Assigned: 04 March

Homework #7 – Protein Motif Search API

EE 547: Spring 2025

Due: Monday, 24 March at 23:59. Submission instructions will follow separately on Brightspace.

1. Extend your Protein Management API by integrating PostgreSQL for metadata storage, implementing advanced search functionalities, and adding a motif search feature through sequence fragmentation.

   Introduction

   Proteins are essential biomolecules that perform a vast array of functions in living organisms, including catalyzing metabolic reactions, providing structural support, and regulating cellular processes. A fundamental aspect of understanding protein function lies in identifying specific sequences known as *motifs*—short, conserved patterns associated with particular biological activities or structural features. Detecting these motifs within protein sequences is crucial for understanding protein roles, interactions, and mechanisms.

   Advancements in computational biology streamline the analysis of large protein datasets and enable efficient motif discovery and sequence analysis. Fragmenting protein sequences into smaller segments enhances the detection of motifs that may be obscured in full-length sequences due to structural complexity or variability. Efficient storage, retrieval, and querying of protein data are vital for bioinformatics applications, aiding in complex analyses and contributing to a deeper understanding of protein biology.

   API Specification

   Extend your existing API by adding new endpoints and modifying existing ones to incorporate database functionalities and search capabilities. Your completed implementation should no longer use a JSON file for storing data.

   - `GET /api/proteins/search`

     Description: Search for proteins based on various criteria.

     Query Parameters:

     - `name`: String. Search proteins by name using partial matches.
     - `molecularWeight[operator]`: Number. Filter proteins by molecular weight.
       * Operators: `gt`, `gte`, `lt`, `lte`, `eq`
       * Example: `molecularWeight[gt]=50000`
     - `sequenceLength[operator]`: Integer. Filter proteins by sequence length.
       * Example: `sequenceLength[lte]=1000`

  - `motif`: String. Amino acid motif to search within protein sequences.
  - `sort`: String. Field and direction to sort by, in the format `field:direction`.
    * Allowed fields: `name`, `createdAt`, `molecularWeight`, `sequenceLength`
    * Direction: `asc` or `desc`. Default is `asc`.
    * Example: `sort=name:desc`

  Response:

  - 200 OK: Successfully retrieved the list. Body: Array of Protein objects matching the search criteria.
  - 400 Bad Request: Invalid query parameters.

- `GET /api/fragments/:fragmentId`

  Description: Retrieve a specific protein fragment by its ID.

  Parameters:

  - `fragmentId`: String (UUID)

  Response:

  - 200 OK: Successfully retrieved the fragment. Body: Fragment object.
  - 404 Not Found: Fragment with the given ID does not exist.

- `GET /api/proteins/:proteinId/fragments`

  Description: Retrieve all fragments of a specific protein sorted by start position (ascending).

  Parameters:

  - `proteinId`: String (UUID)

  Response:

  - 200 OK: Successfully retrieved the list of fragments. Body: Array of Fragment objects.
  - 404 Not Found: Protein with the given ID does not exist.

- `POST /api/proteins`

  Modification: When a new protein is created, its metadata (excluding the sequence) should be stored in PostgreSQL. Fragment the sequence and store fragments in PostgreSQL.

- `GET /api/proteins`

Modification: Retrieve the list of proteins by fetching metadata from PostgreSQL instead of the previous method.

Data Types and Formats

The API uses the following data structures:

- `Protein` **Object** (API Response):

```
{
  "proteinId": String (UUID),
  "name": String (1-100 characters, required),
  "description": String (0-1000 characters, optional),
  "molecularWeight": Number (positive float),
  "sequenceLength": Integer (number of amino acids),
  "createdAt": String (ISO 8601 date-time),
  "updatedAt": String (ISO 8601 date-time),
  "sequenceUrl": String (URL to download the sequence file)
}
```

- `Fragment` **Object** (API Response):

```
{
  "fragmentId": String (UUID),
  "proteinId": String (UUID),
  "sequence": String (2-50 characters, uppercase A-Z),
  "startPosition": Integer,
  "endPosition": Integer,
  "motifs": [String],  // List of motifs found in the fragment
  "secondaryStructure": String (H, E, or C, same length as sequence),
  "confidenceScores": [Number],  // Array of confidence scores (0-1)
  "createdAt": String (ISO 8601 date-time),
  "url": String (URL to access this fragment)
}
```

Note on API Response Transformation

The `Fragment` object in the API response includes arrays for `motifs` and `confidenceScores`, while in the database these are stored in a separate `motifs` table. When implementing the API endpoints, you will need to transform the normalized relational data into the denormalized response structure expected by clients.

This typically involves a simple JOIN operation to retrieve both fragment data and related motifs, followed by grouping and formatting this data to match the expected API response structure. For example, when retrieving a specific fragment, you would JOIN the `fragments` and `motifs` tables, then aggregate the motif data into arrays within your application code.

