BLG317E PROJECT REPORT

Film Recommendation System

Eda Selin Küçükkara
Artificial Intelligence and Data Engineering Department
Istanbul Technical University
150210325
kucukkara21@itu.edu.tr

Revna Altınöz
Artificial Intelligence and Data Engineering Department
Istanbul Technical University
150220756
altinoz20@itu.edu.tr

I. Overview of the Project Idea

The Film Recommendation System is designed to enhance users' movie-watching experiences by providing personalized recommendations, managing watch histories, and enabling detailed movie exploration. It incorporates key functionalities such as rating and reviewing movies, associating movies with genres, and maintaining robust user profiles. This system uses a MySQL database with RESTful APIs developed using Flask to perform all required operations efficiently.

The application aims to serve as a robust backend for a movie platform, ensuring secure and seamless management of movie data, user interactions, and recommendation functionalities.

II. ER Diagram and Data Model Explanations

A. ER Diagram

The ER diagram for the Film Recommendation System depicts the relationships among the following main entities:

- **User**: Attributes: `user_id` (PK), `user_name`, `email`, `password`, `preferences`.
- **Movie**: Attributes: `movie_id` (PK), `title`, `description`, `duration`.
- **Genre**: Attributes: `genre_id` (PK), `genre_name`.
- **Movie_Genre** (Weak Entity): Composite Key: (`movie_id`, `genre_id`), Foreign Keys: `movie_id` → Movie, `genre_id` → Genre.
- **Rating**: Attributes: `rating_id` (PK), `user_id` (FK), `movie_id` (FK), `score`.
- **Review**: Attributes: `review_id` (PK), `user_id` (FK), `movie_id` (FK), `review_text`.
- **Watch_History**: Attributes: `history_id` (PK), `user_id` (FK), `movie_id` (FK).
- **Recommendation**: Attributes: `recommendation_id` (PK), `user_id` (FK), `movie_id` (FK).

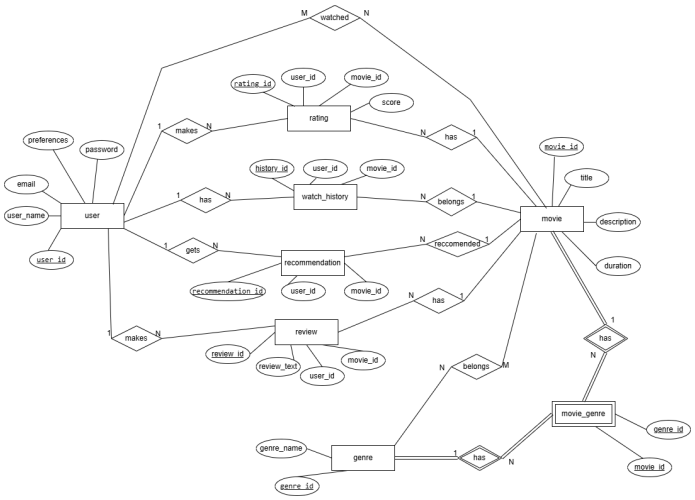

Fig. 1. ER Diagram of the Database

B. Data Model

Each table in the database is designed with appropriate constraints and relationships to ensure data integrity and efficient query execution.

- **user Table**: Contains user details and preferences. Enforces unique constraints on `user_name` and `email`.
- **movie Table**: Stores movie metadata such as title, description, and duration.
- **genre Table**: Maintains a list of distinct genres.
- **movie_genre Table**: Maps movies to their respective genres using a composite primary key.
- **rating Table**: Stores user-provided ratings for movies, with constraints ensuring valid scores between 1.0 and 5.0.
- **review Table**: Contains user-written reviews for movies.
- **watch_history Table**: Logs movies watched by users.
- **recommendation Table**: Stores recommended movies for users based on preferences or activity.

