Name: Nikhil Dnyandev Gaikwad

E-mail: nikhilgaikwad9850@gmail.com

Branch: Computer Science

College Name: JSPM Rajarshi Shahu College of Engineering, Pune

1. Database Query Optimization

Scenario: You have a PostgreSQL database with a users table and an orders table. The orders table has a foreign key reference to the users table. Write a SQL query to fetch the top 5 users with the highest total order amount in the last month. Assume that the orders table has columns user_id, amount, and order_date.

Requirements:

- **Write the SQL query.**

```
SELECT u.*, SUM(o.amount) AS total_order_amount
FROM users u
JOIN orders o ON u.id = o.user_id
WHERE o.order_date >= NOW() - INTERVAL '1 month'
GROUP BY u.id
ORDER BY total_order_amount DESC
LIMIT 5;
```

- **Explain how you would index the tables to optimize the query.**

To optimize query performance through indexing, I would follow a structured approach:

1. **Analyze Query Patterns**: First, I'd examine the most frequently executed queries, focusing on the columns used in WHERE, JOIN, ORDER BY, and GROUP BY clauses. This helps identify which columns could benefit most from indexing.
2. **Select Index Types**:
    - **Single-Column Indexes**: I'd create indexes on individual columns that are often queried alone.

- **Composite Indexes**: For queries that filter or sort by multiple columns, I would create composite indexes. The order of columns is important; I would place the most selective columns first to improve efficiency.

3. **Assess Index Selectivity**: I would target columns with high selectivity (i.e., many unique values), as these are more effective for reducing the number of rows scanned during queries.
4. **Use Covering Indexes**: If a query retrieves only specific columns, I would consider creating covering indexes that include those columns. This allows the database to satisfy the query directly from the index, bypassing the need to access the table itself.
5. **Review Execution Plans**: I would utilize tools like EXPLAIN to analyze the execution plans of my queries. This would help identify potential performance bottlenecks, such as full table scans that could be optimized with indexing.
6. **Monitor Index Usage**: After implementing indexes, I would monitor their usage. If certain indexes are rarely used or not at all, I'd consider removing them, as they can slow down write operations.
7. **Balance Read and Write Operations**: I'd keep in mind that while indexes enhance read performance, they can impact write operations (like INSERT and UPDATE). I'd strive for a balance based on the application's specific workload—whether it leans more toward reads or writes.
8. **Regular Maintenance**: Finally, I'd establish a regular maintenance schedule to rebuild or reorganize indexes, especially for tables that undergo frequent updates. This helps ensure that indexes remain efficient over time.

1. Index on orders.user_id:

   CREATE INDEX idx_orders_user_id ON orders (user_id);

   This index is crucial because it speeds up the JOIN operation between the users and orders tables. When the database executes the JOIN, it can quickly locate all orders for a specific user without scanning the entire orders table.

2. Index on orders.order_date:

   CREATE INDEX idx_orders_order_date ON orders (order_date);

   This index helps optimize the WHERE clause that filters orders from the last month. It allows the database to quickly identify and retrieve only the relevant orders without scanning the entire table.

3. Composite index on orders (user_id, order_date):

   CREATE INDEX idx_orders_user_id_order_date ON orders (user_id, order_date);

   This composite index is particularly useful because it covers both the JOIN condition and the WHERE clause filter. It allows the database to satisfy both conditions using a single index, which can be more efficient than using two separate indexes.

4. Index on users.id:

   CREATE INDEX idx_users_id ON users (id);

This index is likely already present as it's probably the primary key of the users table. If not, adding this index will speed up the JOIN operation from the users' side.

5. Consider a covering index:

   CREATE INDEX idx_orders_covering ON orders (user_id, order_date, amount);

   This covering index includes all columns from the orders table that are used in the query. It can potentially eliminate the need to access the actual table data, as all required information is in the index itself.

   Additional Optimization Considerations:

6. Partial index: If a significant portion of orders are older than a month, you could create a partial index:

   CREATE INDEX idx_recent_orders ON orders (user_id, order_date, amount) WHERE order_date >= NOW() - INTERVAL '1 month';

   This index only includes recent orders, making it smaller and potentially faster for this specific query.

2. REST API Design

Scenario: You need to design a REST API endpoint to retrieve user profile information. The endpoint should support fetching a user's profile by their username and include optional query parameters for filtering the results.

Requirements: v

- ### Define the endpoint URL and HTTP method.

