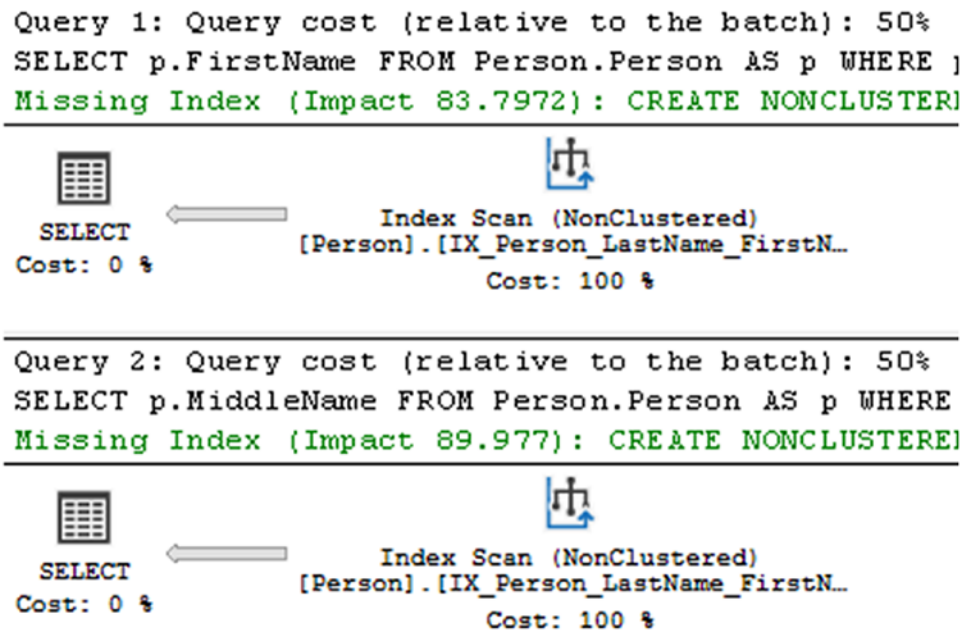

Figure 19-20. *Table scans caused by a lack of indexes*

The differences are primarily caused by the estimated rows to be returned. While both queries are going against the same index and scanning it, each one still has a different predicate and estimated rows, as you can see in Figure 19-21.

Figure 19-21. *Different estimated rows because of differences in the WHERE clause*

Since the column `Person.MiddleName` can contain `NULL`, the data returned is incomplete. This is because, by definition, although a `NULL` value meets the necessary criteria of not being in any way equal to `'B'`, you can't return `NULL` values in this manner. An added `OR` clause is necessary. That would mean modifying the second query like this:

```
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'
      OR p.MiddleName IS NULL;
```

Also, as shown in the missing index statements in the execution plan in Figure 19-19, these two queries can benefit from having indexes created on their tables. Creating test indexes like the following should satisfy the requirements:

```
CREATE INDEX TestIndex1 ON Person.Person (MiddleName);

CREATE INDEX TestIndex2 ON Person.Person (FirstName);
```

When the queries are reexecuted, Figure 19-22 shows the resultant execution plan for the two SELECT statements.

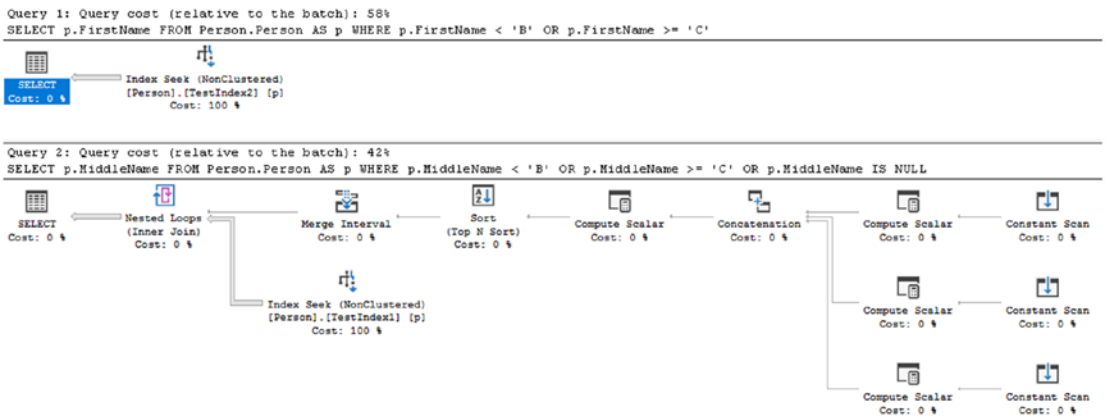

Figure 19-22. *Effect of the IS NULL option being used*

As shown in Figure 19-22, the optimizer was able to take advantage of the index TestIndex2 on the Person.FirstName column to get an Index Seek operation. Unfortunately, the requirements for processing the NULL columns were very different. The index TestIndex1 was not used in the same way. Instead, three constants were created for each of the three criteria defined within the query. These were then joined together through the Concatenation operation, sorted and merged prior to seeking the index three times through the Nested Loop operator to arrive at the result set. Although

it appears, from the estimated costs in the execution plan, that this was the less costly query (42 percent compared to 58 percent), performance metrics tell a different story.

```
Reads: 43
Duration: 143ms
```

vs.

```
Reads: 68
Duration: 168ms
```

Be sure to drop the test indexes that were created.

```
DROP INDEX TestIndex1 ON Person.Person;

DROP INDEX TestIndex2 ON Person.Person;
```

As much as possible, you should attempt to leave NULL values out of the database. However, when data is unknown, default values may not be possible. That's when NULL will come back into the design. I find NULLs to be unavoidable, but they are something to minimize as much as you can.