Implementation Requirements

(a) **Authentication and Authorization**

All API endpoints require authentication using the X-User-ID header. If the X-User-ID header is missing or invalid, the server returns a 401 Unauthorized error.

Here is an example of how to include the header in an API request using cURL:

```
curl -X GET https://api.example.com/api/proteins \
     -H "X-User-ID: your-user-id" \
     -H "Content-Type: application/json" \
     -H "Accept: application/json"
```

And here is an example simple authentication layer using Express middleware:

```
async function authenticateUser(req, res, next) {
  // Extract the user ID from request headers
  const userId = req.header('X-User-ID');

  // Handle missing authentication
  if (!userId) {
    // Return appropriate HTTP status code for unauthenticated requests
    // ...
  }

  try {
    // Query the database to verify if the user exists
    // ...

    // Check query results and handle unauthorized access
    // ...

    // If authorized, attach user information to the request object
    // for use in downstream middleware and route handlers
    req.user = {
      // Add relevant user data here
    };

    // Continue to the next middleware or route handler
    next();
  } catch (error) {
    // Log the error for debugging
    console.error('Authentication error:', error);

    // Send appropriate error response
    // ...
  }
}

// Apply the middleware to protect API routes
// Consider which routes need protection
app.use('/api', authenticateUser);
```

(b) **Environment Configuration**

For this assignment, you will use environment variables to manage our application configuration. This approach keeps sensitive information like database credentials out of the source code and allows for different configurations across environments without code changes.

Using dotenv for Configuration Management

The dotenv package is a simple module that loads environment variables from a .env file into `process.env`, making them accessible throughout your application.

First, install the dotenv package:

```
npm install dotenv
```

Create a `.env` file in the root directory of your project with the following variables (you may need to adjust values based on your local settings):

```
PORT=3000
MAX_PROTEIN_LENGTH=2000
PG_HOST=localhost
PG_PORT=5432
PG_DATABASE=protein_db
PG_USER=postgres
PG_PASSWORD=password
```

To use `dotenv` in your application:

```
// Load dotenv at the top of your main file
const dotenv = require('dotenv');
dotenv.config();

// Now you can access environment variables
const port = process.env.PORT || 3000;
console.log(`Server will run on port ${port}`);
```

Benefits of using dotenv:

- Keeps sensitive data like credentials separate from your code.
- Makes it easy to change configuration between environments.
- Prevents accidental commits of sensitive information to version control.

(c) **PostgreSQL Configuration**

i. **Running PostgreSQL in a Docker Container**

Use a PostgreSQL instance running in a Docker container for local development. To start a PostgreSQL container with data persistence and exposed port:

```
docker run -d \
  -p 5432:5432 \
  -e POSTGRES_PASSWORD=password \
  -e POSTGRES_USER=postgres \
  -e POSTGRES_DB=protein_db \
  -v /path/to/datadir:/var/lib/postgresql/data \
  --name postgres \
  postgres:latest
```

Replace `/path/to/datadir` with the path to a directory on your host machine where you want PostgreSQL data to be stored. You may also need to consult the PostgreSQL Docker Hub page for the latest image tag and changes to the supported environment variables.

Ensure that the database is created and that you create the tables (*i.e.,* schema) before proceeding.

Connecting to PostgreSQL

Use the pg package to interact with PostgreSQL from your application. Initialize the database connection before starting the server using a connection pool:

```
// Load environment variables
const dotenv = require('dotenv');
dotenv.config();

const express = require('express');
const { Pool } = require('pg');
const app = express();

// Set up configuration variables
const PORT = process.env.PORT || 3000;
const MAX_PROTEIN_LENGTH = process.env.MAX_PROTEIN_LENGTH || 2000;

// Create connection pool
const pool = new Pool({
  host: process.env.PG_HOST,
  port: process.env.PG_PORT,
  database: process.env.PG_DATABASE,
  user: process.env.PG_USER,
  password: process.env.PG_PASSWORD
});

// Test the database connection
pool.query('SELECT NOW()', (err, res) => {
  if (err) {
```

```
    console.error('Error connecting to PostgreSQL:', err);
    process.exit(1);
  } else {
    console.log('Connected to PostgreSQL');

    // Start the server after the database connection is established
    app.listen(PORT, () => {
      console.log(`Server running on port ${PORT}`);
    });
  }
});

// Example route using the connection pool and pagination
app.get('/api/proteins', async (req, res) => {
  try {
    // Add pagination similar to Homework #3
    const limit = parseInt(req.query.limit) || 10;
    const offset = parseInt(req.query.offset) || 0;

    const result = await pool.query(
      'SELECT * FROM proteins ORDER BY created_at DESC LIMIT $1 OFFSET $2',
      [limit, offset]
    );
    res.json(result.rows);
  } catch (error) {
    console.error('Database error:', error);
    res.status(500).json({ error: 'Internal Server Error' });
  }
});
```

