Software Documentation

Database schema structure

Our database consists of the following tables and keys:

1. Movies (id, overview (FI), release_date, tagline, title, vote_avg, vote_count)
2. Movie_keywords(movie_id (FK to Movies.id), keyword_id (FK to Keywords.id))
3. Keywords (id , name)
4. Genres_movies(genre_id (FK to Genres.id), movie_id (FK to Movies.id))
5. Genres(id, name)
6. Actor_movies(actor_id (FK to Person.id), movie_id (FK to Movies.id))
7. MovieCrew(movie_id(FK to Movies.id), crew_id(FK to Person.id), job)
8. Person(id, name, gender)

Database design justification

We started with two csv files, one that had lots of details about each movie (including the details we kept in the Movies table), and another that showed the cast and crew for each movie.

One such attribute was a list of keywords for each movie. To support this many-to-many relationship, we had to create a table that holds movie-keyword tuple, and so to save on space (4 bytes for each integer vs up to 51 bytes per keyword) we decided to use the keyword id integer instead of the keyword itself and store the keyword-id tuples in a separate table.

We used a similar justification for creating the Genres (each movie had a list of genres) and Genre_movies tables.

Finally, we saw an is-a relationship with the cast and crew. They all had unique id numbers (even when you take the union of the two groups), gender value and name. Because of the many-to-many relationship that existed between movies, cast members, and crew members, in addition for our need to determine between the two groups, we created MovieCrew and Actor_movies lists.

Each crew member was associated with a distinct job. We decided to store the job name directly with each line instead of using a job id number, because such information was not made available by our data source. We decided against creating our own id system due to the huge amount of processing power we would have to use prior to inserting the data into the database. In a future version of our database, we may consider taking this on so that our database would use less storage and store the job names in a separate table like Genre and Keywords.

We decided against creating a pure is-a relationship between person and crew and cast, because our queries that accessed crew and cast info always needed info on which movie they appeared in. So, to create more efficient queries, we kept this information together as

opposed to creating Crew and Cast tables, as well as movie_crew and movie_cast tables and performing a union between the two when querying.

Database optimizations and index usage

An optimization we did is using the is-a relationship between person and MovieCrew and Actor_movie so gender and name info isn't stored more than necessary.

In Movies, we have the following indexes: on (tagline, overview, keyword), (overview), (release_date), (vote_avg), (vote_count) and the primary key id. The first two indexes are full text indexes, which help us optimize some queries we did.

In Actor_movies we have a primary key on actor id and movie id together since there is a many-to-many relationship there, and so we also created an index on actor_id and movie_id so that we could optimize a query that joins on these fields.

In a similar fashion, our primary key for Genres_movies are on movie id and genre id together due to their many-to-many relationship. To optimize queries on joins genres, and joins on movies, we created an index for each column. We also created an index on keyword_id in Movie_keywords for the same reason.

In MovieCrew, our primary key is on the tuple movie_id and crew_id, and we created indexes on crew_id and on the crew_id, job tuple. The index on crew_id helps us with group by queries on different crew members, and the job index helps us filter by specific jobs.

The queries

1. **Actor besties**: takes 2 input genders as well as year, returns two actor names with the according genders, and count of movies they appeared in together that year. We check the input values against a gender dictionary and year range so that the query will work. The query joins two instances of the person and actor_movies table so that we can compare two actors. The index on movie_id helps optimize the join between Movies and Actor_movies. We then filter by year and gender, which is optimized by the index on release_date. By using a range comparison with the full release date instead of using the YEAR function on the release date, we take advantage of the index we created.

2. **Suspicious Movie Recommendations**: given a genre as input, we return five movies with the highest rating for that genre, where the vote count is less than average for that genre. We create two subqueries and then a main query. The first one gets the average vote count for the input genre. We filter by genre name as soon as possible to avoid multiple calculations for multiple genres. The next subquery selects all movies within the genre that also have a vote count less than the average vote count. The WHERE in this subquery is optimized by the index we created on vote count. The final query simply sorts and limits what we returned from the second query.

3. **These Crew Members Ruined the Movie**: given a job title input, we return the names of 5 crew members and the average rating of the movies that they worked on in that job. We join the Person, MovieCrew, and Movies tables. Indexes on person.id, MovieCrew.crew_id, MovieCrew.movie_id and PK on movies.id help optimize these

joins. We then filter on job name (job index helps here) and made sure the movies they worked on had a higher-than-average vote count for quality control (index on vote count optimizes this). Finally we group by MovieCrew.crew_id so we can get the average movie rating for each crew member, optimized by the movies.vote_avg index.

4. **Movie Lookup**: takes a string of keywords as an input, returns ten movies' title, release year, and rating. We only pull info from the Movies table, and filter using the keywords on the overview, tagline, and title fields. The full text index we created on these three fields together helps optimize the search. We also filter for release dates that are not null (helped by index on the release date) and then sort descending on release year.
5. **Movie Keywords**: Given an input of genre or movie title, we return similar movies to the genre or title. If given a genre, we first use a separate query to fetch one random movie from the genre. Our user inputs the genre name so we first fetch its genre id from the genres table. We then feed the id into a new query that fetches the title of one random movie. We do this separately to avoid having to match on genre name in the same query we fetch a random movie name, since there is no index on it and so that would take an unnecessary amount of time.

   Now we can run the 3rd query which runs no matter the input: we filter on movie name, join the movies, movie_keywords, and keywords tables, and return the keywords associated with the movie. Finally, we turn the input into a keyword string that we feed to the Movie Lookup query (query 4).

A note on data safety: we included checks in each function to prevent SQL injections. The first 2 queries can only take inputs from a dropdown menu defined in our UI, and the rest must not have non-English characters in them.

Code structure, API usage, and the general flow of the application.

We created the database using 2 well-known CSV files from TMDB on Kaggle.

We first created the tables in create_db_script using mysql.connector, then added/ removed indexes and tables as we played around with our queries. For this reason, the file only shows the scripts we ran to create tables but not the indexes, as we added and erased them.

We then pulled relevant data from the CSVs in api_data_retrieve. We created separate functions for each table, except for the Person, MovieCrew,and Actor_movie tables. We filled out those tables together to take advantage of the similar info we needed for those tables and cut back on runtime. Our functions are:

```
movies_table(movie_reader, mycursor, mydb)
Keywords_table(movie_reader, mycursor, mydb)
Movie_keywords_table(movie_reader, mycursor, mydb)
genres_table(movie_reader, mycursor, mydb)
Genres_movies_table(movie_reader, mycursor, mydb)
Person_Cast_MovieCrew_MoviesActors_tables(crew_reader, mycursor, mydb)
```

Each function takes the cursor that we created with mysql.connector, as well as the relevant CSV readers. In the function movies_table, we had to reformat the dates in the CSV so that

it would match what SQL needs. We ran all the functions in the Main function, where we also created the cursor and CSV readers.

All our queries are in queries_db_script, and we then import this file in queries_execution. Here, we showed many examples of both legal and illegal queries.