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

www.EBooksWorld.ir

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;
```


www.EBooksWorld.ir

Figure 19-16 shows the execution plan.

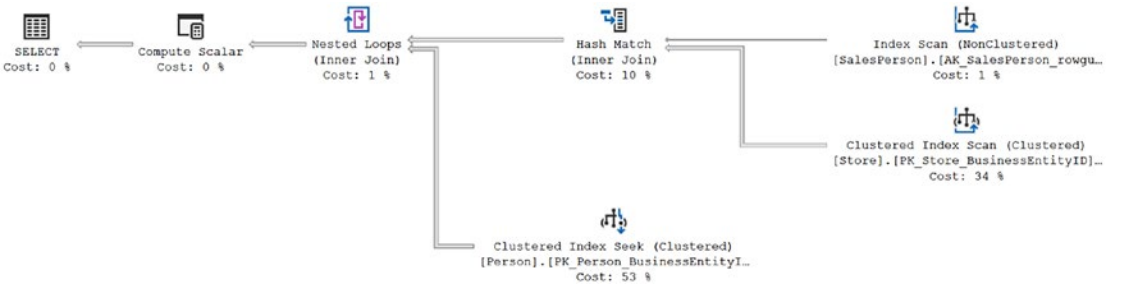

Figure 19-16. *Execution plan showing choices made by the optimizer*

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);
```


www.EBooksWorld.ir

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

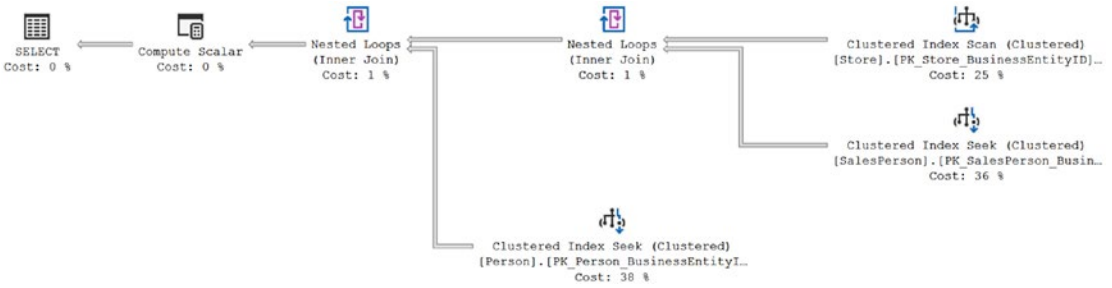

Figure 19-17. *Changes made by using the JOIN query hint*

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;
```


www.EBooksWorld.ir

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

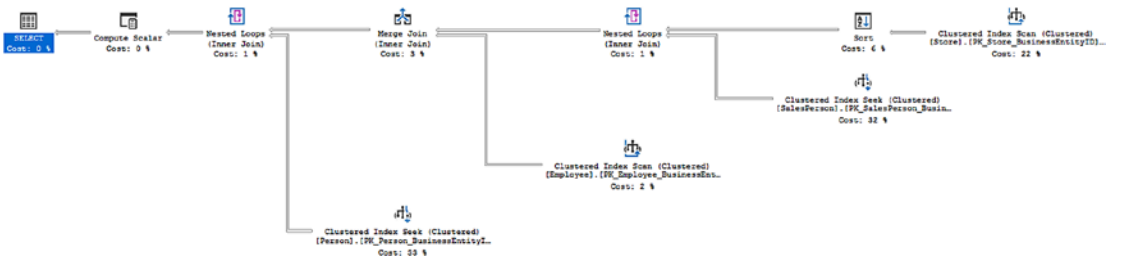

Figure 19-18. *More changes from using the LOOP join hint*

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.

www.EBooksWorld.ir

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
  ```


www.EBooksWorld.ir

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.

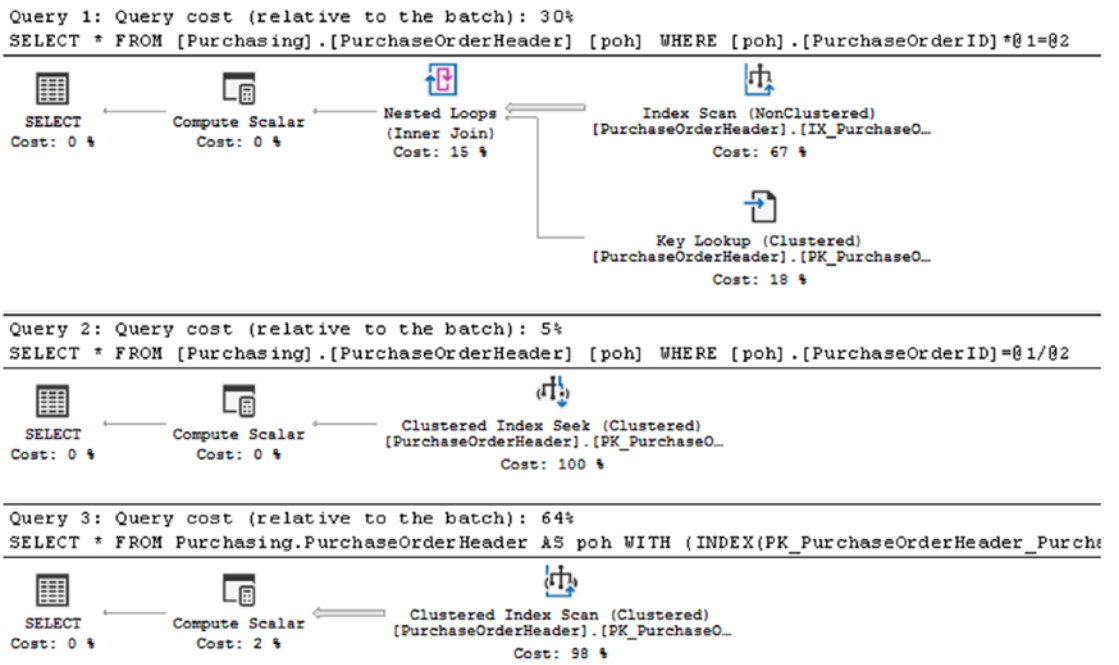

Figure 19-19. *Cost of a query with and without different INDEX hints*

www.EBooksWorld.ir

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.

www.EBooksWorld.ir

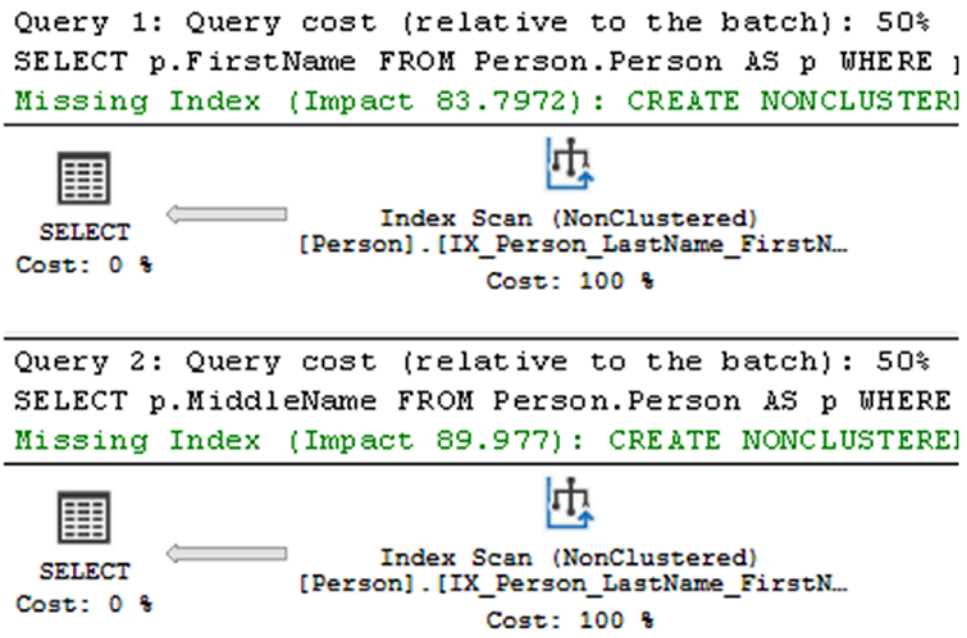

Figure 19-20. *Table scans caused by a lack of indexes*

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';
```


www.EBooksWorld.ir

```
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.

