

# Query Optimisation in SQL Server

## Introduction to Query Optimization and Processing

Query optimization is a critical aspect of database management aimed at improving the efficiency and performance of SQL queries. When dealing with large datasets and complex operations, optimizing queries becomes essential to enhance response times, reduce resource consumption, and improve overall application performance.

## Why Query Optimization is Needed

Query optimization is necessary to address several key challenges in database management:

1. **Performance Improvement:** Optimized queries execute faster, reducing response times and enhancing user experience.
2. **Resource Utilization:** Efficient queries consume fewer system resources, such as CPU and memory, making applications more scalable.
3. **Scalability:** Optimized queries scale better with increasing data volumes and user loads.
4. **Cost Reduction:** By reducing resource consumption and improving efficiency, query optimization can lower operational costs.

## Metrics Affecting Query Optimization

Several metrics impact query optimization:

1. **Execution Time:** The time taken to execute a query, influenced by indexing, query structure, and database configuration.
2. **Resource Consumption:** CPU and memory usage during query execution, which can be optimized through efficient query plans.
3. **I/O Operations:** Disk read/write operations, minimized by optimizing data access patterns and indexing strategies.
4. **Concurrency:** Handling multiple concurrent queries efficiently to maintain system responsiveness.

## Techniques of Query Optimisation:

Till now, we have seen what is query optimisation and different measures to analyse the query performance. Now, we will see the techniques to optimize the query performance in SQL. There are some useful practices to reduce the cost. However, the process of optimization is iterative. One needs to write the query, check query performance using io statistics or execution plan, and optimize it. This cycle needs to be followed iteratively for query optimization. The SQL Server finds the optimal and minimal plan to execute the query.

# 1. Introduction to Indexing

Indexing in databases is akin to creating an index at the back of a book, where topics are listed alphabetically with page numbers for quick access. Similarly, a database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional space and slower write operations. Indexes can be created on one or more columns of a table and are essential for efficient query processing, especially in large databases.

## Query Optimization with Indexing

The goal is to demonstrate how indexing can improve query performance by reducing the time taken to search and sort data within a table.

### Initial Query

The initial query retrieves all orders for a specific customer and sorts them by the order date. The query is as follows:

```
SELECT *
FROM SalesLT.SalesOrderHeader
WHERE CustomerID = 12345
ORDER BY OrderDate DESC;
```

### Analysis of the Query

The query involves two main operations that can benefit from indexing:

1. **Filtering:** The `WHERE` clause filters the results based on the `CustomerID`.
2. **Sorting:** The `ORDER BY` clause sorts the results by the `OrderDate`.

Without indexing, the database needs to scan the entire table to find rows that match the `CustomerID` and then sort those rows by `OrderDate`, which can be time-consuming for large tables.

### Indexing Strategy

To optimize this query, two indexes were created:

1. **Single-Column Index on `CustomerID`:** This index helps the database quickly locate all rows with the specified `CustomerID`.
2. **Composite Index on `CustomerID` and `OrderDate`:** This index assists both in filtering by `CustomerID` and in sorting by `OrderDate`, further reducing the need for a separate sort operation.

## Creating the Indexes

The following SQL commands were executed to create the indexes:

### 1. Single-Column Index:

```
CREATE INDEX idx_CustomerID  
ON SalesLT.SalesOrderHeader (CustomerID);
```

### 2. Composite Index:

```
CREATE INDEX idx_CustomerID_OrderDate  
ON SalesLT.SalesOrderHeader (CustomerID, OrderDate);
```

## Optimized Query

With the indexes in place, the optimized query remains the same:

```
SELECT *  
FROM SalesLT.SalesOrderHeader  
WHERE CustomerID = 12345  
ORDER BY OrderDate DESC;
```

The presence of the indexes allows the database to efficiently locate and sort the relevant rows, significantly improving query performance.

## Benefits of Indexing

- **Speed:** Indexes reduce the number of rows the database needs to scan, speeding up data retrieval.
- **Efficiency:** The composite index on `CustomerID` and `OrderDate` optimizes both the filtering and sorting operations in a single step.
- **Scalability:** Indexes are particularly beneficial for large tables, where full table scans would otherwise be prohibitively slow.

