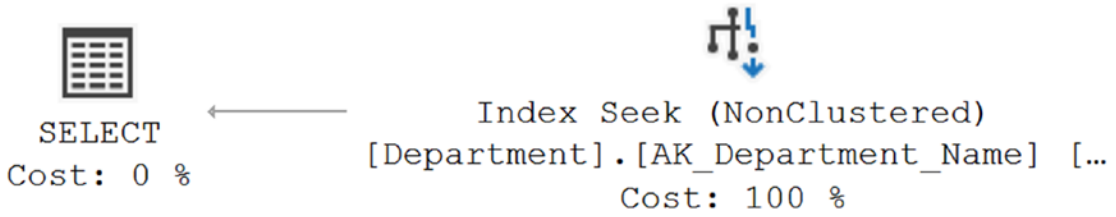

Figure 19-9. *Execution plan showing the benefit of not using the SUBSTRING function on a WHERE clause column*

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.

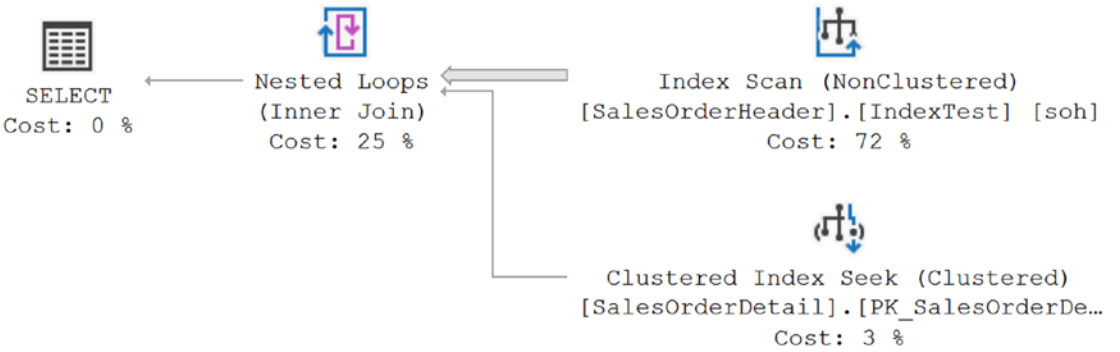

Figure 19-10. *Execution plan showing the detrimental effect of using the DATEPART function on a WHERE clause column*

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.

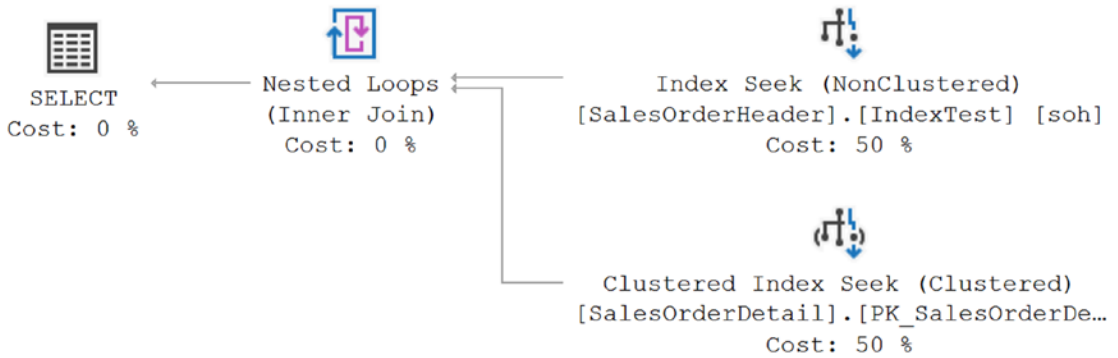

Figure 19-11. *Execution plan showing the benefit of not using the CONVERT function on a WHERE clause column*

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.

