

Lab: Query Rewriting & Index Optimization

Scenario:

Consider a large e-commerce database with a Orders table containing millions of records. A frequently executed query is:

```
SELECT
    CustomerID,
    SUM(TotalAmount) AS TotalSales
FROM
    Orders
WHERE
    OrderDate BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY
    CustomerID;
```

This query calculates total sales for each customer within a specific year.

Performance Analysis:

Without appropriate indexes, this query might perform slowly due to a full table scan. To improve performance, we can analyze the execution plan and identify potential bottlenecks.

Optimization Steps:

Create an Index:

Create a nonclustered index on the OrderDate column:

```
CREATE NONCLUSTERED INDEX IX_Orders_OrderDate ON Orders (OrderDate);
```

This index will help the query optimizer efficiently locate rows within the specified date range.

Review Execution Plan:

- Re-run the query and examine the execution plan.
- Look for any full table scans or inefficient join operations.

- If the index is being used effectively, you should see an Index Seek operation on the IX_Orders_OrderDate index.

Consider Data Distribution:

- If the OrderDate column is highly skewed (e.g., many orders in a few specific months), partitioning the table might further improve performance.

Revised Query

```
SELECT
    CustomerID,
    SUM(TotalAmount) AS TotalSales
FROM
    Orders
WHERE
    OrderDate BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY
    CustomerID;
```

Database Query Performance Analysis & Tuning

Objective

To enhance understanding of query performance and optimization techniques in a database environment.

Tasks

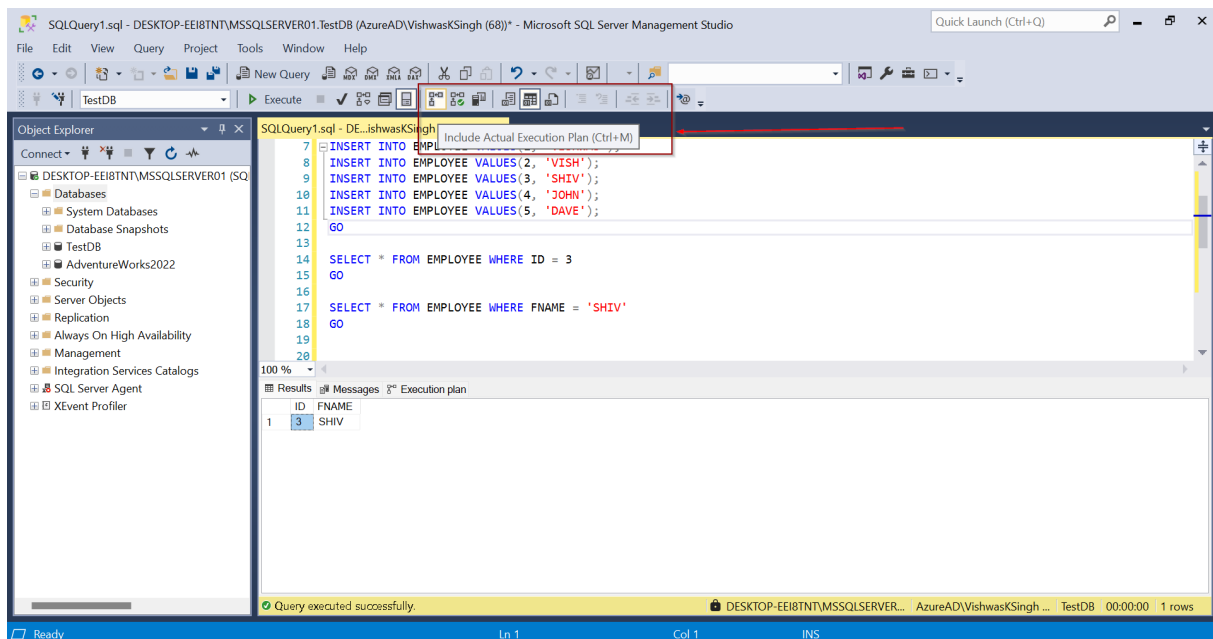
Task 1: Create an empty database

```
CREATE DATABASE TestDB;  
GO
```

Task 2: Create a table and insert data into it

```
USE TestDB  
GO  
  
CREATE TABLE EMPLOYEE(ID INT PRIMARY KEY, FNAME VARCHAR(30))  
GO  
  
INSERT INTO EMPLOYEE VALUES(1, 'VISHWAS');  
INSERT INTO EMPLOYEE VALUES(2, 'VISH');  
INSERT INTO EMPLOYEE VALUES(3, 'SHIV');  
INSERT INTO EMPLOYEE VALUES(4, 'JOHN');  
INSERT INTO EMPLOYEE VALUES(5, 'DAVE');  
GO
```