Clustered Index Scan (Clustered)	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated Execution Mode	Row
Storage	RowStore
Estimated Operator Cost	0.0033172 (23%)
Estimated I/O Cost	0.003125
Estimated Subtree Cost	0.0033172
Estimated CPU Cost	0.0001922
Estimated Number of Executions	1
Estimated Number of Rows to be Read	32
Estimated Number of Rows for All Executions	1
Estimated Number of Rows Per Execution	1
Estimated Row Size	4233 B
Ordered	False
Node ID	3
<b>Predicate</b>	
[AdventureWorksLT2022].[SalesLT].[SalesOrderHeader].[CustomerID]= (12345)	
<b>Object</b>	
[AdventureWorksLT2022].[SalesLT].[SalesOrderHeader].[PK_SalesOrderHeader_SalesOrderID]	
<b>Output List</b>	
[AdventureWorksLT2022].[SalesLT].[SalesOrderHeader].SalesOrderID, [AdventureWorksLT2022].[SalesLT].[SalesOrderHeader].RevisionNumber, [AdventureWorksLT2022].[SalesLT].[SalesOrderHeader].OrderDate, [AdventureWorksLT2022].[SalesLT].[SalesOrderHeader].DueDate, [AdventureWorksLT2022].[SalesLT].[SalesOrderHeader].ShipDate, [AdventureWorksLT2022].[SalesLT].[SalesOrderHeader].Status, [AdventureWorksLT2022].[SalesLT].[SalesOrderHeader].OnlineOrderFlag, [AdventureWorksLT2022].[SalesLT].[SalesOrderHeader]...	

**Before Indexing**

Index Seek (NonClustered)	
Scan a particular range of rows from a nonclustered index.	
Physical Operation	Index Seek
Logical Operation	Index Seek
Estimated Execution Mode	Row
Storage	RowStore
Estimated Operator Cost	0.0032831 (50%)
Estimated I/O Cost	0.003125
Estimated Subtree Cost	0.0032831
Estimated CPU Cost	0.0001581
Estimated Number of Executions	1
Estimated Number of Rows to be Read	1
Estimated Number of Rows for All Executions	1
Estimated Number of Rows Per Execution	1
Estimated Row Size	23 B
Ordered	True
Node ID	3
<b>Object</b>	
[AdventureWorksLT2022].[SalesLT].[SalesOrderHeader].[idx_CustomerID_OrderDate]	
<b>Output List</b>	
[AdventureWorksLT2022].[SalesLT].[SalesOrderHeader].SalesOrderID, [AdventureWorksLT2022].[SalesLT].[SalesOrderHeader].OrderDate, [AdventureWorksLT2022].[SalesLT].[SalesOrderHeader].CustomerID	
<b>Seek Predicates</b>	
Seek Keys[1]: Prefix: [AdventureWorksLT2022].[SalesLT].[SalesOrderHeader].CustomerID = Scalar Operator((12345))	

**After Indexing**

From the above images, after indexing we can clearly see the reduction in:

- Estimated CPU Cost
- Estimated number of rows to be read
- Estimated row size

## Conclusion

Indexing is a powerful technique for optimizing SQL queries. By creating appropriate indexes on the `CustomerID` and `OrderDate` columns of the `SalesLT.SalesOrderHeader` table, we significantly improved the performance of a query that retrieves and sorts sales orders for a specific customer. This example demonstrates the importance of analyzing query patterns and strategically applying indexes to enhance database performance.

## 2. Introduction to Selection in SQL

Selection in SQL involves retrieving specific rows from a database table based on a condition specified in the `WHERE` clause. Effective selection techniques can significantly improve query performance by reducing the amount of data processed and returned by the query.

### Query Optimization with Selection Techniques

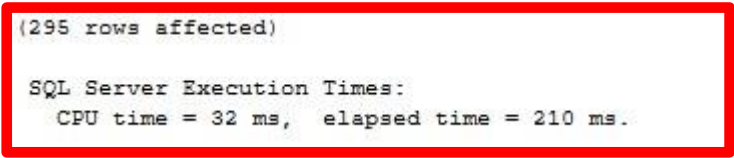
The goal is to demonstrate how careful selection of columns and conditions can improve query performance.