ii. **Database Schema and Setup**

Before running the application, set up the required database schema. You should use the following SQL statements to create the necessary tables:

```
-- Enable UUID support
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

-- Create proteins table
CREATE TABLE proteins (
  protein_id UUID DEFAULT uuid_generate_v4() PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  description VARCHAR(1000),
  molecular_weight FLOAT CHECK (molecular_weight > 0),
  sequence_length INTEGER,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  sequence_url VARCHAR(255)
);

-- Create fragments table
```

```
CREATE TABLE fragments (
  fragment_id UUID DEFAULT uuid_generate_v4() PRIMARY KEY,
  protein_id UUID REFERENCES proteins(protein_id) ON DELETE CASCADE,
  sequence VARCHAR(50) CHECK (sequence ~ '^[A-Z]{2,50}$'),
  start_position INTEGER,
  end_position INTEGER,
  secondary_structure VARCHAR(50) CHECK (secondary_structure ~ '^[HEC]+$'),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  url VARCHAR(255)
);

-- Create motifs table
CREATE TABLE motifs (
  motif_id UUID DEFAULT uuid_generate_v4() PRIMARY KEY,
  fragment_id UUID REFERENCES fragments(fragment_id) ON DELETE CASCADE,
  motif_pattern VARCHAR(50) NOT NULL,
  motif_type VARCHAR(50),
  start_position INTEGER,
  end_position INTEGER,
  confidence_score FLOAT CHECK (confidence_score >= 0 AND confidence_score <= 1)
    ,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Create users table
CREATE TABLE users (
  id VARCHAR(50) PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  role VARCHAR(20) DEFAULT 'basic' CHECK (role IN ('admin', 'basic'))
);

-- Insert default users
INSERT INTO users (id, name, role) VALUES
('admin-user-001', 'Admin User', 'admin'),
('user-001', 'Basic User', 'basic');

-- Create indexes for better query performance
CREATE INDEX idx_proteins_name ON proteins(name);
CREATE INDEX idx_fragments_protein_id ON fragments(protein_id);
CREATE INDEX idx_fragments_sequence ON fragments(sequence);
-- Create motif indexes
CREATE INDEX idx_motifs_fragment_id ON motifs(fragment_id);
CREATE INDEX idx_motifs_pattern ON motifs(motif_pattern);
```

Schema Constraints and Validation

The schema includes several CHECK constraints that serve useful validation purposes:

- In the `proteins` table, the `molecular_weight` CHECK ensures that only positive values are stored, preventing invalid scientific data.

- In the `fragments` table:
  - The `sequence` CHECK constraint (`^[A-Z]{2,50}$`) ensures that the fragment only contains valid uppercase amino acid letters (A-Z) and has a length between 2-50 characters.
  - The `secondary_structure` CHECK (`^[HEC]+$`) ensures it only contains valid structural codes (H for helix, E for sheet, C for coil).
- In the `motifs` table:
  - The `confidence_score` CHECK ensures scores stay within the valid probability range of 0 to 1.

These constraints provide data integrity at the database level rather than relying solely on application validation. PostgreSQL will reject operations that would violate these constraints so anticipate handling these errors. This approach follows the principle that data validation should occur as close as possible to the data storage layer.

iii. **Transactions in PostgreSQL**

Database transactions are essential for maintaining data integrity, especially when performing multiple related operations. A transaction is a single unit of work that either completes entirely or not at all, ensuring your database remains in a consistent state.

The key properties of transactions, often referred to as ACID properties, are:

- **Atomicity**: All operations in a transaction complete successfully, or none of them do.
- **Consistency**: The database moves from one valid state to another.
- **Isolation**: Transactions operate independently without interference.
- **Durability**: Once a transaction is committed, it remains so even in case of system failure.

Here's how to implement transactions in your PostgreSQL operations:

```
// Example: Creating a protein and its fragments in a transaction
async function createProteinWithFragments(proteinData, sequence) {
  try {
    // Begin transaction
    await pool.query('BEGIN');

    // Insert protein data
    const proteinResult = await pool.query(
      `INSERT INTO proteins(name, description, molecular_weight, sequence_length
    , sequence_url)
       VALUES($1, $2, $3, $4, $5) RETURNING protein_id`,
      [proteinData.name, proteinData.description, proteinData.molecularWeight,
       proteinData.sequenceLength, proteinData.sequenceUrl]
    );
```

```
    const proteinId = proteinResult.rows[0].protein_id;

    // Create and store fragments
    await fragmentAndStoreSequence(proteinId, sequence);

    // Commit transaction
    await pool.query('COMMIT');

    return proteinId;
  } catch (error) {
    // Rollback in case of any error
    await pool.query('ROLLBACK');
    console.error('Transaction failed:', error);
    throw error;
  }
}
```

Using transactions ensures that related operations (like creating a protein and its fragments) are completed together, preventing partial updates that could leave your database in an inconsistent state.

