Comprehensive Report On Query Optimization

**1.Introduction**

Query optimization is a critical aspect of database management that focuses on improving the efficiency of SQL queries. The primary goal of query optimization is to reduce the response time and resource consumption of database queries, thereby enhancing the overall performance of the database system. This process involves several key steps and techniques:

**1. Understanding Query Execution:**

* **Query Parsing:** The database management system (DBMS) first parses the SQL query to check for syntax errors and to create a parse tree.
* **Query Optimization:** The parsed query is then passed to the query optimizer, which generates an optimal execution plan. This plan outlines the most efficient way to access and manipulate the data.
* **Query Execution:** The DBMS executes the query based on the chosen execution plan, retrieving the requested data.

**1. Execution Plan:**

* The execution plan is a roadmap that the DBMS follows to execute a query. It includes steps such as table scans, index scans, joins, and sort operations.
* **Cost-Based Optimization:** The optimizer evaluates multiple execution plans and selects the one with the lowest estimated cost, considering factors like CPU, memory, and I/O.

**2. Significance :**

Query optimization is crucial for several reasons, particularly in ensuring the efficient and effective operation of database systems. Here are some key reasons why query optimization is needed:

**1. Performance Improvement:**

* **Reduced Query Execution Time:** Optimized queries execute faster, providing quicker responses to users and applications.
* **Efficient Resource Utilization:** Optimization minimizes the consumption of system resources (CPU, memory, I/O operations), which helps in maintaining overall system performance.

**2. Scalability:**

* As the amount of data grows, unoptimized queries can become increasingly slow and resource-intensive. Optimization ensures that queries remain performant even as the database scales.

**3. Cost Efficiency:**

* Efficient queries reduce the need for additional hardware or cloud resources, leading to cost savings on infrastructure and maintenance.
* Optimized queries also reduce the wear and tear on hardware, potentially extending the life of physical components.

**4. User Experience:**

* Fast query responses contribute to a positive user experience, particularly in real-time applications such as e-commerce, where slow performance can lead to user frustration and loss of business.

**3. Process of Query Execution**

In Microsoft SQL Server, the process of query execution involves several critical steps, from parsing the user's query to delivering the final results. Here’s a detailed overview of how SQL Server handles query execution:

**1. Query Parsing**

* **Lexical Analysis:** The query is tokenized into keywords, identifiers, operators, and literals.
* **Syntax Analysis:** The parser checks the query syntax against SQL Server's grammar rules. If there are syntax errors, they are reported back to the user.

**2. Query Rewrite**

* SQL Server's Query Optimizer might rewrite the query to a more efficient form. This could involve:
  + Simplifying expressions.
  + Transforming subqueries into joins.
  + Merging views and inline functions.

**3. Query Execution Plan Generation**

* The chosen execution plan is converted into a sequence of physical operations that SQL Server's storage engine can execute. This includes decisions about:
  + Access methods (e.g., index seek, table scan).
  + Join strategies (e.g., nested loop join, hash join, merge join).
  + Order of operations.

**4. Query Execution**

* **Execution Engine:** SQL Server's execution engine carries out the operations specified in the plan.
* **Operators:** Common operators include scans (index or table), seeks, joins, sorts, and aggregations.

**5. Fetching Data**

* SQL Server retrieves the necessary data from the storage engine. This involves:
  + **Access Methods:** Using appropriate methods based on the execution plan (e.g., index seek for fast lookups).
  + **Join Operations:** Performing joins if the query involves multiple tables.

**6. Data Retrieval**

* The execution engine collects the results of the operations, performing any necessary intermediate steps like sorting or aggregation.

**7. Result Delivery**

* The final results are formatted and sent back to the client application that submitted the query.

**8. Execution Feedback**

* **Statistics and Logging:** SQL Server can provide detailed statistics about the execution, such as:
  + Execution time.
  + I/O statistics (reads/writes).
  + CPU usage.
  + Actual execution plan versus the estimated plan.

**4. Data Set Used :**

In the SQL Server to manage a **hospital database** named **HospitalDB**. This database includes two primary tables: Doctors and Patients. The Doctors table stores information about doctors, including their ID, name, and gender. The Patients table stores information about patients, including their ID, name, disease, recovery rate, treatment amount, and the doctor responsible for their treatment. This database setup allows for efficient tracking and management of patient treatment and doctor performance, with an emphasis on query optimization to enhance performance.

**5.** **Optimization Techniques :**

1.Indexing:

* Clustered Indexes: Organize data within the table to improve retrieval speed.
* Non-Clustered Indexes: Provide quick access to data using pointers to the data stored in a clustered index or heap.
* Covering Indexes: Include all columns needed for a query, eliminating the need to access the table data.

2. Use numeric fields to store numeric numeric values instead of character data types.

3. Replace UNION with UNION ALL where we don’t have duplicate results.

4**.** SELECT Only Required Columns: Avoid using SELECT \* and specify only the columns needed.

5. Use Proper Joins: : Choose the most efficient join type (INNER JOIN, LEFT JOIN, etc.).. Ensure that your joins are correct and not causing duplicate rows. Review the join conditions and only join the necessary tables.

6. Avoid Nested Loops: Replace nested loops with set-based operations where possible.

7. Avoid Selecting Distint Elements : Using SELECT DISTINCT in SQL queries can sometimes be a necessary solution to eliminate duplicate rows from result sets, but it can also be a performance bottleneck, especially when working with large datasets. Here are some optimization techniques and alternatives to avoid using SELECT DISTINCT

8. Use aggregate functions like GROUP BY when you need to summarize data. This can often be a more efficient way to get distinct results.

6. QUERY 1 : Select Patients with Treatment Amounts Above the Median.

SQL CODE :

WITH MedianCalc AS (

SELECT TreatmentAmount,

NTILE(2) OVER (ORDER BY TreatmentAmount) AS Tile

FROM Patients

),

MedianValue AS (

SELECT AVG(TreatmentAmount) AS Median

FROM MedianCalc

WHERE Tile = 1

)

-- Step 2: Select patients with treatment amounts above the median

SELECT p.PatientID, p.PatientName, p.Disease, p.RecoveryRate, p.TreatmentAmount

FROM Patients p

WHERE p.TreatmentAmount > (SELECT Median FROM MedianValue)

Order By PatientName;

6.1 Before :

A computer screen shot of a number

Description automatically generated

6.2 Optimised Query :

-- Step 1: Calculate the median treatment amount

WITH SortedPatients AS (

SELECT

TreatmentAmount,

ROW\_NUMBER() OVER (ORDER BY TreatmentAmount) AS RowAsc,

ROW\_NUMBER() OVER (ORDER BY TreatmentAmount DESC) AS RowDesc,

COUNT(\*) OVER () AS TotalCount

FROM Patients

),

MedianValue AS (

SELECT AVG(TreatmentAmount) AS Median

FROM SortedPatients

WHERE RowAsc = (TotalCount + 1) / 2 OR RowAsc = (TotalCount + 2) / 2

)

SELECT

p.PatientID,

p.PatientName,

p.Disease,

p.RecoveryRate,

p.TreatmentAmount

FROM Patients p

WHERE p.TreatmentAmount > (SELECT Median FROM MedianValue)

ORDER BY p.PatientName;

6.3 AFTER :

A white background with black text

Description automatically generated

6.4 Techniques Used :

1. The **ROW\_NUMBER()** function is used twice: once in ascending order and once in descending order. This allows for precise identification of median positions regardless of the total count of rows (even or odd).
2. **COUNT(\*) OVER** () AS TotalCount calculates the total number of rows in the dataset. This is crucial for accurately determining the middle values needed for median calculation.
3. The MedianValue CTE accurately identifies the middle row(s) using:

**WHERE RowAsc = (TotalCount + 1) / 2 OR RowAsc = (TotalCount + 2) / 2**

1. For odd counts, it selects the middle row directly.
2. For even counts, it selects the two middle rows and averages their TreatmentAmount values.

7.QUERY 2 : A list of patients who have the highest treatment amount for each disease.