#### Initial Query

The initial query is as follows:

```
SET STATISTICS TIME ON
SELECT * FROM SalesLT.Product
```

#### Analysis of the Query

A screenshot of SQL Server execution statistics for the initial query. The text is enclosed in a red rectangular border. It shows that 295 rows were affected. The SQL Server Execution Times are: CPU time = 32 ms, and elapsed time = 210 ms.

```
(295 rows affected)

SQL Server Execution Times:
    CPU time = 32 ms,  elapsed time = 210 ms.
```

The query involves several operations that can benefit from optimization:

1. **Column Selection:** The `SELECT *` clause retrieves all columns, which can be inefficient if only a subset of columns is needed.

#### Selection Strategy

To optimize this query, we can:

1. Select only the necessary columns instead of all columns.
2. Add more specific filtering conditions if applicable.

### Optimized Query with Specific Column Selection

By selecting only the columns that are needed, we reduce the amount of data retrieved and processed.

#### Selecting Specific Columns

```
SET STATISTICS TIME ON
SELECT ProductNumber, Name, Color, Weight FROM SalesLT.Product
```

This reduces the amount of data processed and transferred, improving query performance.

```
(295 rows affected)

SQL Server Execution Times:
    CPU time = 0 ms,  elapsed time = 106 ms.
```

## Explanation of the Optimized Query

1. **Column Selection:** By selecting only the necessary columns we reduce the amount of data retrieved, which speeds up the query.

## Benefits of Efficient Selection

- **Speed:** Reduces the amount of data processed and transferred, speeding up the query.
- **Resource Utilization:** Efficient selection uses fewer resources, making the query more efficient.
- **Relevance:** Ensures that only relevant data is retrieved, making the query results more useful.

Clustered Index Scan (Clustered)	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated Execution Mode	Row
Storage	RowStore
Estimated I/O Cost	0.0771991
Estimated Operator Cost	0.0776806 (100%)
Estimated CPU Cost	0.0004815
Estimated Subtree Cost	0.0776806
Estimated Number of Executions	1
Estimated Number of Rows for All Executions	295
Estimated Number of Rows Per Execution	295
Estimated Number of Rows to be Read	295
<b>Estimated Row Size</b>	<b>4273 B</b>
Ordered	false
Node ID	0
<b>Object</b>	
[AdventureWorksLT2022].[SalesLT].[Product].[PK_Product_ProductID]	
<b>Output List</b>	
[AdventureWorksLT2022].[SalesLT].[Product].ProductID,	
[AdventureWorksLT2022].[SalesLT].[Product].Name,	
[AdventureWorksLT2022].[SalesLT].[Product].ProductNumber,	
[AdventureWorksLT2022].[SalesLT].[Product].Color,	
[AdventureWorksLT2022].[SalesLT].[Product].StandardCost,	
[AdventureWorksLT2022].[SalesLT].[Product].ListPrice,	
[AdventureWorksLT2022].[SalesLT].[Product].Size,	
[AdventureWorksLT2022].[SalesLT].[Product].Weight,	
[AdventureWorksLT2022].[SalesLT].[Product].ProductCategoryID,	
[AdventureWor...	

Before Selection

Clustered Index Scan (Clustered)	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated Execution Mode	Row
Storage	RowStore
Estimated I/O Cost	0.0771991
Estimated Operator Cost	0.0776806 (100%)
Estimated CPU Cost	0.0004815
Estimated Subtree Cost	0.0776806
Estimated Number of Executions	1
Estimated Number of Rows for All Executions	295
Estimated Number of Rows Per Execution	295
Estimated Number of Rows to be Read	295
<b>Estimated Row Size</b>	<b>110 B</b>
Ordered	False
Node ID	0
<b>Object</b>	
[AdventureWorksLT2022].[SalesLT].[Product].[PK_Product_ProductID]	
<b>Output List</b>	
[AdventureWorksLT2022].[SalesLT].[Product].Name,	
[AdventureWorksLT2022].[SalesLT].[Product].ProductNumber,	
[AdventureWorksLT2022].[SalesLT].[Product].Color,	
[AdventureWorksLT2022].[SalesLT].[Product].Weight	

After Selection

From the above images, we can clearly see that the **Estimated Row Size** has decreased. This happens because we are not reading all the columns, instead we are reading only selected columns.

