Lab Session 09

9. Query Optimization

Aim:

The aim of this experiment is to understand and apply various query optimization strategies to enhance the performance of database queries.

Description:

This involves analyzing the execution plans of queries, identifying bottlenecks, and implementing optimization techniques such as indexing, query rewriting, and the use of efficient algorithms. By the end of the experiment, students should be able to:

- Understand the importance of query optimization in database management systems (DBMS) and how it impacts overall system performance.
- Analyze query execution plans to identify performance bottlenecks and inefficient operations.
- Implement indexing strategies to improve the speed of data retrieval operations.
- Apply query rewriting techniques to transform queries into more efficient forms without changing their semantics.
- Evaluate the performance improvements achieved through various optimization strategies by comparing query execution times before and after optimization.

Pre-Requisites: PostgreSQL, TerraER Tool, Windows/ Ubuntu/CentOS/Debian, DBMS Concepts.

Pre-Lab:

1) What is query optimization in the context of a DBMS?

Choosing the fastest way to run a SQL query.

2) Why is query optimization important for database performance?

It boosts speed and reduces resource use.

3) Explain the role of the query optimizer in a DBMS.

Picks the best plan to run a query efficiently.

4) What is an execution plan in the context of query optimization?

A detailed map of how the query will run.

5) What is query rewriting, and how can it improve query performance?

Changing a query to make it faster without changing the result.

6) What is the purpose of analyzing query execution plans?

To find and fix slow parts of a quer

In-Lab:

1) Query optimization strategies for improving query performance.

a) Analyzing Execution Plans

Objective: Understand how to generate and analyze execution plans to identify performance bottlenecks.

Steps:

1. Create and Populate Tables

2. Write Complex SQL Queries

3. Generate Execution Plans.

Questions:

1) Create tables for employee and department with appropriate key constraints?

2) Write Complex SQL Queries Involving Joins and Subqueries.

3) Use the EXPLAIN command to generate execution plans for each query.

b). Identifying Performance Bottlenecks

Objective: Identify performance bottlenecks in SQL queries and understand their impact on query performance.

Questions:

1) Create tables for customers and orders with appropriate key constraints?

2) To write SQL queries with potential performance issues, identify these issues using the EXPLAIN command, and understand their impact on query performance.

c). Query Optimization

Query Optimization in a Sales Database

Objective: To optimize a query in a sales database by identifying and resolving performance bottlenecks.

Apply Optimization Techniques:

- Add indexes on frequently queried columns.
- Rewrite complex joins and subqueries.

Use EXISTS instead of IN for subqueries when appropriate

1) **Create tables for customers and orders with appropriate key constraints?**
2) **Write a query to retrieve sales data for products in a specific category**

3) **Create appropriate indexes to optimize the query.**

4) **Measure and compare the performance before and after optimization using execution plans and query execution times.**

5) **Optimize SQL queries based on identified performance bottlenecks to improve query efficiency.**

a) Analyzing Execution Plans

1) Create Tables for Employee and Department

```
CREATE TABLE Department (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(100)
);

CREATE TABLE Employee (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    salary DECIMAL(10, 2),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES Department(dept_id)
);
```

2) Complex SQL Query

```
SELECT e.emp_name, e.salary, d.dept_name
FROM Employee e
JOIN Department d ON e.dept_id = d.dept_id
WHERE e.salary > (
    SELECT AVG(salary) FROM Employee WHERE dept_id =
e.dept_id
);
```

3) Execution Plan

```
EXPLAIN
SELECT e.emp_name, e.salary, d.dept_name
FROM Employee e
JOIN Department d ON e.dept_id = d.dept_id
WHERE e.salary > (
    SELECT AVG(salary) FROM Employee WHERE dept_id =
e.dept_id
);
```

b) Identifying Performance Bottlenecks

1) Create Tables for Customers and Orders

```
CREATE TABLE Customer (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(100)
);

CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    order_date DATE,
    customer_id INT,
    amount DECIMAL(10, 2),
    FOREIGN KEY (customer_id) REFERENCES
Customer(customer_id)
);
```

2) Query with Performance Issue

```
SELECT *
FROM Customer
WHERE customer_id IN (
    SELECT customer_id FROM Orders WHERE amount > 1000
);
```

c) Query Optimization in Sales Database

1) Create Tables

```
CREATE TABLE Product (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100),
    category VARCHAR(100)
);
```

```
CREATE TABLE Sales (
    sale_id INT PRIMARY KEY,
    product_id INT,
    customer_id INT,
    sale_date DATE,
    amount DECIMAL(10, 2),
    FOREIGN KEY (product_id) REFERENCES Product(product_id),
    FOREIGN KEY (customer_id) REFERENCES
Customer(customer_id)
);
```

2) Sales Query for Specific Category

```
SELECT p.product_name, s.amount, s.sale_date
FROM Sales s
JOIN Product p ON s.product_id = p.product_id
WHERE p.category = 'Electronics';
```

3) Create Indexes

```
CREATE INDEX idx_product_category ON Product(category);
CREATE INDEX idx_sales_product ON Sales(product_id);
```

4) Analyze with Execution Plan

```
EXPLAIN ANALYZE
SELECT p.product_name, s.amount, s.sale_date
FROM Sales s
JOIN Product p ON s.product_id = p.product_id
WHERE p.category = 'Electronics';
```

5) Optimized SQL Queries

```
SELECT *
FROM Customer c
WHERE EXISTS (
   SELECT 1 FROM Orders o
   WHERE o.customer_id = c.customer_id AND o.amount > 1000
);
```

```
SELECT customer_name
FROM Customer c
WHERE EXISTS (
   SELECT 1 FROM Orders o
   WHERE o.customer_id = c.customer_id AND o.amount > 1000
);
```

```
SELECT DISTINCT c.customer_name
FROM Customer c
JOIN Orders o ON c.customer_id = o.customer_id
WHERE o.amount > 1000;
```

Viva-Voce Questions:

1) What is the significance of using indexes in SQL queries, and how do they improve query performance?

Indexes speed up data retrieval by avoiding full table scans. They are essential for faster searches, efficient joins, and enforcing constraints.

2) Explain how the EXPLAIN statement can be used to identify performance bottlenecks in SQL queries.

EXPLAIN shows how a query runs—whether it uses indexes, how tables are joined, and how many rows are scanned. It's used to spot performance issues like full table scans or slow joins.

3) What are composite indexes, and when should they be used in query optimization?

Composite indexes cover multiple columns. Use them when queries filter or sort by those columns together. The order of columns in the index matters.

4) How can query performance be affected by the use of functions in the WHERE clause, and what are the alternatives to mitigate this impact?

Using functions (like UPPER() or YEAR()) in WHERE disables indexes, causing slow queries. Instead, use precomputed columns or rewrite queries to avoid functions.

5) Explain the importance of query execution time measurement and the methods to measure and compare query performance before and after optimization.

Measuring execution time helps track performance before and after optimization. Use tools like EXPLAIN ANALYZE, slow query logs, or app-level timing to compare query speed.

Students Signature

(For Evaluator's use only)

Comment of the Evaluator (if Any)	Evaluator's Observation Marks Secured:_____ out of _____ Full Name of the Evaluator: Signature of the Evaluator Date of Evaluation: