Database Systems Courses - Homework 3

By:

Shay Fux, 313452252

Nathan Bloch, 316130707

Database Scheme Structure

Movies(movieID, title, adult, overview, popularity, releaseDate, runtime)

- movieID - the API's movie ID (The Movie DB API).
- title - the movie's title.
- adult - boolean value which indicates if the movie is appropriate for non-adults.
- overview - short plane text overview of the movie.
- popularity - numeric value that represents popularity of the movie.
- releaseDate - release date of the movie.
- runtime - movie length.

MovieGenres(movieID, genreID)

- movieID - the API's movie ID (The Movie DB API).
- genreID - ID of the genre.

Genres(genreID, genreName)

- genreID - ID of the genre.
- genreName - the name of the genre.

Actors(actorID, actorName , popularity)

- actorID - the API's actor ID (The Movie DB API).
- actorName - the actor's name.
- popularity - numeric value that represents popularity of the actor.

MoviesActors(movieID, actorID)

- movieID - the API's movie ID (The Movie DB API).
- actorID - the API's actor ID (The Movie DB API).

Directors(directorID, directorName)

- directorID - API's director ID (The Movie DB API).
- directorName - the director's name.

MovieDirector(movieID, directorID)

- movieID - the API's movie ID (The Movie DB API).
- directorID - API's director ID (The Movie DB API).

MovieCollection(<u>movieID</u>, collectionID)

- movieID - the API's movie ID (The Movie DB API).
- collectionID - the API's collection ID(The Movie DB API) which the movie belongs to.

Collection(<u>collectionID</u>, collectionName)

- collectionID - the API's collection ID(The Movie DB API) which the movie belongs to.
- collectionName - the collection's name.

<u>Database Design Explanation</u>

The database schema we created tried to achieve all the best-practices we have learned in class. The database schema we tried to achieve is a normalized schema. There are two types of tables in the DB: one type of tables is tables that hold unique data, and the second type is tables that have a role of connecting between tables from the previous type. In that way, we achieved a normalized schema, in which on one hand there is no data duplication since each table holds unique data, but on the other hand there is no data loss due to the ability to join tables.

We preferred this schema over others due to its simplicity in our opinion, as we think this schema is the most straight-forward solution.

<u>Description of Main Queries</u>

1. **Movie Recommendation by Collection**
    - Given a movie title, the query returns all movies which are from the same collection as the input movie title. The user can use this functionality to find movie recommendations.
    - To optimize this query, we added an index on the 'title' column of the 'Movies' table. This allows a more efficient way to find the corresponding movie ID.
    - The DB supports this query by containing MovieCollection and Movies tables. The Movies table helps in retrieving the movie's ID based on the input title, then the MovieCollection helps in finding the relevant collection of the input title, and finally the MovieCollection table is used again to find all movies within that collection.
2. **Top Movies of Actor**
    - Given an actor name, the query returns the TOP-5 popular movies in which the given actor played. The user can use this functionality to get the names of the most popular movies in which his favorite actor played.
    - To optimize this query, we added an index on the 'actorName' column of the 'Actors' table. This allows a more efficient way to find the corresponding actor ID.

- The DB supports this query by containing the Actors, MoviesActors and Movies tables. The Actors table is used to retrieve the input actor's ID. Then, the MoviesActors table is used to retrieve all movie IDs in which the given actor played(using the actor's ID). Finally, the Movies table is used to retrieve all movie names of the relevant movie IDs.

3. **Top movies in genre and with specific runtime**
   - Given a genre and runtime X, the query returns the TOP-5 popular movies in the genre with runtime $\leq X$. The user can use this functionality to get the names of the most popular movies in the genre he specifies that don't exceed his movie's runtime limitation.
   - To optimize this query, we added an index on the 'genreName' column of the 'Genres' table. This allows a more efficient way to find the corresponding genre ID. However, the Genres table is extremely small(19 rows) and thus the index is very small in terms of memory and therefore not very significant.
   - The DB supports this query by containing the Genres, MoviesGenres and Movies tables. The Genres table is used to retrieve the input genre's ID. Then, the MoviesGenres table is used to retrieve all movie IDs in the given genre(using the genre's ID). Finally, the Movies table is used to retrieve all movie names of the relevant movie IDs.
4. **Searching for Movies using a search key in the movies overview (Fulltext Query)**
   - Given a search key, the query returns the TOP-5 movies ranked by popularity which include the search key in their overview.
   - To optimize this query, we added a fulltext index on the 'overview' column of the 'Movies' table. This allows a much more efficient way to search in text using given keywords as required in this query.
   - The DB supports this query by containing the Movies table, that holds the title, popularity and overview of the movie.
5. **Top directors between range of years**
   - Given start date X and end date Y, the query returns the TOP-5 directors in terms of average movie's popularity which they have directed and their reales date is between X and Y.
     The user can use this functionality to get the names of the most successful directors at the range of years he specified.
   - To optimize this query, we added an index on the 'releaseDate' column of the 'Movies' table. This allows a more efficient way to handle the range condition on the release date.
   - The DB supports this query by containing the Directors, MovieDirector and Movies tables. The MovieDirector table is used and the Movies table is used to retrieve the directorIDs that have directed movies whose release date is

between the user range input. Finally, the Directorstable is used to retrieve all directors' names of the relevant directors' IDs.