When it is unavoidable and you will be dealing with NULL values, keep in mind that you can use a filtered index that removes NULL values from the index, thereby improving the performance of that index. This was detailed in Chapter 7. Sparse columns offer another option to help you deal with NULL values. Sparse columns are primarily aimed at storing NULL values more efficiently and therefore reduce space—at a sacrifice in performance. This option is specifically targeted at business intelligence (BI) databases, not OLTP databases where large amounts of NULL values in fact tables are a normal part of the design.

Declarative Referential Integrity

Declarative referential integrity is used to define referential integrity between a parent table and a child table. It ensures that a record in the child table exists only if the corresponding record in the parent table exists. The only exception to this rule is that the child table can contain a NULL value for the identifier that links the rows of the child table to the rows of the parent table. For all other values of the identifier in the child, a corresponding value must exist in the parent table. In SQL Server, DRI is implemented

using a PRIMARY KEY constraint on the parent table and a FOREIGN KEY constraint on the child table.

With DRI established between two tables and the foreign key columns of the child table set to NOT NULL, the SQL Server optimizer is assured that for every record in the child table, the parent table has a corresponding record. Sometimes this can help the optimizer improve performance because accessing the parent table is not necessary to verify the existence of a parent record for a corresponding child record.

To understand the performance benefit of implementing declarative referential integrity, let's consider an example. First, eliminate the referential integrity between two tables, Person.Address and Person.StateProvince, using this script:

```
IF EXISTS (   SELECT *
              FROM sys.foreign_keys
              WHERE object_id = OBJECT_ID(N'[Person].[FK_Address_State
              Province_StateProvinceID]')
                    AND parent_object_id = OBJECT_ID(N'[Person].[Address]'))
    ALTER TABLE Person.Address
    DROP CONSTRAINT FK_Address_StateProvince_StateProvinceID;
```

Consider the following SELECT statement:

```
SELECT a.AddressID,
       sp.StateProvinceID
FROM Person.Address AS a
    JOIN Person.StateProvince AS sp
        ON a.StateProvinceID = sp.StateProvinceID
WHERE a.AddressID = 27234;
```

Note that the SELECT statement fetches the value of the StateProvinceID column from the parent table (Person.Address). If the nature of the data requires that for every product (identified by StateProvinceId) in the child table (Person.StateProvince) the parent table (Person.Address) contains a corresponding product, then you can rewrite

the preceding SELECT statement as follows to reference the Address table instead of the StateProvince table for the StateProvinceID column:

```
SELECT a.AddressID,
       a.StateProvinceID
FROM Person.Address AS a
    JOIN Person.StateProvince AS sp
        ON a.StateProvinceID = sp.StateProvinceID
WHERE a.AddressID = 27234;
```

Both SELECT statements should return the same result set. After removing the foreign key constraint, the optimizer generates the same execution plan for both the SELECT statements, as shown in Figure 19-23.