  Endpoint URL and HTTP Method:

  GET /users/{username}/profile

  HTTP Method: GET

  Endpoint URL: /users/{username}/profile

  Path Parameter: username (required)

  Explanation:

  We use the GET method because we're retrieving data, not modifying it.

  The endpoint URL is structured to clearly indicate that we're retrieving a user's profile information. The {username} path parameter is used to identify the specific user.

  By using a path parameter for the username, we make it easy to cache and optimize the request.

  Optional Query Parameters:

  To support filtering the results, we can add optional query parameters. Here are a few examples:

  fields: a comma-separated list of fields to include in the response (e.g., fields=name,email,location)

  expand: a boolean flag to include additional information, such as the user's friends or followers (e.g., expand=true)

  lang: a language code to return the profile information in a specific language (e.g., lang=fr)

  These query parameters can be added to the URL as follows:

  GET /users/{username}/profile?fields=name,email&expand=true&lang=fr

- ### Specify the expected request parameters and response structure.

  #### Request Parameters

  - **Path Parameter**:
    username: (required) The unique identifier for the user whose profile is being retrieved.
  - **Query Parameters** (optional):

- fields: (optional) A comma-separated list of specific fields to include in the response. For example, fields=firstName,lastName,email.
- includePosts: (optional, boolean) A flag indicating whether to include the user's posts in the response. Defaults to false.

Response Structure

- **Success Response** (HTTP Status 200):
  json
  Copy code
  ```
  {
   "username": "john_doe",
   "firstName": "John",
   "lastName": "Doe",
   "email": "john@example.com",
   "bio": "Software Developer",
   "posts": [] // Only included if includePosts=true
  }
  ```

- **Error Response** (HTTP Status 404 - Not Found):
  json
  Copy code
  ```
  {
   "error": "User not found",
   "message": "No user exists with the username 'john_doe'."
  }
  ```

- **Error Response for Bad Request** (HTTP Status 400):
  json
  Copy code
  ```
  {
   "error": "Invalid input",
   "message": "Username cannot be empty and must conform to naming conventions."
  }
  ```

- **Explain how you would handle errors (e.g., user not found) and validate input.**

  Input Validation:

  Ensure the username is provided and is not empty. It should conform to specific naming conventions (e.g., alphanumeric characters, underscores, and a maximum length).

Validate the fields parameter to ensure it only contains valid field names (e.g., firstName, lastName, etc.).

Validate the includePosts parameter to ensure it is a boolean.

Error Handling:

If the username does not exist in the database, return a 404 Not Found status with an appropriate error message.

For any invalid inputs (like empty usernames or incorrect field names), return a 400 Bad Request status with details about the validation errors.

3. TypeScript Type Inference

Scenario: Consider the following TypeScript function:

```typescript
function getUserInfo(userId: number): Promise<{ name: string; age: number }> {
    // Simulate an async API call
    return fetch(`/api/users/${userId}`)
        .then(response => response.json());
}
```

What TypeScript feature is being used to specify the return type of the getUserInfo function?

The feature being used to specify the return type of the getUserInfo function is **explicit return type annotation**. In TypeScript, you can explicitly define the return type of a function using a colon followed by the type after the function parameters. This helps with type safety and clarity, making it clear what type the function is expected to return.

Example:

typescript
Copy code
function getUserInfo(username: string): UserInfo {
  // function implementation
}

In this case, UserInfo is the specified return type, which should be defined elsewhere in the code, typically as an interface or type.

How would you modify this function to handle cases where the API response might be missing the age property.

To modify the function to handle cases where the API response might be missing the age property, you can use optional properties in your TypeScript interface definition for the return type. Here's an example of how you could define the UserInfo type and modify the function:

typescript

Copy code

interface UserInfo {

  username: string;

  firstName: string;

```
    lastName: string;
    email: string;
    age?: number; // Make age optional
}
async function getUserInfo(username: string): Promise<UserInfo> {
    const response = await fetch(`https://api.example.com/users/${username}`);
    const data: UserInfo = await response.json();
    if (data.age === undefined) {
        data.age = null; // or any default value you want to assign
    }
    return data;
}
```

Key Changes:

Optional Property: The age property is defined as optional using the ? syntax, which indicates that it may or may not be present in the API response.

Handling Missing Property: Inside the function, we check if data.age is undefined. If it is, we can set it to null or any default value as needed.