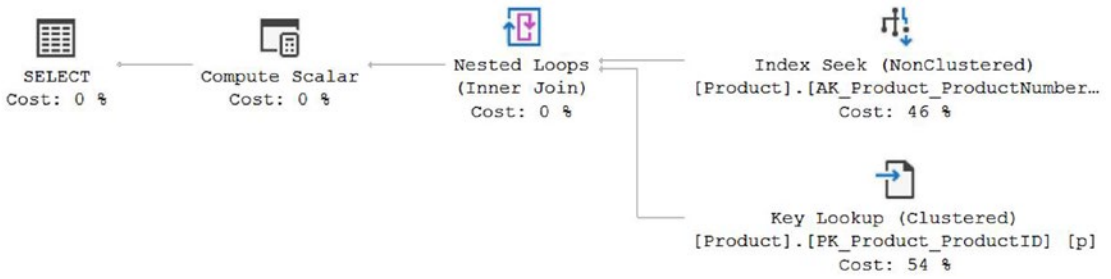

Figure 19-12. *Execution plan with a scalar UDF*

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.

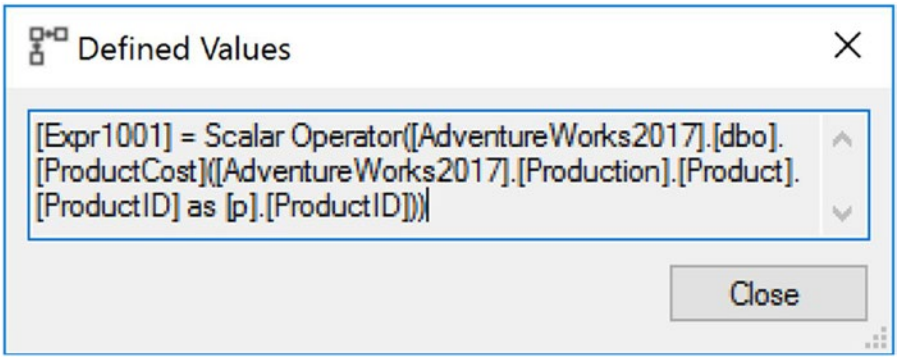

Figure 19-13. *Compute Scalar operator properties showing the scalar UDF at work*

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.

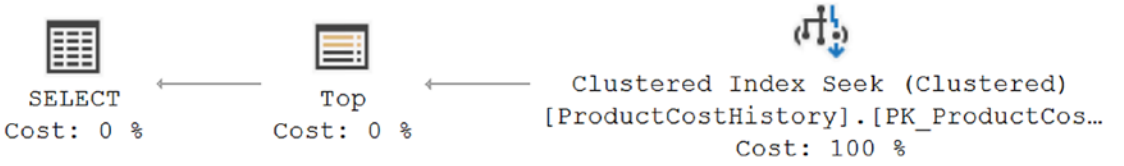

Figure 19-14. *Execution plan of the scalar function*

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.