7.1 SQL Code :

SELECT p1.PatientID, p1.PatientName, p1.Disease, p1.RecoveryRate, p1.TreatmentAmount

FROM Patients p1

INNER JOIN Patients p2 ON p1.Disease = p2.Disease

WHERE p1.TreatmentAmount = (SELECT MAX(p3.TreatmentAmount) FROM Patients p3 WHERE p3.Disease = p1.Disease)

ORDER BY p1.PatientName, p1.TreatmentAmount;

7.2 BEFORE :

A white screen with black text

Description automatically generated

7.3 Optimized Query :

-- Step 1: Calculate the maximum TreatmentAmount per Disease

WITH MaxTreatmentByDisease AS (

SELECT Disease, MAX(TreatmentAmount) AS MaxTreatmentAmount

FROM Patients

GROUP BY Disease

)

-- Step 2: Select patients with the maximum TreatmentAmount for their Disease

SELECT p.PatientID, p.PatientName, p.Disease, p.RecoveryRate, p.TreatmentAmount

FROM Patients p

INNER JOIN MaxTreatmentByDisease mtd ON p.Disease = mtd.Disease AND p.TreatmentAmount = mtd.MaxTreatmentAmount

ORDER BY p.PatientName, p.TreatmentAmount;

A white background with black text

Description automatically generated

7.4 Techniques Used :

1. **Common Table Expression (CTE)**: By using a CTE, the maximum TreatmentAmount per disease is computed only once. This reduces the number of redundant calculations compared to the original query.
2. **Reduction of Redundant Calculations**: The original query recalculates the maximum TreatmentAmount for each row, which is inefficient. The optimized query calculates it once and reuses the result.
3. **Simplification**: The optimized query is more straightforward and easier to read. The join condition is clear, and there is no need for an additional self-join as in the original query.

8. QUERY 3: Calculate total number of patients with the same disease and total treatment amount for that disease.

8.1 SQL CODE :

WITH DetailedPatients AS (

SELECT p.PatientID, p.PatientName, p.Disease, p.RecoveryRate, p.TreatmentAmount,

(SELECT COUNT(\*) FROM Patients p2 WHERE p2.Disease = p.Disease) AS PatientCount,

(SELECT SUM(p3.TreatmentAmount) FROM Patients p3 WHERE p3.Disease = p.Disease) AS TotalTreatmentAmount

FROM Patients p

)

SELECT dp.PatientID, dp.PatientName, dp.Disease, dp.RecoveryRate, dp.TreatmentAmount,

dp.PatientCount, dp.TotalTreatmentAmount

FROM DetailedPatients dp

INNER JOIN Patients p ON dp.Disease = p.Disease

ORDER BY dp.TreatmentAmount DESC;

8.2 BEFORE :

A white background with black text

Description automatically generated

8.3 Optimized query :

WITH DiseaseAggregates AS (

SELECT Disease,

COUNT(\*) AS PatientCount,

SUM(TreatmentAmount) AS TotalTreatmentAmount

FROM Patients

GROUP BY Disease

)

SELECT p.PatientID, p.PatientName, p.Disease, p.RecoveryRate, p.TreatmentAmount,

da.PatientCount, da.TotalTreatmentAmount

FROM Patients p

INNER JOIN DiseaseAggregates da ON p.Disease = da.Disease

ORDER BY p.TreatmentAmount DESC;

A black text with numbers and a white background

Description automatically generated with medium confidence8.4 Techniques Used :

1. The final SELECT statement joins Patients with the precomputed aggregates in DiseaseAggregates using INNER JOIN. This is more efficient and straightforward than joining the detailed information derived from repeated subqueries.
2. The ORDER BY clause remains the same, ensuring the final result is ordered by TreatmentAmount in descending order, which is efficient due to the reduced dataset after optimization.
3. By using DiseaseAggregates, the need for subqueries in the main selection is eliminated. The counts and sums are computed once per disease rather than once per patient.
4. The DiseaseAggregates CTE pre-computes the PatientCount and TotalTreatmentAmount for each Disease using aggregate functions (COUNT(\*) and SUM(TreatmentAmount)). This aggregation is done in a single scan of the Patients table, reducing the computational overhead significantly.

9. QUERY 4 : The query retrieves details about patients who are treated by the top 2 doctors (with even DoctorIDs) in terms of patient count.

