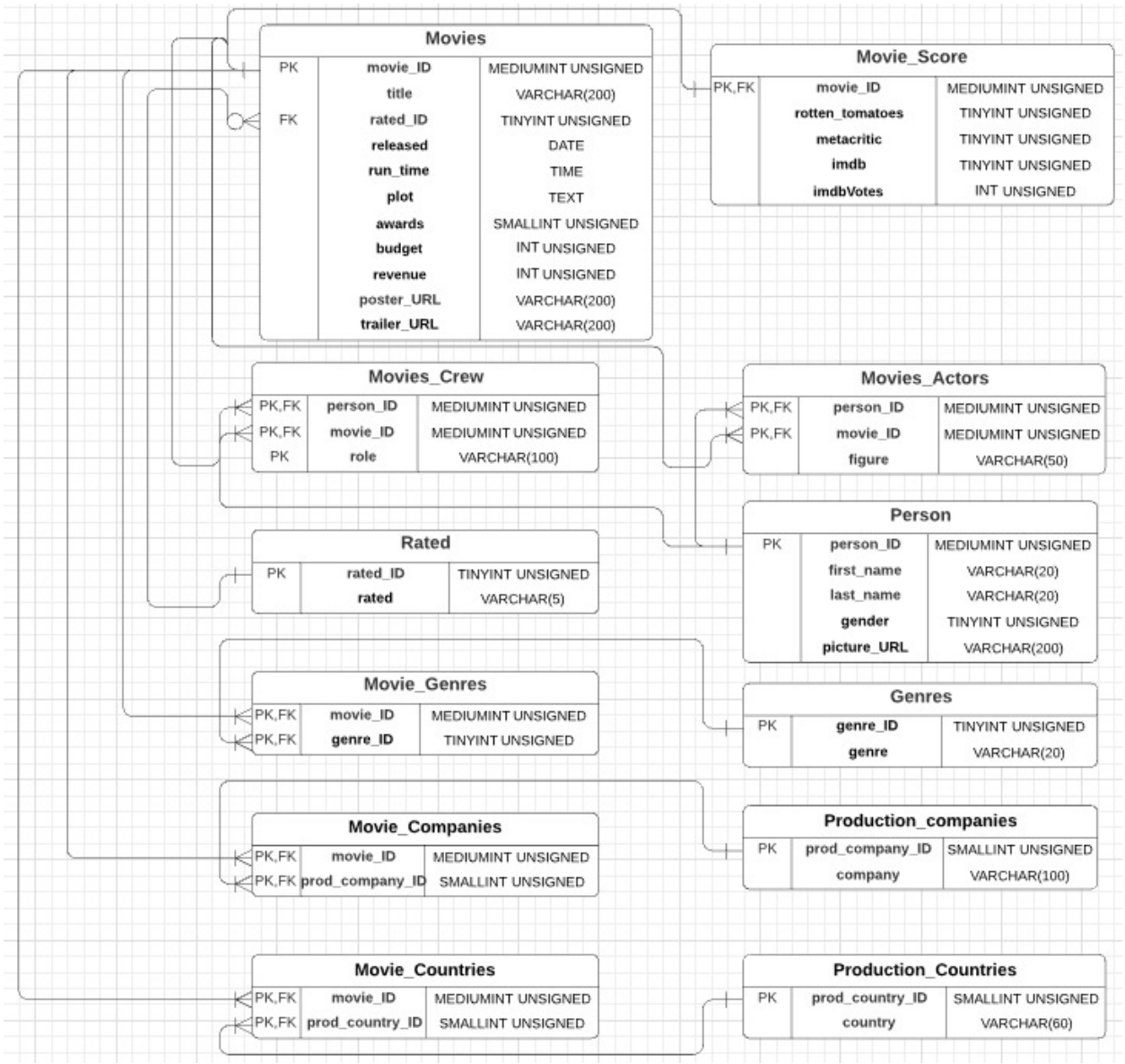
Oz Zafar, Mor Polani, Dvir Solomon, Daniel Kotler

Databases Course Project - FindMeAMovie

Software Documentation

DB scheme structure



Database design explanation – our main entities in the DB are movies and Person (actors and crew), so the correlated tables were created.
Each movie usually contains many actors and usually more than one from each of the following: genre, production company and production country. So, in order to prevent redundancy, we created 3 tables respectively connecting them and the movies.
Although each movie contains a single value of rating (PG, PG-13, etc..) - we decided to separate them into different tables in order to consume less memory (instead of storing a string multiple times, we replace it with an ID which is relatively smaller. Then we created an additional table which the connection id->rating is stored **once**.
Same for the movie's score, but the reason for the separating is convenient; the Score attributes could be attributes in the Movies table, but they are all related to the same topic (score) - so we decided to create an additional table.

