Database Management Systems

System Documentation

Database Schema:

- Green marks tables added to create many to many relations

1. movies (**id**, name, date_published, description, budget, box_office, avg_vote, num_of_votes, revenue (created automatically))

• Attributes:

- Id: int Primary Key
- name: varchar(255)
- date_published: date
- description: text
- budget: unsigned int
- box_office: unsigned int
- avg_vote: unsigned float (2,1)
- num_of_votes: unsigned int
- revenue: bigint derived from (COALESCE (box_office, 0) – COALESCE (budget, 0))

(The unsigned constraints to only positive values)

• Declared Indices:

- Full Text: description
- Full Text: name
- B-Tree Index: date_published
- B-Tree Index: revenue
- Hash Index: name

2. genres (**id**, name)

• Attributes:

- Id: int Primary Key
- name: varchar (255)

• Declared Indices:

- Hash Index: name

• Constraints:

- unique name

3. movieGenres (**movie_id**, **genre_id**):

• Attributes:

- movie_id: int
- genre_id: int
- Primary Key is combination of (movie_id, genre_id)

• Foreign Keys:

- movie_id -> movies.id
- genre_id -> genres.id

4. actors (**id**, name)

• Attributes:

- Id: int Primary Key
- name: varchar (255)

• Constraints:

- unique name

5. movieActors (**movie_id**, **actor _id**):

• Attributes:

- movie_id: int
- actor_id: int
- Primary Key is combination of (movie_id, actor_id)

• Foreign Keys:

- movie_id -> movies.id
- actor_id -> actors.id

6. directors (**id**, name)

• Attributes:

- Id: int Primary Key
- name: varchar (255)

• Constraints:

- unique name

7. movieDirectors (**movie_id**, **director_id**):

Attributes:

- movie_id: int
- director_id: int
- Primary Key is combination of (movie_id, director_id)

• Foreign Keys:

- movie_id -> movies.id
- director_id -> director.id

Main Queries

The first two are Full-Text queries, and the last 3 are very complex queries using nested queries, group by, aggregation and more.

Query 1 – Full Text - Word Description Search:

```
SELECT name AS film_title ,revenue, date_published, avg_vote, description
FROM movies
WHERE MATCH(description) AGAINST(%s)
    AND avg_vote >= %s
ORDER BY revenue DESC
```

Parameters: word_searched_in_description (str), avg_vote_threshold (int)

Output: Query returns rows of films that description contains the { word_searched_in_description} which average user vote is higher or equal to { avg_vote_threshold} it then displays the columns as described in the query SELECT above.

Query 2 – Full Text - Word Prefix Film Title Search:

```
SELECT name AS film_title, revenue, date_published, avg_vote,
num_of_votes , description
FROM movies
WHERE MATCH(name) AGAINST(%s IN BOOLEAN MODE)
    AND num_of_votes >= %s
ORDER BY revenue DESC;
```

Parameters: word_searched_in_title(str), num_of_votes_threshold(int)

Output: Query returns rows of films that name (film title) contains word that { word_searched_in_title } is a prefix of, which has at least the amount of user vote that is equal or higher to { num_of_votes_threshold } it then displays the columns as described in the query SELECT above.

Query 3 – Complex Query - Top Actors Career Revenue by Film Name:

```
WITH actor_and_number_of_movies AS(
SELECT actor_id , count(*) AS num_of_movies
FROM movieActors
GROUP BY actor_id
)
SELECT actors.name, AVG(revenue) as average_movies_revenue
FROM movies, movieActors, actors
WHERE movies.id = movieActors.movie_id
AND actors.id = movieActors.actor_id
AND actors.id  IN (
    SELECT actor_id
    FROM actor_and_number_of_movies
    WHERE num_of_movies >= %s
)
AND actors.id IN(
    SELECT actor_id
    FROM actors, movieActors, movies
        WHERE actors.id = movieActors.actor_id
        AND movies.id = movieActors.movie_id
        AND movies.name = %s
)
GROUP BY actors.id, actors.name
ORDER BY average_movies_revenue DESC
LIMIT %s
```

Parameters: min_film_threshold(int), movie_name(str), num_of_top_actors(int)

Output: Query returns actors that played a specific film, by that film name { movie_name} ordered by actors average career movies revenue: that is defined as the revenue of the average film the played at.

With the ability to set a minimum amount of films the actor played at { min_film_threshold} and limit the amount of actors { num_of_top_actors}, the actors are ordered by their descending average revenue, the actor name and career average revenue is returned.

Purpose: May be used to identify actors the consistently are part of revenue producing films.

Query 4 – Complex Query – Top Hiring Directors by Genre:

```
WITH director_and_number_of_movies AS (
    SELECT movieDirectors.director_id, COUNT(*) AS num_of_movies
    FROM movieDirectors, genres, movies, movieGenres
    WHERE movieDirectors.movie_id = movies.id
    AND movieGenres.movie_id = movies.id
    AND movieGenres.genre_id = genres.id
    AND genres.name = %s
    GROUP BY director_id
),
actor_average_movie_revenue AS (
    SELECT actors.id, actors.name, AVG(movies.revenue) AS avg_actor_revenue
    FROM movieActors
    JOIN actors ON movieActors.actor_id = actors.id
    JOIN movies ON movieActors.movie_id = movies.id
    JOIN movieGenres ON movieGenres.movie_id = movies.id
    JOIN genres ON movieGenres.genre_id = genres.id
    WHERE genres.name = %s
    GROUP BY actors.id, actors.name
)
SELECT directors.id, directors.name, AVG(actor_average_movie_revenue.avg_actor_revenue) AS director_actors_average
FROM actor_average_movie_revenue
    JOIN movieActors ON movieActors.actor_id = actor_average_movie_revenue.id
    JOIN movies ON movieActors.movie_id = movies.id
    JOIN movieDirectors ON movieDirectors.movie_id = movies.id
    JOIN directors ON movieDirectors.director_id = directoras.id
    JOIN movieGenres ON movieGenres.movie_id = movies.id
    JOIN genres ON movieGenres.genre_id = genres.id

AND directors.id IN (
    SELECT director_id
    FROM director_and_number_of_movies
    WHERE num_of_movies >= %s
)
GROUP BY directors.id, directors.name
ORDER BY director_actors_average DESC
LIMIT %s;
```

