Backend and Database Architecture for Dynamic Assessment

Use Cases:

- Convert a direct graph used for Dynamic Assessments like GMAT into a running database design and backend logic.
- Send batches of questions from BE to FE with max no of questions being N.
- Next batch depends on answers to questions from all previous questions from all previous batches.
- Single/Multiple Selects.
- Next Batch Traversal.
- Previous Batch Traversal.
- Assessment ends at the leaf node.
- Score being positive for correct answer and negative for an incorrect answer.

Explanation and Workflow:

1. Examination Initialization:

1) **User Registration/Authentication:**
    a) User registers/logs in.
    b) System checks user details from the users table and authenticates the user.
    c) After successful authentication, the system initializes current_batch_id to 1 for new users.
2) **Instructions Page:**
    a) Before starting the examination, display an instructions page with examination guidelines.
    b) Allow the user to begin the examination when they are ready.

2. Examination Progress:

3) **Fetching Questions:**
    a) Based on the current_batch_id from the users table, the system fetches the first batch of questions.
    b) Questions are displayed along with their respective options.
    c) User attempts the questions and selects answers.
4) **Answer Submission:**
    a) After answering each question from the current batch, the user submits their responses.
    b) Answers are saved in the user_answers table.
    c) Using the decision graph in Neo4j, the system determines if the user will proceed to another batch or end the examination based on the criteria provided.
    d) Update the current_batch_id in the users_assessment_session table based on Neo4j's decision graph.

5) **Batch Navigation:**
    a) Users progress from one batch to the next based on their answers and the decision graph.
    b) If necessary, users can navigate to a previous batch to review or change their answers.
    c) The system ensures the user follows the rules defined in the decision graph. For example, if the last question of batch 1 can lead to batch 2 or batch 3, the system will decide which batch to display next based on the user's all previous responses.
6) **End of Examination:**
    a) Once the user completes all the required batches or reaches an end node in the decision graph, the examination ends.
    b) The system checks Neo4j to confirm if there are no more PROCEED_TO relationships.

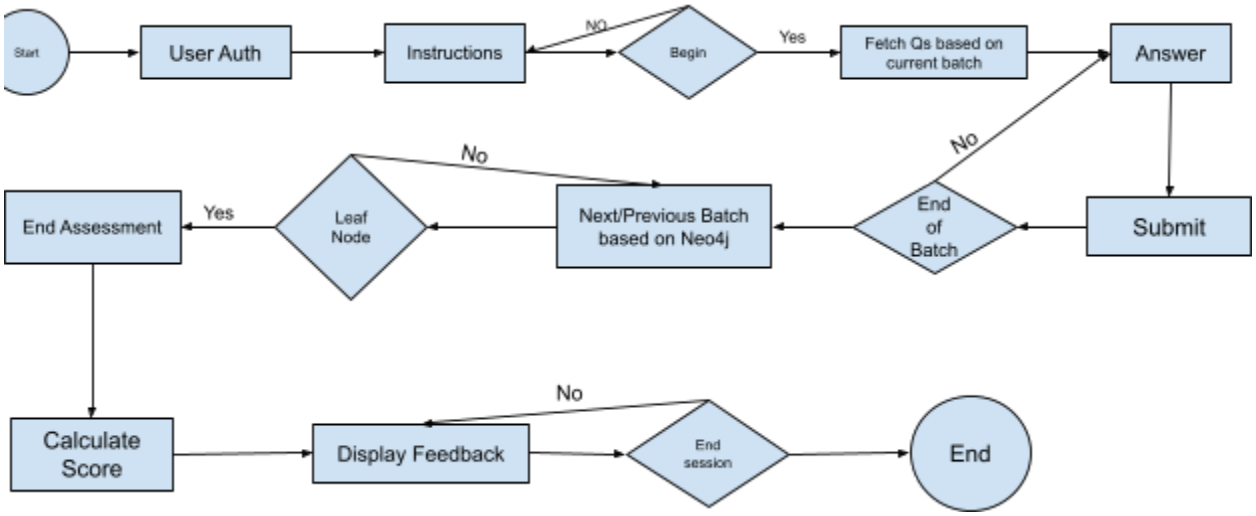
3. Post-Examination:

7) **Score Calculation:**
    a) After ending the examination, the system calculates the user's score.
    b) Retrieve all the user's answers from the user_answers table.
    c) Cross-check each answer with the is_correct field in the options table.
    d) Sum up the scores from the correct answers.
8) **Feedback Display (Additional):**
    a) Display the total score to the user.
    b) Optionally, provide feedback on each question (correct answers, explanations, etc.) based on examination settings.
9) **History and Analytics (Additional):**
    a) Users can view their past examination attempts, scores, and feedback.
    b) Admin or instructors can view user analytics, batch performance, common mistakes, etc., to improve future batches or training materials.