DB optimization performed

Indexes:
The following is a list of the indexes we have defined:

- Indexes: every PK – default, Movie.released, Movies.run_time, Movie.budget, Movie.awards.
- Full Text Query Indexes: Movies.title, Movies.plot, Movies. Person.first_name, Person.last_name.

These indexes were defined for the following reasons:
1.1. Any column defined as PK is indexed. This is the default mode of MYSQL.
1.2. Queries that required text search directly raised the need for a full-text index. Hence the fields written above under Full Text Query Indexes are indexed as such.
1.3. For **every** entrance to the site, there is a query that works behind the scenes to bring the recommended movies for the user. Because of its high frequency, we decided that each attribute in the query (Movie.released, Movies.run_time), would be indexed to expedite the use of it.
1.4. We reviewed the attributes that are searched in our queries and indexed those that are used most frequently - Movie.released (which is indexed due to a previous section anyway), Movie.budget and Movie.awards.

Creating linking tables
As mentioned above, we used linking tables for some attributes in order to reduce the amount of memory in usage - we use IDs instead of Strings that are relatively bigger. Moreover, those

strings are repeated several times in different tuples, which increase the importance of that optimization.

7 main queries

Query 1

Given user choices: genres (G), length range (T1-T2) and years range (Y1-Y2) the query returns selects movies with length between T1,T2 that were filmed between Y1,Y2, and are categorized under at least one of genre in G. The query then selects up to 10 most popular movies from each category (popularity is measured by the average of the movie score in three different sites, calculated in “recommendations” VIEW) and returns them ordered by category and then by popularity descending.

- One movie can be categorized under more than one genre.

```
def recommendations_query(self, user_genres, min_len, max_len, start_year, end_year):
    user_genres_string = self.parse_genres("r", user_genres)

    query = f"""
    SELECT r.movie_ID, r.title, GROUP_CONCAT(r.genre) AS genres, r.released, r.run_time, r.popularity,
           r.poster_URL
    FROM Recommendations r
    WHERE ({user_genres_string}) AND r.popularity_rank <= 10
        AND EXTRACT(YEAR FROM r.released) BETWEEN {start_year} AND {end_year}
        AND (EXTRACT(HOUR FROM r.run_time)*60+EXTRACT(MINUTE FROM r.run_time))
        BETWEEN {min_len} AND {max_len}
    GROUP BY r.movie_ID, r.title, r.released, r.run_time, r.popularity
    ORDER BY genres, r.popularity DESC
    """
    rows = self.execute_sql(query)
    return rows
```

```
#VIEW for recommendations query
def recommendations_view(self):

    query= """
    CREATE OR REPLACE VIEW Recommendations AS
        SELECT g.genre, m.movie_ID, m.title, m.released, m.run_time,
          (ms.rotten_tomatoes+ms.metacritic+ms.imdb)/3 AS popularity,
          ROW_NUMBER() OVER(Partition BY g.genre ORDER BY (ms.rotten_tomatoes+ms.metacritic+ms.imdb)/3 DESC)
            AS popularity_rank, m.poster_URL
        FROM Movies m, Movie_Genres mg, Genres g, Movie_Score ms
        WHERE m.movie_ID = mg.movie_ID AND mg.genre_ID = g.genre_ID
        AND m.movie_ID = ms.movie_ID
    """
    self.execute_sql(query)
```

We optimized this query by building the “Recommendations” VIEW and using it for the query. This VIEW joins the relevant tables for the query and applying window function

ROW_NUMBER() to rank the popularity in each category. The query then selects the relevant rows according to the parameters passed by the user.

