**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);

1. **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

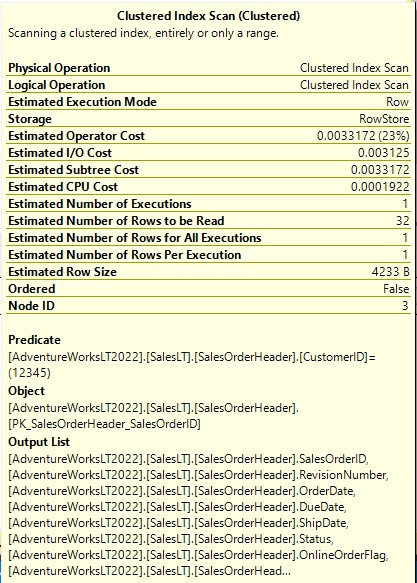
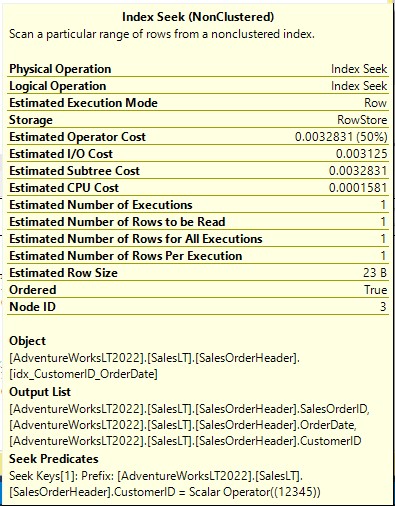
WHERE CustomerID = 12345

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.

**Before Indexing 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.