4. Concurrency Handling

Scenario: You are developing a Node.js application where users can book events. The event booking system should prevent double bookings for the same event.

Requirements:

- **Describe how you would handle concurrent booking requests to ensure that an event cannot be booked more than once at the same time.**

  1. Handling Concurrent Booking Requests

  Application Code:

- Mutex Locking: Use a locking mechanism to ensure that only one request can process the booking for a specific event at a time. You can use libraries like async-lock or a similar solution to create a lock around the booking logic.
- Atomic Transactions: Leverage atomic transactions to perform the check and insert operation in a single step. This ensures that if two requests check availability at the same time, only one can succeed.

  Example Using Mutex:

  javascript

  Copy code

```
const AsyncLock = require('async-lock');

const lock = new AsyncLock();

async function bookEvent(eventId, userId) {
    return lock.acquire(eventId, async () => {
        const event = await getEventById(eventId);
        if (event.isBooked) {
            throw new Error('Event is already booked.');
        }
        await createBooking(eventId, userId);
    });
}
```

- **Suggest a strategy for implementing this in both the application code and the database schema.**

  Schema Design:

- Bookings Table: Create a Bookings table that includes fields for eventId, userId, and a timestamp for the booking. You might also include a unique constraint on the combination of eventId and timestamp to ensure that no two bookings can occur for the same event at the same time.

  sql

  Copy code

  ```
  CREATE TABLE Bookings (
    id SERIAL PRIMARY KEY,
    eventId INT NOT NULL,
    userId INT NOT NULL,
    bookingTime TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE (eventId, bookingTime)
  );
  ```

  Implementation Strategy:

- Use of Unique Constraints: The unique constraint on eventId and bookingTime prevents double bookings at the same timestamp. Adjust the granularity of bookingTime as necessary to your application's requirements.
- Optimistic Locking: Alternatively, you could implement optimistic locking where you check if the event is still available before finalizing the booking. If another booking occurs after the check, you can reject the booking attempt.

  Example with Optimistic Locking:

1. Check if the event is booked:

   sql

   Copy code

   ```
   SELECT * FROM Events WHERE id = $1 AND isBooked = false;
   ```

2. If available, book the event:

   sql

   Copy code

   ```
   INSERT INTO Bookings (eventId, userId) VALUES ($1, $2);
   UPDATE Events SET isBooked = true WHERE id = $1;
   ```

3. Handle potential conflicts in your application code by catching errors when attempting to insert a duplicate booking.

5. Error Handling in Node.js

Scenario: You have the following Node.js function that processes a user login request:

```javascript
async function loginUser(email, password) {
    try {
        const user = await findUserByEmail(email);
        if (!user) {
            throw new Error('User not found');
        }
        const isPasswordValid = await verifyPassword(user, password);
        if (!isPasswordValid) {
            throw new Error('Invalid password');
        }
        return { success: true, user };
    } catch (error) {
        // Handle error
    }
}
```

1. **How would you improve the error handling in this function to provide more specific error responses to the client?**

   1. Improving Error Handling

   Here's how you might structure your login function to enhance error handling:

   javascript

   Copy code

   async function login(username, password) {

   try {

   const user = await findUserByUsername(username);

   if (!user) {

   throw new Error('USER_NOT_FOUND');

   }

   const isValidPassword = await verifyPassword(user.password, password);

   if (!isValidPassword) {

```
            throw new Error('INVALID_PASSWORD');
        }
        const token = generateToken(user);
        return { success: true, token };

    } catch (error) {
        handleLoginError(error);
    }
}
function handleLoginError(error) {
    switch (error.message) {
        case 'USER_NOT_FOUND':
            return { status: 404, message: 'User not found.' };
        case 'INVALID_PASSWORD':
            return { status: 401, message: 'Invalid password.' };
        default:
            return { status: 500, message: 'An unexpected error occurred.' };
    }
}
```

Key Improvements:

- **Specific Error Messages**: Instead of a generic error message, specific error messages are thrown for different failure points (USER_NOT_FOUND and INVALID_PASSWORD).
- **Centralized Error Handling**: The handleLoginError function centralizes error handling, making it easier to manage different types of errors and their responses.

2. Explain the importance of differentiating between different types of errors (e.g., user not found vs. invalid password).

Differentiating between various types of errors is crucial for several reasons:

1. **User Experience**: Providing specific error messages helps users understand what went wrong. For instance, if a user is informed that their password is incorrect, they

can take appropriate action (like trying again). In contrast, a vague error message may lead to frustration.