6. **Top combination of actor & director movies**
    - Given actor name and director name , the query returns the TOP-10 popular movies in which the given actor played and the given director directed. The user can use this functionality to get the names of the most popular movies such his favorite actor and director have participated in .
    - To optimize this query, we added an index on the 'actorName' column of the 'Actors' table and on the 'directorName' column of 'Directors' table. This allows a more efficient way to find the corresponding actor ID and director ID.
    - The DB supports this query by containing the Actors, MoviesActors , Directors, MovieDirector and Movies tables. The Actors and Directors tables are used to retrieve the input actor's ID and director's ID respectively. Then, the MoviesActors and MovieDirector tables are used to retrieve all movie IDs in which the given actor and director have both been participating (using the actor's ID and director's ID). Finally, the Movies table is used to retrieve all movie names of the relevant movie IDs.
7. **Movies of Top actors directed by specific director**
    - Given director name , genre name and range of years the query returns the Top-10 movies in terms of the average actors' popularity who participated in them , such that the given director directed them , their release date is between the given range and their genre is as the given genre name.
    - Many of the previously stated indexes can optimize this query:
        - the index from bullet (6) regarding the directors names.
        - the index from bullet (3) regarding the genres names.
        - the index from bullet (5) regarding the release dates.
    - The DB supports this query by containing Movies , MovieDirector , Directors, MovieGenres , Genres , MoviesActors and Actors tables in order to yield the designated Movies titles in accordance to the user parameters.

<u>Code Structure</u>

The project could be roughly partitioned into 3 parts:

- **Data Retrieval through API**

  The API we used to retrieve our data is The Movie Database API (https://developers.themoviedb.org/3). We saved the data into a JSON formatted file so we could only retrieve the data once from the API. We used 3 different APIs from that website, and we saved the retrieved data in the following files:
    - movies.json - holding data about movies in general.
    - actors.json - holding data about the actors in each movie.
    - directors.json - holding data about the directors in each movie.

  The relevant code is in SRC/API-DATA-RETRIEVE/DATA-API-TO-JSON.py.

- **DB Creation + Data Insertion**
  We used Python in order to connect to the DB and create the DB tables. The tables we created were exactly as stated in the DB schema previously. However, we assigned primary keys as required. The code is in the CREATE_DB_SCHEMA.py file inside the SRC/CREATE-DB-SCRIPT folder.
  In addition, we used Python in order to insert data into the DB as well. Since all our data is stored in the JSON files, we implemented the insertion by parsing the JSON files and retrieving the relevant data for each table. The code is in the INSERT_DATA_DB.py file inside the SRC/CREATE-DB-SCRIPT folder.

- **DB Querying("the app")**
  We implemented only the backend of the DB Querying part. The queries were implemented in the SRC/queries.py file. This python file includes 7 methods that build the queries based on the required input. The methods return a string that is the query for the DB. The DB querying is done in the SRC/show_queries_output.py file. This file includes 7 methods that, given an input from the user, build the query using the input, connect to the DB, execute the query on the DB, show the results back, and close the connection to the DB.
  These parts are however only the backend of the app and are relevant only to the server side, and therefore were not designed for the client.

Description of API + API Usage

The API we used to retrieve our data is The Movie Database API (https://developers.themoviedb.org/3). We saved the data into a JSON formatted file so we could only retrieve the data once from the API. We used 3 APIs different from that website.

The first API is https://api.themoviedb.org/3/discover/movie, that given a page index returns the 20 movie ids that are in that page. This API allowed us to find all movie ids.

The second API was https://api.themoviedb.org/3/movie/{movie_id}, that given a movie id returns all information about that movie. We used that API to retrieve general information about the movies using the movie ids from the previous step. We saved the retrieved data in the movies.json file. We retrieved data for about 10000 movies.

The third API was https://api.themoviedb.org/3/movie/{movie_id}/credits, that given a movie id returns all information about the credits for the movie. We used that API to retrieve data about the actors and directors of each movie. We saved the retrieved data in the actors.json & directors.json files.

General Flow of the Application

The app is an interactive recommender app. The user logs into the website, and then he sees a nicely designed welcome screen in which he can choose which recommendation option he would like to use. Then, he enters the relevant input and submits the form. The form is then sent from the client to the server side where the relevant query(including the user input) will be executed on the DB. Then, the results of the query will be returned to the client side and be shown to him in a tabular way.