LESSON 45
Database Optimization Techniques

WEEK 09

Introduction to Database Optimization

- **What is Database Optimization?**
  - Database optimization refers to the process of improving the performance of a database system to ensure fast query execution, efficient data retrieval, and effective resource utilization.
- **Why is it important?**
  - Proper optimization leads to reduced load times, better user experience, and lower operational costs.
- **Types of Optimization:**
  - Query optimization, index optimization, schema design optimization, etc.

Query Optimization Techniques

- **Why optimize queries?**
  - Queries can be slow due to inefficient joins, missing indexes, or unnecessary data fetching.
- **How to optimize queries?**
  - Use SELECT only for necessary columns: Avoid SELECT *.
  - Limit the use of joins: Prefer subqueries or indexed joins when possible.
  - Avoid nested queries in SELECT statements: Flatten queries where possible.
- **Example:**
  - Inefficient: SELECT * FROM employees WHERE salary > 100000
  - Optimized: SELECT employee_id, name FROM employees WHERE salary > 100000

Indexing for Performance

- **What are indexes?**
  - Indexes are special lookup tables that speed up data retrieval.
- **How to use indexes effectively?**
  - Create indexes on frequently queried columns (like WHERE, JOIN, and ORDER BY).
  - Use composite indexes for queries that filter by multiple columns.
  - Avoid excessive indexing: Too many indexes slow down INSERT, UPDATE, and DELETE operations.
- **Example:**
  - Create an index on email column: CREATE INDEX idx_email ON users(email);

Schema Design Optimization

- **Why schema design matters?**
  - Proper schema design ensures efficient storage and easy data retrieval.
- **How to optimize schema?**
  - Normalize data to avoid redundancy and maintain consistency.
  - De-normalize when necessary to reduce complex joins.
  - Use partitioning for large tables to improve query performance.
  - Data types: Choose appropriate data types for columns to save space and increase performance.
- **Example:**
  - Using INT instead of BIGINT for columns that don't require such a large range.

Caching for Database Optimization

- **What is caching?**
  - Caching involves storing the results of expensive database queries in memory to speed up repeated access.
- **How to use caching?**
  - Cache query results: Cache frequent queries or complex data calculations.
  - Use application-level caching: Store the results of common database queries in Redis or Memcached.
- **Example:**
  - Cache product details in Redis for quick access in an e-commerce app.

Database Normalization and De-normalization

- **What is normalization?**
  - Normalization is the process of organizing a database to reduce redundancy and improve data integrity.
  - 1NF, 2NF, 3NF: Levels of normalization.
- **When to de-normalize?**
  - De-normalization can improve performance in read-heavy systems by reducing the need for complex joins.
- **Example:**
  - Normalized: Split user and orders tables.
  - De-normalized: Combine user and orders to reduce JOIN overhead in read queries.

Query Execution Plan Analysis

- **What is a Query Execution Plan (QEP)?**
  - A QEP shows how a database will execute a query, including the order of operations and the use of indexes.
- **How to analyze QEP?**
  - Look for full table scans, missing indexes, and expensive operations.
  - Use the EXPLAIN statement to generate the query execution plan.
- **Example:**
  - EXPLAIN SELECT * FROM orders WHERE user_id = 123;

Database Partitioning

- **What is partitioning?**
  - Partitioning involves splitting a large table into smaller, more manageable pieces while keeping the data logically intact.
- **Types of partitioning:**
  - Range Partitioning: Based on a specific range of values (e.g., date ranges).
  - List Partitioning: Based on a list of discrete values.
  - Hash Partitioning: Distributes data evenly across partitions.
- **Example:**
  - Partitioning an orders table by year: CREATE TABLE orders_2021 PARTITION BY RANGE(year);

Reducing Deadlocks and Lock Contention

❖ What are deadlocks?

- Deadlocks occur when two or more transactions are waiting for each other to release locks, resulting in a stalemate.

❖ How to avoid deadlocks?