## **Conclusion**

Efficient selection techniques are crucial for optimizing SQL queries. By selecting only the necessary columns and refining filtering conditions, we significantly improved the performance of a query that retrieves sales orders for a specific customer in the AdventureWorksLT database. This example demonstrates the importance of careful column selection and precise filtering conditions in enhancing database performance.

### 3. Introduction to Avoiding SELECT DISTINCT in SQL

Using `SELECT DISTINCT` in SQL queries ensures that the result set contains only unique rows, which can be resource-intensive and impact performance, especially for large datasets. This report demonstrates how avoiding `SELECT DISTINCT` and optimizing queries can lead to performance improvements.

#### Query Optimization by Avoiding SELECT DISTINCT

We will compare the performance of queries with and without `SELECT DISTINCT` using SQL Server's `SET STATISTICS TIME ON` command to measure execution time.

#### Initial Query with SELECT DISTINCT

The initial query retrieves all unique combinations of Name, Color, StandardCost, and Weight from the `SalesLT.Product` table.

```
SET STATISTICS TIME ON
SELECT DISTINCT Name, Color, StandardCost, Weight
FROM SalesLT.Product;
```

#### Analysis of the Query

The query uses `SELECT DISTINCT` to ensure that the result set contains only unique rows. This can be inefficient because it requires the database to scan all rows and eliminate duplicates.

#### Alternative Query Without SELECT DISTINCT

```
SET STATISTICS TIME ON
SELECT Name, Color, StandardCost, Weight, SellStartDate, SellEndDate
FROM SalesLT.Product
```

#### Comparison of Query Performance

To compare the performance, we will execute both queries with `SET STATISTICS TIME ON` to measure the CPU time and elapsed time.

#### Initial Query Execution

```
(295 rows affected)

SQL Server Execution Times:
    CPU time = 0 ms,  elapsed time = 201 ms.
```



## Optimized Query Execution

```
(295 rows affected)

SQL Server Execution Times:
    CPU time = 0 ms,  elapsed time = 84 ms.
```

## Benefits of Avoiding SELECT DISTINCT

- **Speed:** Avoiding `SELECT DISTINCT` can reduce the time needed to scan and compare rows.
- **Efficiency:** Using `GROUP BY` can be a more efficient way to retrieve unique values.
- **Resource Utilization:** Reduces the computational resources required to process the query.

## Conclusion

Avoiding `SELECT DISTINCT` when it is not necessary can lead to significant performance improvements in SQL queries. By using alternative strategies such as `GROUP BY`, we can achieve the same results more efficiently. This example demonstrates the importance of analyzing query requirements and optimizing accordingly to enhance database performance.

## 4. Introduction to IN vs. EXISTS in SQL

In SQL, both `IN` and `EXISTS` are used to check for the existence of rows in a subquery. However, they differ in how they handle `NULL` values and can have different performance characteristics. This report compares the performance of using `IN` versus `EXISTS` in SQL Server, using `SET STATISTICS TIME ON` to measure execution time.

### Query Optimization Using IN

Objective is the optimization of SQL queries using `IN` to check for existence in the AdventureWorksLT database.

#### Query with IN

The initial query retrieves `ProductNumber`, `Name`, and `Color` from the `SalesLT.Product` table, filtering by `ProductID` using `IN` with a subquery from `SalesLT.ProductDescription`.

```
SET STATISTICS TIME ON
SELECT ProductNumber, Name, Color FROM SalesLT.Product
WHERE ProductID IN
(SELECT ProductID FROM SalesLT.ProductDescription)
```

#### Analysis of the Query

The `IN` operator checks for the existence of a value in a subquery result set. It is straightforward but may perform differently based on the number of rows returned by the subquery.

#### Optimized Query with EXISTS

The optimized query uses `EXISTS` to check for the existence of rows in the subquery from `SalesLT.ProductDescription`.

```
SET STATISTICS TIME ON
SELECT ProductNumber, Name, Color FROM SalesLT.Product
WHERE EXISTS
(SELECT ProductID FROM SalesLT.ProductDescription)
```

#### Explanation of the Optimized Query

1. **IN:** `IN` checks for the existence of a value in a subquery result set.
2. **EXISTS:** `EXISTS` checks for the existence of rows in a subquery, which can sometimes be more efficient, especially when dealing with large result sets or complex queries.

