Advanced Database

Khaula Molapo

20240001

Week: 3

1.

A retail company uses a business intelligence (BI) system to analyze sales from multiple branches across different regions. The database stores millions of transaction records including product details, timestamps, customer data, and payment methods. Over time, management notices that dashboard reports take too long to load, especially when generating monthly revenue summaries and customer behavior analytics. The slow performance is caused by poorly indexed tables, large joins between sales and customer tables, and inefficient aggregate queries. To optimize performance, the database administrator introduces indexing on frequently filtered columns such as product_id and transaction_date. Partitioning is applied to separate historical and recent data, reducing scan time. Materialized views are created for frequently requested summary reports so that the system retrieves precomputed results instead of recalculating data each time. Query rewriting is also used to eliminate unnecessary subqueries and redundant joins. As a result, report generation becomes faster, system load decreases, and decision-makers can access near real-time analytics.

2.

Cost-based optimization (CBO) is a database query optimization technique where the database management system evaluates multiple possible execution plans and selects the one with the lowest estimated cost. The cost refers to expected resource usage such as CPU processing time, disk I/O, and memory consumption. The optimizer uses statistics about tables, indexes, and data distribution to predict how long each plan will take. Instead of following fixed rules, CBO compares alternatives like join order, access paths, and index usage before deciding on the best approach. This method improves performance because it adapts to actual data patterns and workload conditions. However, its effectiveness depends heavily on accurate database statistics.

3.

Example Query:

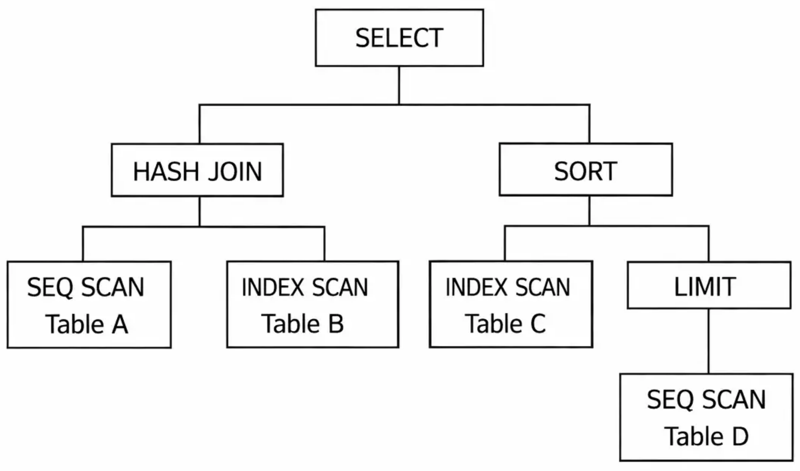
```
SELECT c.customer_name, SUM(s.amount) AS total_spent FROM customers c JOIN sales s ON
c.customer_id = s.customer_id WHERE s.transaction_date >= '2025-01-01' GROUP BY
c.customer_name ORDER BY total_spent DESC;
```

Reflection: While testing this query in PostgreSQL, the initial execution required a full table scan on the sales table, which increased processing time significantly as data volume grew.

After adding indexes on frequently filtered and joined columns, the execution plan changed to use index scans instead of sequential scans. This reduced disk I/O and improved response time noticeably. Using EXPLAIN ANALYZE helped identify bottlenecks and confirm performance improvements. I also learned that selecting only necessary columns reduces memory usage and speeds up sorting operations.

4.

The query plan diagram visually represents how PostgreSQL executes a query step by step. It begins with scanning tables using either sequential scans or index scans depending on available indexes. After retrieving relevant rows, the system performs join operations to combine data from related tables based on matching keys. The diagram also shows aggregation stages where calculations such as SUM or COUNT are applied to grouped records. Sorting operations may appear later in the plan when results must be ordered before being displayed. Understanding the query plan helps identify performance issues such as unnecessary scans, inefficient join methods, or excessive sorting operations.


SELECT
HASH JOIN
SORT
SEQ SCAN Table A
INDEX SCAN Table B
INDEX SCAN Table C
LIMIT
SEQ SCAN Table D