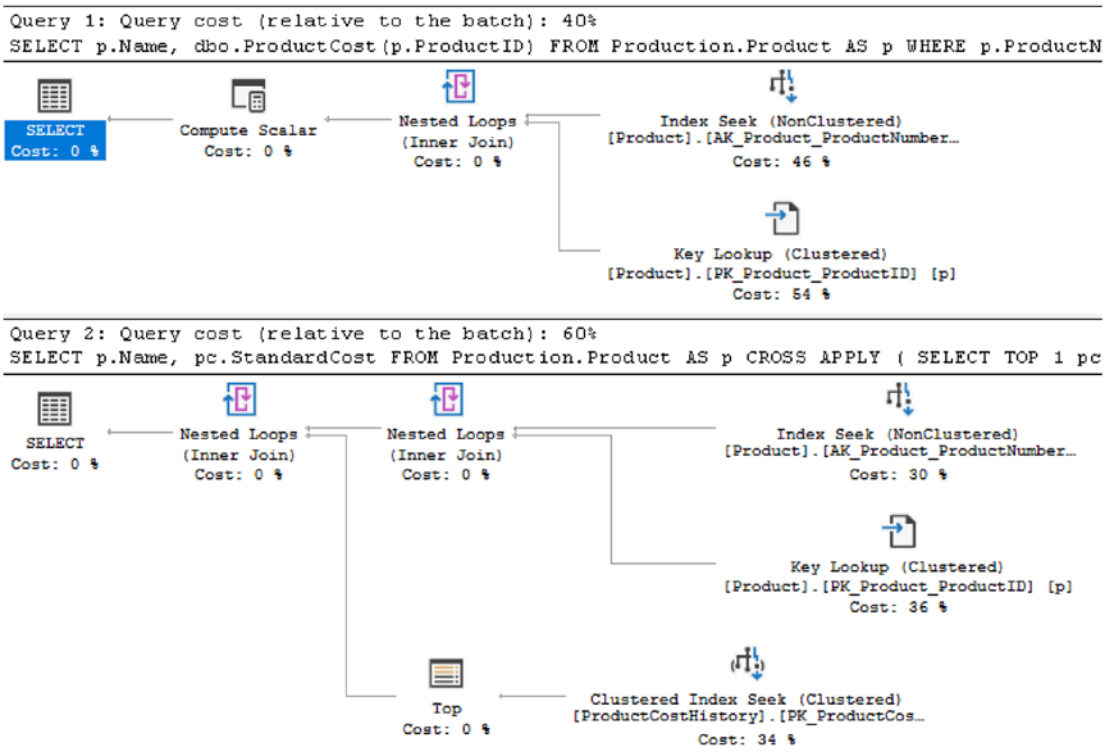

Figure 19-15. *An execution plan with a scalar function and one without a scalar function*

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.

Since it is usually difficult to outsmart the optimizer, the usual recommendation is to avoid optimizer hints. Some hints can be extremely beneficial (for example, `OPTIMIZE FOR`), but others are beneficial in only very specific circumstances. Generally, it is beneficial to let the optimizer determine a cost-effective processing strategy based on the data distribution statistics, indexes, and other factors. Forcing the optimizer (with hints) to use a specific processing strategy hurts performance more often than not, as shown in the following examples for these hints:

- `JOIN` hint
- `INDEX` hint

JOIN Hint

As explained in Chapter 6, the optimizer dynamically determines a cost-effective `JOIN` strategy between two data sets based on the table/index structure and data. Table 19-2 summarizes the `JOIN` types supported by SQL Server 2017 for easy reference.

Table 19-2. *JOIN Types Supported by SQL Server 2017*

JOIN Type	Index on Joining Columns	Usual Size of Joining Tables	Presorted JOIN Clause
Nested loops	Inner table a must	Small	Optional
	Outer table preferable		
Merge	Both tables a must	Large	Yes
	Optimal condition: clustered or covering index on both		
Hash	Inner table *not* indexed	Any	No
		Optimal condition: inner table large, outer table small	
Adaptive	Uses either hash or loops depending on the data being returned by the query	Variable, but usually very large because it currently works only with columnstore indexes	Depends on join type

SQL Server 2017 introduced the new join type, the adaptive join. It's really just a dynamic determination of either the nested loops or the hash, but that adaptive processing methodology effectively makes for a new join type, which is why I've listed it here.

Note The outer table is usually the smaller of the two joining tables.

You can instruct SQL Server to use a specific `JOIN` type by using the `JOIN` hints in Table 19-3.

Table 19-3. *JOIN Hints*

JOIN Type	JOIN Hint
Nested loop	`LOOP`
Merge	`MERGE`
Hash	`HASH`
	`REMOTE`

There is no hint for the adaptive join. There is a hint for a `REMOTE` join. This is used when one of the tables in a join is remote to the current database. It allows you to direct which side of the `JOIN`, based on the input size, should be doing the work.

To understand how the use of `JOIN` hints can affect performance, consider the following `SELECT` statement:

```
SELECT s.Name AS StoreName,
       p.LastName + ', ' + p.FirstName
FROM Sales.Store AS s
    JOIN Sales.SalesPerson AS sp
        ON s.SalesPersonID = sp.BusinessEntityID
    JOIN HumanResources.Employee AS e
        ON sp.BusinessEntityID = e.BusinessEntityID
    JOIN Person.Person AS p
        ON e.BusinessEntityID = p.BusinessEntityID;
```

Figure 19-16 shows the execution plan.