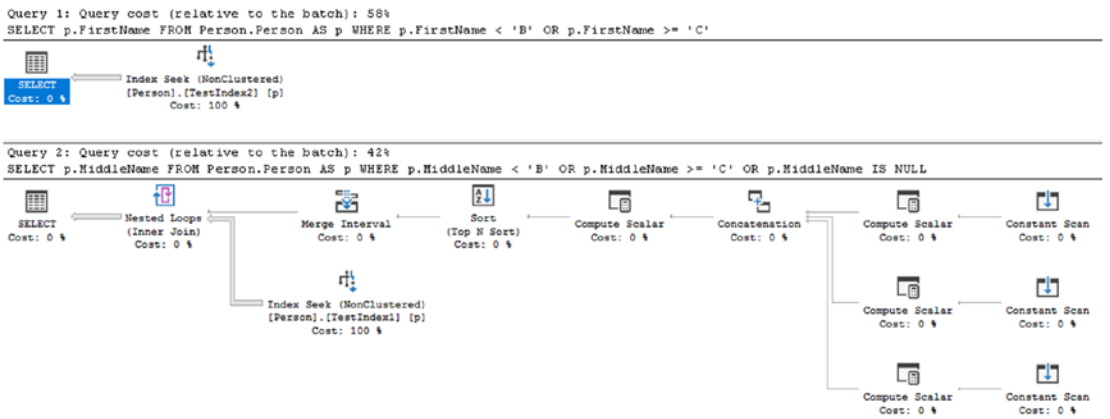

Figure 19-22. *Effect of the IS NULL option being used*

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

www.EBooksWorld.ir

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

www.EBooksWorld.ir

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

www.EBooksWorld.ir

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.

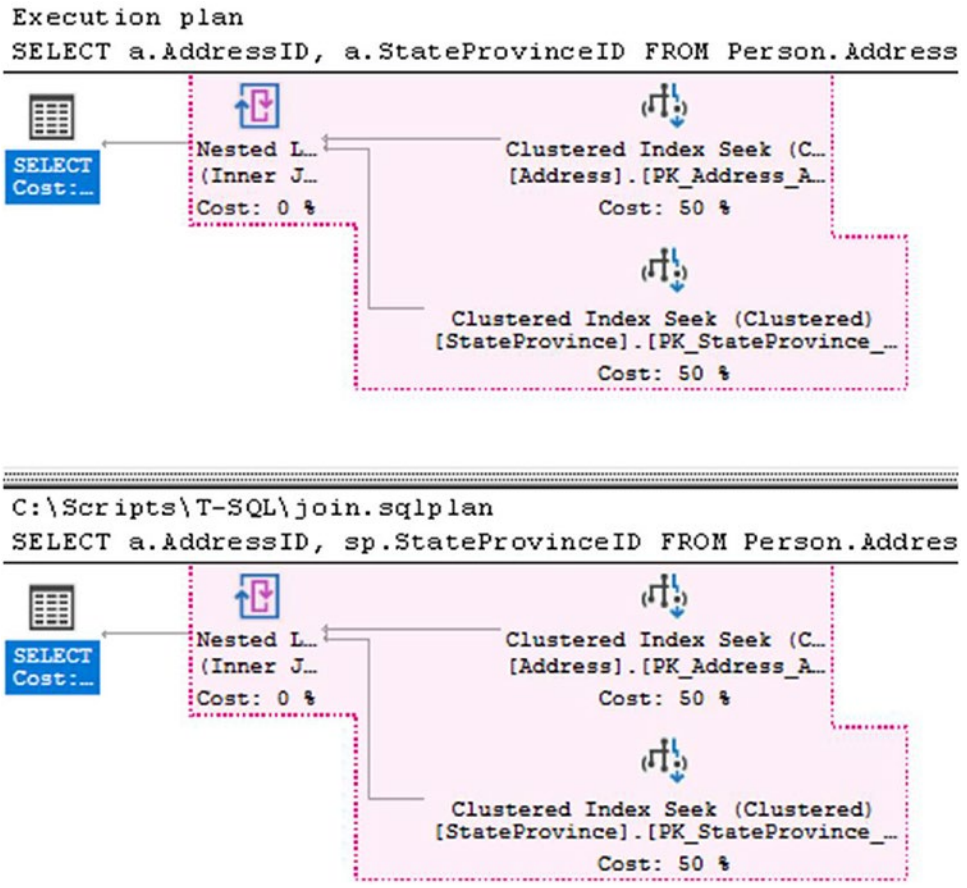

Figure 19-23. *Execution plan when DRI is not defined between the two tables*

www.EBooksWorld.ir

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.

www.EBooksWorld.ir

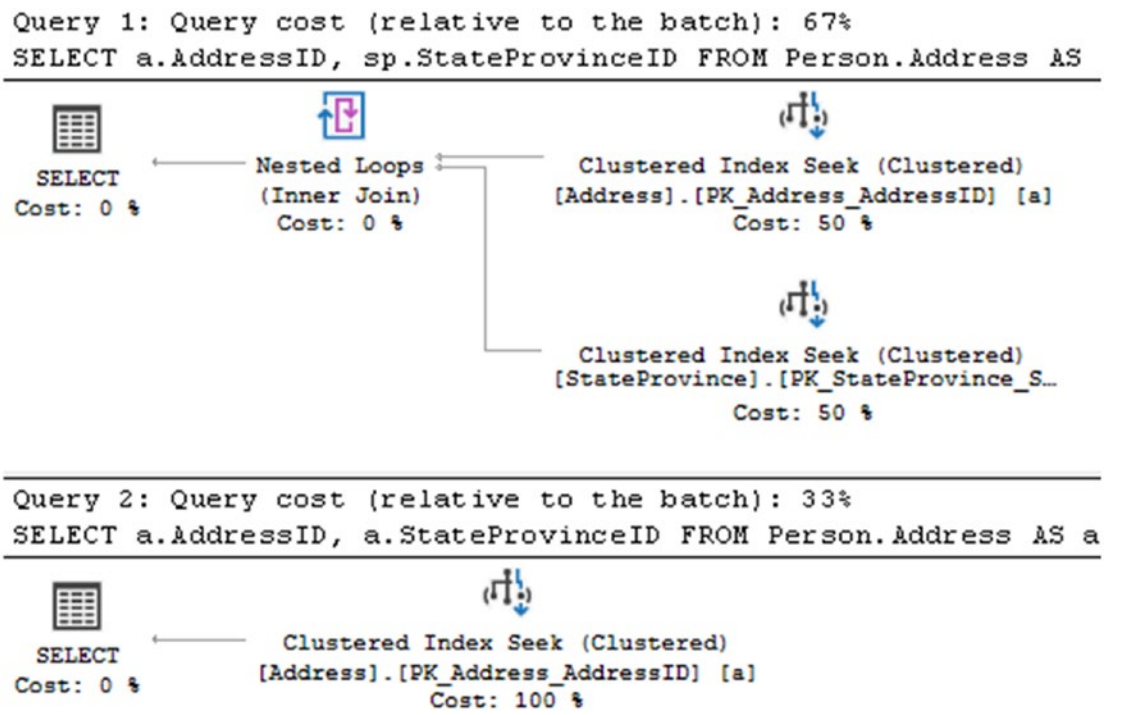

Figure 19-24. *Execution plans showing the benefit of defining DRI between the two tables*

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

www.EBooksWorld.ir

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.

www.EBooksWorld.ir

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`.