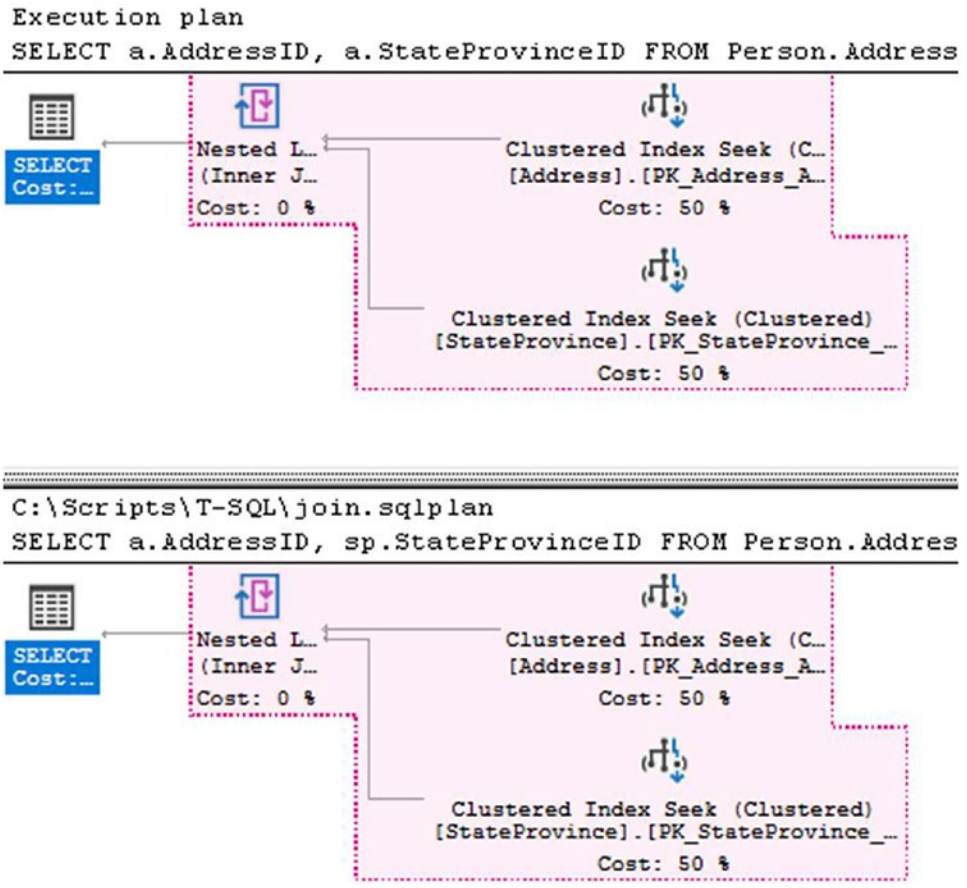

Figure 19-23. *Execution plan when DRI is not defined between the two tables*

To understand how declarative referential integrity can affect query performance, replace the FOREIGN KEY dropped earlier.

```
ALTER TABLE Person.Address WITH CHECK
ADD CONSTRAINT FK_Address_StateProvince_StateProvinceID
    FOREIGN KEY
    (
        StateProvinceID
    )
    REFERENCES Person.StateProvince
    (
        StateProvinceID
    );
```

Note There is now referential integrity between the tables.

Figure 19-24 shows the resultant execution plans for the two SELECT statements.

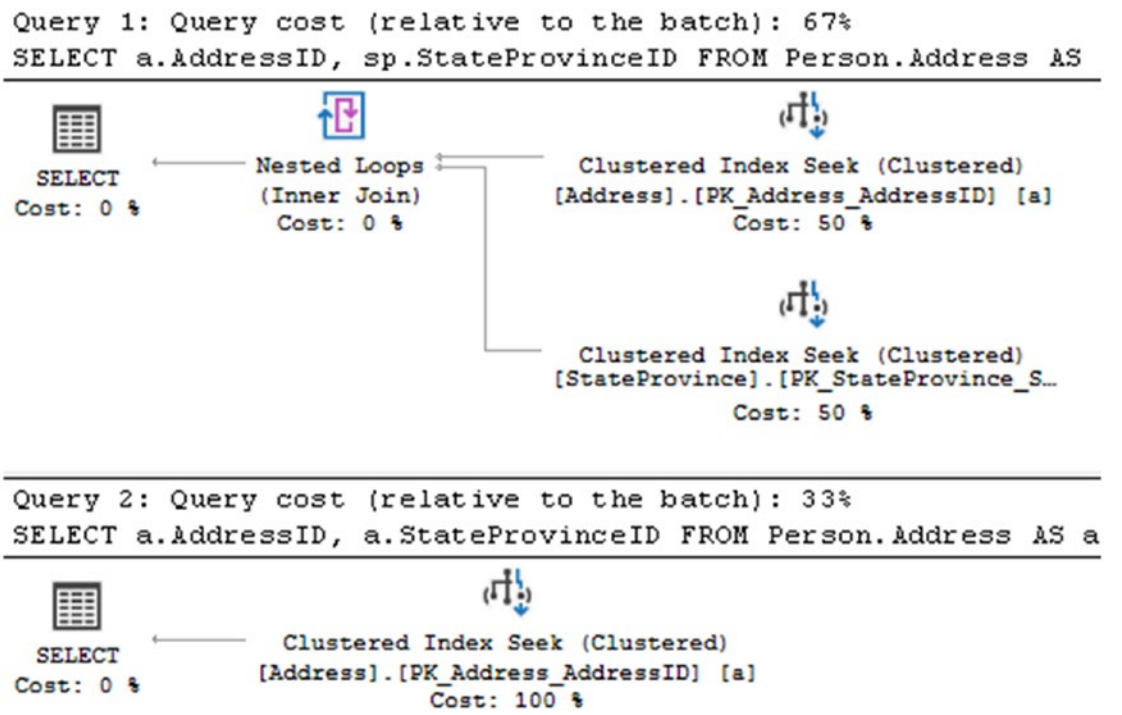

Figure 19-24. *Execution plans showing the benefit of defining DRI between the two tables*

As you can see, the execution plan of the second `SELECT` statement is highly optimized: the `Person.StateProvince` table is not accessed. With the declarative referential integrity in place (and `Address.StateProvince` set to `NOT NULL`), the optimizer is assured that for every record in the child table, the parent table contains a corresponding record. Therefore, the `JOIN` clause between the parent and child tables is redundant in the second `SELECT` statement, with no other data requested from the parent table.

You probably already knew that domain and referential integrity are Good Things, but you can see that they not only ensure data integrity but also improve performance. As just illustrated, domain and referential integrity provide more choices to the optimizer to generate cost-effective execution plans and improve performance.