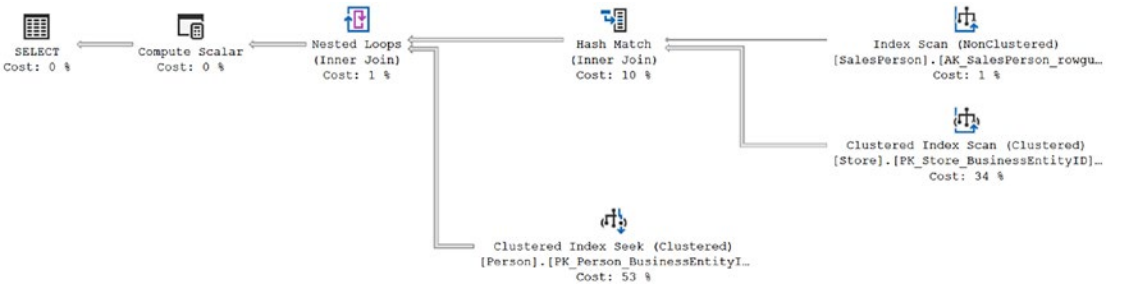

Figure 19-16. *Execution plan showing choices made by the optimizer*

As you can see, SQL Server dynamically decided to use a `LOOP JOIN` to add the data from the `Person.Person` table and to add a `HASH JOIN` for the `Sales.Salesperson` and `Sales.Store` tables. As demonstrated in Chapter 6, for simple queries affecting a small result set, a `LOOP JOIN` generally provides better performance than a `HASH JOIN` or `MERGE JOIN`. Since the number of rows coming from the `Sales.Salesperson` table is relatively small, it might feel like you could force the `JOIN` to be a `LOOP` like this:

```
SELECT s.Name AS StoreName,
       p.LastName + ',   ' + p.FirstName
FROM Sales.Store AS s
    JOIN Sales.SalesPerson AS sp
        ON s.SalesPersonID = sp.BusinessEntityID
    JOIN HumanResources.Employee AS e
        ON sp.BusinessEntityID = e.BusinessEntityID
    JOIN Person.Person AS p
        ON e.BusinessEntityID = p.BusinessEntityID
OPTION (LOOP JOIN);
```

When this query is run, the execution plan changes, as you can see in Figure 19-17.

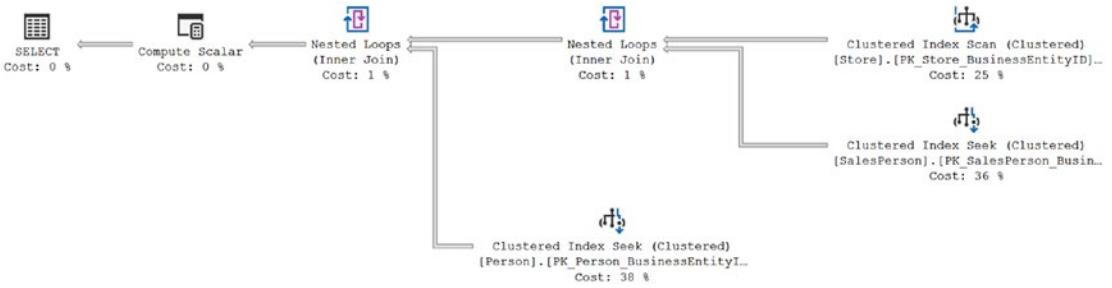

Figure 19-17. *Changes made by using the JOIN query hint*

Here are the corresponding `performance` outputs for each query:

- With no `JOIN` hint:

  ```
  Reads: 2364
  Duration: 84ms
  ```

- With a `JOIN` hint:

  ```
  Reads: 3740
  Duration: 97ms
  ```

You can see that the query with the `JOIN` hint takes longer to run than the query without the hint. It also adds a number of reads. You can make this even worse. Instead of telling all hints used in the query to be a `LOOP` join, it is possible to target just the one you are interested in, like so:

```
SELECT s.Name AS StoreName,
       p.LastName + ',   ' + p.FirstName
FROM Sales.Store AS s
    INNER LOOP JOIN Sales.SalesPerson AS sp
        ON s.SalesPersonID = sp.BusinessEntityID
    JOIN HumanResources.Employee AS e
        ON sp.BusinessEntityID = e.BusinessEntityID
    JOIN Person.Person AS p
        ON e.BusinessEntityID = p.BusinessEntityID;
```