#### Comparison of Query Performance

To compare the performance, we will execute both queries with `SET STATISTICS TIME ON` to measure execution time.

## Initial Query Execution with IN

```
(295 rows affected)

SQL Server Execution Times:
    CPU time = 0 ms,  elapsed time = 63 ms.
```

## Optimized Query Execution with EXISTS

```
(295 rows affected)

SQL Server Execution Times:
    CPU time = 0 ms,  elapsed time = 56 ms.
```

## Benefits of Using EXISTS

- **Performance:** `EXISTS` can sometimes be more efficient, especially when the subquery returns a large number of rows.
- **Clarity:** Clearly defines the intent to check for the existence of rows in a subquery.
- **Optimization:** Optimizes performance by avoiding unnecessary data retrieval and comparisons.

## Conclusion

Using `EXISTS` to check for the existence of rows in a subquery can sometimes offer better performance compared to `IN`, especially in SQL Server. This report demonstrates the importance of choosing the appropriate method based on query requirements and optimizing SQL queries for better performance and efficiency.

## 5. Introduction to INNER JOIN vs. WHERE Clause in SQL

SQL queries can use either the `INNER JOIN` clause or the `WHERE` clause to combine data from multiple tables. Understanding when to use each method is crucial for optimizing query performance. This report compares the performance of `INNER JOIN` and `WHERE` clause approaches using SQL Server's `SET STATISTICS IO ON` and `SET STATISTICS TIME ON` commands to measure I/O and execution time.

### Query Optimization Using INNER JOIN

Objective is the optimization of SQL queries using `INNER JOIN` and `WHERE` clause approaches in the AdventureWorksLT database.

#### Initial Query with WHERE Clause

The initial query retrieves `Name`, `Color`, and `ListPrice` from the `SalesLT.Product` table, filtering by `ProductCategoryID` using the `WHERE` clause to join with `SalesLT.ProductCategory`.

```
SET STATISTICS IO ON;
SET STATISTICS TIME ON;

SELECT p.Name, Color, ListPrice
FROM SalesLT.Product p, SalesLT.ProductCategory pc
WHERE p.ProductCategoryID = pc.ProductCategoryID;
```

#### Analysis of the Query

The query uses the `WHERE` clause for joining, which is a traditional approach but can be less clear and less optimized compared to `INNER JOIN`.

#### Optimized Query with INNER JOIN

The optimized query uses `INNER JOIN` to join `SalesLT.Product` and `SalesLT.ProductCategory` on `ProductCategoryID`.

```
SET STATISTICS IO ON;
SET STATISTICS TIME ON;

SELECT p.Name, Color, ListPrice
FROM SalesLT.Product p
INNER JOIN SalesLT.ProductCategory pc
ON p.ProductCategoryID = pc.ProductCategoryID;
```

## Explanation of the Optimized Query

1. **INNER JOIN:** `INNER JOIN` explicitly states how tables are related (`ON p.ProductCategoryID = pc.ProductCategoryID`), making it clearer and potentially more optimized by the query optimizer.
2. **WHERE Clause:** Using the `WHERE` clause for joining is an older approach and can be less optimized, especially in more complex queries.

## Comparison of Query Performance

To compare the performance, we will execute both queries with `SET STATISTICS IO ON` and `SET STATISTICS TIME ON` to measure I/O and execution time.

### Initial Query Execution with WHERE Clause

```
(295 rows affected)
Table 'Workfile'. Scan count 0, logical reads 0, p
Table 'Worktable'. Scan count 0, logical reads 0,
Table 'Product'. Scan count 1, logical reads 103,
Table 'ProductCategory'. Scan count 1, logical rea

SQL Server Execution Times:
    CPU time = 0 ms,  elapsed time = 122 ms.
```

### Optimized Query Execution with INNER JOIN

```
(295 rows affected)
Table 'Workfile'. Scan count 0, logical reads 0,
Table 'Worktable'. Scan count 0, logical reads 0,
Table 'Product'. Scan count 1, logical reads 103,
Table 'ProductCategory'. Scan count 1, logical re

SQL Server Execution Times:
    CPU time = 0 ms,  elapsed time = 50 ms.
```

## Benefits of Using INNER JOIN

- **Clarity:** `INNER JOIN` provides clearer syntax, explicitly stating how tables are joined.
- **Optimization:** `INNER JOIN` can be more optimized by the query optimizer, potentially improving query performance.
- **Modern Approach:** `INNER JOIN` is a more modern and preferred approach for SQL queries.