To achieve the performance benefit of DRI, as mentioned previously, the foreign key columns in the child table should be `NOT NULL`. Otherwise, there can be rows (with foreign key column values as `NULL`) in the child table with no representation in the parent table. That won't prevent the optimizer from accessing the primary table (`Prod`) in the previous query. By default—that is, if the `NOT NULL` attribute isn't mentioned for

a column—the column can have NULL values. Considering the benefit of the NOT NULL attribute and the other benefits explained in this section, always mark the attribute of a column as NOT NULL if NULL isn't a valid value for that column.

You also must make sure you are using the WITH CHECK option when building your foreign key constraints. If the NOCHECK option is used, these are considered to be untrustworthy constraints by the optimizer, and you won't realize the performance benefits that they can offer.

Summary

As discussed in this chapter, to improve the performance of a database application, it is important to ensure that SQL queries are designed properly to benefit from performance-enhancement techniques such as indexes, stored procedures, database constraints, and so on. Ensure that queries don't prevent the use of indexes. In many cases, the optimizer has the ability to generate cost-effective execution plans irrespective of query structure, but it is still a good practice to design the queries properly in the first place. Even after you design individual queries for great performance, the overall performance of a database application may not be satisfactory. It is important not only to improve the performance of individual queries but also to ensure that they don't use up the available resources on the system. The next chapter will cover how to reduce resource usage within your queries.

CHAPTER 20

Reduce Query Resource Use

In the previous chapter you focused on writing queries in such a way that they appropriately used indexes and statistics. In this chapter, you'll make sure you're writing a queries in such a way that they don't use your resources in inappropriate ways. There are approaches to writing queries that avoid using memory, CPU, and I/O, as well as ways to write the queries that use more of these resources than you really should. I'll go over a number of mechanisms to ensure your resources are used optimally by the queries under your control.

In this chapter, I cover the following topics:

- Query designs that are less resource-intensive
- Query designs that use the procedure cache effectively
- Query designs that reduce network overhead
- Techniques to reduce the transaction cost of a query

Avoiding Resource-Intensive Queries

Many database functionalities can be implemented using a variety of query techniques. The approach you should take is to use query techniques that are resource friendly and set-based. These are a few techniques you can use to reduce the footprint of a query:

- Avoid data type conversion.
- Use `EXISTS` over `COUNT(*)` to verify data existence.
- Use `UNION ALL` over `UNION`.

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_20

- Use indexes for aggregate and sort operations.
- Be cautious with local variables in a batch query.
- Be careful when naming stored procedures.

I cover these points in more detail in the next sections.

Avoid Data Type Conversion

SQL Server allows, in some instances (defined by the large table of data conversions available in Books Online), a value/constant with different but compatible data types to be compared with a column's data. SQL Server automatically converts the data from one data type to another. This process is called *implicit data type conversion.* Although useful, implicit conversion adds overhead to the query optimizer. To improve performance, use a value/constant with the same data type as that of the column to which it is compared.

To understand how implicit data type conversion affects performance, consider the following example:

```
IF EXISTS (   SELECT *
              FROM sys.objects
              WHERE object_id = OBJECT_ID(N'dbo.Test1'))
    DROP TABLE dbo.Test1;

CREATE TABLE dbo.Test1 (Id INT IDENTITY(1, 1),
                        MyKey VARCHAR(50),
                        MyValue VARCHAR(50));
CREATE UNIQUE CLUSTERED INDEX Test1PrimaryKey ON dbo.Test1 (Id ASC);
CREATE UNIQUE NONCLUSTERED INDEX TestIndex ON dbo.Test1 (MyKey);
GO

SELECT TOP 10000
       IDENTITY(INT, 1, 1) AS n
INTO #Tally
FROM master.dbo.syscolumns AS scl,
     master.dbo.syscolumns AS sc2;

INSERT INTO dbo.Test1 (MyKey,
                       MyValue)
```

```
SELECT TOP 10000
       'UniqueKey' + CAST(n AS VARCHAR),
       'Description'
FROM #Tally;

DROP TABLE #Tally;


SELECT t.MyValue
FROM dbo.Test1 AS t
WHERE t.MyKey = 'UniqueKey333';

SELECT t.MyValue
FROM dbo.Test1 AS t
WHERE t.MyKey = N'UniqueKey333';
```

After creating the table `Test1`, creating a couple of indexes on it, and placing some data, two queries are defined. Both queries return the same result set. As you can see, both queries are identical except for the data type of the variable equated to the `MyKey` column. Since this column is `VARCHAR`, the first query doesn't require an implicit data type conversion. The second query uses a different data type from that of the `MyKey` column, requiring an implicit data type conversion and thereby adding overhead to the query performance. Figure 20-1 shows the execution plans for both queries.