9.1 SQL CODE :

WITH DoctorStats AS (

SELECT

D.DoctorID,

D.DoctorName,

COUNT(P.PatientID) AS TotalPatients,

AVG(P.RecoveryRate) AS AvgRecoveryRate,

SUM(P.TreatmentAmount) AS TotalTreatmentCost,

ROW\_NUMBER() OVER (PARTITION BY D.DoctorID ORDER BY COUNT(P.PatientID) DESC) AS RankByPatientCount

FROM

Doctors D

LEFT JOIN

Patients P ON D.DoctorID = P.DoctorID

WHERE

D.DoctorID % 2 = 0 -- Select only doctors with even DoctorID

GROUP BY

D.DoctorID, D.DoctorName

),

TopDoctors AS (

SELECT

DoctorID,

DoctorName,

TotalPatients,

AvgRecoveryRate,

TotalTreatmentCost,

RankByPatientCount

FROM

DoctorStats

WHERE

RankByPatientCount <= 2 -- Select top 2 doctors by patient count

)

SELECT

P.PatientID,

P.PatientName,

P.Disease,

P.RecoveryRate,

P.TreatmentAmount,

D.DoctorID,

D.DoctorName,

DS.TotalPatients,

DS.AvgRecoveryRate,

DS.TotalTreatmentCost

FROM

Patients P

JOIN

Doctors D ON P.DoctorID = D.DoctorID

JOIN

DoctorStats DS ON D.DoctorID = DS.DoctorID

JOIN

TopDoctors TD ON D.DoctorID = TD.DoctorID

ORDER BY

DS.TotalPatients DESC, D.DoctorID, P.PatientID;

9.2 BEFORE :

A screenshot of a computer program

Description automatically generated

9.3 Optimized Query :

WITH DoctorStats AS (

SELECT

D.DoctorID,

D.DoctorName,

COUNT(P.PatientID) AS TotalPatients,

AVG(P.RecoveryRate) AS AvgRecoveryRate,

SUM(P.TreatmentAmount) AS TotalTreatmentCost

FROM

Doctors D

LEFT JOIN

Patients P ON D.DoctorID = P.DoctorID

WHERE

D.DoctorID % 2 = 0 -- Select only doctors with even DoctorID

GROUP BY

D.DoctorID, D.DoctorName

),

RankedDoctors AS (

SELECT

DoctorID,

DoctorName,

TotalPatients,

AvgRecoveryRate,

TotalTreatmentCost,

ROW\_NUMBER() OVER (ORDER BY TotalPatients DESC) AS RankByPatientCount

FROM

DoctorStats

)

SELECT

P.PatientID,

P.PatientName,

P.Disease,

P.RecoveryRate,

P.TreatmentAmount,

D.DoctorID,

D.DoctorName,

DS.TotalPatients,

DS.AvgRecoveryRate,

DS.TotalTreatmentCost

FROM

Patients P

JOIN

Doctors D ON P.DoctorID = D.DoctorID

JOIN

DoctorStats DS ON D.DoctorID = DS.DoctorID

JOIN

RankedDoctors RD ON D.DoctorID = RD.DoctorID

WHERE

RD.RankByPatientCount <= 2 -- Select top 2 doctors by patient count

ORDER BY

A screenshot of a computer program

Description automatically generated DS.TotalPatients DESC, D.DoctorID, P.PatientID;

A screenshot of a computer code

Description automatically generated

9.4 Techniques Used :

1. The original query uses ROW\_NUMBER with a PARTITION BY clause, which is unnecessary for ranking doctors by patient count. The optimized query removes this partitioning, simplifying the calculation.
2. The DoctorStats CTE aggregates necessary statistics (TotalPatients, AvgRecoveryRate, TotalTreatmentCost) without computing ROW\_NUMBER within this step.
3. The RankedDoctors CTE performs ranking using ROW\_NUMBER without partitioning, ensuring that the ranking is global across all doctors with even DoctorIDs.
4. The final query joins Patients, Doctors, and precomputed DoctorStats and RankedDoctors only once, eliminating redundant joins and ensuring efficient data retrieval.
5. The filtering for the top 2 doctors by patient count is done directly in the RankedDoctors CTE, making the query more efficient and easier to understand.

10. QUERY 5 : A report that includes doctors' total treatment amounts and details of patients with above-average recovery rates.