2. **Security**: Specific error messages help prevent information leakage. For example, distinguishing between a user not found and an invalid password can reduce the risk of username enumeration attacks, where an attacker tries to guess valid usernames based on the error responses.
3. **Debugging and Monitoring**: Differentiating error types aids developers in identifying issues quickly. For example, if you notice a high number of USER_NOT_FOUND errors in your logs, it may indicate a problem with user registration or data integrity.
4. **API Design**: For APIs, returning specific error codes and messages allows clients to handle errors appropriately in their applications. This can facilitate better error handling strategies on the client side.

6. REST API Rate Limiting

Scenario: You are tasked with implementing rate limiting for a REST API endpoint to prevent abuse. The rate limit should be applied per IP address.

Requirements:

- **Describe a basic approach to implement rate limiting in a Node.js application.**

Basic Approach to Implement Rate Limiting

1. **Define Rate Limits**: Determine the allowed number of requests per time period (e.g., 100 requests per hour per IP address). This will help you set the parameters for your rate limiting.
2. **Track Requests**: Use a data structure to track the number of requests made by each IP address within the defined time frame. This can be done in memory, in a database, or using a caching mechanism.
3. **Check Requests**: For each incoming request, check the tracked requests for the user's IP address:
    - If the request count is within the limit, process the request.
    - If the limit is exceeded, return a response indicating that the rate limit has been exceeded.
4. **Reset Count**: Implement a mechanism to reset the request count after the time period has elapsed.

Example Implementation

Here’s a simple example using a middleware approach:

javascript

Copy code

```
const rateLimit = require('express-rate-limit');

const express = require('express');

const app = express();

const limiter = rateLimit({

  windowMs: 60 * 60 * 1000, // 1 hour

  max: 100, // limit each IP to 100 requests per windowMs

  message: 'Too many requests from this IP, please try again later.',

});
```

```
app.use('/api/', limiter);

app.get('/api/resource', (req, res) => {
  res.send('Resource data');
});


app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Libraries and Strategies for Rate Limiting

1. **Express Rate Limit**:
    - **Library**: express-rate-limit
    - **Description**: This is a simple and widely used middleware for Express applications. It allows you to easily implement rate limiting with options for customizing the window size,

- **What libraries or strategies would you use to handle rate limiting?**

Libraries and Strategies for Rate Limiting

1. **Express Rate Limit**:
    - **Library**: express-rate-limit
    - **Description**: This is a simple and widely used middleware for Express applications. It allows you to easily implement rate limiting with options for customizing the window size, maximum requests, and response messages.
2. **Redis for Distributed Rate Limiting**:
    - **Library**: rate-limiter-flexible or redis
    - **Description**: If your application is distributed across multiple instances, consider using Redis to store request counts. rate-limiter-flexible is a versatile library that can be configured to work with Redis, providing a centralized way to manage rate limits across instances.
3. **Custom In-Memory Solutions**:
    - If your application is simple and doesn't require distributed rate limiting, you could implement a custom in-memory solution using a JavaScript object or a Map to track request counts and timestamps.

7. API Pagination

Scenario: You need to implement pagination for an endpoint that returns a list of products. The endpoint should support pagination using limit and offset query parameters.

Requirements:

- **Write a brief description of how you would implement pagination in a Node.js REST API.**

To implement pagination for an endpoint that returns a list of products, I would follow these steps:

1. **Define Query Parameters**: Use limit and offset as query parameters to control the number of products returned and the starting point of the results.
    - limit: Number of products to return per request.
    - offset: The number of products to skip before starting to collect the result set.
2. **Modify the Database Query**: Update the database query to utilize the limit and offset parameters when fetching products. This can be done using SQL with LIMIT and OFFSET clauses, or equivalent methods in NoSQL databases.
3. **Response Structure**: Include metadata in the response to provide information about the total number of products, the current page, and the total pages. This helps clients understand the context of the paginated data.

Example Implementation

Here's a simple example using Express and a SQL database:

javascript

Copy code

```
app.get('/api/products', async (req, res) => {
  const limit = parseInt(req.query.limit) || 10;
  const offset = parseInt(req.query.offset) || 0;
  try {
    const [products, total] = await Promise.all([
      db.query('SELECT * FROM products LIMIT ? OFFSET ?', [limit, offset]),
      db.query('SELECT COUNT(*) as count FROM products'),
    ]);
    const totalCount = total[0].count;
```

