Software Documentation

DB Scheme Structure

1. **titles** (film_id, title)
2. **people** (person_id, name)
3. **actors** (film_id, actor_id)

   Foreign keys:

   - film_id → titles:film_id
   - actor_id → people:person_id
4. **directors** (film_id, director_id)

   Foreign keys:

   - film_id → titles:film_id
   - director_id → people:person_id
5. **writers** (film_id, writer_id)

   Foreign keys:

   - film_id → titles:film_id
   - writer_id → people:person_id
6. **genres** (film_id, genre)

   Foreign keys:

   - Film_id → titles:film_id
7. **runtime** (film_id, lengthInMinutes)

   Foreign keys:

   - Film_id → titles:film_id
8. **plot** (film_id, plot)

   Foreign keys:

   - Film_id → titles:film_id
9. **imdb** (film_id, score)

   Foreign keys:

   - Film_id → titles:film_id
10. **wiki** (film_id, url)

    Foreign keys:

    - Film_id → titles:film_id
11. **boxOffice** (film_id, revenue)

    Foreign keys:

    - Film_id → titles:film_id

In designing our database, we decided to utilize the IMDb ID returned by the OMDb API, because it's a single value that uniquely identifies each movie. The format of the actual IMDd ID is 2 t's followed by 7 digits (e.g., tt0138749). For simplicity, we converted the field to be strictly numeric by stripping the 'tt'. This became the primary key in our 'titles' table and was used as a foreign key in nearly every other table to connect attributes of a given film.

Another primary key is person_id. We recognized that there is an overlap between directors, actors and writers – some actors also direct for example. Consequently, we constructed a people table so that the names could be stored only once. The primary key, person_id, is used as a foreign key in the actors, directors, and writers tables to specify the specific role held by a person for a particular film. For instance, Leonardo Decaprio, whose person_id = 6831, was an actor in 'Blood Diamond' but a writer for "The 11th Hour."

Because someone could act in, direct, or write for multiple films, and a single film could have multiple writers, actors or directors, the primary key in the actors, directors, and writers tables is the combination of their 2 foreign key attributes (the film_id with the actor_id, director_id, or writer_id respectively).

DB Optimizations

To make our tables as light as possible we set the attribute types to be the smallest possible that fits our data. We could do this because the application should not rely on continued usage as required. Before creating the tables we ran over each feature and checked what the heaviest value is. For instance, what's the longest movie title or the highest box office revenue. This is how we determined a VARCHAR() tight upper bound for strings, and which type of UNSIGNED INT or DECIMAL is required for numeric attributes (such as TINYINT, SMALLINT, MEDIUMINT and DECIMAL(3,1) we used).

Aside from constricting the attribute types for efficiency, we also ensured that our data decomposition to smaller tables would be lossless. Our use of two attributes in each relation ensures BCNF. As the application should not rely on continued usage, using BCNF decomposition obtains that our set of relations is in a normal form with no anomalies.

Moreover, the design of tables helps optimize our queries. First, our tables' size is small as possible thanks to NOT NULL and NO Duplicates policies with use of thin efficient-as-possible data types. Second, each of our tables was defined with suitable primary key that generates B tree indexes automatically. Third, the design decision to use film_id and person_id as foreign keys makes the lookup for movies and professionals faster than a name-string-matching approach. All optimizations described above, help to make our application's queries much faster thanks to lowering the cost of searching in tables and performing joins between them.

Lastly, we used full text index on the plot attribute to enable efficient querying of movie themes. For instance, if someone wishes to get information on movies about war we can efficiently search for this keyword in the plots of movies in our database to find relevant films.

Description of 7 main queries

1. For each genre, who is the actor/s that appeared in the most number of movies of this genre and how many did they appear in?
    a. If a user is interested in producing a film of a certain genre, or multiple films of different genres, this query informs him which actors have a lot of experience and would be good to cast.
    b. This query was optimized by running 2 subqueries to get the max number of films per genre by actor_id, rather than actor name which would necessitate thousands of string comparisons. Once this aggregate data is known we retrieve the actor name via the actor_id.
    c. The actor_id sits in a table, actors, along with film_id, which is a foreign key shared by the genre table. This is how we determine how many films of a certain genre an actor appeared in. And then since actor_id is a foreign key of the people table, we can retrieve the actor's name from people.name.