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

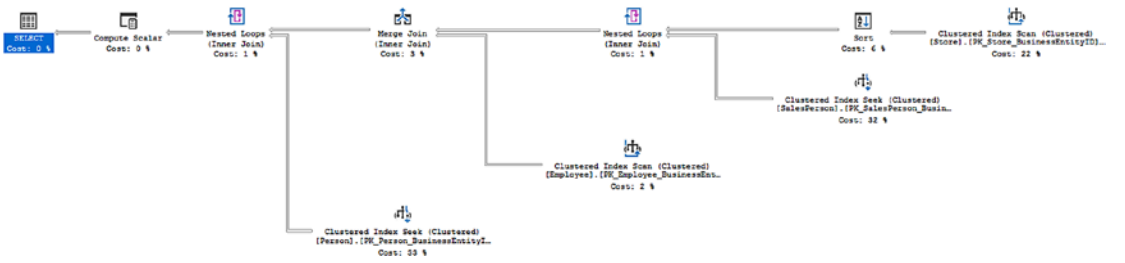

Figure 19-18. *More changes from using the LOOP join hint*

As you can see, there are now four tables referenced in the query plan. There have been four tables referenced through all the previous executions, but the optimizer was able to eliminate one table from the query through the simplification process of optimization (referred to in Chapter 8). Now the hint has forced the optimizer to make different choices than it otherwise might have and removed simplification from the process. The reads degrade although the execution time improved slightly over the previous query.

```
Reads: 3749
Duration: 86ms
```

`JOIN` hints force the optimizer to ignore its own optimization strategy and use instead the strategy specified by the query. `JOIN` hints can hurt query performance because of the following factors:

- Hints prevent autoparameterization.
- The optimizer is prevented from dynamically deciding the joining order of the tables.

Therefore, it makes sense to not use the `JOIN` hint but to instead let the optimizer dynamically determine a cost-effective processing strategy. There are exceptions, of course, but the exceptions must be validated through thorough testing.

INDEX Hints

As mentioned earlier, using an arithmetic operator on a WHERE clause column prevents the optimizer from choosing the index on the column. To improve performance, you can rewrite the query without using the arithmetic operator on the WHERE clause, as shown in the corresponding example. Alternatively, you may even think of forcing the optimizer to use the index on the column with an INDEX hint (a type of optimizer hint). However, most of the time, it is better to avoid the INDEX hint and let the optimizer behave dynamically.

To understand the effect of an INDEX hint on query performance, consider the example presented in the "Avoid Arithmetic Operators on the WHERE Clause Column" section. The multiplication operator on the PurchaseOrderID column prevented the optimizer from choosing the index on the column. You can use an INDEX hint to force the optimizer to use the index on the OrderID column as follows:

```
SELECT *
FROM Purchasing.PurchaseOrderHeader AS poh WITH (INDEX(PK_
PurchaseOrderHeader_PurchaseOrderID))
WHERE poh.PurchaseOrderID * 2 = 3400;
```

Note the relative cost of using the INDEX hint in comparison to not using the INDEX hint, as shown in Figure 19-18. Also, note the difference in the number of logical reads shown in the following performance metrics:

- No hint (with the arithmetic operator on the WHERE clause column):

  ```
  Reads: 11
  Duration: 210mcs
  ```

- No hint (without the arithmetic operator on the WHERE clause column):

  ```
  Reads: 2
  Duration: 105mcs
  ```

- INDEX hint:

  ```
  Reads: 44
  Duration: 380mcs
  ```

From the relative cost of execution plans and number of logical reads, it is evident that the query with the INDEX hint actually impaired the query performance. Even though it allowed the optimizer to use the index on the PurchaseOrderID column, it did not allow the optimizer to determine the proper index-access mechanism. Consequently, the optimizer used the index scan to access just one row. In comparison, avoiding the arithmetic operator on the WHERE clause column and not using the INDEX hint allowed the optimizer not only to use the index on the PurchaseOrderID column but also to determine the proper index access mechanism: INDEX SEEK.