- Access tables in a consistent order: Avoid transactions locking different tables in different orders.
- Use proper transaction isolation levels: Avoid unnecessary long-running transactions.
- Keep transactions short and fast: Commit or rollback transactions as quickly as possible.

Using Stored Procedures and Triggers

- **What are stored procedures?**
  - Stored procedures are precompiled SQL queries stored in the database for reuse.
- **Benefits:**
  - Reduce network traffic.
  - Improve execution speed by reducing query parsing time.
- **What are triggers?**
  - Triggers automatically execute a predefined action when certain database events occur (INSERT, UPDATE, DELETE).
- **Example:**
  - Use a stored procedure to handle order processing.

Ministry of SMEs and Startups KOSME Korea SMEs and Startups Agency LIKELION JOBKOREA

Backup and Disaster Recovery Optimization

- **Why backup is important for performance?**
  - Efficient backup strategies prevent database lockups during backup and recovery.
- **Backup strategies:**
  - Use incremental backups to avoid full backups each time.
  - Use replicas to offload read queries and backups.
- **Disaster recovery:**
  - Ensure fast recovery using proper database replication and failover mechanisms.

Monitoring Database Performance

- **Key performance indicators (KPIs) to monitor:**
  - Query response time, CPU usage, memory usage, disk I/O, and network latency.
- **Tools for monitoring:**
  - pgAdmin (PostgreSQL), MySQL Workbench (MySQL), Database Performance Analyzer.
  - New Relic, Datadog, and Prometheus for full-stack monitoring.

Best Practices for Database Optimization

- Use proper indexing.
- Avoid unnecessary joins and subqueries.
- Regularly analyze and update your query execution plans.
- Optimize schema design with normalization and de-normalization as needed.
- Use caching for frequently accessed data.Partition large tables for improved query performance.

Query Optimization Example – Avoiding SELECT *

- **Problem:**
  - Using SELECT * retrieves all columns from a table, even if only a few columns are needed.
- **Example (Inefficient Query):**
  - SELECT * FROM users WHERE status = 'active';
  - This retrieves all columns for every active user, which is unnecessary and can slow down the query.
- **Optimized Query:**
  - SELECT id, name, email FROM users WHERE status = 'active’;
  - Only the necessary columns (id, name, email) are selected, improving performance.

Using Indexes for Faster Query Execution

- **Problem:**
  - Without indexes, the database has to perform a full table scan to find the relevant rows.
- **Example (No Index):**
  - SELECT name FROM products WHERE price > 100;
  - If there is no index on price, the database has to scan every row.
- **Solution (Add Index):**
  - CREATE INDEX idx_price ON products(price);
  - This index speeds up queries filtering by price, as the database can now use the index rather than scanning the entire table.

Query Optimization Example – Using Joins Efficiently

- **Problem:**
  - Joins with large datasets can be slow if indexes are not properly used.
- **Example (Inefficient Join):**
  - **SELECT users.name, orders.amountFROM users**
    JOIN orders ON users.id = orders.user_id;
  - If the users.id and orders.user_id columns are not indexed, the join operation can be slow.
- **Solution (Add Indexes):**
  - CREATE INDEX idx_users_id ON users(id);
  - CREATE INDEX idx_orders_user_id ON orders(user_id);
  - By indexing the id column in users and user_id in orders, the join operation becomes faster.

Example of Database Partitioning – Range Partitioning

- **Problem:**
  - A large table with billions of rows can lead to slow query performance, especially on a specific range of data.
- **Example (No Partitioning):**
  - A single orders table contains records for all years.
  - SELECT * FROM orders WHERE order_date BETWEEN '2022-01-01' AND '2022-12-31';
  - This query might be slow if the orders table has millions of rows.
- **Solution (Partitioning by Year):**
  - CREATE TABLE orders_2022 PARTITION BY RANGE(order_date) (

    PARTITION p2022 VALUES LESS THAN ('2023-01-01')

    );
  - By partitioning the table by year, the query can quickly access the data for the year 2022 without scanning the entire table.