2. Return a list of actors who appeared in more than one of the X highest revenue films (where X is an integer provided by the user, for example X=100).
    a. If a user wants to produce a movie that will be a box office success, they may want to consider casting these actors who have been in multiple movies that generated a lot of revenue.
    b. The query was optimized by running a subquery to first select the most profitable films, and then going over the actors by actor_id, to see how many they appeared in. Only once we isolate the actors that have appeared in more than one of the most profitable films by actor_id, we get the actor names. This is more efficient than using actor name throughout the entire query.
    c. The database design supports this query because film_id is a foreign key to both the actors table and boxOffice table. Thus, we are able to use film_id to connect an actor_id to the X highest revenue movies – a subset of boxOffice. Then since actor_id is a foreign key to people.person_id, we are able to get the actors names from people.name.

3. Return a list of directors and the actors they have worked with multiple times. The results are ordered alphabetically by director and then in descending order of the actors they worked with most.
    a. If a user wants to work with particular directors, they should also consider hiring actors with whom those directors already have a rapport. This query enables the user to see which pairs of actor-director feel comfortable working together – which will likely contribute to the film's success. Also, knowing that someone they like working with is also going to be a part of the film, the director or actor may be more likely to agree to the project in the first place.
    b. The query was optimized by running a subquery to isolate pairs of director_id and actor_id that have worked together more than once. Once we have this subset of pairs, we get the director and actor names, which is more efficient than from the start grouping by director name and actor name, and then returning only pairs that occur more than once.
    c. Since film_id is a foreign key in both the directors table and actors table, we can see which directors.director_id and actors.actor_id have worked together on the same film. Then since director_id and actor_id are both foreign keys of people.people_id, we can get their names from people.name.

4. How many films directed by X got an IMDb score above Y (where X is a director name and Y is a number input by the user)
    a. If a user is considering a number of directors to work with and wants their movie to be liked, they should hire a director who has many highly liked movies.
    b. The aggregate function used in this query has very little data to work with since before it is called, the WHERE clause isolates the small subset of data that interests the user.
    c. Since people.person_id is a foreign key of directors.director_id, and film_id is a foreign key in directors, the films directed by director X can be easily found via a single join. Moreover, since film_id is also a foreign key of the imdb table, via a second on film_id we can identify the relevant movies by the directors (those which received imdb.score >= Y). We ultimately group by director_id.

5. Return Wikipedia pages of films directed by the director whose movies have generated the highest total box office revenue. Display them in decreasing order of box office revenue.
    a. This query is useful for people who want to make a movie that will do great at the box office. They may want to hire the director whose movies have been the most successful in this sense, or they may just want to develop a film that is similar to what this director has done. In either case, it would be useful to read up on the director's prior work.
    b. The query has a subquery to identify the director_id whose films have the highest total box office revenue. All other joins are performed in accordance with this subset of data and the actual name of the director is only used at the very end of the query, not throughout which would be less efficient.
    c. Because film_id is a foreign key in both the directors table and boxOffice table, we are able to identify, thanks to this key, the total box office revenue of movies directed by director_id. Then the director_id's are ordered from highest total box office revenue to least and only the first result is selected. The director_id selected by this subquery is used to get the relevant Wikipedia pages. film_id is also a foreign key of the wiki table, so by joining directors, wiki, boxOffice and title on film_id we can get the titles, box office revenue and relevant Wikipedia URL of movies made by that director_id. Lastly, since director_id is a foreign key of people, we can get the director's name from people.name.

6. What is the average length in minutes of genre X movies who are in the top 10th percentile in terms of box office revenue (where X is a genre input by the user, such as 'Action' or 'Comedy')?
    a. If a user is making a certain genre of movie, they can look up what, on average, was the length of the highest earning movies of this genre. They should aim to make their movie around this length.
    b. This query made use of a sub-query to first identify the relevant movies, and then only applied aggregate functions to the subset of data.
    c. PERCENT_RANK() was applied over boxOffice.revenue to determine the percentile each movie belongs to. Only movies of the relevant genre were given a rank, and these movies were found thanks to film_id, which is a foreign key for both boxOffice and the genres table.

7. What is the average revenue of movies that deal with X (where X is a keyword input by the user, for instance 'war')?
    a. This query enables the user to check whether a theme they are considering is profitable on average or not.
    b. This query was optimized via full text index on the plot attribute.
    c. The full text query identifies plots of movies that deal with the specified theme – movies for which X appears in plot.plot. Then via the foreign key plot.film_id, which is also the key of boxOffice, we can obtain the revenue of each film and return the average.

Code Structure

The code is distributed to 5 python files used for data preparation, data insertion to the DB and for querying the DB.