© Grant Fritchey 2018
G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_20

www.EBooksWorld.ir

- 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)
```


www.EBooksWorld.ir

```
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.

www.EBooksWorld.ir

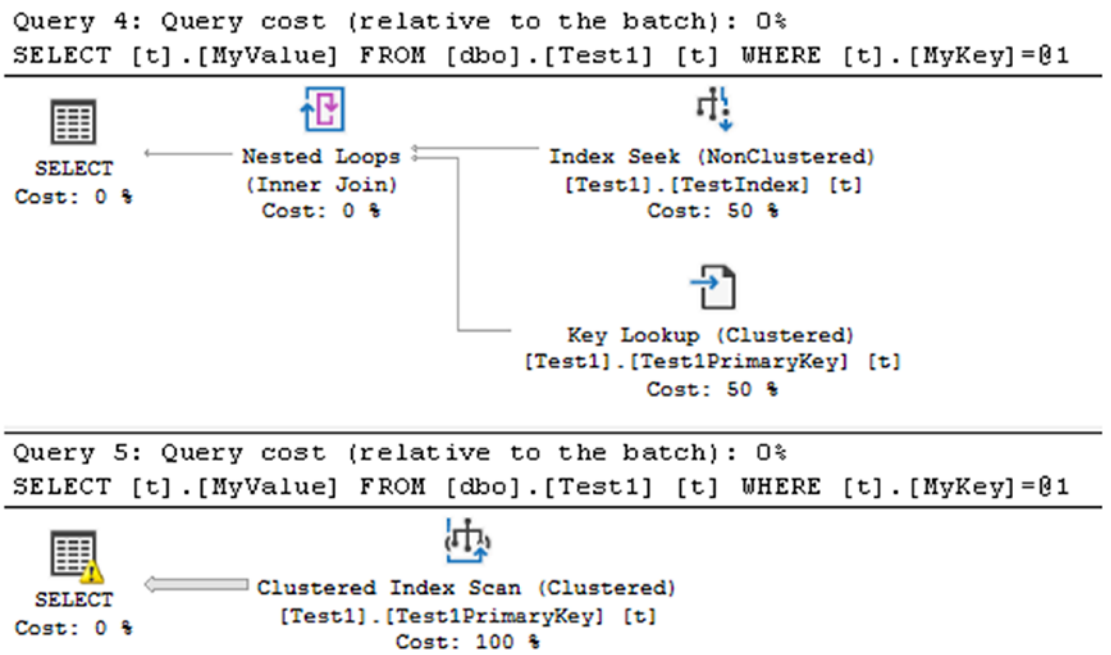

Figure 20-1. *Plans for a query with and without implicit data type conversion*

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.

www.EBooksWorld.ir

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';
```


www.EBooksWorld.ir

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
```

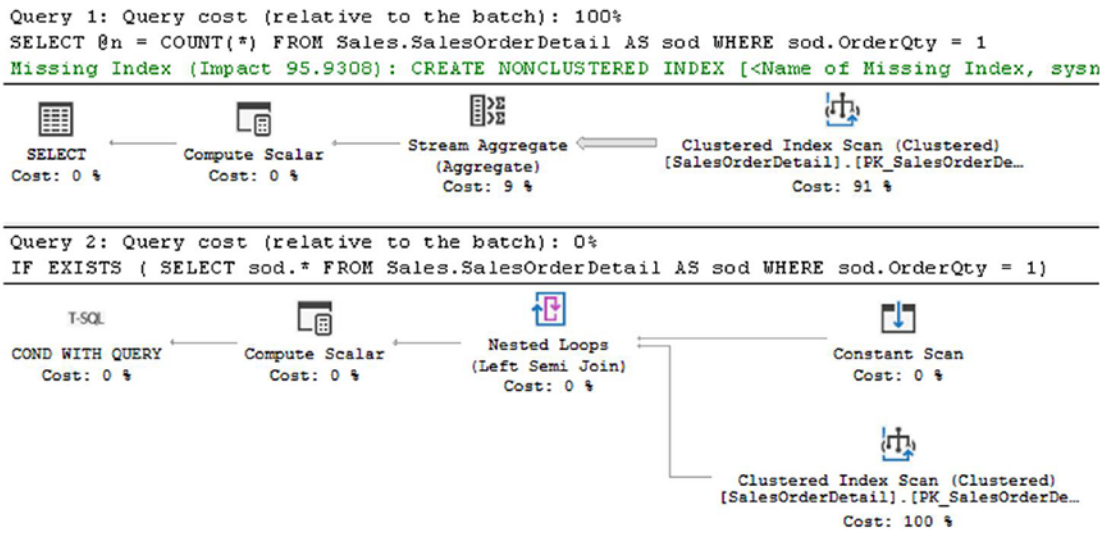

Figure 20-2. *Difference between COUNT and EXISTS*

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
```


www.EBooksWorld.ir

```
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;
```

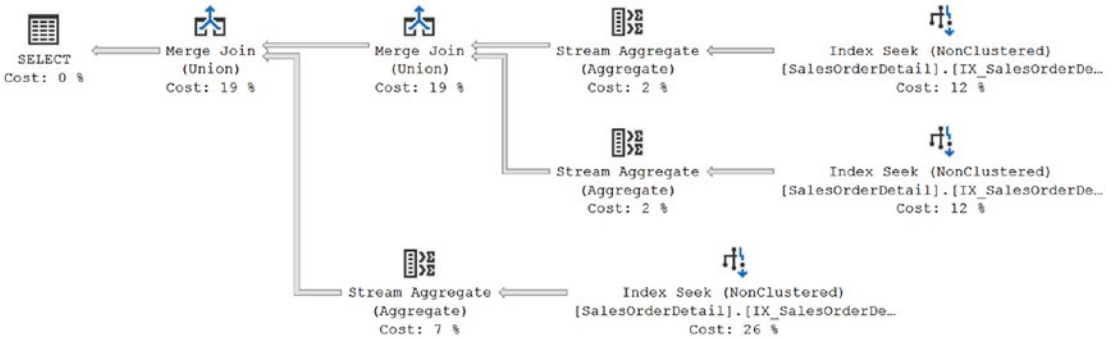

Figure 20-3. *The execution plan of the query using the UNION clause*

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.

www.EBooksWorld.ir

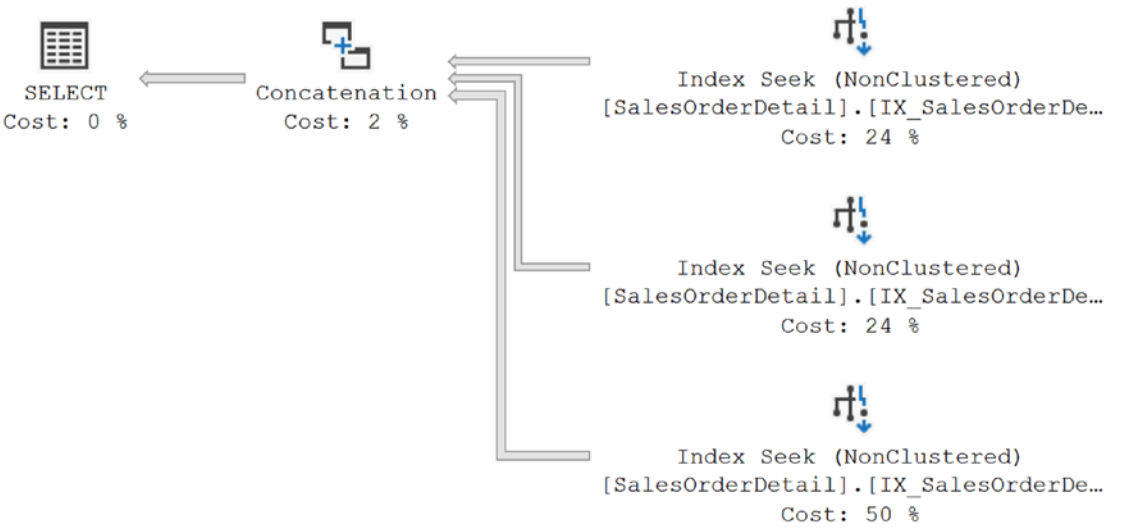

Figure 20-4. *The execution plan of the query using UNION ALL*

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

www.EBooksWorld.ir

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;
```