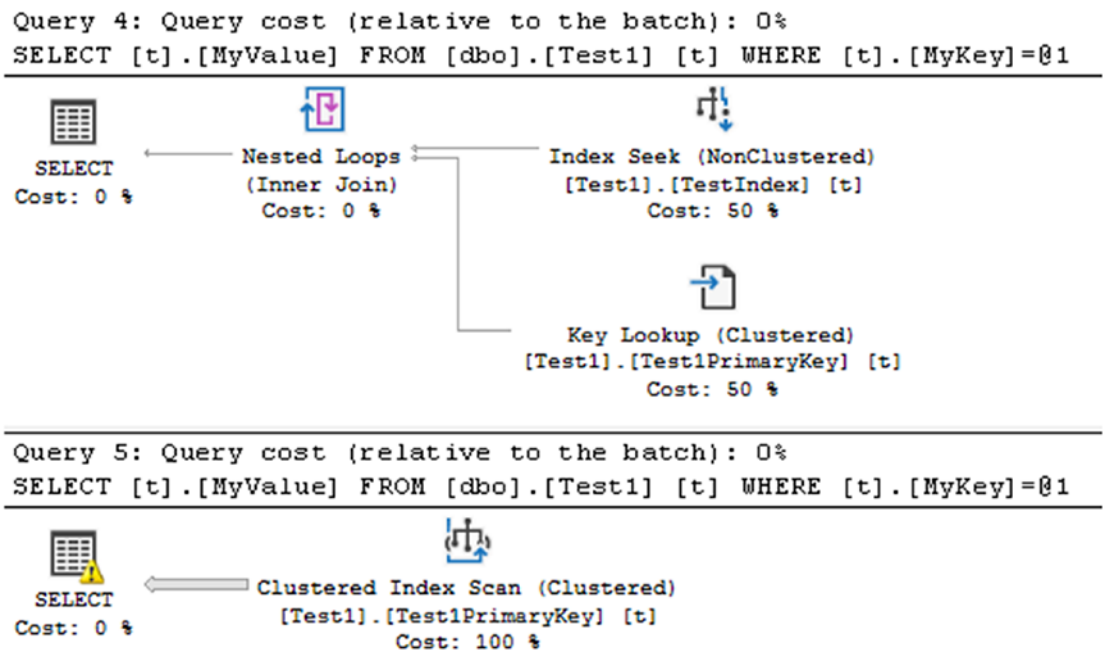

Figure 20-1. *Plans for a query with and without implicit data type conversion*

The complexity of the implicit data type conversion depends on the precedence of the data types involved in the comparison. The data type precedence rules of SQL Server specify which data type is converted to the other. Usually, the data type of lower precedence is converted to the data type of higher precedence. For example, the `TINYINT` data type has a lower precedence than the `INT` data type. For a complete list of data type precedence in SQL Server, please refer to the MSDN article "Data Type Precedence" (`http://bit.ly/1cN7AYc`). For further information about which data type can implicitly convert to which data type, refer to the MSDN article "Data Type Conversion" (`http://bit.ly/1j7kIJf`).

Note the warning icon on the `SELECT` operator. It's letting you know that there's something questionable in this query. In this case, it's the fact that there is a data type conversion operation. The optimizer lets you know that this might negatively affect its ability to find and use an index to assist the performance of the query. This can also be a false positive. If there are conversions on columns that are not used in any of the predicates, it really doesn't matter at all that an implicit, or even an explicit, conversion has occurred.

To see the specific process in place, look to the properties of the `Clustered Index Scan` operator and the `Predicate` value. Mine is listed as follows:

```
CONVERT_IMPLICIT(nvarchar(50),[AdventureWorks2017].[dbo].[Test1].[MyKey] as
[t].[MyKey],0)=[@1]
```

The duration went from about 110 microseconds on average to 1,400 microseconds, and the reads went from 4 to 56.

When SQL Server compares a column value with a certain data type and a variable (or constant) with a different data type, the data type of the variable (or constant) is always converted to the data type of the column. This is done because the column value is accessed based on the implicit conversion value of the variable (or constant). Therefore, in such cases, the implicit conversion is always applied on the variable (or constant).

As you can see, implicit data type conversion adds overhead to the query performance both in terms of a poor execution plan and in added CPU cost to make the conversions. Therefore, to improve performance, always use the same data type for both expressions.

Use EXISTS over COUNT(*) to Verify Data Existence

A common database requirement is to verify whether a set of data exists. Usually you'll see this implemented using a batch of SQL queries, as follows:

```
DECLARE @n INT;
SELECT @n = COUNT(*)
FROM Sales.SalesOrderDetail AS sod
WHERE sod.OrderQty = 1;
IF @n > 0
    PRINT 'Record Exists';
```

Using `COUNT(*)` to verify the existence of data is highly resource-intensive because `COUNT(*)` has to scan all the rows in a table. `EXISTS` merely has to scan and stop at the first record that matches the `EXISTS` criterion. To improve performance, use `EXISTS` instead of the `COUNT(*)` approach.

```
IF EXISTS (   SELECT sod.*
              FROM Sales.SalesOrderDetail AS sod
              WHERE sod.OrderQty = 1)
    PRINT 'Record Exists';
```

The performance benefit of the EXISTS technique over the COUNT(*) technique can be compared using the query performance metrics, as well as the execution plan in Figure 20-2, as you can see from the output of running these queries.

```
COUNT Duration: 8.9ms
Reads: 1248
EXISTS Duration: 1.7ms
Reads: 17
```

