Term: Fall 2023 **Subject:** Computer Science & Engineering (CSE) **Number:** 512

Course Title: Distributed **D**atabase **S**ystems (CSE 512)

Team: PVS

Part 3: Query Processing and Optimization Techniques

Problem Statement: Implement query processing and optimization techniques to enhance the performance of the distributed database system with your chosen topic.

Task completed: Distributed Indexing, Query Optimization

1. **Distributed Indexing:**

   Distributed indexing is crucial for improving the performance and scalability of databases, especially in distributed or large-scale systems. Indexing enhances query performance by enabling faster data retrieval and efficient search operations. Implementing distributed indexing in MySQL or any distributed database system is important for several reasons:

   1. Improved Query Performance: Indexing allows databases to quickly locate and retrieve specific data, reducing the time required for query execution. In a distributed environment, efficient indexing ensures that queries can be executed across multiple nodes without excessive latency.

   2. Scalability: As data grows, distributing indexes across multiple nodes helps distribute the query load and scale horizontally, enabling databases to handle larger volumes of data and more concurrent users.

   3. Fault Tolerance: Distributed indexing can improve fault tolerance by replicating index data across nodes. If one node fails, others can still serve queries, preventing a complete system outage.

   4. Load Balancing: Distributing indexes can help balance the query load across nodes, preventing bottlenecks and ensuring optimal resource utilization.

   Implementing distributed indexing in MySQL involves several strategies and technologies:

   1. Partitioning: MySQL supports various partitioning strategies (range, hash, list) to divide large tables into smaller, more manageable parts. These partitions can have their own indexes, and data can be distributed across nodes based on the chosen partitioning scheme.

2. Sharding: Sharding involves horizontally partitioning data across multiple database instances or nodes. Each shard can have its own indexes, and queries can be directed to the relevant shard based on a predefined sharding key.

3. MySQL Cluster: MySQL Cluster (NDB) is a distributed, shared-nothing architecture that supports automatic sharding and replication. It offers built-in distributed indexing and partitioning capabilities, allowing indexes to be distributed across nodes for better performance and scalability.

4. Use of NoSQL Solutions: In some cases, integrating NoSQL solutions like Apache Cassandra or MongoDB, which inherently support distributed indexing and scaling, alongside MySQL for specific types of data can be beneficial.

5. Consistent Hashing and Middleware Solutions: Consistent hashing algorithms help distribute data and indexes across nodes in a consistent manner, reducing the need for rebalancing when nodes are added or removed. Middleware solutions or database proxies can also manage distributed queries and indexing.

2. Query Optimization:

Query optimization is crucial for maximizing the efficiency and performance of database systems like MySQL. Optimized queries lead to faster response times, reduced resource consumption, and improved overall system performance. Here's why query optimization is important and how it can be implemented in MySQL:

Importance of Query Optimization:

1. Performance Improvement: Well-optimized queries execute faster, reducing the time it takes to retrieve data and respond to user requests. This enhances the user experience and enables the system to handle more concurrent users or transactions.

2. Resource Utilization: Optimized queries consume fewer system resources such as CPU, memory, and disk I/O. This helps in efficient resource utilization and scalability, allowing the database to handle larger workloads without performance degradation.

3. Cost Reduction: Improved query performance reduces operational costs by requiring fewer hardware resources and minimizing the need for extensive system tuning or upgrades.

Techniques for Query Optimization in MySQL:

1. Use Indexes: Properly designed and utilized indexes can significantly enhance query performance. Identify columns frequently used in WHERE, JOIN, or ORDER BY clauses and create indexes on these columns to speed up data retrieval.

2. Query Rewriting: Reformulate queries to use more efficient syntax and minimize unnecessary operations. Utilize optimized SQL constructs and avoid redundant or inefficient code.

3. Optimize Joins: Ensure that JOIN operations are well-optimized by using appropriate join types (e.g., INNER JOIN, LEFT JOIN) and indexing columns involved in joins. Avoid excessive joins and consider denormalizing data if it improves performance.

4. Avoid SELECT: Instead of selecting all columns from a table, specify only the required columns. This reduces the amount of data fetched and improves query performance.

5. Limit and Pagination: Use LIMIT to restrict the number of rows returned, especially when querying large datasets. Implement pagination for displaying results in chunks rather than fetching the entire dataset at once.

6. Query Cache: Enable MySQL query caching to store the results of frequently executed queries in memory, reducing the need to reprocess identical queries.

7. Optimize Table Structures: Normalize or denormalize database tables based on specific use cases. Normalize to minimize data redundancy and denormalize to improve query performance by reducing JOIN operations.

8. Monitor and Analyze Performance: Use MySQL's built-in tools like EXPLAIN to analyze query execution plans and identify potential bottlenecks. Monitor query performance regularly and optimize based on the identified issues.

9. Use Stored Procedures and Prepared Statements: Stored procedures and prepared statements can improve performance by reducing network traffic and providing reusable query execution plans.

In our project we implemented the distributed indexing and query optimization by doing the appropriate partitioning and including the WHERE clause as well as ORDER BY for faster query processing. We also performed data replication for faster data retrieval. Each partition can have its own indexes, making it easier to access specific subsets of data quickly. Utilizing the WHERE clause efficiently helps in narrowing down the data that needs to be retrieved, reducing the amount of unnecessary data processing. Additionally, using ORDER BY in queries can benefit from indexes, as sorted data can be retrieved more efficiently. Replicating data across multiple nodes ensures data availability and improves fault tolerance. It allows queries to be served from multiple replicas, reducing the load on individual nodes, and improving data retrieval speed, especially for read-heavy workloads. By combining distributed indexing, query optimization techniques, and data replication, our project has likely achieved significant improvements in query performance, scalability, fault tolerance, and overall system efficiency in a distributed environment.

```
STOCK_PRICE_HISTORICAL_DATA_TABLE_PARTITION_QUERY = ("ALTER TABLE StockPriceHistory "
                                                     "PARTITION BY LIST COLUMNS(StockSymbol) ("
                                                     "PARTITION aapl VALUES IN ('AAPL'),"
                                                     "PARTITION googl VALUES IN ('GOOGL'),"
                                                     "PARTITION msft VALUES IN ('MSFT'),"
                                                     "PARTITION tcs VALUES IN ('TCS'));")
pc.horizontal_partitioning(conn, STOCK_PRICE_HISTORICAL_DATA_TABLE_PARTITION_QUERY)
```

Here, horizontal partitioning has been created on stock symbol so that while inserting the data or retrieving the data the query processing can be made faster. Also, this partitioning creates indices which can again help identify the location of the cluster of the data.