Example of Database De-normalization

- **Problem:**
  - Complex joins between normalized tables can result in slow performance, especially in read-heavy applications.
- **Example (Normalized Schema):**
  - In a normalized schema, orders and products are stored in separate tables:

    ```
    SELECT orders.order_id, products.product_name
    FROM orders
    JOIN products ON orders.product_id = products.product_id;
    ```
  - This query requires joining two tables, which may be inefficient if there are many rows.

Example of Database De-normalization

- **Solution (De-normalization):**
  - Combine the orders and products tables into one to eliminate the need for a join:

```
CREATE TABLE orders_with_product AS
SELECT orders.order_id, orders.product_id, products.product_name
FROM orders
JOIN products ON orders.product_id = products.product_id;
```

  - Now, the orders_with_product table contains all necessary information, reducing the need for joins.

Example of Using Caching for Database Optimization

- Problem:
  - Frequent queries on the same data can cause unnecessary load on the database.
- Example (Without Caching):
  - SELECT COUNT(*) FROM orders WHERE status = 'shipped';

Ministry of SMEs and Startups · KOSME Korea SMEs and Startups Agency · LIKELION · JOBKOREA

Example of Using Caching for Database Optimization

❖ Solution (Using Redis for Caching):

- Cache the result of this query using Redis:

```
const redis = require('redis');
const client = redis.createClient();

// Check if the result is cached
client.get('shipped_orders_count', (err, reply) => {
  if (reply) {
    console.log('Cached result:', reply);
  } else {
    // If not cached, run the query and cache the result
    const result = db.query('SELECT COUNT(*) FROM orders WHERE status = "ship
    client.setex('shipped_orders_count', 3600, result); // Cache for 1 hour
  }
});
```

Example of Optimizing Query with Execution Plan

- **Problem:**
  - Without analyzing the query execution plan, you may not realize that a query is performing inefficient operations.

    SELECT * FROM orders WHERE customer_id = 123;
  - This query may perform a full table scan if there is no index on customer_id.
- **Solution (Using EXPLAIN to Analyze Execution Plan):**
  - EXPLAIN SELECT * FROM orders WHERE customer_id = 123;
  - The output will show whether the query is using an index or performing a full table scan. If it's using a full table scan, create an index:
  - CREATE INDEX idx_customer_id ON orders(customer_id);

Example of Using Stored Procedures for Performance

- **Problem:**
  - Repeatedly executing the same complex query in an application can lead to inefficient performance.
- **Example (Without Stored Procedure):**
  - The same query is executed in the application code:

    ```
    SELECT * FROM orders WHERE customer_id = 123 AND status = 'shipped';
    ```
- **Solution (Using Stored Procedure):**
  - Create a stored procedure for the query:

    ```
    CREATE PROCEDURE GetShippedOrders(customer_id INT)
    BEGIN
        SELECT * FROM orders WHERE customer_id = customer_id AND status = 'shipped';
    END;
    ```
  - Now, the query is precompiled and executed more efficiently.

Example of Indexing with Composite Index

- **Problem:**
  - A query filtering by multiple columns may not be optimized if there are no composite indexes.
- **Example (No Composite Index):**
  - SELECT * FROM orders WHERE customer_id = 123 AND status = 'shipped';
  - This query may require scanning the entire table if there are no indexes on both customer_id and status.
- **Solution (Create Composite Index):**
  - CREATE INDEX idx_customer_status ON orders(customer_id, status);
  - This composite index speeds up queries filtering by both customer_id and status.

Conclusion and Best Practices

- **Key Takeaways:**
  - Always analyze query performance and execution plans.
  - Use indexing strategically and avoid over-indexing.
  - Normalize data but consider de-normalization in certain cases for performance.
  - Cache frequently queried data to reduce database load.
  - Use stored procedures for frequently executed, complex queries.
- **Best Practices:**
  - Monitor performance regularly.
  - Keep queries simple and avoid unnecessary complexity.
  - Test database performance in production-like environments.