Query 2

Given a movie score (X) and range of years (Y1-Y2) the query returns for each actor the number of films he took a part in that got score X or above (while score is defined same as “popularity” mentioned above and is calculated in the Popular_Actors VIEW) and were filmed between the years Y1 to Y2.

```
def popular_actors_query(self, movie_score, start_year, end_year):

    query = f"""
    SELECT person_ID, first_name, last_name, gender, COUNT(*) AS amount_of_movies, picture_URL
    FROM Popular_Actors pa
    WHERE EXTRACT(YEAR FROM pa.movie_released) BETWEEN {start_year} AND {end_year}
        AND movie_popularity >= {movie_score}
    GROUP BY person_ID, first_name, last_name, picture_URL
    ORDER BY amount_of_movies DESC, first_name, last_name
    """

    rows = self.execute_sql(query)
    return rows
```

```
#VIEW for popular_actors query
def popular_actors_view(self):

    query="""
    CREATE OR REPLACE VIEW Popular_Actors AS
        SELECT p.person_ID, p.first_name, p.last_name, p.gender, p.picture_URL, m.movie_ID,
            m.released AS movie_released,
            (ms.rotten_tomatoes +  ms.metacritic + ms.imdb)/3 AS movie_popularity
        FROM Person p, Movies_Actors ma, Movie_Score ms, Movies m
        WHERE p.person_ID = ma.person_ID AND ma.movie_ID = ms.movie_ID AND ma.movie_ID=m.movie_ID
    """
    self.execute_sql(query)
```

We optimized this query by building the “Popular_Actors” VIEW and using it for the query. This VIEW joins the relevant tables for the query and calculating the popularity for each movie. The query then selects the relevant rows according to the parameters passed by the user.

Query 3

Given number of movies (X) and list of genres (Y) the query returns couples of director-actor who did together X or more movies while each movie counted is under a genre that appears in Y. For each couple the query also returns the number of movies they did together and the genres of these movies (each genre coming from one or more movies).

```
def director_actor_coupling_query(self, number_of_movies, user_genres):
    user_genres_string = self.parse_genres("dagm", user_genres)

    query = f"""
    SELECT director_ID, director_first_name, director_last_name, director_pic, actor_ID, actor_first_name,
        actor_last_name, actor_pic, SUM(co_operations) AS co_operations,
        GROUP_CONCAT(genre SEPARATOR "<br>") AS user_genres
    FROM Director_Actor_Genre_NumOfMovies dagm
    WHERE ({user_genres_string})
    GROUP BY director_ID, director_first_name, director_last_name, actor_ID, actor_first_name,
        actor_last_name
    HAVING co_operations >= {number_of_movies}
    ORDER BY co_operations DESC
    """

    rows = self.execute_sql(query)
    return rows
```

```
# VIEW FOR director_actor_coupling
def director_actor_coupling_view(self):

    query = """
    CREATE OR REPLACE VIEW Director_Actor_Genre_NumOfMovies AS
        SELECT g.genre_ID, g.genre, COUNT(*) AS co_operations, p1.person_ID AS director_ID,
            p1.first_name AS director_first_name, p1.last_name AS director_last_name,
            p1.picture_URL AS director_pic,p2.person_ID AS actor_ID, p2.first_name AS actor_first_name,
            p2.last_name AS actor_last_name, p2.picture_URL AS actor_pic
        FROM Person p1, Person p2, Movies_Crew mc, Movies_Actors ma, Movies m, Movie_Genres mg, Genres g
        WHERE p1.person_ID=mc.person_ID AND p2.person_ID=ma.person_ID AND
            mc.role = "Director" AND mc.movie_ID=m.movie_ID AND ma.movie_ID=m.movie_ID
            AND m.movie_ID = mg.movie_ID AND mg.genre_ID = g.genre_ID
        GROUP BY g.genre_ID, g.genre, p1.person_ID, p1.first_name, p1.last_name, p2.person_ID,
            p2.first_name, p2.last_name
        """

    self.execute_sql(query)
```

We optimized this query by building the “Director_Actor_Genre_NumOfMovies” VIEW and using it for the query. This VIEW joins the relevant tables for the query, grouping records by director-actor couples and calculating the number of their co-operations.