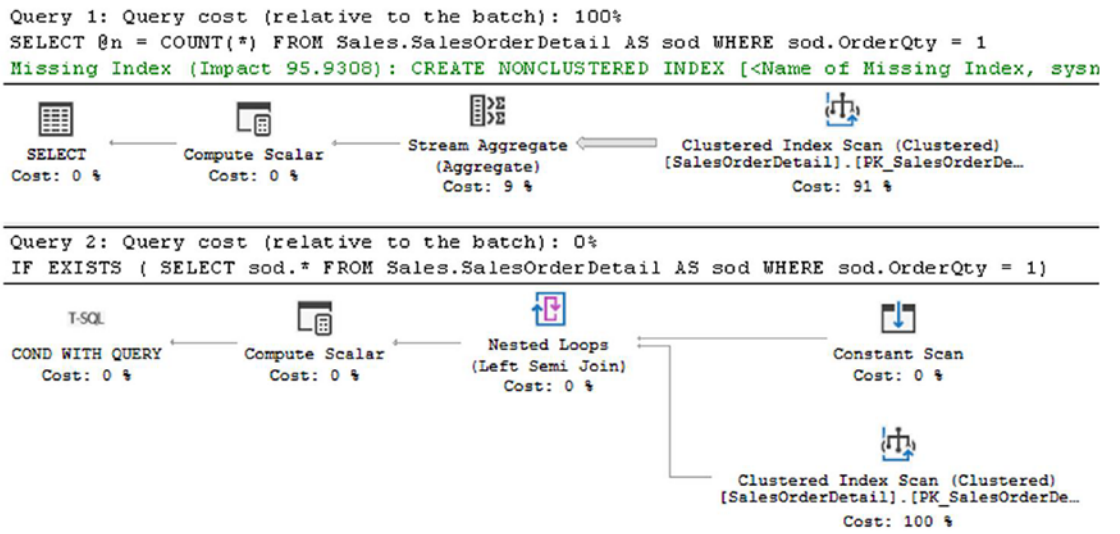

Figure 20-2. *Difference between COUNT and EXISTS*

As you can see, the EXISTS technique used only 17 logical reads compared to the 1,246 used by the COUNT(*) technique, and the execution time went from 8.9ms to 1.7ms. Therefore, to determine whether data exists, use the EXISTS technique.

Use UNION ALL Instead of UNION

You can concatenate the result set of multiple SELECT statements using the UNION clause as follows, as shown in Figure 20-3:

```
SELECT sod.ProductID,
       sod.SalesOrderID
FROM Sales.SalesOrderDetail AS sod
```

```
WHERE sod.ProductID = 934
UNION
SELECT sod.ProductID,
       sod.SalesOrderID
FROM Sales.SalesOrderDetail AS sod
WHERE sod.ProductID = 932
UNION
SELECT sod.ProductID,
       sod.SalesOrderID
FROM Sales.SalesOrderDetail AS sod
WHERE sod.ProductID = 708;
```

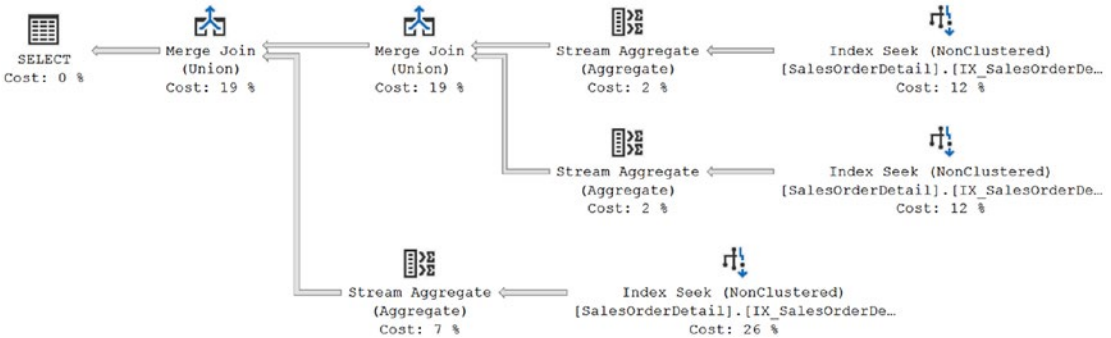

Figure 20-3. *The execution plan of the query using the UNION clause*

The `UNION` clause processes the result set from the three `SELECT` statements, removing duplicates from the final result set and effectively running `DISTINCT` on each query, using the `Stream Aggregate` to perform the aggregation. If the result sets of the `SELECT` statements participating in the `UNION` clause are exclusive to each other or you are allowed to have duplicate rows in the final result set, then use `UNION ALL` instead of `UNION`. This avoids the overhead of detecting and removing any duplicates and therefore improves performance, as shown in Figure 20-4.