4. Log Out / End Session:

10) **Completion:**
    a) After reviewing their scores and feedback, users can choose to log out or end their session.
    b) Optionally, provide users with certificates or badges based on their scores.

Flow Diagram:



Data Structure and Implementation of /next_batch API

a) **Request:**
   i) Header:
      (1) Authentication Token (JWT or similar to identify the user).
   ii) Body:
      (1) user_id: Unique user identifier (if not decoded from the token).
      (2) session_id: Unique user session identifier

b) **Response:**
   i) Body:
      (1) batch_id: The ID of the next batch.
      (2) question: question object.

2. Logic for Determining the Next Question:

a. Fetch Current User Data:
- Retrieve the user's current_batch_id and all the answers of previously answered questions, based on the user_id or the decoded user ID from the token.

b. Determine Neo4j Node:
- Using the last_question_id, identify the corresponding node in the Neo4j database.

c. Traverse the PROCEED_TO Relationship:
- Traverse the PROCEED_TO relationship from the identified node.

- If there's a condition attached to this relationship (based on the last_answer_option), follow the path that matches the user's answer.
- If there's no condition, or if the condition doesn't match the user's answer, follow the default progression.

d. Fetch Next Batch ID:

- The end node of the traversed relationship will have a batch_id associated with it. This is the next batch the user should progress to.

e. Update User's Current Batch in PostgreSQL:

- Update the current_batch_id for the user in the users table in PostgreSQL to reflect the new batch they are on.

f. Fetch Next Batch Questions from PostgreSQL:

- Retrieve all questions corresponding to the new batch_id from the questions table in PostgreSQL.

g. Send Response:

- Return the next batch_id and its corresponding questions in the response body.

****This approach makes the progression logic highly dynamic. Any changes to the progression rules or conditions would be done directly in the Neo4j database, and the application would adapt in real-time without any code changes. This ensures flexibility and scalability.****

2) **APIs that the FE will be interacting with:**

a) **POST - /start_assessment**

Initializes a new assessment session for the user and returns the first batch of questions.

- Sessions starts → entry in user_assessment_session table
- Request Params: {
  user_id: value
  }

- Response: {
  batch:{
  number: 1,
  questions: [
  {
  question_id: “”,
  text: “”,
  marks: “”,
  answer: {
  options: [{text:””, is_correct: false/true}],

```
                                        }
                                    }]
                            }
                    }
```

b) POST - /answer_question

Receives the user's answer to a question, saves it in the user_anwers table.

- Gets stored into user_answers table

```
Request Params: {
        question_id: “”,
        answer_id: “”,
        batch: 1,
}
Response: {
        status: success
}
```

c) GET - /:session_id/next_batch

Retrieves the next batch of questions based on the current session's progress.

- Based on the answers of questions of the previous batch, which we fetch from the user_answers table and travers using question_traversal table.
- BE will create a list based on traversed questions, and send the list to FE.

```
Request Params: {
        session_id: “”
}

Response: {
                batch:{
                        number: 2,
                        questions: [
                                {
                                        question_id: “”,
                                        text: “”,
                                        marks: “”,
                                        answer: {
                                                options: [{text:””, is_correct:
                                false/true}],

                                        }
                                }]
                }
```

d) GET - /:session_id/previous_batch

Retrieves the previous batch of questions based on the current session's progress.

- Reverse traversing, using user_answers table, and using user_assessment_session table, we can query on the user_anwers table to retrieve the list of Qs from the previous bath. From the options table, we can always show options associated with the Qs.

Request Params: {
session_id: “”
}

Response: {
batch:{
number: 1,
questions: [
{
question_id: “”,
text: “”,
marks: “”,
answer: {
options: [{text:””, is_correct:
false/true}],

}
}]
}

e) GET - /:session_id/get_score

Calculates and returns the user's score for the completed assessment session.

- User answers → correct → score = score +1;
- User answers → incorrect → score = score - 1;
- After each answer submission, score gets updated in the user_assessment_session table.
- At the end, we can just query on the score column from this table, and show the score.

Request Params: {
session_id: “”
}

Response: {
score: “”
}

f) POST - /end_assessment

Ends the assessment following the completion of sending all the batches to FE and getting termination confirmation from the FE.

- For that particular user, assessment has ended, and the flag in user_assessment_session table can be turned to ‘completed’.