Query 4

Given budget (X) and number of production companies (Y) the query returns the total budget and names of directors which the total budget of movies they did, while each movie in the list was produced by at least Y production companies, is at least X. For each director in the list we also present the movies he directed with more than X production companies and their budget. For each director the movies he directed are indexed by their budget from highest to lowest.

```
def directors_movies_budget_query(self, total_budget, companies_number):

    query = f"""
    SELECT t.person_ID, t.first_name AS director_first_name, t.last_name AS director_last_name,
        t.picture_URL AS director_picture, t.movie_ID, t.title, t.num_of_companies, t.budget, t.total_budget,
        t.poster_URL AS movie_poster, t.movie_index, t.max_index
    FROM (
        SELECT person_ID, first_name, last_name, title, num_of_companies, budget, picture_URL, poster_URL,
        movie_ID,
        SUM(budget) OVER w total_budget,
        ROW_NUMBER() OVER(PARTITION BY person_ID, first_name, last_name, picture_URL ORDER BY budget DESC)
            movie_index,
        COUNT(*) OVER w max_index
        FROM Movie_numOfCompanies_Director mcd
        WHERE mcd.num_of_companies >={companies_number}
        WINDOW w AS (PARTITION BY person_ID, first_name, last_name, picture_URL)
      ) t
    WHERE t.total_budget>={total_budget}
    ORDER BY director_first_name, director_last_name, budget DESC
    """
```

```
# VIEW FOR directors_movies_budget
def directors_movies_budget_view(self):

    query ="""
    CREATE OR REPLACE VIEW Movie_numOfCompanies_Director AS
        SELECT m.movie_ID, m.title, rm.num_of_companies, m.budget, mc.person_ID, p.first_name, p.last_name,
            p.picture_URL, m.poster_URL
        FROM Movies m, Person p, Movies_Crew mc, (
            SELECT m.movie_ID, COUNT(*) as num_of_companies
            FROM Movies m, Movie_Companies mc
            WHERE m.movie_ID = mc.movie_ID
            GROUP BY m.movie_ID
          ) rm
        WHERE mc.role ="Director" AND mc.movie_ID = m.movie_ID
        AND m.movie_ID=rm.movie_ID AND mc.person_ID = p.person_ID
    """

    self.execute_sql(query)
```

We optimized this query by building the “Movie_NumOfCompanies_Director” VIEW and using it for the query. This VIEW joins the relevant tables for the query and calculating the number of production companies for each movie. Another optimization was in the query itself – we used 2 window functions over the same window, so we defined the same window for both of them.

Query 5

Given budget (X) and number of movie awards (Y) the query returns country-movie couples which “movie” was filmed in “country”, its budget was X or above and won at least Y awards. For each country the query returns at most 15 movies. The movies are selected by the number of awards they won and their budget, using the ROW_NUMBER() window function.

```
def countries_movies_query(self, movie_budget, movie_awards):

    query = f"""
    SELECT cmn.country, cmn.movie_ID, cmn.title, cmn.budget, cmn.awards, cmn.poster_URL AS movie_poster
    FROM(
        SELECT pc.country, m.title, m.budget, m.awards, m.poster_URL, m.movie_ID,
            ROW_NUMBER() OVER(Partition BY pc.country
            ORDER BY m.awards DESC, m.budget DESC) AS ranked_budget
        FROM Production_Countries pc, Movie_Countries mc, Movies m
        WHERE pc.prod_country_ID = mc.prod_country_ID AND mc.movie_ID = m.movie_ID
                    AND m.budget >= {movie_budget} AND m.awards >= {movie_awards}
        ) cmn
    WHERE cmn.ranked_budget <= 15
    ORDER BY cmn.country, cmn.awards DESC, cmn.budget DESC
    """

    rows = self.execute_sql(query)
    return rows
```

We optimized this query by using indices for columns “movie_ID”, “budget”, ”awards” in the Movies table, “prod_country_ID”, “country” in the Production_Countries and “prod_country_ID” in the Movies_Countries table.

Our DB design supports this query with the table Movie_Countries which connects records from Movies table to records in Production_Countries by movie_ID and prod_country_ID.

Query 6

Given a year (X) the query returns for each actor in how many films he played from year X and on, and the total number of awards those movies won.

```
def actors_movies_awards_query(self, start_year):

    query = f"""
    SELECT p.person_ID, p.first_name, p.last_name, COUNT(*) AS number_of_movies_played ,
        SUM(m.awards) AS total_awards, p.picture_URL
    FROM Person p, Movies_Actors ma, Movies m
    WHERE p.person_ID = ma.person_ID AND ma.movie_ID = m.movie_ID
        AND EXTRACT(YEAR FROM m.released) > {start_year}
    GROUP BY p.person_ID, p.first_name, p.last_name, p.gender
    ORDER BY total_awards DESC, number_of_movies_played DESC
    """

    rows = self.execute_sql(query)
    return rows
```

We optimized this query by using indices for columns “person_ID” in Person table, “person_ID”, “movie_ID” in Movies_Actors table and “movie_ID” in Movies table.
Our DB design supports this query with the table Movies_Actors which connects records from Movies table to records in Person table by movie_ID and person_ID.