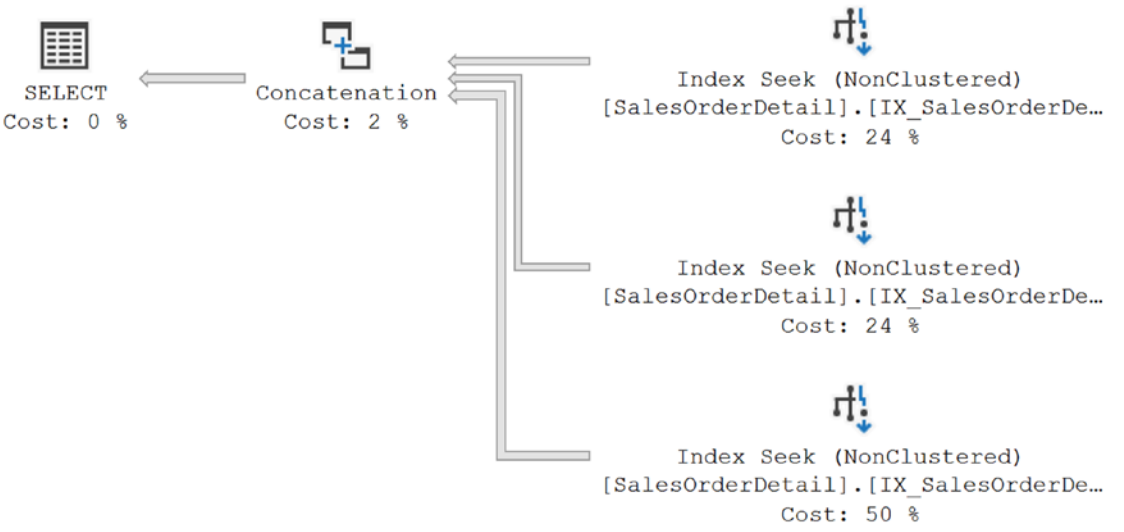

Figure 20-4. *The execution plan of the query using UNION ALL*

As you can see, in the first case (using `UNION`), the optimizer aggregated the records to eliminate the duplicates while using the `MERGE` to combine the result sets of the three `SELECT` statements. Since the result sets are exclusive to each other, you can use `UNION ALL` instead of the `UNION` clause. Using the `UNION ALL` clause avoids the overhead of detecting duplicates and joining the data and thereby improves performance.

The query performance metrics tell a similar story going from 125ms on the `UNION` query to 95ms on the `UNION ALL` query. Interestingly enough, the reads are the same at 20. It's the different processing needed for one query above and beyond that needed for the other query that makes a difference in performance in this case.

Use Indexes for Aggregate and Sort Conditions

Generally, aggregate functions such as `MIN` and `MAX` benefit from indexes on the corresponding column. They benefit even more from columnstore indexes as was demonstrated in earlier chapters. However, even standard indexes can assist with some aggregate queries. Without any index of either type on the columns, the optimizer has to scan the base table (or the rowstore clustered index), retrieve all the rows, and perform a

stream aggregate on the group (containing all rows) to identify the MIN/MAX value, as shown in the following example (see Figure 20-5):

```
SELECT  MIN(sod.UnitPrice)
FROM    Sales.SalesOrderDetail AS sod;
```

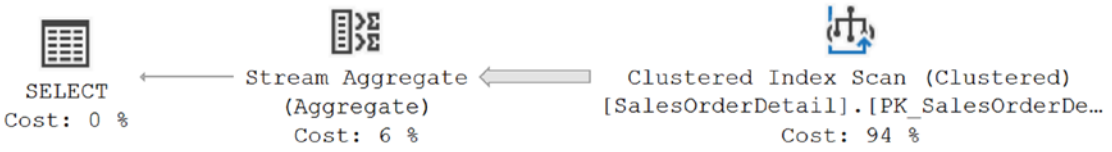

Figure 20-5. *A scan of the entire table filtered to a single row*

The performance metrics of the SELECT statement using the MIN aggregate function are as follows:

```
Duration: 15.8ms
Reads: 1248
```

The query performed more than 1,200 logical reads just to retrieve the row containing the minimum value for the UnitPrice column. You can see this represented in the execution plan in Figure 20-5. A huge fat row comes out of the Clustered Index Scan operation only to be filtered to a single row by the Stream Aggregate operation. If you create an index on the UnitPrice column, then the UnitPrice values will be presorted by the index in the leaf pages.

```
CREATE INDEX TestIndex ON Sales.SalesOrderDetail (UnitPrice ASC);
```

The index on the UnitPrice column improves the performance of the MIN aggregate function significantly. The optimizer can retrieve the minimum UnitPrice value by seeking to the topmost row in the index. This reduces the number of logical reads for the query, as shown in the corresponding metrics and execution plan (see Figure 20-6).

```
Duration: 97 mcs
Reads: 3
```

Figure 20-6. *An index radically improves performance*

Similarly, creating an index on the columns referred to in an ORDER BY clause helps the optimizer organize the result set fast because the column values are prearranged in the index. The internal implementation of the GROUP BY clause also sorts the column values first because sorted column values allow the adjacent matching values to be grouped quickly. Therefore, like the ORDER BY clause, the GROUP BY clause also benefits from having the values of the columns referred to in the GROUP BY clause sorted in advance.

Just to repeat, for most aggregate queries, a columnstore index will likely result in even better performance than a regular rowstore index. However, in some circumstances, a columnstore index could be a waste of resources, so it's good to know there may be options, depending on the query and your structures.