Parameters: genre_name(str), num_of_min_films(int), num_of_top_directors(int)

Output: Query returns directors which directed at least { num_of_min_films} of films in a specific {genre_name}, ordering them by their actors hired in the films average career revenue in these types of genre of films. The amount of directors is limited by { num_of_top_directors}

Purpose: Identifying directors that hire well in a particular genre.

Query 5 – Complex Query – Top Film Genres and Most Popular Film in Genre in a Date Window:

```
WITH movie_in_dates AS (
    SELECT id
    FROM movies
    WHERE '2003-2-5' <= date_published
    AND date_published <= '2003-2-18'
),
top_film_in_date AS (
    SELECT genres.id AS genre_id, movies.id AS movie_id, movies.name AS top_film, movies.revenue AS max_revenue
    FROM movies
    JOIN movieGenres ON movieGenres.movie_id = movies.id
    JOIN genres ON movieGenres.genre_id = genres.id
    WHERE movies.id IN (SELECT id FROM movie_in_dates)
    AND (movies.revenue, genres.id) IN (
        SELECT MAX(movies.revenue), genres.id
        FROM movies
        JOIN movieGenres ON movieGenres.movie_id = movies.id
        JOIN genres ON movieGenres.genre_id = genres.id
        WHERE movies.id IN (SELECT id FROM movie_in_dates)
        GROUP BY genres.id
    )
)
SELECT genres.name, AVG(movies.revenue) AS avg_genre_revenue, top_film_in_date.top_film as top_film_in_genre
FROM movie_in_dates
    JOIN movieGenres ON movieGenres.movie_id = movie_in_dates.id
    JOIN genres ON genres.id = movieGenres.genre_id
    JOIN movies ON movies.id = movie_in_dates.id
    JOIN top_film_in_date ON top_film_in_date.genre_id = genres.id
GROUP BY genres.id, genres.name, top_film_in_date.top_film
ORDER BY avg_genre_revenue DESC
LIMIT 3 ;
```

Parameters: start_date(date), end_date(date), num_of_top_genres(int)

Output: Query returns film genres ordered by the descending average revenue of films in that genre in time window starting at {start_date} and ending at {end_date} along with the highest revenue film in that time

revenue of the genre. Limited the amount of rows returned by { num_of_top_genres}

Purpose: Identifying optimal release date for movies based on previous releases and genre.

Database Design

The database design is in BCNF to reduce data storage while allowing for flexibility in adding more functionality to the web application in the future, the tables: movieGenres, movieActors, movieDirectors are made to implement the many to many relation in a storage efficient way, the design allows for simple maintenance and modification of attributes, for example if we want to rename a genre we can change it in the genre table, a one row change without needing to change rows in the movies table for example if it was designed in another way.

One drawback in our design is the combination of different data, as you can see in our 4th query for example, combining different tables can create a very complex query, having to worry about in the 4th query about actors movies and directors forces us to deal with 6 tables and their different relations, storing all data in one table may allow for easier use. There is some redundant information that we store for possible future use cases.

The data types defined are around the smallest possible that can support the data populated.

The data type constraints defined ensure that the data inserted is logical, for example, ratings can't be negative or genre name must be unique.

Database Optimization:

- Full Text index on description: optimizes search efficiency for query 1 for word search in description
- Full Text index on name: optimizes search efficiency for query 2 for prefix search in film titles, according to documentation MYSQL I read

FULL TEXT index optimizes word search for prefix too, furthermore, empirically seen by how fast the query results are.

- Hash index on name in genres: optimizes query 4 which searches by genre name.
- Hash index on name in movies: optimizes query 3 which searches by specific movie name.
- BTREE index on revenue: query 1 and query 2 order by revenue so it optimizes them, also revenue is used extensively in query 3,4,5 so adding an index on it may allow MYSQL to perform more efficient algorithms.
- BTREE index on date: query 5 filters movies by date range the index optimizes that search therefore improving its run time

API Usage:

API date is sourced from GitHub user sahildit IMDB-Movies-Extensive-Dataset-Analysis which contains extensive film data. The csv file was modified with many changes to make it fit our use case and format. (encoding, removing non ascii characters, changing dates format and more) and filtered movies that answer:

• Movies under 5000 votes

• movies without budget or box office information

• movies after 1985

• movies with budget that isn't denominated in USD

For a total of 5772 rows in data set (which also contains for example around 38,000 actors.)

Code Structure:

In /src/create_db_script.py: straightforward creation of the database with a connector

/src/api_data_retrieve.py: parsing the data from cleaned_trimmed_fixed_data4.csv, using a helper function run_query to execute many queries, populating each table at a time, using hash_maps to

cache information that we would like to use for the next populating such as defined name : id for actors, movies, genres, directors, we manually enforce specific id's for this to work. The insert queries are straightforward.

In /src/queries_db_script.py Using a helper function function for select queries, we define our 5 main queries with their parameters and added checks that the queries are given correct parameters.

In /src/queries_execution.py We run 3 examples for each query in functions defined for each query. In our main we call all of them and print it using pandas layout printing helper function we defined, was structured in a way that is easy to read.