Week 3, Backend for high load

Azamat Serek, PhD, Assist.Prof.

Database design and optimization

Database often becomes the bottleneck in software performance. Having an optimized database is essential for high performing systems.

Indexing

Create Indexes: Indexes are data structures that provide a quick lookup mechanism, significantly improving query performance. They work by creating a sorted data structure that allows the database engine to quickly locate the rows that satisfy a WHERE clause. While indexes speed up SELECT queries, they may slow down write operations, so it's crucial to strike a balance between read and write performance.

```
CREATE INDEX idx_username ON users(username);
```

Composite indexes

Use Composite Indexes: Composite indexes involve multiple columns and are useful for queries that filter or sort based on multiple conditions. This reduces the need for separate indexes on each column and improves the efficiency of the query planner.

```
CREATE INDEX idx_name_age ON employees(name, age);
```

Normalization

Normalization: This process organizes data to minimize redundancy and dependency, reducing the likelihood of data anomalies. By breaking down large tables into smaller, related ones, normalization ensures data consistency. However, it may lead to more complex queries.

Example of 3rd Normal Form

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(100),
    address VARCHAR(255)
);

CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

Denormalization

Denormalization: While normalization reduces redundancy, denormalization introduces controlled redundancy to improve query performance, especially for read-heavy operations. This can involve adding redundant columns or tables strategically.

```
CREATE TABLE denormalized_orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    customer_name VARCHAR(100),
    order_date DATE
);
```

Query optimization

Optimize Queries: Regularly analyze and optimize frequently used queries. Use tools like EXPLAIN to understand the query execution plan and identify areas for improvement.

```
EXPLAIN SELECT * FROM orders WHERE customer_id = 123;
```

*Avoid SELECT *:* Retrieve only the necessary columns rather than selecting all columns. This reduces the amount of data transferred and processed, improving query performance.

```
SELECT order_id, order_date FROM orders WHERE customer_id = 123;
```

Partitioning

Partition Tables: Partitioning involves dividing large tables into smaller, more manageable pieces. This can significantly improve query performance by allowing the database engine to work on smaller subsets of data, leading to faster query execution.

```
CREATE TABLE sales (
    sale_id INT PRIMARY KEY,
    sale_date DATE,
    amount DECIMAL(10, 2)
) PARTITION BY RANGE (YEAR(sale_date)) (
    PARTITION p0 VALUES LESS THAN (1990),
    PARTITION p1 VALUES LESS THAN (2000),
    PARTITION p2 VALUES LESS THAN (2010),
    PARTITION p3 VALUES LESS THAN (2020),
    PARTITION p4 VALUES LESS THAN (MAXVALUE)
);
```

Caching

Query Caching: Implement a caching mechanism to store the results of frequently executed queries. This reduces the load on the database by serving cached results, enhancing response times.

```
-- Pseudocode
DECLARE @cacheKey NVARCHAR(255) = 'query_cache_key';
DECLARE @cachedResult NVARCHAR(MAX);

SET @cachedResult = REDIS.GET(@cacheKey);

IF @cachedResult IS NULL
BEGIN
    -- Execute the query and store the result in the cache
    SET @cachedResult = EXECUTE_QUERY('SELECT * FROM large_table');
    REDIS.SET(@cacheKey, @cachedResult, EXPIRY_TIME);
END
```

Object Caching: Cache frequently accessed objects or data in the application layer to minimize database queries. This can be achieved using in-memory caching libraries or frameworks.

```
from django.core.cache import cache

def get_user_data(user_id):
    # Try to fetch user data from cache
    user_data = cache.get(f'user_{user_id}')

    if user_data is None:
        # If not in cache, fetch from the database
        user_data = User.objects.get(id=user_id)

        # Store the data in cache for future requests
        cache.set(f'user_{user_id}', user_data, TIMEOUT)

    return user_data
```

Regular Maintenance:

Update Statistics: Keeping statistics up-to-date is crucial for the query planner to make informed decisions about execution plans. Regularly update statistics to ensure accurate and efficient query optimization.

```
-- Update statistics for a table
UPDATE STATISTICS table_name;
```

Data Archiving: Archive or purge old data that is no longer needed. This can improve query performance and reduce storage requirements, especially in systems with large historical datasets.

```
-- Archive data older than a certain date
DELETE FROM historical_data WHERE date < '2020-01-01';
```

Hardware optimization

Optimize Server Configuration: Adjust database server settings and configurations based on the workload and hardware capabilities. This includes parameters such as buffer sizes, cache settings, and connection limits.

```
-- Example: Increase the size of the query cache
SET GLOBAL query_cache_size = 256M;
```

Use SSDs: Consider using Solid State Drives (SSDs) for storage. SSDs provide faster data access compared to traditional Hard Disk Drives (HDDs), resulting in improved overall database performance.

Concurrency Control:

Isolation Levels: Adjust isolation levels based on the requirements of your application. Isolation levels control the visibility of changes made by one transaction to other transactions. Choosing the appropriate isolation level is crucial for balancing consistency and performance.

```
-- Set isolation level to READ COMMITTED
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

Connection Pooling:

Use Connection Pooling: Reuse database connections to avoid the overhead of establishing new connections for each request. Connection pooling helps manage and reuse database connections efficiently.

```
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:mysql://localhost:3306/database");
config.setUsername("username");
config.setPassword("password");
config.setMaximumPoolSize(10);

HikariDataSource dataSource = new HikariDataSource(config);
```

Database Design:

Efficient Schema Design: Design the database schema with performance in mind. Optimize data types, use appropriate constraints, and minimize unnecessary relationships. A well-designed schema can significantly impact query efficiency.

```
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(255),
    price DECIMAL(10, 2),
    -- Additional columns as needed
);
```

Monitoring and Profiling:

Regular Monitoring: Implement monitoring tools to track database performance over time. Regularly monitor key metrics such as CPU usage, memory usage, and query execution times to identify potential issues.

```
SHOW STATUS LIKE 'cpu%';
```

Profiling Queries: Profile and analyze the performance of individual queries to pinpoint bottlenecks. Tools like the MySQL Performance Schema can provide detailed insights into query execution.

Profiling

Profiling Queries: Profile and analyze the performance of individual queries to pinpoint bottlenecks. Tools like the MySQL Performance Schema can provide detailed insights into query execution.

```
-- Enable Performance Schema
SET GLOBAL performance_schema = ON;

-- Profile a specific query
SELECT * FROM orders WHERE customer_id = 123;
```

References

https://danielfoo.medium.com/11-database-optimization-techniques-97fdbed1b627