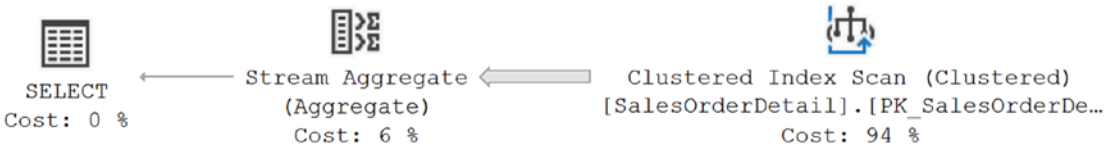

Figure 20-5. *A scan of the entire table filtered to a single row*

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*

www.EBooksWorld.ir

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;
```


www.EBooksWorld.ir

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

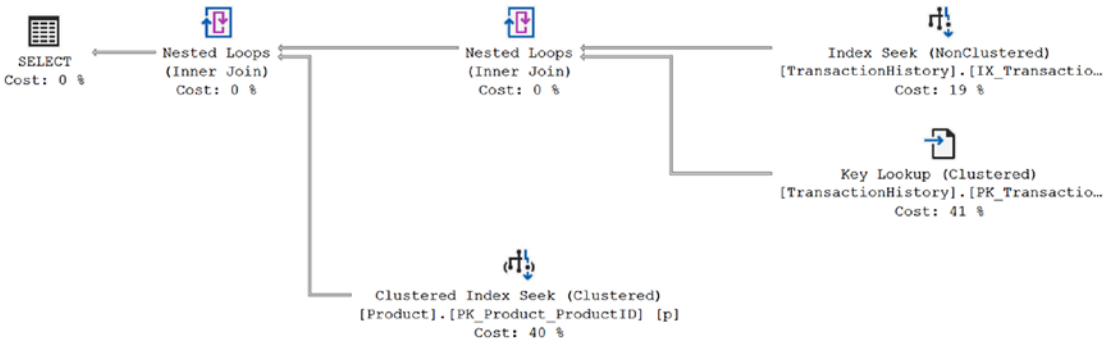

Figure 20-7. *Execution plan showing the effect of a local variable in a batch query*

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;
```


www.EBooksWorld.ir

Figure 20-8 shows the result.

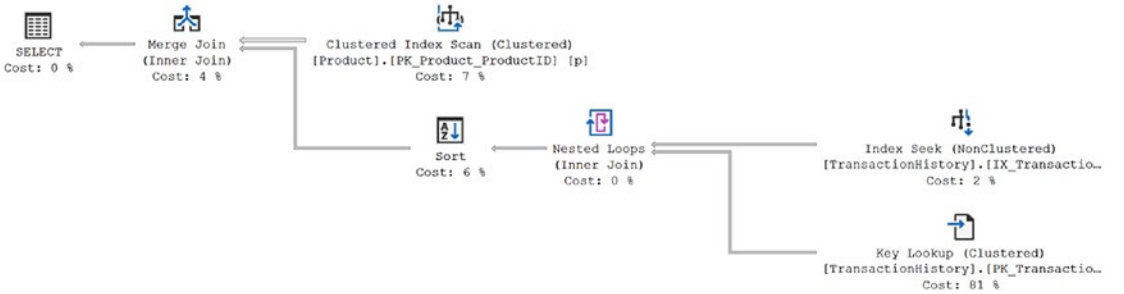

Figure 20-8. *Execution plan for the query when the local variable is not used*

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.

www.EBooksWorld.ir

The estimated number of rows, at the bottom of Figure 20-9, is 3.05628. It's exactly the same as the calculation. Note, though, that the actual number of rows, at the top of Figure 20-9, is 48. This becomes important. If we look at the same properties on the same operator in the second plan, the one in Figure 20-8, we'll see that the estimated and actual number of rows are identical at 48. In this case, the optimizer decided that 48 rows returned was too many to be able to perform well through an `Index Seek` against the `Product` table. Instead, it opted to use an ordered scan (which you can verify through the properties of the `Index Scan` operator) and then a merge join.

Actual Number of Rows	48
Actual Rebinds	0
Actual Rewinds	0
Actual Time Statistics	
Defined Values	[Adventur
Description	Scan a par
Estimated CPU Cost	0.0001604
Estimated Execution Mode	Row
Estimated I/O Cost	0.003125
Estimated Number of Executions	1
Estimated Number of Rows	3.05628

Figure 20-9. *Estimated versus actual number of rows*

In point of fact, the first plan was faster; however, it did result in higher I/O output. This is where we have to exercise caution. In this case, the performance was a little better, but if the system was under load, especially if it was under I/O strain, then the second plan is likely to perform faster, with fewer contentions on resources, since it has a lower number of reads overall. The caution comes from identifying which of these plans is better in particular circumstances.

To avoid this potential performance problem, use the following approach. Don't use a local variable as a filter criterion in a batch for a query like this. A local variable

www.EBooksWorld.ir

is different from a parameter value, as demonstrated in Chapter 17. Create a stored procedure for the batch and execute it as follows:

```
CREATE OR ALTER PROCEDURE ProductDetails (@id INT)
AS
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;
GO

EXEC ProductDetails @id = 1;
```

This approach can backfire. The process of using the values passed to a parameter is referred to as *parameter sniffing*. Parameter sniffing occurs for all stored procedures and parameterized queries automatically. Depending on the accuracy of the statistics and the values passed to the parameters, it is possible to get a bad plan using specific values and a good plan using the sampled values that occur when you have a local variable. Testing is the only way to be sure which will work best in any given situation. However, in most circumstances, you're better off having accurate values rather than sampled ones. For more details on parameter sniffing, see Chapter 17.

As a general guideline, it's best to avoid hard-coding values. If the values have to change, you may have to change them in a lot of code. If you do need to code values within your queries, local variables let you control them from a single location at the top of the batch, making the management of the code easier. However, local variables, as we've just seen, when used for data retrieval can affect plan choice. In that case, parameter values are preferred. You can even set the parameter value and provide it with a default value. These will still be sniffed as regular parameters.

www.EBooksWorld.ir

Be Careful When Naming Stored Procedures

The name of a stored procedure does matter. You should not name your procedures with a prefix of sp_. Developers often prefix their stored procedures with sp_ so that they can easily identify the stored procedures. However, SQL Server assumes that any stored procedure with this exact prefix is probably a system stored procedure, whose home is in the master database. When a stored procedure with an sp_ prefix is submitted for execution, SQL Server looks for the stored procedure in the following places in the following order:

- In the master database
- In the current database based on any qualifiers provided (database name or owner)
- In the current database using dbo as the schema, if a schema is not specified

Therefore, although the user-created stored procedure prefixed with sp_ exists in the current database, the master database is checked first. This happens even when the stored procedure is qualified with the database name.

