**Query optimization**

**1. Indexing**

Use Indexes: Indexes can significantly speed up data retrieval. Ensure that columns used in WHERE, JOIN, and ORDER BY clauses are indexed.

-- Creating an index on the 'email' column in the 'users' table

CREATE INDEX idx\_email ON users(email);

**Scenario:** If you frequently query users by email, adding an index on the email column speeds up the search process.

Avoid Over-Indexing: While indexes improve read performance, they can slow down INSERT, UPDATE, and DELETE operations. Only index the columns that are frequently searched.

-- Avoid creating unnecessary indexes

CREATE INDEX idx\_age ON users(age);

**Scenario:** If you rarely search by age, avoid indexing this column as it could slow down INSERT and UPDATE operations.

**2. Query Structure**

Select Only Required Columns: Avoid SELECT \*. Specify only the columns you need.

-- Bad practice: Selecting all columns

SELECT \* FROM users;

-- Good practice: Selecting only necessary columns

SELECT first\_name, last\_name, email FROM users;

Scenario: If you only need the user’s name and email, specifying only these columns reduces the amount of data processed.

Use EXPLAIN: Use the EXPLAIN command to analyze how MySQL executes your queries and adjust them based on the output.

-- Using EXPLAIN to understand query execution

EXPLAIN SELECT first\_name FROM users WHERE email = 'example@example.com';

**Scenario:** This helps you see if MySQL is using an index or if there’s a full table scan.

Avoid Subqueries: Wherever possible, use JOIN instead of subqueries, as JOIN is usually faster.

-- Subquery approach (slower)

SELECT id FROM users WHERE id IN (SELECT user\_id FROM orders WHERE total > 100);

-- JOIN approach (faster)

SELECT users.id FROM users JOIN orders ON users.id = orders.user\_id WHERE orders.total > 100;

Scenario: Using JOIN is generally faster than a subquery, especially with large datasets.

Optimize Joins: Ensure that joined columns are indexed and try to limit the number of joins.

-- Ensure columns used in JOINs are indexed

CREATE INDEX idx\_user\_id ON orders(user\_id);

Scenario: Indexing the user\_id in the orders table speeds up the JOIN operation.

**3. Data Types and Schema Design**

Choose Proper Data Types: Use the smallest data type that can store your data. For example, use TINYINT instead of INT when the range of values permits.

-- Using smaller data types

CREATE TABLE products (

id INT UNSIGNED AUTO\_INCREMENT PRIMARY KEY,

stock TINYINT UNSIGNED,

price DECIMAL(10,2)

);

**Scenario:** TINYINT for stock levels is appropriate if stock values range between 0-255, saving space and improving performance.

Normalization: Normalize your database to avoid redundancy, but also consider denormalization in cases where it significantly improves performance.

-- Normalized schema

CREATE TABLE customers ( id INT PRIMARY KEY,

name VARCHAR(100));

CREATE TABLE orders (

id INT PRIMARY KEY,

customer\_id INT,

FOREIGN KEY (customer\_id) REFERENCES customers(id)

);

**Scenario:** This design avoids redundant data but can be slower for complex queries. Consider denormalization where needed for performance.

**4. Query Cache**

Enable Query Cache: If your queries are read-heavy, enable MySQL’s query cache. However, note that query cache is removed in MySQL 8.0, so this is only applicable to older versions.

SET GLOBAL query\_cache\_size = 1048576;

query\_cache\_type = ON;

Scenario: If you run the same query multiple times, enabling query cache can improve performance.

**5. Partitioning and Sharding**

Partitioning: If your tables are very large, consider partitioning them to improve performance.

-- Partitioning a large table by range

CREATE TABLE orders ( id INT, order\_date DATE, customer\_id INT, amount DECIMAL(10,2) )

PARTITION BY RANGE (YEAR(order\_date)) (

PARTITION p0 VALUES LESS THAN (2000),

PARTITION p1 VALUES LESS THAN (2010),

PARTITION p2 VALUES LESS THAN MAXVALUE );

Scenario: Partitioning the orders table by year can make queries faster by scanning only relevant partitions.

Sharding: For extremely large databases, consider database sharding to distribute data across multiple servers.

**Scenario:** If your database grows too large for a single server, you might shard it. For example, users with IDs 1-10000 on one server and 10001-20000 on another.

**6. Optimizing Specific Queries**

Limit the Result Set: Use LIMIT to limit the number of rows returned, especially in large tables.

-- Limiting the number of rows returned

SELECT \* FROM orders ORDER BY order\_date DESC LIMIT 10;

**Scenario:** If you only need the most recent 10 orders, using LIMIT reduces the number of rows processed.

Avoid Wildcards in LIKE: Full wildcards (e.g., %term%) can be slow. If possible, use term% to allow MySQL to use an index.

-- Slow: Full wildcard search

SELECT \* FROM users WHERE email LIKE '%example.com%';

-- Faster: Leading wildcard search

SELECT \* FROM users WHERE email LIKE 'example.com%';

**Scenario:** The second query allows MySQL to use an index if available, making it faster.

Use IN Instead of Multiple OR: For conditions checking against multiple values, IN is often faster than multiple OR conditions.

-- Using OR (slower)

SELECT \* FROM users WHERE age = 20 OR age = 25 OR age = 30;

-- Using IN (faster)

SELECT \* FROM users WHERE age IN (20, 25, 30);

**Scenario:** Using IN simplifies the query and is usually faster than multiple OR conditions.

**7. Connection Management**

Persistent Connections: Use persistent connections to avoid the overhead of establishing new connections repeatedly.

Connection Pooling: Implement connection pooling to manage multiple database connections efficiently.

**8. Monitoring and Adjustments**

Monitor Slow Queries: Enable MySQL’s slow query log to identify queries that are taking too long.

Analyze Query Performance: Regularly review query performance and adjust as your data grows and usage patterns change.

**9. Hardware and Configuration**

Optimize Server Configuration: Adjust MySQL server settings like innodb\_buffer\_pool\_size, query\_cache\_size, and others based on your workload.

Upgrade Hardware: In some cases, upgrading hardware (CPU, RAM, SSDs) can significantly improve performance.

**10. Use Performance Tools**

MySQL Workbench: Use MySQL Workbench for visual query analysis and optimization.

Third-Party Tools: Tools like Percona Toolkit, pt-query-digest, and others can help in optimizing and analyzing MySQL performance.