10.1 SQL CODE :

WITH DoctorTreatmentTotals AS (

SELECT

D.DoctorID,

D.DoctorName,

SUM(P.TreatmentAmount) AS TotalTreatmentAmount

FROM

Doctors D

JOIN

Patients P ON D.DoctorID = P.DoctorID

GROUP BY

D.DoctorID, D.DoctorName

),

PatientAboveAvgRecovery AS (

SELECT

P.PatientID,

P.PatientName,

P.Disease,

P.RecoveryRate,

P.TreatmentAmount,

P.DoctorID -- Include DoctorID in the select list

FROM

Patients P

JOIN (

SELECT AVG(RecoveryRate) AS AvgRecoveryRate

FROM Patients

) AS AvgData ON P.RecoveryRate > AvgData.AvgRecoveryRate

)

SELECT

DT.DoctorID,

DT.DoctorName,

DT.TotalTreatmentAmount,

PAR.PatientID,

PAR.PatientName,

PAR.Disease,

PAR.RecoveryRate,

PAR.TreatmentAmount

FROM

DoctorTreatmentTotals DT

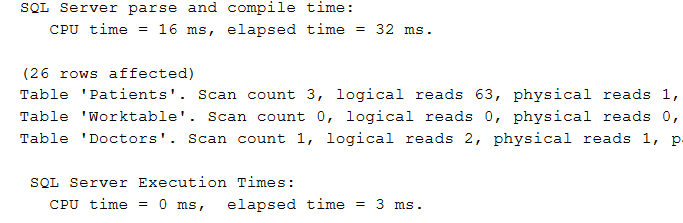
JOIN

PatientAboveAvgRecovery PAR ON DT.DoctorID = PAR.DoctorID -- Join on DoctorID

ORDER BY

DT.DoctorID, PAR.PatientID;

10.2 BEFORE :



10.3 AFTER :

CREATE INDEX idx\_DoctorID ON Patients(DoctorID);

CREATE INDEX idx\_RecoveryRate ON Patients(RecoveryRate);

WITH AvgRecoveryRate AS (

SELECT AVG(RecoveryRate) AS AvgRecoveryRate

FROM Patients

),

DoctorTreatmentTotals AS (

SELECT

D.DoctorID,

D.DoctorName,

SUM(P.TreatmentAmount) AS TotalTreatmentAmount

FROM

Doctors D

JOIN

Patients P ON D.DoctorID = P.DoctorID

GROUP BY

D.DoctorID, D.DoctorName

),

PatientAboveAvgRecovery AS (

SELECT

P.PatientID,

P.PatientName,

P.Disease,

P.RecoveryRate,

P.TreatmentAmount,

P.DoctorID

FROM

Patients P

CROSS JOIN

AvgRecoveryRate A

WHERE

P.RecoveryRate > A.AvgRecoveryRate

)

SELECT

DT.DoctorID,

DT.DoctorName,

DT.TotalTreatmentAmount,

PAR.PatientID,

PAR.PatientName,

PAR.Disease,

PAR.RecoveryRate,

PAR.TreatmentAmount

FROM

DoctorTreatmentTotals DT

JOIN

PatientAboveAvgRecovery PAR ON DT.DoctorID = PAR.DoctorID

ORDER BY

DT.DoctorID, PAR.PatientID;

A white background with black text

Description automatically generated

* 1. Techniques Used :

1. **Indexing**: Creating indexes on DoctorID and RecoveryRate significantly improves the performance of join and filtering operations. Indexes allow the database to quickly find rows with specific values, reducing the time needed for these operations.
2. The average recovery rate is computed once in the AvgRecoveryRate CTE, rather than being recalculated multiple times during the join operation. This reduces computational overhead and simplifies the main query.
3. Using a CROSS JOIN with the precomputed average recovery rate ensures that the average is only calculated once and used efficiently in the filtering condition. This approach simplifies the logic and improves performance.
4. The final SELECT statement joins DoctorTreatmentTotals and PatientAboveAvgRecovery on DoctorID, ensuring that the required data is retrieved efficiently. By using precomputed values and indexes, the join operations are optimized for better performance.

11. QUERY 6 : Summarize the key statistics for each doctor regarding their patient count, average recovery rate, and total treatment cost.