To understand the effect of prefixing sp_ to a stored procedure name, consider the following stored procedure:

```
IF EXISTS ( SELECT  *
            FROM    sys.objects
            WHERE   object_id = OBJECT_ID(N'[dbo].[sp_Dont]')
                    AND type IN (N'P', N'PC') )
    DROP PROCEDURE  [dbo].[sp_Dont]
GO
CREATE PROC [sp_Dont]
AS
PRINT 'Done!'
GO
--Add plan of sp_Dont to procedure cache
EXEC AdventureWorks2017.dbo.[sp_Dont] ;
GO
--Use the above cached plan of sp_Dont
EXEC AdventureWorks2012.dbo.[sp_Dont] ;
GO
```


www.EBooksWorld.ir

The first execution of the stored procedure adds the execution plan of the stored procedure to the procedure cache. A subsequent execution of the stored procedure reuses the existing plan from the procedure cache unless a recompilation of the plan is required (the causes of stored procedure recompilation are explained in Chapter 10). Therefore, the second execution of the stored procedure `spDont` shown in Figure 20-10 should find a plan in the procedure cache. This is indicated by an `SP:CacheHit` event in the corresponding Extended Events output.

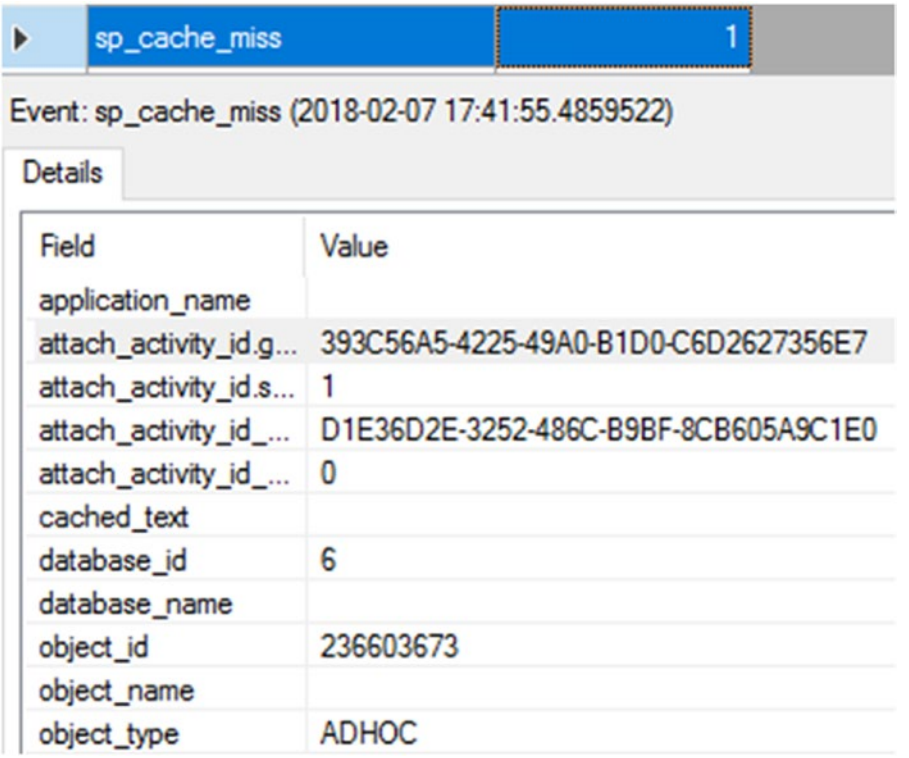

Figure 20-10. *Extended Events output showing the effect of the sp_ prefix on a stored procedure name*

Note that an `SP:CacheMiss` event is fired before SQL Server tries to locate the plan for the stored procedure in the procedure cache. The `SP:CacheMiss` event is caused by SQL Server looking in the master database for the stored procedure, even though the execution of the stored procedure is properly qualified with the user database name.

This aspect of the `sp_` prefix becomes more interesting when you create a stored procedure with the name of an existing system stored procedure.

www.EBooksWorld.ir

```
CREATE OR ALTER PROC sp_addmessage @param1 NVARCHAR(25)
AS
PRINT  '@param1 =  '  + @param1 ;
GO

EXEC AdventureWorks2017.dbo.[sp_addmessage]   'AdventureWorks';
```

The execution of this user-defined stored procedure causes the execution of the system stored procedure `sp_addmessage` from the `master` database instead, as you can see in Figure 20-11.

```
Msg 8114, Level 16, State 5, Procedure sp_addmessage, Line 4009
Error converting data type varchar to int.
```

Figure 20-11. *Execution result for stored procedure showing the effect of the sp_ prefix on a stored procedure name*

Unfortunately, it is not possible to execute this user-defined stored procedure. You can see now why you should not prefix a user-defined stored procedure's name with `sp_`. Use some other naming convention. From a pure performance standpoint, this is a trivial improvement. However, if you have high volume and response time is critical, it is one more small point in your favor if you avoid the `sp_` naming standard.

Reducing the Number of Network Round-Trips

Database applications often execute multiple queries to implement a database operation. Besides optimizing the performance of the individual query, it is important that you optimize the performance of the batch. To reduce the overhead of multiple network round-trips, consider the following techniques:

- Execute multiple queries together.
- Use `SET NOCOUNT`.

Let's look at these techniques in a little more depth.

www.EBooksWorld.ir

Execute Multiple Queries Together

It is preferable to submit all the queries of a set together as a batch or a stored procedure. Besides reducing the network round-trips between the database application and the server, stored procedures also provide multiple performance and administrative benefits, as described in Chapter 16. This means the code in the application needs to be able to deal with multiple result sets. It also means your T-SQL code may need to deal with XML data or other large sets of data, not single-row inserts or updates.

Use SET NOCOUNT

You need to consider one more factor when executing a batch or a stored procedure. After every query in the batch or the stored procedure is executed, the server reports the number of rows affected.

```
(<Number> row(s) affected)
```

This information is returned to the database application and adds to the network overhead. Use the T-SQL statement `SET NOCOUNT` to avoid this overhead.

```
SET NOCOUNT ON <SQL queries> SET NOCOUNT OFF
```

Note that the `SET NOCOUNT` statement doesn't cause any recompilation issue with stored procedures, unlike some `SET` statements, as explained in Chapter 18.

Reducing the Transaction Cost

Every action query in SQL Server is performed as an *atomic* action so that the state of a database table moves from one *consistent* state to another. SQL Server does this automatically, and it can't be disabled. If the transition from one consistent state to another requires multiple database queries, then atomicity across the multiple queries should be maintained using explicitly defined database transactions. The old and new states of every atomic action are maintained in the transaction log (on the disk) to ensure *durability,* which guarantees that the outcome of an atomic action won't be lost once it completes successfully. An atomic action during its execution is *isolated* from other database actions using database locks.

www.EBooksWorld.ir

Based on the characteristics of a transaction, here are two broad recommendations to reduce the cost of the transaction:

- Reduce logging overhead.
- Reduce lock overhead.

Reduce Logging Overhead

A database query may consist of multiple data manipulation queries. If atomicity is maintained for each query separately, then a large number of disk writes are performed on the transaction log. Since disk activity is extremely slow compared to memory or CPU activity, the excessive disk activity can increase the execution time of the database functionality. For example, consider the following batch query:

```
--Create a test table
IF (SELECT  OBJECT_ID('dbo.Test1')
   ) IS NOT NULL
    DROP TABLE dbo.Test1;
GO
CREATE TABLE dbo.Test1 (C1 TINYINT);
GO

--Insert 10000 rows
DECLARE @Count INT = 1;
WHILE @Count <= 10000
    BEGIN
        INSERT  INTO dbo.Test1
                (C1)
        VALUES  (@Count % 256);
        SET @Count = @Count + 1;
    END
```

Since every execution of the INSERT statement is atomic in itself, SQL Server will write to the transaction log for every execution of the INSERT statement.

www.EBooksWorld.ir

An easy way to reduce the number of log disk writes is to include the action queries within an explicit transaction.

```
DECLARE @Count INT = 1;
DBCC SQLPERF(LOGSPACE);
BEGIN TRANSACTION
WHILE @Count <= 10000
    BEGIN
        INSERT  INTO dbo.Test1
                (C1)
        VALUES  (@Count % 256) ;
        SET @Count = @Count + 1 ;
        END
COMMIT
DBCC SQLPERF(LOGSPACE);
```

The defined transaction scope (between the `BEGIN TRANSACTION` and `COMMIT` pair of commands) expands the scope of atomicity to the multiple `INSERT` statements included within the transaction. This decreases the number of log disk writes and improves the performance of the database functionality. To test this theory, run the following T-SQL command before and after each of the `WHILE` loops:

```
DBCC SQLPERF(LOGSPACE);
```

This will show you the percentage of log space used. On running the first set of inserts on my database, the log went from 3.2 percent used to 3.3 percent. When running the second set of inserts, the log grew about 6 percent.