## Conclusion

Using `INNER JOIN` instead of the `WHERE` clause for joining tables can lead to improved query performance and clarity. This report demonstrates the importance of using efficient SQL syntax and understanding when to use different join methods to optimize database queries.

## 6. Introduction to TOP vs. No Limiting in SQL

SQL queries often need to restrict the number of rows returned, especially for performance reasons or to display a subset of data. This report compares the performance of using `TOP` to limit rows versus not limiting rows in SQL Server, using `SET STATISTICS IO ON` and `SET STATISTICS TIME ON` to measure I/O and execution time.

### Query Optimization Using TOP

Objective is the optimization of SQL queries using `TOP` to limit rows in the AdventureWorksLT database.

#### Query without Limiting

The initial query retrieves `Name`, `Color`, and `ListPrice` from the `SalesLT.Product` table without limiting the number of rows.

```
SET STATISTICS IO ON;
SET STATISTICS TIME ON;

SELECT Name, Color, ListPrice
FROM SalesLT.Product;
```

#### Analysis of the Query

The query retrieves all rows from the `SalesLT.Product` table, which can be inefficient if only a subset of rows is needed.

#### Optimized Query with TOP

The optimized query uses `TOP 10` to limit the results to the first 10 rows from the `SalesLT.Product` table.

```
SET STATISTICS IO ON;
SET STATISTICS TIME ON;

SELECT TOP 10 Name, Color, ListPrice
FROM SalesLT.Product;
```

#### Explanation of the Optimized Query

1. **TOP:** `TOP 10` limits the number of rows returned to 10, optimizing performance by reducing the amount of data processed and transferred.
2. **No Limiting:** Without `TOP 10`, the query retrieves all rows, which may be unnecessary if only a small subset is needed.

#### Comparison of Query Performance

To compare the performance, we will execute both queries with `SET STATISTICS IO ON` and `SET STATISTICS TIME ON` to measure I/O and execution time.

## Initial Query Execution without TOP

```
(295 rows affected)
Table 'Product'. Scan count 1, logical reads

SQL Server Execution Times:
    CPU time = 0 ms,  elapsed time = 78 ms.
```

## Optimized Query Execution with TOP

```
(10 rows affected)
Table 'Product'. Scan count 1, logical reads 5,

SQL Server Execution Times:
    CPU time = 0 ms,  elapsed time = 3 ms.
```

## Benefits of Using TOP

- **Performance:** TOP 10 reduces the amount of data processed and transferred, improving query performance.
- **Efficiency:** Limits the results to only the necessary rows, optimizing resource utilization.
- **Clarity:** Clearly defines the intent to retrieve a specific number of rows, making the query more understandable.

## Conclusion

Using TOP 10 to limit the number of rows returned can improve query performance and optimize resource usage in SQL Server. This report demonstrates the importance of efficiently retrieving data and using SQL syntax to achieve optimal query performance.

## 7. Introduction to Loops vs. Bulk Insert/Update in SQL

In SQL Server, loops (`WHILE` loop) are commonly used for iterative operations, whereas bulk insert/update operations handle large volumes of data more efficiently. This report compares the performance of using loops versus bulk insert/update operations in SQL Server, using `SET STATISTICS TIME ON` to measure execution time.

### Query Optimization Using Loops

Objective is the optimization of SQL queries using a `WHILE` loop to insert records into the `SalesLT.ProductDescription` table in the AdventureWorksLT database.

### Query with WHILE Loop

The initial query uses a `WHILE` loop to iteratively insert records into the `SalesLT.ProductDescription` table.

```
SET STATISTICS TIME ON;

DECLARE @Counter INT = 1;

WHILE (@Counter <= 10)
BEGIN
    PRINT 'The counter value is = ' + CONVERT(VARCHAR, @Counter);

    INSERT INTO [SalesLT].[ProductDescription]
        ([Description]
        , [rowguid]
        , [ModifiedDate])
    VALUES
        ('This is great'
        , NEWID()
        , '2010-12-01');

    SET @Counter = @Counter + 1;
END
```

### Analysis of the Query

The `WHILE` loop iteratively inserts records one by one, which can be less efficient for large datasets due to the overhead of multiple transactions and logging operations.

