Software Documentation

DB Scheme Structure :

movie(id, title, year, usa_gross_income) - Information about movies
person(id, name, birthplace, children) - Information about actors and actresses
principal(id, movie_id, person_id, category) - Information about roles in movies - actors, actresses, directors, writers etc.
rating(id, total_votes, mean_vote) - Information about the rating of movies
genre(id, movie_id, genre) - Information about genres of movies
best_picture_winners(id) - Information about movies that won in the best picture award.

We collected datasets online that would match our goal - creating a BCNF db scheme.
At first, genres were inside a nested list on 'movie' dataset, so we split it in order to maintain 1NF.
We tried to split it into many tables in order to keep BCNF and allow easy updates.

DB Optimizations:

1. indices - we indexed relevant columns for our queries:
    a. movie.year
    b. movie.usa_gross_income
    c. person.children
    d. rating.total_votes
    e. rating.mean_vote
    f. genre.genre
    g. principal.category
    h. person.name
    i. person.birthplace

    Those indices allow us to search, filter and find in range much faster, giving our users better performance. Indices are especially useful in our service, since the queries are dynamic and their output changes by the user input.
    The **disadvantage** of indexing many fields is that db updates will be slower. We assumed we should focus on query performance rather than db updates which won't happen as often as query execution.
2. Primary keys - for each table, we defined a primary key, named "id"
3. Foreign keys - we had some data anomalies in our dataset, therefore we used foregin keys to provide referential integrity:
    a. principal(person_id) references person(id)
    b. principal(movie_id) references movie(id)

    c. genre(movie_id) references movie(id)
4. Normalization - as mentioned in **db scheme** description, our DB is in BCNF.
   We could have designed the tables to better match our queries (demand less joins) and increase performance, but we thought less data redundancy would make our app more scalable and allow more queries to be added later on.
5. NOT NULL - we learned that using it as much as possible enables better use of indexes.
   We decided to add it after we created the tables, so we altered the following :
    a. movie.title, movie.year
    b. person.name, person.children(defaults 0)
    c. principal.category,
    d. rating.total_votes, rating.mean_vote
    e. Genre.genre

   Other fields are primary keys (not null by default), foreign keys (mapped to a primary key) or fields which we allow nulls (e.g gross income)

Queries description:

1. Popularity(v,y)
    a. find the most popular movie in each genre, between movies that have at least (v) votes, which came out in year (y).
    b. Optimization - indexing the table “movie” by “year” column to efficiently find movies which came out in the specific year, and “rating” table by column “total_votes” to efficiently find movies that their number of votes is in the desired range.
2. Actors(g)
    a. count actors and actresses who played in movies which got a mean rating of at least (g). split it by the amount of children.
    b. Optimization - indexing the table “rating” by “mean_vote” column to efficiently find movies with rating in the desired range, and “person” table by “children” column to efficiently split the output by number of children.
3. The most vivid writer - hard-coded query, no user input
    a. find the top 10 writers who wrote movies in the maximal number of different genres. assumption - multi-genre movies count as 1 for each genre.
    b. Optimization - indexing the table “genre” by “genre” column to efficiently find how many genres each writer wrote, and “principal” table by “category” column to efficiently go through the writers.
4. Winning *role*(role)

    a. find the (role) who participated in most award winning movies. For example - if the user input is “director” the query would find the director who directed most award winning movies.
    b. Optimization - indexing the table “principal” by “category” column to efficiently go through the role requested.
5. *a,b,c*(a,b,c)
    a. Find all movies that their usa gross income was more than (a) million usd, got an average vote of at least (b), at least (c) actors and actresses participated in the movie.
    b. Optimization - indexing the table “movie” by “usa_gross_income” column to efficiently find movies that have income in the desired range, indexing “rating” table by “mean_vote” to efficiently find movies with rating in the desired range, and “principal” table by “category” to efficiently count number of actors and actresses in each movie.
6. How popular is the name *name*(name)
    a. for each movie, count the number of actors and actresses that played in it named (name), in descending order
    b. Optimization - full text query, “name” column in “person” table is a fulltext index.
7. Number of actors and actresses from each city in country *c* (c)
    a. for country (c), get the number of actors and actresses born in each city, for cities whose number of actors>0), in descending order.
    b. Optimization - indexing “person” table by “birthplace” to efficiently go through each one’s birthplace.

Code structure:

- API-DATA-RETRIEVE.py
  Contains the API data fetch and table creation.
- CREATE-DB-SCHEME.py
  Contains tables creation from our datasets (`init_db()`),
  adding NOT NULL (`alter_not_null()`),
  and indices creation (`create_indicies()`).
  Tables are created by using the ‘create_table’ generalized function.
  The genres table is created separately because it splits a list to different rows.
- queries.py
  Contains our dynamic queries (listed in same order as above), which can be executed by a server given a user’s input. The server will call `get_query_result` which wraps the relevant query_x function that returns a sql command.

Example - server will run `get_query_result(query7('israel'))` and process the query output.

Description of our API:

We used the IMDB API by RapidAPI. We fetched an imdb-ids list (which match our datasets) of best picture award winners, and created a table 'best_picture_winners'. This data gives us a different perspective of a "successful" movie, besides user ratings.

The rest of the data is based on csv tables we found online, from a db called "imdb-extensive-dataset" on kaggle, but it seems like since we started the project, the tables were removed from the site (all pages lead to 404).

General flow of the application:

1. The user inserts the inputs for the queries.
2. The client sends the inputs to the server.
3. The server calls our queries.py query-execute-API, which executes the desired queries, based on the user inputs. The data is retrieved from our database and returned to the server.
4. The server passes the outputs to the client.
5. The outputs are available to the user via the UI.