Be Cautious with Local Variables in a Batch Query

Often, multiple queries are submitted together as a batch, avoiding multiple network round-trips. It's common to use local variables in a query batch to pass a value between the individual queries. However, using local variables in the WHERE clause of a query in a batch doesn't allow the optimizer to generate an efficient execution plan in all cases.

To understand how the use of a local variable in the WHERE clause of a query in a batch can affect performance, consider the following batch query:

```
DECLARE @Id INT = 67260;
SELECT  p.Name,
            p.ProductNumber,
            th.ReferenceOrderID
    FROM    Production.Product AS p
    JOIN    Production.TransactionHistory AS th
            ON th.ProductID = p.ProductID
    WHERE   th.ReferenceOrderID = @Id;
```

Figure 20-7 shows the execution plan of this SELECT statement.

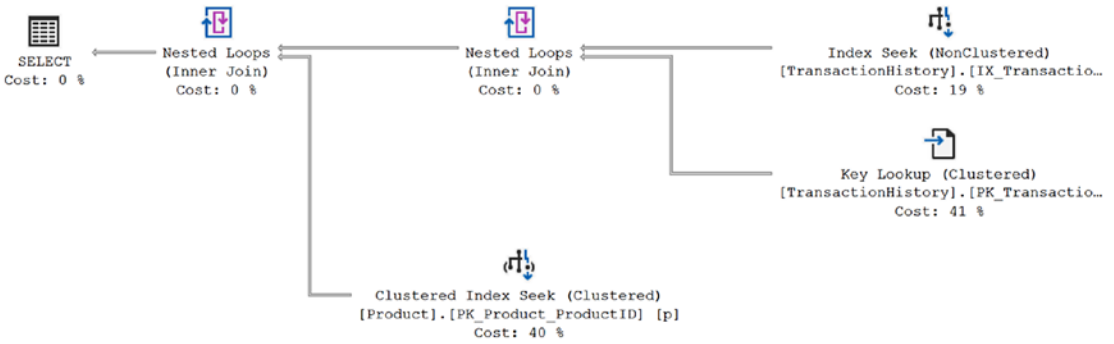

Figure 20-7. *Execution plan showing the effect of a local variable in a batch query*

As you can see, an Index Seek operation is performed to access the rows from the Production.TransactionHistory primary key. A Key Lookup against the clustered index is necessary through the loops join. Finally, a Clustered Index Seek against the Product table adds to the result set through another loops join. If the SELECT statement is executed without using the local variable, by replacing the local variable value with an appropriate constant value as in the following query, the optimizer makes different choices:

```
SELECT  p.Name,
            p.ProductNumber,
            th.ReferenceOrderID
    FROM    Production.Product AS p
    JOIN    Production.TransactionHistory AS th
            ON th.ProductID = p.ProductID
    WHERE   th.ReferenceOrderID = 67260;
```

Figure 20-8 shows the result.

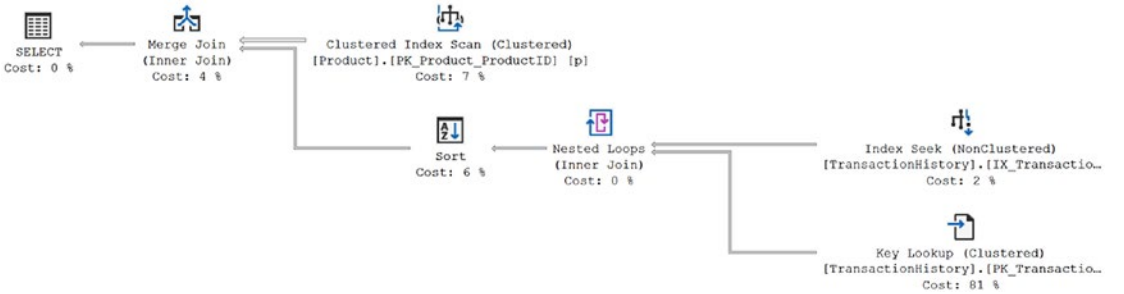

Figure 20-8. *Execution plan for the query when the local variable is not used*

You have a completely different execution plan. Parts of it are similar. You have the same `Index Seek` and `Key Lookup` operators, but their data is joined to a `Clustered Index Scan` and a `Merge Join`. Comparing plans quickly becomes problematic when considering performance, so let's look to the performance metrics to see whether there are differences. First, here's the information from the initial query with the local variable:

```
Duration: 696ms
Reads: 242
```

Then here's the second query, without the local variable:

```
Duration: 817ms
Reads: 197
```

The plan with the local variable results in somewhat faster execution, 696ms to 817ms, but, in quite a few more reads, 242 to 197. What causes the disparity between the plans and the differences in performance? It all comes down to the fact that a local variable, except in the event of a statement-level recompile, cannot be known to the operator. Therefore, instead of a specific count of the number of rows taken from values within the statistics, a calculated estimate is done based on the density graph.

So, there are 113,443 rows in the `TransactionHistory` table. The density value is 2.694111E-05. If we multiply them together, we arrive at the value 3.05628. Now, let's take a look at the execution plan estimated number of rows from the first execution plan (the one in Figure 20-8) to see the estimated number of rows.