11.1 SQL CODE :

WITH DoctorStats AS (

SELECT

D.DoctorID,

D.DoctorName,

COUNT(P.PatientID) AS TotalPatients,

AVG(P.RecoveryRate) AS AvgRecoveryRate,

SUM(P.TreatmentAmount) AS TotalTreatmentCost,

ROW\_NUMBER() OVER (ORDER BY COUNT(P.PatientID) DESC) AS RankByPatientCount

FROM

Doctors D

LEFT JOIN

Patients P ON D.DoctorID = P.DoctorID

GROUP BY

D.DoctorID, D.DoctorName

),

PatientDetails AS (

SELECT

P.PatientID,

P.PatientName,

P.Disease,

P.RecoveryRate,

P.TreatmentAmount,

P.DoctorID

FROM

Patients P

)

SELECT

DS.DoctorID,

DS.DoctorName,

DS.TotalPatients,

DS.AvgRecoveryRate,

DS.TotalTreatmentCost,

PD.PatientID,

PD.PatientName,

PD.Disease,

PD.RecoveryRate AS PatientRecoveryRate,

PD.TreatmentAmount AS PatientTreatmentAmount

FROM

DoctorStats DS

LEFT JOIN

PatientDetails PD ON DS.DoctorID = PD.DoctorID

ORDER BY

DS.TotalPatients DESC, DS.DoctorID, PD.PatientID;

11.2 BEFORE :

A white background with black text

Description automatically generated

11.3 AFTER :

CREATE INDEX idx\_DoctorID ON Patients(DoctorID);

CREATE INDEX idx\_PatientID ON Patients(PatientID);

WITH DoctorStats AS (

SELECT

D.DoctorID,

D.DoctorName,

COUNT(P.PatientID) AS TotalPatients,

AVG(P.RecoveryRate) AS AvgRecoveryRate,

SUM(P.TreatmentAmount) AS TotalTreatmentCost,

ROW\_NUMBER() OVER (ORDER BY COUNT(P.PatientID) DESC) AS RankByPatientCount

FROM

Doctors D

LEFT JOIN

Patients P ON D.DoctorID = P.DoctorID

GROUP BY

D.DoctorID, D.DoctorName

)

SELECT

DS.DoctorID,

DS.DoctorName,

DS.TotalPatients,

DS.AvgRecoveryRate,

DS.TotalTreatmentCost,

P.PatientID,

P.PatientName,

P.Disease,

P.RecoveryRate AS PatientRecoveryRate,

P.TreatmentAmount AS PatientTreatmentAmount

FROM

DoctorStats DS

LEFT JOIN

Patients P ON DS.DoctorID = P.DoctorID

ORDER BY

DS.TotalPatients DESC, DS.DoctorID, P.PatientID;

A computer screen shot of a computer code

Description automatically generated

* 1. **Techniques Used :**

**1.**  Creating indexes on DoctorID and PatientID significantly improves the performance of join operations. Indexes allow the database to quickly find rows with specific values, reducing the time needed for these operations.

**2. Removing Redundant CTE**: The PatientDetails CTE is removed because it is unnecessary. The patient details can be directly retrieved in the final join, simplifying the query.

**3.** The final query directly joins DoctorStats with Patients, eliminating the need for the intermediate PatientDetails CTE. This reduces redundancy and improves query performance.

**4.** Efficient Use of Window Functions :The use of ROW\_NUMBER() to rank doctors by patient count is retained but simplified by directly ordering the counts. This ensures that the ranking is efficient and straightforward.

**12. CONCLUSION :**

In this SQL project, we developed a comprehensive reporting system for a hospital database, focusing on analyzing the performance of doctors and the treatment outcomes of their patients. The project involved several key steps, including the creation of indexes, the use of Common Table Expressions (CTEs), and the joining of tables to generate detailed reports.

We explored methods to measure query execution time across different database systems, ensuring our solution is performant and scalable.

Tools like EXPLAIN ANALYZE, SHOW PROFILE, and SET STATISTICS TIME were suggested for performance profiling.

This SQL project has laid a solid foundation for a robust reporting and analysis system within a hospital setting. The methodologies and techniques applied ensure efficient data processing, insightful reporting, and a scalable framework for future enhancements. This system can significantly contribute to improved decision-making, enhanced patient care, and optimized resource management in a hospital environment.