Task 3: Enable Include Actual Execution Plan



Task 4: Execute the Below Query

```
SELECT * FROM EMPLOYEE WHERE FNAME = 'SHIV'
GO
```

Observe the execution Plan

SQL Query Optimization

The screenshot displays the Microsoft SQL Server Enterprise Manager interface. The central pane shows a query execution plan for a query named 'Query 1: Query cost (relative)'. The query is 'SELECT * FROM [EMPLOYEE] WHERE [FNAME] = [1]'. The execution plan shows a 'Clustered Index Scan (Clustered)' operation. The execution statistics pane on the right provides detailed information about the query's performance, including the number of rows read, the number of rows for all executions, and the estimated cost. The estimated cost is 100%, indicating a full table scan. The actual number of rows read is 5, and the number of rows for all executions is 1. The estimated row size is 198 bytes. The estimated CPU cost is 0.0001625. The estimated I/O cost is 0.003125. The estimated operator cost is 0.0032875 (100%). The estimated subtree cost is 0.0032875. The estimated number of executions is 1. The number of rows for all executions is 1. The number of rows per execution is 1. The estimated number of rows to be read is 5. The estimated row size is 198 bytes. The actual rebinds are 0. The actual rewinds are 0. The ordered flag is false. The node ID is 0. The predicate is '[TestDB].[dbo].[EMPLOYEE].[FNAME] = [1]'. The object is '[TestDB].[dbo].[EMPLOYEE]'. The output list is '[TestDB].[dbo].[EMPLOYEE].[ID], [TestDB].[dbo].[EMPLOYEE].[FNAME]'. The status bar at the bottom indicates 'Query executed successfully'.

The query searches a record using a non index value or a non primary value. Hence the query searches the whole table. When scaled this is a performance bottleneck.

This can be improved by

1. Creating an index over the column
2. using an existing indexed column(Primary Key)

Task 5: Creating an optimized query

```
SELECT * FROM EMPLOYEE WHERE ID = 3
GO
```

Execute the query and observe the execution plan

SQL Query Optimization

The screenshot displays the SQL Server Management Studio interface. The left pane shows the Object Explorer with the TestDB database selected. The center pane shows a query window with the following SQL code:

```
7 INSERT INTO EMPLOYEE VALUES (1, 'John', 'Doe', '1980-01-01', '1000', '1000')
8 INSERT INTO EMPLOYEE VALUES (2, 'Jane', 'Doe', '1980-01-01', '1000', '1000')
9 INSERT INTO EMPLOYEE VALUES (3, 'John', 'Doe', '1980-01-01', '1000', '1000')
10 INSERT INTO EMPLOYEE VALUES (4, 'Jane', 'Doe', '1980-01-01', '1000', '1000')
11 INSERT INTO EMPLOYEE VALUES (5, 'John', 'Doe', '1980-01-01', '1000', '1000')
12 GO
13
14 SELECT * FROM EMPLOYEE WHERE ID = 1
15 GO
16
17 SELECT * FROM EMPLOYEE WHERE ID = 2
18 GO
19
20
```

The right pane shows the execution plan for the query. The main operation is a 'Clustered Index Seek (Clustered)'. The plan details the physical and logical operations, execution mode, storage, and various cost metrics. Red arrows point to the 'Actual Number of Rows Read' and 'Estimated Number of Rows for All Executions' fields.

Property	Value
Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Actual Number of Rows Read	1
Actual Number of Rows for All Executions	1
Actual Number of Batches	0
Estimated Operator Cost	0.0032831 (100%)
Estimated I/O Cost	0.003125
Estimated Subtree Cost	0.0032831
Estimated CPU Cost	0.0001581
Estimated Number of Executions	1
Number of Executions	1
Estimated Number of Rows for All Executions	1
Estimated Number of Rows to be Read	1
Estimated Number of Rows Per Execution	1
Estimated Row Size	30 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0
Object	[TestDB].[dbo].[EMPLOYEE].[PK_EMPLOYEE_3214EC2763ECF99A]
Output List	[TestDB].[dbo].[EMPLOYEE].ID, [TestDB].[dbo].[EMPLOYEE].FNAME
Seek Predicates	Seek Keys(1): Prefix: [TestDB].[dbo].[EMPLOYEE].ID = Scalar Operator (CONVERT_IMPLICIT(nt,@1,0))

Hence, Query performance can be improved by rewriting the query using indexed columns

Similarly, We can apply the above steps on Northwind/AdventureWorks Database

We can also observe the query performance using activity monitor