Data Preparation

1. kaggle_21st_century_movies.py - uses data from the Wikipedia Movie Plots dataset and filters it to movies that have been released from the year 2000. In our DB, the Wikipedia page for each movie was taken from this dataset.
2. OMDB_API.py - most of our data came from using the OMDB API. This code creates a csv file from the API responses as follows: it takes from the filtered Kaggle dataset the title and release year of each movie (which are both creates a movie unique key in our case), and requests more attributes about this movie from the API. At the end of the process, we get a csv file containing all the data we wanted for each movie.
3. create_people_list.py - this code takes all the people names from the fully combined csv file (directors, producers, and actors) and creates a new csv containing all people names and their person_id. Using the person_id (a small int data type) allows us to optimize queries containing people.

Data Insertion

4. insert_into_tables.py - This code assumes that all the relevant tables have been already created (in our case we made SQL queries to create them in advance) and insert the relevant data to each table.

Querying the DB

5. app_demo.py - This code contains our data queries and demonstaint a flow of usage for a client wanting to get insights from past movies data while creating her new movie.

Description of the API and how it was used

We used the **OMDb API**, a RESTful web service for obtaining data on any movie from IMDb. The API can be called either by IMDb ID or movie title, along with 0 or more optional parameters, such as year of release or type of result to return (movie, series, episode).

The other source of our data is a csv file that we downloaded from kaggle (wiki_movie_plots_dedupped.csv). This dataset contains descriptions (release year, title, origin, director, cast, genre, wikipedia page link, and plot) for nearly 35,000 movies dating back to 1901. We decided to only take data for movies released in 2000 or later, since they are the most relevant to our application. We also decided that we'd take attributes common to both data sources (provided both by the API and the csv) from the API. Thus, we were left with a subset of this Kaggle csv, containing just the (1) title, (2) release year, and (3) Wikipedia page for nearly 6,800 movies released in the year 2000 or later (generated by **kaggle_21st_century_movies.py**). We used this csv to call the API.

Specifically, we were interested in using the API to get additional data on each movie in the csv – such as IMDb id, runtime, box office revenue, writers, etc. From the csv we took the title along with the release year to call the API because we understood that the title alone might not be unique. Since the API has a daily limit of 1000 calls per day and we had nearly 6,800 movies in our csv, we needed to generate 7 API keys. Then we wrote a python script that ran over the csv and called the API, switching to a new key once the limit was reached on the old one (**OMDB_API.py**).

The returned data, along with the Wikipedia URL from the csv data source, were written to an output file. In parallel we also performed some pre-processing on the data. For instance, the API returns runtime as a string like "153 min" so we reformatted and wrote to the output file simply "153." Another example is box office revenue: the API returns a string like "$15,000,000" and we simplified the result to "15000000."

Occasionally there were pairs of 'movie title' + 'release year' for which the API didn't return a result. Thus, the generated file (**movies_data_combined.csv**) from which we built our database contains data for 4408 unique movies released in the 21st century.

General flow of the application

Our app is cross-platform, so users are free to choose the medium that's most convenient and comfortable for them. Regardless of platform – mobile or desktop – the user experience is the same. Users receive answers to their movie-related questions by typing a question in our search bar or utilizing any one of our 3 modules.

Each module leads users to a new window, where they can view decisions they have made before while building the movie, and get relevant information to make future decisions. For example, in the Crew Builder module, the system will offer actors for roles based on the genre of the film and its plot as well as according to the director and other staff members with whom the actor has previously worked successfully. Users can play with the decisions and see what impact they will have on the film and recommendations for future decisions.

Working within a module means that the user makes decisions and saves them in order to progress in creating the film. Working with the search bar is more exploratory and expands the knowledge, sometimes even regardless of the specific movie the user is currently working on.

Beneath each of the modules on the main page there are suggested searches. For instance, "best length for a comedy movie" is recommended for the "statistics" module. These recommendations are updated regularly to ensure that our users are prompted to ask what are likely to be the most relevant questions. New users of the app will be shown the most popular searches made in each module. Old users on the other hand will be shown questions that are tailored to them based on their search histories. Of course, users aren't required to search what we tell them; they are encouraged to get creative with the search bar.

In any case, the question text is converted to a query for our database by a free natural text engine. Thanks to the database design and optimizations explained above, results are returned quickly and displayed to the user in a separate window. A link to the results page can be saved for future reference or be an input to another application. The results can also be shared by the user with his or her colleagues. Our app conveniently has buttons that link to WhatsApp, Messenger, Slack, and email because we understand that movie making is collaborative.