(d) **Protein Fragment Analysis**

When a new protein is added:

i. Break the sequence into overlapping fragments. Use a sliding window approach with a window size of 15 amino acids and a step size of 5 (i.e., overlap of 10 amino acids). For example, given the sequence `ABCDEFGHIJKLMNO`, the fragments would be:

- `ABCDEFGHIJKLMNO` (positions 1-15)
- `FGHIJKLMNOPQRST` (positions 6-20)
- Continue until the end of the sequence.

ii. Secondary Structure Prediction: Use the GOR method to predict the secondary structure for each fragment.

iii. Store each fragment in the fragments table in PostgreSQL, with motifs and confidence scores stored as JSONB data.

```
// Fragment a protein sequence and store fragments
async function fragmentAndStoreSequence(proteinId, sequence) {
  // Configuration for sliding window approach
  const windowSize = 15;
  const stepSize = 5;

  try {
```

```
      // 1. Iterate through sequence with sliding window
      for (let i = 0; i < sequence.length - windowSize + 1; i += stepSize) {
        // 2. Extract fragment and calculate positions
        // ...

        // 3. Analyze fragment characteristics
        const secondaryStructure = predictSecondaryStructure(/* ... */);

        // 4. Identify motifs in current fragment
        // ...

        // 5. Prepare data for database storage
        // Note: JSON data requires stringification

        // 6. Execute database insertion
        // Use parameterized query for security
        // ...
      }
    } catch (error) {
      // 7. Handle errors appropriately
      console.error('Fragmentation error:', error);
      throw error; // Allow transaction to handle rollback
    }
  }
```

(e) **Search Functionality**

For the `/api/proteins/search` endpoint, you should implement a search function that can filter proteins based on name, molecular weight, sequence length, and motif patterns. You'll need to construct SQL queries dynamically based on the provided search parameters.

The implementation should handle:

i. Name search with partial matching

ii. Numeric range filtering for molecular weight and sequence length

iii. Motif pattern matching within protein fragments

iv. Sorting results by specified fields

Relational Queries for Motif Search

When implementing the search functionality with the new relational schema, you will need to write queries that join multiple tables. For example, to search for proteins containing a specific motif pattern:

```
-- Find proteins containing a specific motif pattern
SELECT DISTINCT p.*
FROM proteins p
```

```
-- note: INNER is optional, default JOIN is INNER JOIN
INNER JOIN fragments f ON p.protein_id = f.protein_id
INNER JOIN motifs m ON f.fragment_id = m.fragment_id
WHERE m.motif_pattern SIMILAR TO 'a';
```

Note the use of `DISTINCT` to prevent duplicate proteins in the results, since a protein might contain multiple fragments matching the criteria.

The API should support the following predefined motifs for motif search:

i. **N-glycosylation site**: `N[^P][ST][^P]`

   An N followed by any amino acid except P, then either S or T, and again any amino acid except P.

ii. **Casein kinase II phosphorylation site**: `[ST].{2}[DE]`

   Either S or T, followed by any two amino acids, then either D or E.

iii. **Tyrosine kinase phosphorylation site**: `[RK].{0,2}[DE]`

   Either R or K, followed by 0 to 2 of any amino acid, then either D or E.

PostgreSQL's regular expression functionality works well for these pattern matching operations.

Implementation Considerations

- **Parameterized Queries**

  Always use parameterized queries with the $1, $2 syntax to prevent SQL injection attacks:

```
const result = await pool.query(
  'SELECT * FROM proteins WHERE name = $1',
  [name]
);
```

- **Error Handling**

  Add basic error handling for database operations:

```
try {
  const result = await pool.query('SELECT * FROM proteins');
  return result.rows;
} catch (error) {
  console.error('Database error:', error);
  throw new Error('Database operation failed');
}
```

PostgreSQL errors provide detailed information beyond the generic message, allowing for more sophisticated error handling. When catching database errors, you can access properties like `error.code` (containing PostgreSQL's SQLSTATE error codes) and `error.constraint` (identifying which constraint was violated). This enables implementing specific recovery strategies for different error conditions, such as unique constraint violations (23505), foreign key violations (23503), or check constraint failures (23514). For production applications, consider examining these properties to provide more meaningful feedback to users or to implement automatic retry mechanisms for transient errors.

- **Performance Considerations**
  - Use the indexes defined in the schema for efficient querying
  - Implement pagination for the GET /api/proteins endpoint
  - Use database joins to retrieve related data in a single query when possible