### Optimized Query with Bulk Insert

The optimized query uses a single `INSERT` statement with multiple value sets (`VALUES`) to perform bulk insert into the `SalesLT.ProductDescription` table.



```

SET STATISTICS TIME ON;

INSERT INTO [SalesLT].[ProductDescription]
    ([Description]
    , [rowguid]
    , [ModifiedDate])
VALUES
    ('This is great', NEWID(), '2010-12-01'),
    ('New news', NEWID(), '2010-12-01'),
    ('Awesome product.', NEWID(), '2010-12-01'),
    ('Another product.', NEWID(), '2010-12-01'),
    -- Add more rows as needed
    ('Awesome product.', NEWID(), '2010-12-01');

```

## Explanation of the Optimized Query

1. **Bulk Insert:** Using a single `INSERT` statement with multiple value sets (`VALUES`) reduces the overhead of multiple transactions and logging operations, making it more efficient for large datasets.
2. **WHILE Loop:** The `WHILE` loop inserts records iteratively, which can lead to increased execution time and resource consumption.

## Comparison of Query Performance

To compare the performance, we will execute both queries with `SET STATISTICS TIME ON` to measure execution time.

### Initial Query Execution with WHILE Loop

```

SQL Server parse and compile time:
    CPU time = 0 ms, elapsed time = 4 ms.

SQL Server Execution Times:
    CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:
    CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:
    CPU time = 0 ms, elapsed time = 0 ms.
The counter value is = 1

SQL Server Execution Times:
    CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
    CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:
    CPU time = 15 ms, elapsed time = 7 ms.

(1 row affected)

SQL Server Execution Times:
    CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:
    CPU time = 0 ms, elapsed time = 0 ms.

```

```

The counter value is = 2

SQL Server Execution Times:
    CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
    CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:
    CPU time = 0 ms, elapsed time = 0 ms.

(1 row affected)

SQL Server Execution Times:
    CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:
    CPU time = 0 ms, elapsed time = 0 ms.
The counter value is = 3

SQL Server Execution Times:
    CPU time = 0 ms, elapsed time = 0 ms.
SQL Server parse and compile time:
    CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:
    CPU time = 0 ms, elapsed time = 0 ms.

```

## Optimized Query Execution with Bulk Insert

```
SQL Server parse and compile time:
  CPU time = 15 ms, elapsed time = 39 ms

SQL Server Execution Times:
  CPU time = 0 ms,  elapsed time = 0 ms.
SQL Server parse and compile time:
  CPU time = 0 ms, elapsed time = 0 ms.

SQL Server Execution Times:
  CPU time = 0 ms,  elapsed time = 3 ms.

(5 rows affected)
```

## Benefits of Using Bulk Insert

- **Performance:** Bulk insert operations are generally faster for inserting large volumes of data due to reduced transactional overhead.
- **Efficiency:** Reduces the number of transactions and logging operations, optimizing resource usage.
- **Scalability:** Scales better for large datasets compared to iterative operations like loops.

## Conclusion

Using bulk insert/update operations instead of loops (`WHILE` loop) can significantly improve performance and efficiency in SQL Server, especially for handling large datasets. This report highlights the importance of choosing the appropriate method based on data volume and optimizing SQL queries for better performance.

# Report Conclusion and Key Takeaways

In conclusion, optimizing SQL queries is crucial for improving database performance, enhancing application scalability, and reducing operational costs. By understanding the factors influencing query optimization and employing best practices such as index utilization, proper query structuring, and efficient data access methods, organizations can achieve significant performance gains and better utilize their database resources.

## Key Takeaways:

- **Importance of Optimization:** Query optimization enhances performance, scalability, and reduces costs.
- **Metrics to Consider:** Execution time, resource consumption, I/O operations, and concurrency impact query efficiency.
- **Best Practices:** Use indexing wisely, structure queries efficiently, and optimize data access patterns for optimal performance.

Implementing these strategies ensures efficient database operations, supporting high-performance applications and better user experiences.