```
        const totalPages = Math.ceil(totalCount / limit);
        res.json({
          products,
          totalCount,
          totalPages,
          currentPage: Math.floor(offset / limit) + 1,
        });
      } catch (error) {
        res.status(500).json({ message: 'Internal Server Error' });
      }
    });
```

Explain how you would handle large data sets and ensure that pagination is efficient.

Handling Large Data Sets Efficiently

1. **Use Indexing**: Ensure that the relevant columns (e.g., product IDs) are indexed in the database. This speeds up the retrieval of products and improves query performance.
2. **Limit the Returned Data**: Only select the necessary fields instead of retrieving all columns. This reduces the amount of data processed and sent over the network.
3. **Cache Results**: For frequently accessed data, consider implementing caching mechanisms (like Redis or in-memory caching) to reduce the load on the database and speed up response times.
4. **Avoid Large Offsets**: Large offsets can lead to performance issues as the database needs to skip many rows. Consider using alternative pagination strategies like keyset pagination (also known as cursor-based pagination) that rely on the last fetched item instead of an offset.

   Example for cursor-based pagination:

   - Instead of offset, use a lastProductId query parameter to fetch products greater than this ID.
5. **Monitor Performance**: Regularly analyze the performance of the pagination queries and adjust indexes, query structure, or caching strategies as necessary.

8. Data Validation in TypeScript

Scenario: You have the following TypeScript function that processes user input:

```typescript
interface UserInput {
    name: string;
    email: string;
}

function processUserInput(input: UserInput) {
    // Process the input
}
```

1. **How would you enhance the UserInput interface to include validation rules (e.g., email format)?**

   To enhance the UserInput interface and implement effective validation in a TypeScript application, you can follow these steps:

   1. Enhancing the UserInput Interface

   You can enhance the UserInput interface by adding validation rules as properties. For example, you might define a type for the validation rules and use it in your interface. Here's how you can do it:

   typescript

   Copy code

   interface UserInput {

   username: string;

   email: string;

   password: string;

   }

   interface ValidationRules {

   username: (value: string) => boolean;

   email: (value: string) => boolean;

   password: (value: string) => boolean;

```
}

const validationRules: ValidationRules = {

  username: (value) => /^[a-zA-Z0-9_]{3,20}$/.test(value), // alphanumeric and underscores, 3-20 chars

  email: (value) => /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(value), // basic email format

  password: (value) => value.length >= 8 // at least 8 characters

};
```

2. Describe a strategy for validating user input in a TypeScript application.

To validate user input effectively in a TypeScript application, you can follow this strategy:

Step-by-Step Validation Process

1. **Define the Input Schema**: Use interfaces to define the structure of user input, as shown above.
2. **Create Validation Functions**: Implement validation functions that check whether each input field meets the specified criteria. You can use regular expressions for format validation (e.g., for emails).
3. **Centralized Validation Logic**: Create a centralized function that accepts the user input and runs all validation checks. This function should return an object indicating which fields are valid and which are not.
4. **Error Handling**: For invalid inputs, return detailed error messages to inform users about what went wrong. This could be an object mapping field names to error messages.
5. **Integration with Application Logic**: Call the validation function before processing user input in your application (e.g., before storing data in a database or proceeding with further business logic).

Example Implementation

Here's a simplified example:

typescript

Copy code

```
interface ValidationResult {

  valid: boolean;
```

```
    errors: { [key: string]: string };
}

function validateUserInput(input: UserInput): ValidationResult {
    const errors: { [key: string]: string } = {};
    let valid = true;

    if (!validationRules.username(input.username)) {
        valid = false;
        errors.username = 'Username must be 3-20 characters long and can only contain letters, numbers, and underscores.';
    }
    if (!validationRules.email(input.email)) {
        valid = false;
        errors.email = 'Invalid email format.';
    }
    if (!validationRules.password(input.password)) {
        valid = false;
        errors.password = 'Password must be at least 8 characters long.';
    }
    return { valid, errors };
}
const userInput: UserInput = {
    username: 'user123',
    email: 'user@example.com',
    password: 'password123'
};
const validationResult = validateUserInput(userInput);
if (!validationResult.valid) {
    console.error('Validation failed:', validationResult.errors);
```

```
} else {
    console.log('User input is valid!');
}
```