The best way is to work with sets of data rather than individual rows. A `WHILE` loop can be an inherently costly operation, like a cursor (more details on cursors in Chapter 23). So, running a query that avoids the `WHILE` loop and instead works from a set-based approach is even better.

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


www.EBooksWorld.ir

```
BEGIN TRANSACTION
INSERT INTO dbo.Test1 (C1)
SELECT TOP 1000
       (n % 256)
FROM #Tally AS t
COMMIT
```

Running this query with the `DBCC SQLPERF()` function before and after showed less than .01 percent growth of the used space within the log, and it ran in 41ms as compared to more than 2s for the `WHILE` loop.

One area of caution, however, is that by including too many data manipulation queries within a transaction, the duration of the transaction is increased. During that time, all other queries trying to access the resources referred to in the transaction are blocked. Rollback duration and recovery time during a restore increase because of long transactions.

Reduce Lock Overhead

By default, all four SQL statements (`SELECT`, `INSERT`, `UPDATE`, and `DELETE`) use database locks to isolate their work from that of other SQL statements. This lock management adds performance overhead to the query. The performance of a query can be improved by requesting fewer locks. By extension, the performance of other queries are also improved because they have to wait a shorter period of time to obtain their own locks.

By default, SQL Server can provide row-level locks. For a query working on a large number of rows, requesting a row lock on all the individual rows adds a significant overhead to the lock-management process. You can reduce this lock overhead by decreasing the lock granularity, say to the page level or table level. SQL Server performs the lock escalation dynamically by taking into consideration the lock overheads. Therefore, generally, it is not necessary to manually escalate the lock level. But, if required, you can control the concurrency of a query programmatically using lock hints as follows:

```
SELECT * FROM <TableName> WITH(PAGLOCK)  --Use page level lock
```


www.EBooksWorld.ir

Similarly, by default, SQL Server uses locks for SELECT statements besides those for INSERT, UPDATE, and DELETE statements. This allows the SELECT statements to read data that isn't being modified. In some cases, the data may be quite static, and it doesn't go through much modification. In such cases, you can reduce the lock overhead of the SELECT statements in one of the following ways:

- Mark the database as READONLY.

```
ALTER DATABASE <DatabaseName> SET READ_ONLY
```

  This allows users to retrieve data from the database, but it prevents them from modifying the data. The setting takes effect immediately. If occasional modifications to the database are required, then it may be temporarily converted to READWRITE mode.

```
ALTER DATABASE <DatabaseName> SET READ_WRITE
<Database modifications>
ALTER DATABASE <DatabaseName> SET READONLY
```

- Use one of the snapshot isolations.

  SQL Server provides a mechanism to put versions of data into tempdb as updates are occurring, radically reducing locking overhead and blocking for read operations. You can change the isolation level of the database by using an ALTER statement.

```
ALTER DATABASE AdventureWorks2017 SET READ_COMMITTED_SNAPSHOT ON;
```

- Prevent SELECT statements from requesting any lock.

```
SELECT * FROM <TableName> WITH(NOLOCK)
```

  This prevents the SELECT statement from requesting any lock, and it is applicable to SELECT statements only. Although the NOLOCK hint can't be used directly on the tables referred to in the action queries (INSERT, UPDATE, and DELETE), it may be used on the data retrieval part of the action queries, as shown here:

```
DELETE Sales.SalesOrderDetail
FROM Sales.SalesOrderDetail AS sod WITH (NOLOCK)
    JOIN Production.Product AS p WITH (NOLOCK)
        ON sod.ProductID = p.ProductID
           AND p.ProductID = 0;
```


www.EBooksWorld.ir

Just know that this leads to dirty reads, which can cause duplicate rows or missing rows and is therefore considered to be a last resort to control locking. In fact, this is considered to be quite dangerous and will lead to improper results. The best approach is to mark the database as read-only or use one of the snapshot isolation levels.

This is a huge topic, and a lot more can be said about it. I discuss the different types of lock requests and how to manage lock overhead in the next chapter. If you made any of the proposed changes to the database from this section, I recommend restoring from a backup.

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 are resource friendly and 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 work well with other queries without causing serious blocking issues. In the next chapter, you will look into the different blocking aspects of a database application.

www.EBooksWorld.ir

CHAPTER 21

Blocking and Blocked Processes

You would ideally like your database application to scale linearly with the number of database users and the volume of data. However, it is common to find that performance degrades as the number of users increases and as the volume of data grows. One cause for degradation, especially associated with ever-increasing scale, is blocking. In fact, database blocking is usually one of the biggest enemies of scalability for database applications.

In this chapter, I cover the following topics:

- The fundamentals of blocking in SQL Server
- The ACID properties of a transactional database
- Database lock granularity, escalation, modes, and compatibility
- ANSI isolation levels
- The effect of indexes on locking
- The information necessary to analyze blocking
- A SQL script to collect blocking information
- Resolutions and recommendations to avoid blocking
- Techniques to automate the blocking detection and information collection processes

© Grant Fritchey 2018
G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_21

www.EBooksWorld.ir

Blocking Fundamentals

In an ideal world, every SQL query would be able to execute concurrently, without any blocking by other queries. However, in the real world, queries *do* block each other, similar to the way a car crossing through a green traffic signal at an intersection blocks other cars waiting to cross the intersection. In SQL Server, this traffic management takes the form of the *lock manager,* which controls concurrent access to a database resource to maintain data consistency. The concurrent access to a database resource is controlled across multiple database connections.

I want to make sure things are clear before moving on. Three terms are used within databases that sound the same and are interrelated but have different meanings. These are frequently confused, and people often use the terms incorrectly and interchangeably. These terms are *locking, blocking,* and *deadlocking.* Locking is an integral part of the process of SQL Server managing multiple sessions. When a session needs access to a piece of data, a lock of some type is placed on it. This is different from blocking, which is when one session, or thread, needs access to a piece of data and has to wait for another session's lock to clear. Finally, deadlocking is when two sessions, or threads, form what is sometimes referred to as a *deadly embrace.* They are each waiting on the other for a lock to clear. Deadlocking could also be referred to as a permanent blocking situation, but it's one that won't resolve by waiting any period of time. Deadlocking will be covered in more detail in Chapter 22. So, locks can lead to blocks, and both locks and blocks play a part in deadlocks, but these are three distinct concepts. Please understand the differences between these terms and use them correctly. It will help in your understanding of the system, your ability to troubleshoot, and your ability to communicate with other database administrators and developers.

In SQL Server, a database connection is identified by a session ID. Connections may be from one or many applications and one or many users on those applications; as far as SQL Server is concerned, every connection is treated as a separate session. Blocking between two sessions accessing the same piece of data at the same time is a natural phenomenon in SQL Server. Whenever two sessions try to access a common database resource in conflicting ways, the lock manager ensures that the second session waits until the first session completes its work in conjunction with the management of transactions within the system. For example, a session might be modifying a table record while another session tries to delete the record. Since these two data access requests are incompatible, the second session will be blocked until the first session completes its task.

www.EBooksWorld.ir

On the other hand, if the two sessions try to read a table concurrently, both requests are allowed to execute without blocking, since these data access requests are compatible with each other.