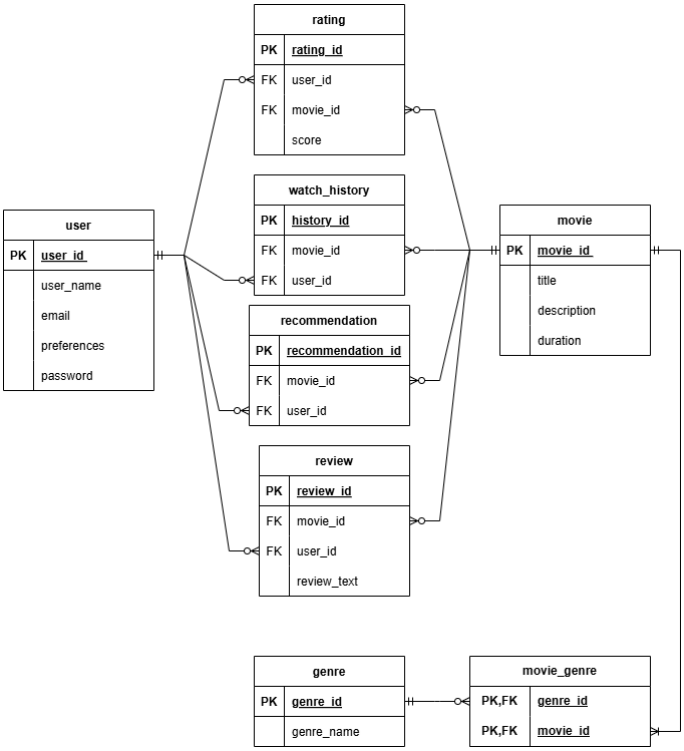

Fig. 2. Data Model of the Database

III. Descriptions of CRUD Operations and Their Implementations

A. user Operations

- **Create**: Register a new user with unique `email` and `user_name`.
- **Read**: Fetch user details by `user_id`.
- **Update**: Update user name e-mail or preferences.
- **Delete**: Remove a user and associated data such as ratings and reviews.

B. movie Operations

- **Create**: Add a new movie to the database.
- **Read**: Retrieve movie details.
- **Update**: Modify movie metadata such as description or duration.
- **Delete**: Remove a movie, ensuring related data integrity.

C. genre Operations

- **Create**: Add a new genre.
- **Read**: List all genres.
- **Update**: Modify genre names.
- **Delete**: Remove a genre if no movies are associated.

D. movie_genre Operations

- **Create**: Associate a movie with a genre.
- **Read**: List all genre associations for a movie.
- **Delete**: Remove a specific movie-genre association.

E. rating Operations

- **Create**: Submit a rating for a movie by a user.
- **Read**: Fetch all ratings for a movie.
- **Update**: Modify a user's rating for a movie.
- **Delete**: Remove a user's rating.

F. review Operations

- **Create**: Submit a review for a movie by a user.
- **Read**: Fetch all reviews for a movie.
- **Update**: Edit a user's review.
- **Delete**: Remove a user's review.

G. watch_history Operations

- **Create**: Log a movie to a user's watch history.
- **Read**: Retrieve the watch history for a user.

H. recommendation Operations

- **Create**: Add a recommended movie for a user.
- **Read**: Fetch all recommendations for a user.
- **Delete**: Remove a recommendation.

I. Complex Relationships

movie_genre associations are managed using the `movie_genre` table, ensuring robust many-to-many relationships.

IV. API Design and Endpoint Explanations

A. Overview

The RESTful API enables seamless interaction with the database. It uses JWT-based authentication for secure access and follows REST best practices.

B. Endpoints

- **user Endpoints**
  - `GET /users`: Fetch all users.
  - `GET /users/<int:user_id>`: Fetch user specified with id.
  - `PUT /users/<int:user_id>`: Update user.
  - `DELETE /users/<int:user_id>`: Delete a user.
- **movie Endpoints**
  - `POST /movies`: Add a new movie can be accessed by admin only.
  - `GET /movies`: Retrieve all movies.
  - `GET /movies/<int:movie_id>`: Retrieve movie by its id.
  - `PUT /movies/<int:movie_id>`: Update a movie.
  - `DELETE /movies/<int:movie_id>`: Delete a movie can be accessed by admin only.
- **genre Endpoints**
  - `POST /genres`: Add a new genre can be accessed by admin only.
  - `GET /genres`: List all genres.
  - `GET /genres/<int:genre_id>`: List genre by its id.

  - PUT /genres/<int:genre_id>: Update a genre can be accessed by admin only.
  - DELETE /genres/<int:genre_id>: Delete a genre can be accessed by admin only.
- **movie_genre Endpoints**
  - POST /movie-genre: Assigning a movie to a genre can be accessed by admin only.
  - GET /movies/<int:movie_id>/genres: Get all genres of a movie.
  - GET /genres/<int:genre_id>/movies: Get all movies of a genre.
  - DELETE /movie-genre: Delete a genre from a movie can be accessed by admin only.
- **rating Endpoints**
  - POST /ratings: Add a rating.
  - GET /movies/<int:movie_id>/ratings: Get all ratings for a movie.
  - PUT /ratings/<int:rating_id>: Update a rating.
  - DELETE /ratings/<int:rating_id>: Delete a rating.
- **review Endpoints**
  - POST /reviews: Add a review.
  - GET /movies/<int:movie_id>/reviews: Get all reviews for a movie.
  - PUT /reviews/<int:review_id>: Update a review.
  - DELETE /reviews/<int:review_id>: Delete a review.
- **watch_history Endpoints**
  - POST /watch-history: Add a movie to the watch history.
  - GET /watch-history: Fetch user's watch history.
- **recommendation Endpoints**
  - POST /recommendations: Give recommendation to a user can be accessed by admin only.
  - GET /recommendations: Retrieve movie recommendations for user.
  - DELETE /recommendations/<int:user_id>/<int:movie_id>: Delete a recommendation can be accessed by admin only.
- **Other Endpoints**
  - POST /register: Register a new user.
  - POST /login: User login.
  - GET /initialize-database: Initialize the database and create tables.