Request Params: {
session_id: “”,
user_id: “”
}

Response: {
status: “success”
}

Database Designs:

- These will be minimum table requirements for providing a scalable database design which will incorporate current scenarios, as well probable future scenarios as well.
- Tables Required with their relationships:

1) users:

- Store user basic/advanced user information.
- All actions will be performed by the user or with respect to a user.
- **Columns (datatype):**
  - id (bigint) (primary key)
  - name (character varying)
  - email (character varying)
  - other details
  - created_at (timestamp with timezone)
  - updated_at (timestamp with timezone)
  - created_by (bigint)
  - updated_by (bigint)

2) questions:

- Store the question with its text, weightage of marks it has, degree of the question and other metadata.
- **Columns (datatype):**
    - id (bigint) (primary key)
    - text (character varying)
    - marks (integer)
    - degree (integer)
    - created_at (timestamp with timezone)
    - updated_at (timestamp with timezone)
    - created_by (bigint)
    - updated_by (bigint)

3) options:

- Stores each option as an individual record for a particular question, with its text and correctness.
- **Columns (datatype):**
    - id (bigint) (primary key)
    - question_id (bigint) (foreign key to the questions table)
    - text (character varying)
    - is_correct (boolean)
    - created_at (timestamp with timezone)
    - updated_at (timestamp with timezone)
    - created_by (bigint)
    - updated_by (bigint)

4) user_answers:

- Stores which option a particular user has selected for which particular question from which particular batch.
- **Columns (datatype):**
    - id (bigint) (primary key)
    - user_id (bigint) (foreing key to the users table)
    - question (bigint) (foreing key to the questions table)
    - option (bigint) (foreign key to the options table)
    - batch (integer)
    - created_at (timestamp with timezone)
    - updated_at (timestamp with timezone)
    - created_by (bigint)
    - updated_by (bigint)

5) exam_metadata:

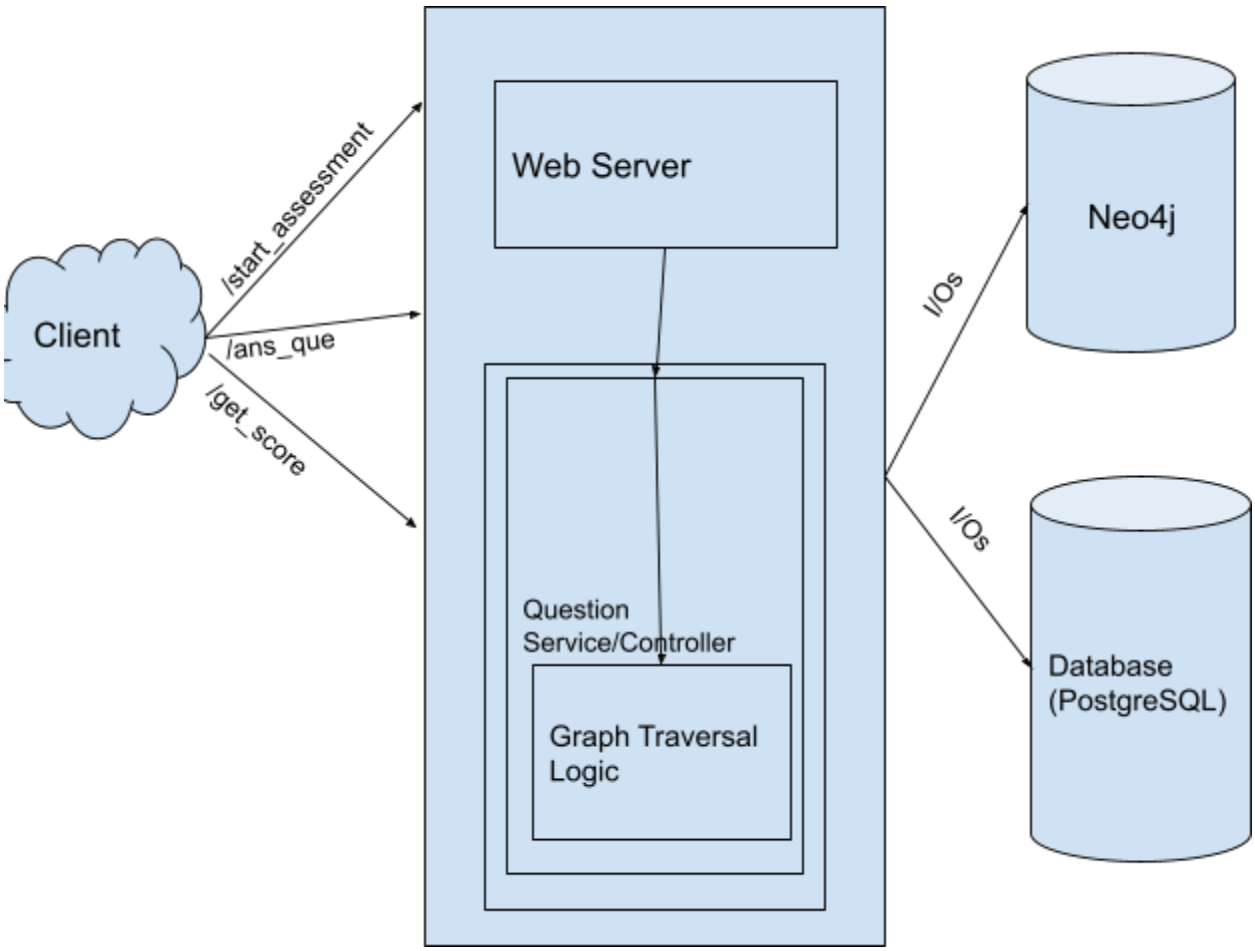
- Stores exam related details such as, version, total number of questions, maximum question per batch, name etc..