Usually, the effect of blocking on a session is quite small and doesn't affect its performance noticeably. At times, however, because of poor query and/or transaction design (or maybe bad luck), blocking can affect query performance significantly. In a database application, every effort should be made to minimize blocking and thereby increase the number of concurrent users who can use the database.

With the introduction of in-memory tables in SQL Server 2014, locking, at least for these tables, takes on whole new dimensions. I'll cover their behavior separately in Chapter 24.

Understanding Blocking

In SQL Server, a database query can execute as a logical unit of work in itself, or it can participate in a bigger logical unit of work. A bigger logical unit of work can be defined using the `BEGIN TRANSACTION` statement along with `COMMIT` and/or `ROLLBACK` statements. Every logical unit of work must conform to a set of four properties called *ACID* properties:

- Atomicity
- Consistency
- Isolation
- Durability

I cover these properties in the sections that follow because understanding how transactions work is fundamental to understanding blocking.

Atomicity

A logical unit of work must be *atomic*. That is, either all the actions of the logical unit of work are completed or no effect is retained. To understand the atomicity of a logical unit of work, consider the following example:

```
USE AdventureWorks2017;
GO
DROP TABLE IF EXISTS dbo.ProductTest;
GO
```


www.EBooksWorld.ir

```
CREATE TABLE dbo.ProductTest (ProductID INT
                                    CONSTRAINT ValueEqualsOne CHECK
(ProductID = 1));
GO
--All ProductIDs are added into ProductTest as a logical unit of work
INSERT INTO dbo.ProductTest
SELECT p.ProductID
FROM Production.Product AS p;
GO
SELECT pt.ProductID
FROM dbo.ProductTest AS pt; --Returns 0 rows
```

SQL Server treats the preceding `INSERT` statement as a logical unit of work. The `CHECK` constraint on column `ProductID` of the `dbo.ProductTest` table allows only the value of `1`. Although the `ProductID` column in the `Production.Product` table starts with the value of `1`, it also contains other values. For this reason, the `INSERT` statement won't add any records at all to the `dbo.ProductTest` table, and an error is raised because of the `CHECK` constraint. Thus, atomicity is automatically ensured by SQL Server.

So far, so good. But in the case of a bigger logical unit of work, you should be aware of an interesting behavior of SQL Server. Imagine that the previous insert task consists of multiple `INSERT` statements. These can be combined to form a bigger logical unit of work, as follows:

```
BEGIN TRAN
 --Start:  Logical unit of work
--First:
INSERT  INTO dbo.ProductTest
        SELECT  p.ProductID
        FROM    Production.Product AS p;
--Second:
INSERT  INTO dbo.ProductTest
VALUES  (1);
COMMIT --End:   Logical unit of work
GO
```


www.EBooksWorld.ir

With the dbo.ProductTest table already created in the preceding script, the BEGIN TRAN and COMMIT pair of statements defines a logical unit of work, suggesting that all the statements within the transaction should be atomic in nature. However, the default behavior of SQL Server doesn't ensure that the failure of one of the statements within a user-defined transaction scope will undo the effect of the prior statements. In the preceding transaction, the first INSERT statement will fail as explained earlier, whereas the second INSERT is perfectly fine. The default behavior of SQL Server allows the second INSERT statement to execute, even though the first INSERT statement fails. A SELECT statement, as shown in the following code, will return the row inserted by the second INSERT statement:

```
SELECT  *
FROM    dbo.ProductTest; --Returns a row with t1.c1 = 1
```

The atomicity of a user-defined transaction can be ensured in the following two ways:

- SET XACT_ABORT ON
- Explicit rollback

Let's look at these briefly.

SET XACT_ABORT ON

You can modify the atomicity of the INSERT task in the preceding section using the SET XACT_ ABORT ON statement.

```
SET XACT_ABORT ON;
GO
BEGIN TRAN
 --Start:  Logical unit of work
--First:
INSERT  INTO dbo.ProductTest
        SELECT  p.ProductID
        FROM    Production.Product AS p;
--Second:
INSERT  INTO dbo.ProductTest
VALUES  (1);
COMMIT
```


www.EBooksWorld.ir

```
 --End:   Logical unit of work GO
SET XACT_ABORT OFF;
GO
```

The SET XACT_ABORT statement specifies whether SQL Server should automatically roll back and abort an entire transaction when a statement within the transaction fails. The failure of the first INSERT statement will automatically suspend the entire transaction, and thus the second INSERT statement will not be executed. The effect of SET XACT_ABORT is at the connection level, and it remains applicable until it is reconfigured or the connection is closed. By default, SET XACT_ABORT is OFF.

Explicit Rollback

You can also manage the atomicity of a user-defined transaction by using the TRY/CATCH error-trapping mechanism within SQL Server. If a statement within the TRY block of code generates an error, then the CATCH block of code will handle the error. If an error occurs and the CATCH block is activated, then the entire work of a user-defined transaction can be rolled back, and further statements can be prevented from execution, as follows:

```
BEGIN TRY
    BEGIN TRAN
    --Start: Logical unit of work
    --First:
    INSERT INTO dbo.ProductTest
    SELECT p.ProductID
    FROM Production.Product AS p

    Second:
    INSERT INTO dbo.ProductTest (ProductID)
    VALUES (1)
    COMMIT --End: Logical unit of work
END TRY
BEGIN CATCH
    ROLLBACK
    PRINT 'An error occurred'
    RETURN
END CATCH
```


www.EBooksWorld.ir

The ROLLBACK statement rolls back all the actions performed in the transaction until that point. For a detailed description of how to implement error handling in SQL Server–based applications, please refer to the MSDN Library article titled "Using TRY...CATCH in Transact SQL" (http://bit.ly/PNlAHF).

Since the atomicity property requires that either all the actions of a logical unit of work are completed or no effects are retained, SQL Server *isolates* the work of a transaction from that of others by granting it exclusive rights on the affected resources. This means the transaction can safely roll back the effect of all its actions, if required. The exclusive rights granted to a transaction on the affected resources block all other transactions (or database requests) trying to access those resources during that time period. Therefore, although atomicity is required to maintain the integrity of data, it introduces the undesirable side effect of blocking.

Consistency

A unit of work should cause the state of the database to travel from one *consistent* state to another. At the end of a transaction, the state of the database should be fully consistent. SQL Server always ensures that the internal state of the databases is correct and valid by automatically applying all the constraints of the affected database resources as part of the transaction. SQL Server ensures that the state of internal structures, such as data and index layout, are correct after the transaction. For instance, when the data of a table is modified, SQL Server automatically identifies all the indexes, constraints, and other dependent objects on the table and applies the necessary modifications to all the dependent database objects as part of the transaction. That means that SQL Server will maintain the physical consistency of the data and the objects.

The logical consistency of the data is defined by the business rules and should be put in place by the developer of the database. A business rule may require changes to be applied on multiple tables, certain types of data to be restricted, or any number of other requirements. The database developer should accordingly define a logical unit of work to ensure that all the criteria of the business rules are taken care of. Further, the developer will ensure that the appropriate constructs are put in place to support the business rules that have been defined. SQL Server provides different transaction management features that the database developer can use to ensure the logical consistency of the data.

So, SQL Server works with the logical, business-defined, constraints that ensure a business-oriented data consistency to create a physical consistency on the underlying structures. The consistency characteristic of the logical unit of work blocks all other

www.EBooksWorld.ir

transactions (or database requests) trying to access the affected objects during that time period. Therefore, even though consistency is required to maintain a valid logical and physical state of the database, it also introduces the same side effect of blocking.