Query 7 (full text query)

Given a string built from words w_1,w_2,…,w_n , genres (G), minimum rating (R) and range of years (Y1-Y2) the query returns movies that w_1,…,w_n appear in their title, have a rating of R or above, were filmed in the years Y1-Y2 and categorized under at least one genre in G.
If Search Type is "Contains" (chosen in UI) then w_1,…,w_n each is treated as a sub-string and the query will return movies such that w_1,…,w_n are pre-fixes in words that appear in the movies title

```
def movies_with_string_in_name_query(self, string_to_search, movie_score, user_genres, start_year,
                                     end_year, sub_string):
    string_to_search_arr = string_to_search.split(" ")
    if len(string_to_search_arr) == 1:
        string_to_search = "+" + string_to_search
        if sub_string:
            string_to_search = string_to_search + "*"
    else:
        for i,string in enumerate(string_to_search_arr):
            string_to_search_arr[i] = "+" + string_to_search_arr[i]
            if sub_string:
                string_to_search_arr[i] = string_to_search_arr[i] + "*"
            # add space after each word
        string_to_search = " ".join(string_to_search_arr)
```

```
user_genres_string = self.parse_genres("g", user_genres)

query = f"""
SELECT m.movie_ID, m.title, GROUP_CONCAT(g.genre),
    (ms.rotten_tomatoes +  ms.metacritic + ms.imdb)/3 AS popularity, m.poster_URL
FROM Movies m, Movie_Score ms, Movie_Genres mg, Genres g
WHERE Match(m.title) AGAINST("{string_to_search}" IN BOOLEAN MODE)
      AND m.movie_ID = ms.movie_ID
      AND (ms.rotten_tomatoes +  ms.metacritic + ms.imdb)/3 >= {movie_score}
      AND m.movie_ID = mg.movie_ID AND mg.genre_ID = g.genre_ID AND ({user_genres_string})
      AND EXTRACT(YEAR FROM m.released) BETWEEN {start_year} AND {end_year}
GROUP BY m.movie_ID, m.title, popularity, m.poster_URL
ORDER BY popularity DESC
"""
rows = self.execute_sql(query)
return rows
```

We optimized this query by using a FULL TEXT index for column “title” in Movies table and indices for columns “movie_ID” in Movies, Movie_Score, Movie_Genres tables, “genre_ID” in Movie_genres and Genres tables, and “released” in Movies table.

Our DB design supports this query with the table Movies_Genres which connects records from Movies table to records in Genres table by movie_ID and genre_ID. Moreover, Movie_Score PK is also a foreign key because it is a PK in Movies. So by that we can get a movie score from Movie_Score using a movie_ID from a record in Movies.

Query 8 (extra full text query)

Given a string built from words w_1,w_2,…,w_n the query returns movies which have actors that w_1,…,w_n appear in their first or last name. For each movie it also returns the number of actors found and their names.

If Search Type is "Contains" (chosen in UI) then w_1,…,w_n each is treated as a sub-string and the query will return movies which have actors that w_1,…,w_n appear as a prefix in their first or last name.

```
def movies_actors_with_string_in_name_query(self, string_to_search, sub_string):

    string_to_search_arr = string_to_search.split(" ")
    if len(string_to_search_arr) == 1:
        string_to_search = "+" + string_to_search
        if sub_string:
            string_to_search = string_to_search + "*"
    else:
        for i, string in enumerate(string_to_search_arr):
            string_to_search_arr[i] = "+" + string_to_search_arr[i]
            if sub_string:
                string_to_search_arr[i] = string_to_search_arr[i] + "*"
        string_to_search = " ".join(string_to_search_arr)
```

```
query = f"""
SELECT m.movie_ID, m.title, COUNT(*) as num_of_actors,
    GROUP_CONCAT(concat(p.first_name, " "), p.last_name SEPARATOR ", "),
    m.poster_URL
FROM Movies m, Movies_Actors ma, Person p
WHERE m.movie_ID = ma.movie_ID AND p.person_ID = ma.person_ID
    AND (Match(p.first_name, p.last_name)
    AGAINST("{string_to_search}" IN BOOLEAN MODE))
GROUP BY m.movie_ID, m.title
ORDER BY num_of_actors DESC
"""

rows = self.execute_sql(query)
return rows
```

We optimized this query by using a FULL TEXT index for columns “first_name”, “last_name” (in this order) in Person table and indices for columns “movie_ID” in Movies, Movie_Actors and “person_ID” in Movie_Actors and Person tables.

Our DB design supports this query with the table Movies_Actors which connects records from Movies table to records in Person table by movie_ID and person_ID.