- **Columns (datatype):**
    - id (bigint) (primary key)
    - version (integer)
    - total_questions (integer)
    - questions_per_batch (integer)
    - exam_name (character varying)
    - created_at (timestamp with timezone)
    - updated_at (timestamp with timezone)
    - created_by (bigint)
    - updated_by (bigint)

6) **user_assessment_session:**
    - Stores assessment session details per user, such as, current question user is on, total questions the user has attempted, current score, time left, current batch etc..
    - **Columns (datatype):**
        - id (bigint)
        - user_id (bigint) (foreing key to the users table)
        - current_question (bigint) (foreing key to the questions table)
        - questions_attempted (integer)
        - score (integer)
        - time_left (integer)
        - current_batch (integer)
        - created_at (timestamp with timezone)
        - updated_at (timestamp with timezone)
        - created_by (bigint)
        - updated_by (bigint)

Backend Architecture and API Logic:

Architecture Diagram:



Broad Approach for the End to End Solution:

if the application grows too big with the development team as well.

Microservices Architecture:

- Divide the system into microservices to ensure modularity, scalability, and easier maintenance. Each microservice can handle specific functionalities, such as user management, question management, assessment logic, scoring, and analytics.

Components:

User Service:
Handles user authentication, registration, and profile management.

Question Service:
Manages the question bank, including question creation, editing, and categorization. It stores the directed graph structure of the assessment.

Assessment Service:
Manages the assessment logic. It navigates through the directed graph based on user responses and keeps track of the user's progress.

Scoring Service:
Calculates the score based on user responses. Handles positive and negative marking.

Analytics Service:
It Gathers data for analysis and reporting, providing insights into user performance, question difficulty, assessment completion rates, etc.

Notification Service:
Sends notifications to users, such as reminders to complete the assessment or score reports.

API Gateway:
Acts as a single entry point for clients, routing requests to appropriate microservices and handling authentication/authorization.

Database:
Choose appropriate databases for each microservice. For example, a graph database (e.g., Neo4j) for the directed graph structure, a relational database (e.g., PostgreSQL) for user data, and a NoSQL database (e.g., MongoDB) for analytics data.

Directed Graph Representation:
Store the directed graph structure of the question paper in a graph database. Each node represents a question, and edges represent possible transitions based on user responses. Include attributes for question text, options, correct answers, marks, and penalties.

Assessment Logic:
The Assessment Service is responsible for guiding users through the assessment. It retrieves the current question based on the user's progress and responses. After each question, it calculates the next question based on the directed graph and user's answers.

User Progress and Responses:
Store user progress and responses in the database. This allows users to resume assessments from where they left off and ensures accurate scoring.

Scoring and Negative Marking:
The Scoring Service calculates the total score based on correct and incorrect answers. It applies positive and negative marking as defined by the assessment rules.

Authentication and Authorization:
Use OAuth2 or JWT for authentication and authorization. Ensure that users can access only the assessments they are eligible for.

Caching and Performance:
Implement caching mechanisms to reduce database load for frequently accessed data, such as question metadata. Consider using caching tools like Redis.

Load Balancing and Auto-scaling:
Deploy microservices on a cloud platform (e.g., AWS, Azure) and use load balancers to distribute traffic. Set up auto-scaling to handle varying loads efficiently.

Monitoring and Logging:
Implement logging and monitoring to track system performance, identify bottlenecks, and detect errors. Tools like Prometheus and Grafana can be helpful.

Security Measures:
Implement security best practices to protect user data and the system. Regularly update dependencies and conduct security audits.

Backup and Disaster Recovery:
Regularly backup the databases and have a disaster recovery plan in place to ensure data integrity and availability.

By prioritizing scalability, security, and user experience when designing the backend architecture, actual implementation details would depend on your technology stack, programming languages, frameworks, and other specific requirements and can differ from what is mentioned above.