Isolation

In a multiuser environment, more than one transaction can be executed simultaneously. These concurrent transactions should be isolated from one another so that the intermediate changes made by one transaction don't affect the data consistency of other transactions. The degree of *isolation* required by a transaction can vary. SQL Server provides different transaction isolation features to implement the degree of isolation required by a transaction.

Note Transaction isolation levels are explained later in the chapter in the "Isolation Levels" section.

The isolation requirements of a transaction operating on a database resource can block other transactions trying to access the resource. In a multiuser database environment, multiple transactions are usually executed simultaneously. It is imperative that the data modifications made by an ongoing transaction be protected from the modifications made by other transactions. For instance, suppose a transaction is in the middle of modifying a few rows in a table. During that period, to maintain database consistency, you must ensure that other transactions do not modify or delete the same rows. SQL Server logically isolates the activities of a transaction from that of others by blocking them appropriately, which allows multiple transactions to execute simultaneously without corrupting one another's work.

Excessive blocking caused by isolation can adversely affect the scalability of a database application. A transaction may inadvertently block other transactions for a long period of time, thereby hurting database concurrency. Since SQL Server manages isolation using locks, it is important to understand the locking architecture of SQL Server. This helps you analyze a blocking scenario and implement resolutions.

Note The fundamentals of database locks are explained later in the chapter in the "Capturing Blocking Information" section.

---

www.EBooksWorld.ir

Durability

Once a transaction is completed, the changes made by the transaction should be *durable*. Even if the electrical power to the machine is tripped off immediately after the transaction is completed, the effect of all actions within the transaction should be retained. SQL Server ensures durability by keeping track of all pre- and post-images of the data under modification in a transaction log as the changes are made. Immediately after the completion of a transaction, SQL Server ensures that all the changes made by the transaction are retained—even if SQL Server, the operating system, or the hardware fails (excluding the log disk). During restart, SQL Server runs its database recovery feature, which identifies the pending changes from the transaction log for completed transactions and applies them to the database resources. This database feature is called *roll forward*.

The recovery interval period depends on the number of pending changes that need to be applied to the database resources during restart. To reduce the recovery interval period, SQL Server intermittently applies the intermediate changes made by the running transactions as configured by the recovery interval option. The recovery interval option can be configured using the `sp_configure` statement. The process of intermittently applying the intermediate changes is referred to as the *checkpoint* process. During restart, the recovery process identifies all uncommitted changes and removes them from the database resources by using the pre-images of the data from the transaction log.

Starting with SQL Server 2016, the default value of the `TARGET_RECOVERY_TIME` has been changed from 0, which means that the database will be doing all automatic checkpoints, to one minute. The default interval for automatic is also one minute, but now, the control is being set through the `TARGET_RECOVERY_TIME` value by default. If you need to change the frequency of the checkpoint operation, use `sp_configure` to change the recovery interval value. Setting this value means that the database is using indirect checkpoints. Instead of relying on the automatic checkpoints, you can use indirect checkpoints. This is a method to basically make the checkpoints occur all the time in order to meet the recovery interval. For systems with an extremely high number of data modifications, you might see high I/O because of indirect checkpoints. Starting in SQL Server 2016, all new databases created are automatically using indirect checkpoints because the `TARGET_INTERVAL_TIME` has been set. Any databases migrated from previous versions will be using whichever checkpoint method they had in that previous version. You may want to change their behavior as well. Using indirect checkpoints can result, for most systems, in a more consistent checkpoint behavior and faster recovery.

www.EBooksWorld.ir

The durability property isn't a direct cause of most blocking since it doesn't require the actions of a transaction to be isolated from those of others. But in an indirect way, it increases the duration of the blocking. Since the durability property requires saving the pre- and post-images of the data under modification to the transaction log on disk, it increases the duration of the transaction and therefore the possibility of blocking.

Introduced in SQL Server 2014 is the ability to reduce latency, the time waiting on a query to commit and write to the log, by modifying the durability behavior of a given database. You can now use delayed durability. This means that when a transaction completes, it reports immediately to the application as a successful transaction, reducing latency. But the writes to the log have not yet occurred. This may also allow for more transactions to be completed while still waiting on the system to write all the output to the transaction log. While this may increase apparent speed within the system, as well as possibly reducing contention on transaction log I/O, it's inherently a dangerous choice. This is a difficult recommendation to make. Microsoft suggests three possible situations that may make it attractive.

- *You don't care about the possible loss of some data*: Since you can be in a situation where you need to restore to a point in time from log backups, by choosing to put a database in delayed durability you may lose some data when you have to go to a restore situation.
- *You have a high degree of contention during log writes*: If you're seeing a lot of waits while transactions get written to the log, delayed durability could be a viable solution. But, you're also going to want to be tolerant of data loss, as discussed earlier.
- *You're experiencing high overall resource contention*: A lot of resource contention on the server comes down to the locks being held longer. If you're seeing lots of contention and you're seeing long log writes or also seeing contention on the log and you have a high tolerance for data loss, this may be a viable way to help reduce the system's contention.

In other works, I recommend using delayed durability only if you meet all those criteria, with the first being the most important. Also, don't forget about the changes to the checkpoint behavior noted earlier. If you're in a high-volume system, with lots of data changes, you may need to adjust the recovery interval to assist with system behavior as well.

www.EBooksWorld.ir

Note Out of the four ACID properties, the `isolation` property, which is also used to ensure atomicity and consistency, is the main cause of blocking in a SQL Server database. In SQL Server, isolation is implemented using locks, as explained in the next section.

Locks

When a session executes a query, SQL Server determines the database resources that need to be accessed; and, if required, the lock manager grants different types of locks to the session. The query is blocked if another session has already been granted the locks; however, to provide both transaction isolation and concurrency, SQL Server uses different lock granularities, as explained in the sections that follow.

Lock Granularity

SQL Server databases are maintained as files on the physical disk. In the case of a traditional nondatabase file such as an Excel file on a desktop machine, the file may be written to by only one user at a time. Any attempt to write to the file by other users fails. However, unlike the limited concurrency on a nondatabase file, SQL Server allows multiple users to modify (or access) contents simultaneously, as long as they don't affect one another's data consistency. This decreases blocking and improves concurrency among the transactions.

To improve concurrency, SQL Server implements lock granularities at the following resource levels and in this order:

- Row (`RID`)
- Key (`KEY`)
- Page (`PAG`)
- Extent (`EXT`)
- Heap or B-tree (`HoBT`)
- Table (`TAB`)
- File (`FIL`)

www.EBooksWorld.ir

- Application (APP)
- MetaData (MDT)
- Allocation Unit (AU)
- Database (DB)

Let's take a look at these lock levels in more detail.

Row-Level Lock

This lock is maintained on a single row within a table and is the lowest level of lock on a database table. When a query modifies a row in a table, an RID lock is granted to the query on the row. For example, consider the transaction on the following test table:

```
DROP TABLE IF EXISTS dbo.Test1;
CREATE TABLE dbo.Test1 (C1 INT);
INSERT INTO dbo.Test1
VALUES (1);
GO

BEGIN TRAN
DELETE dbo.Test1
WHERE C1 = 1;

SELECT dtl.request_session_id,
       dtl.resource_database_id,
       dtl.resource_associated_entity_id,
       dtl.resource_type,
       dtl.resource_description,
       dtl.request_mode,
       dtl.request_status
FROM sys.dm_tran_locks AS dtl
WHERE dtl.request_session_id = @@SPID;
ROLLBACK
```

The dynamic management view sys.dm_tran_locks can be used to display the lock status. The query against sys.dm_tran_locks in Figure 21-1 shows that the DELETE statement acquired, among other locks, an exclusive RID lock on the row to be deleted.

www.EBooksWorld.ir