C. Swagger Documentation

Interactive documentation is integrated in app and deployed locally and hosted at http://127.0.0.1:5000/apidocs when the code running for the app. It provides detailed descriptions of each endpoint, including request and response schemas.

V. Examples of Complex Queries with Explanations

A. Filter Movies by Genre, Duration, and Rating

```
SELECT m.movie_id, m.title, m.description,
m.duration, COALESCE(AVG(r.score), 0)
AS avg_rating
FROM movie m
JOIN movie_genre mg ON m.movie_id = mg.movie_id
JOIN genre g ON mg.genre_id = g.genre_id
LEFT JOIN rating r ON m.movie_id = r.movie_id
WHERE g.genre_name = %s
AND m.duration BETWEEN %s AND %s
GROUP BY m.movie_id
HAVING avg_rating >= %s
```

Explanation: This query filters movies based on genre, duration range, and minimum rating. This query is utilized in the /movies/filter endpoint to provide users with tailored movie recommendations.

B. Top Movies by Genre

```
SELECT m.movie_id, m.title, m.description,
COALESCE(AVG(r.score), 0) AS avg_rating
FROM movie m
JOIN movie_genre mg ON m.movie_id = mg.movie_id
JOIN genre g ON mg.genre_id = g.genre_id
LEFT JOIN rating r ON m.movie_id = r.movie_id
WHERE g.genre_name = %s
ORDER BY avg_rating DESC
LIMIT %s
```

Explanation: This query retrieves the top-rated movies for a specific genre. This query powers the /movies/top endpoint, allowing users to discover highly-rated movies in their preferred genre.

C. Genre Statistics

```
SELECT g.genre_name, COUNT(m.movie_id)
AS movie_count, AVG(r.score) AS avg_rating
FROM genre g
LEFT JOIN movie_genre mg
ON g.genre_id = mg.genre_id
LEFT JOIN movie m ON mg.movie_id = m.movie_id
LEFT JOIN rating r ON m.movie_id = r.movie_id
GROUP BY g.genre_name
ORDER BY movie_count DESC
```

Explanation: This query provides statistics for each genre, including the number of movies and the average rating. This query is utilized in the /genres/statistics endpoint to give users insights into the popularity and quality of different genres.

D. Finding Genres of the Top-Rated Movie

```
SELECT DISTINCT g.genre_id, g.genre_name
FROM genre g
JOIN movie_genre mg
ON g.genre_id = mg.genre_id
```

```
WHERE mg.movie_id IN (
    SELECT m.movie_id
    FROM movie m
    LEFT JOIN rating r
    ON m.movie_id = r.movie_id
    GROUP BY m.movie_id
    HAVING AVG(COALESCE(r.score, 0)) = (
    SELECT MAX(avg_rating)
    FROM (
        SELECT AVG(COALESCE(r.score, 0))
        AS avg_rating
        FROM movie m
        LEFT JOIN rating r
        ON m.movie_id = r.movie_id
        GROUP BY m.movie_id
    ) AS subquery
))
```

Explanation: This query retrieves the genres associated with the highest-rated movie. This query supports the /genres/top-rated-movie endpoint, offering users information about the genres of the best-rated movie.

VI. Challenges and Solutions

A. Challenges

- **Database Design**: Ensuring robust relationships between tables.
  - **Solution**: Used a normalized schema with clear primary and foreign keys.
- **Handling Many-to-Many Relationships**:
  - **Solution**: Created `movie_genre` as a weak entity with a composite key.
- **Secure Authentication**:
  - **Solution**: Implemented JWT-based authentication for API access.
- **Complex Query Performance**:
  - **Solution**: Optimized queries using appropriate indexing and constraints.

VII. Conclusion

The Film Recommendation System successfully achieves its goal of providing a robust backend solution for managing movies, users, and recommendations. By employing a normalized database schema, RESTful APIs, and clear documentation, the project ensures scalability, security, and usability.

This project demonstrates the importance of database normalization in reducing redundancy and improving query efficiency, which directly contributes to a seamless user experience. The inclusion of advanced querying capabilities, such as filtering movies by multiple criteria and retrieving statistical insights, showcases the system's ability to handle complex use cases effectively. Furthermore, the use of JWT-based authentication adds a layer of security, ensuring that user data and system resources remain protected.

The RESTful API design adheres to best practices, making it easy for developers to integrate and extend the system in the future. Comprehensive API documentation using Swagger further enhances usability by providing clear guidance for potential users and collaborators. This report summarizes the development process and highlights key technical and architectural decisions that contribute to the overall robustness and reliability of the system.

Overall, the Film Recommendation System is a scalable and efficient solution capable of meeting the needs of diverse user groups, setting a solid foundation for future enhancements and integrations. The insights gained from this project can serve as valuable learning points for similar applications in the domain of recommendation systems.