Therefore, in general, let the optimizer choose the best indexing strategy for the query and don't override the optimizer behavior using an INDEX hint. Also, not using INDEX hints allows the optimizer to decide the best indexing strategy dynamically as the data changes over time. Figure 19-19 shows the difference between specifying index hints and not specifying them.

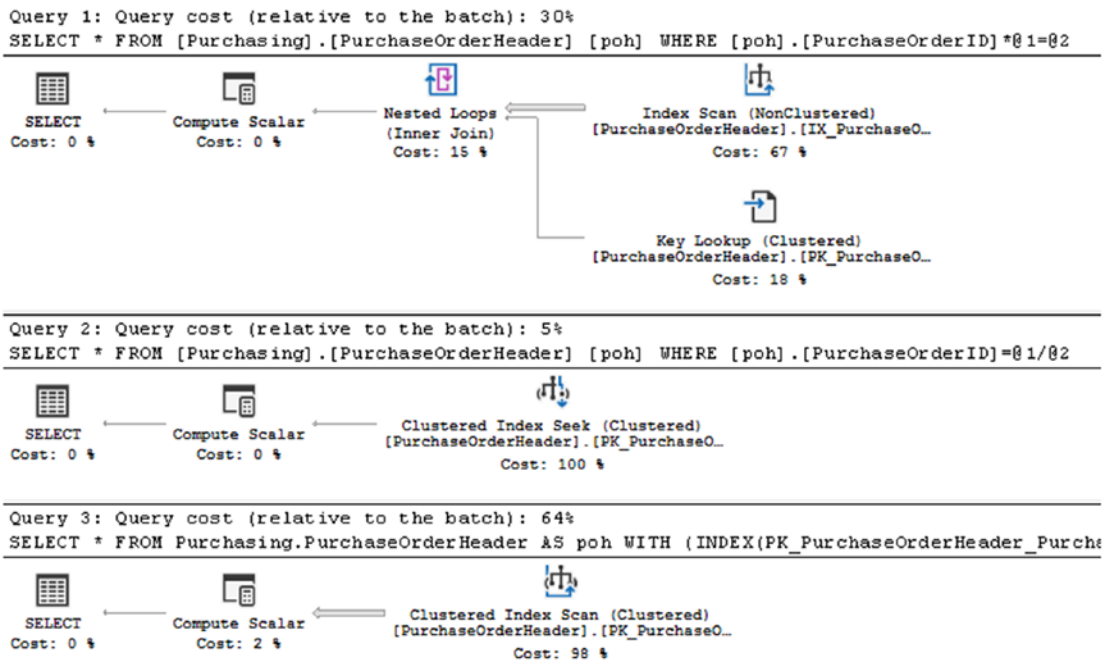

Figure 19-19. *Cost of a query with and without different INDEX hints*

Using Domain and Referential Integrity

Domain and referential integrity help define and enforce valid values for a column, maintaining the integrity of the database. This is done through column/table constraints.

Since data access is usually one of the most costly operations in a query execution, avoiding redundant data access helps the optimizer reduce the query execution time. Domain and referential integrity help the SQL Server optimizer analyze valid data values without physically accessing the data, which reduces query time.

To understand how this happens, consider the following examples:

- The `NOT NULL` constraint
- Declarative referential integrity (DRI)

NOT NULL Constraint

The `NOT NULL` column constraint is used to implement domain integrity by defining the fact that a `NULL` value can't be entered in a particular column. SQL Server automatically enforces this fact at runtime to maintain the domain integrity for that column. Also, defining the `NOT NULL` column constraint helps the optimizer generate an efficient processing strategy when the `ISNULL` function is used on that column in a query.

To understand the performance benefit of the `NOT NULL` column constraint, consider the following example. These two queries are intended to return every value that does not equal `'B'`. These two queries are running against similarly sized columns, each of which will require a table scan to return the data:

```
SELECT p.FirstName
FROM Person.Person AS p
WHERE p.FirstName < 'B'
      OR p.FirstName >= 'C';

SELECT p.MiddleName
FROM Person.Person AS p
WHERE p.MiddleName < 'B'
      OR p.MiddleName >= 'C';
```

The two queries use similar execution plans, as you can see in Figure 19-20.