General Notes:

- We used VIEWS to optimize complex queries that join several big tables and that will be, in our opinion, in common use. For queries that join less tables we preferred to use indexes.
- Regarding full text queries - when searching for words with length < 4 it is not guaranteed to find any matches even if there are in the DB. As written in mysql documentation "ft_min_word_len" is the variable for the minimum length of the word to be included in FULL TEXT index.

- ft_min_word_len

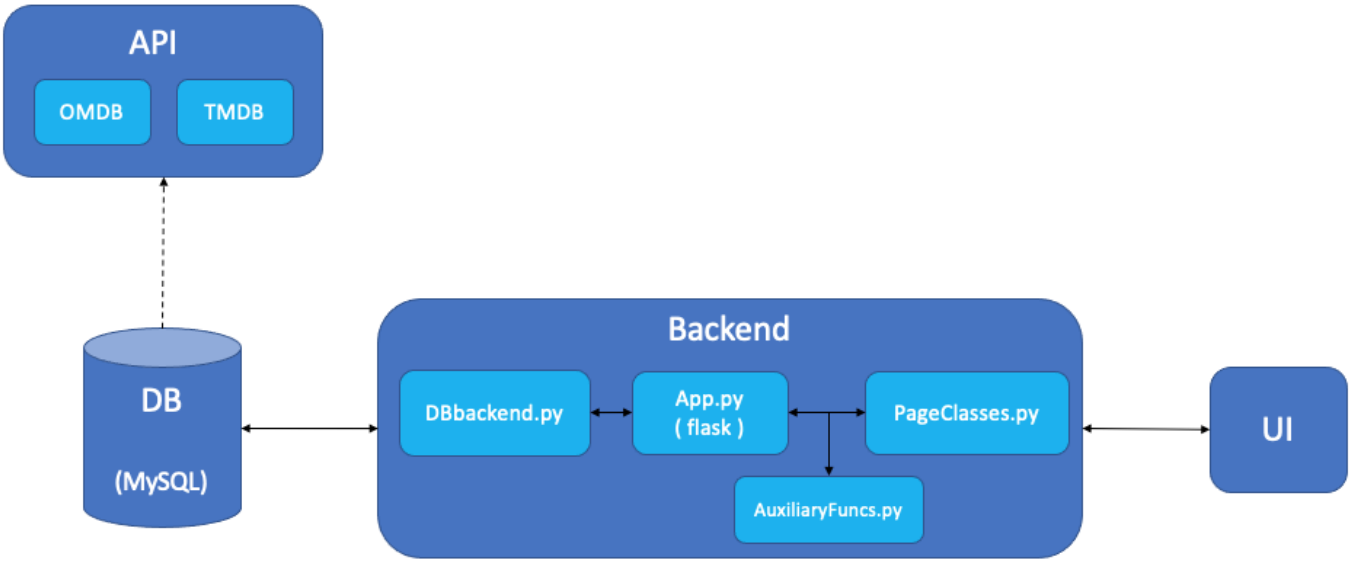
Command-Line Format	--ft-min-word-len=#
System Variable	ft_min_word_len
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	4
Minimum Value	1

https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_ft_min_word_len

Code Structure

Git repository

1. Client - “templates” and “static” folders. All HTML, JavaScript and CSS files are for the app’s client.
2. Server - app.py, PageClasses.py, AuxiliaryFuncs.py and DBbackend.py files. The server connects between requests coming from the client and the desired data from the DB - the client gets information from the server using Flask’s render_template() method and then further communicates with the server with http requests – all can be found in app.py Each page of the UI has an entity in PageClasses.py. In order to generate those page entities, we use AuxiliaryFuncs.py, which generate them from the data in the DB. While DBbackend.py is the single interface of the database: including connection configuration, query results processing, error handling and some SQL related functions used for the data population.



API

1. TMDB API - API of The Movie Database, a collaborative database about movie. We used this API by a python package ‘tmdbv3api’ to retrieve a big number of popular movies. It provides us most of the data required in order to fill the tables.
2. OMDB API – API of Open Media Database, also an online collaborative database about movies. We retrieved its data by a python package ‘omdb’.

For each movie retrieved from the previous API, we used this API in order to fill the Score table, the awards attribute of Movies table and also to fill NULL attributes of TMDB API.

External Libraries

1. Flask
2. mysql.connector
3. Html
4. Css
5. Js
6. jQuery
7. tmdbv3api
8. omdb

General Flow