Group 19

Team Name: Free-Range-Chickens

Team Members: Julia Han, Mike Nguyen, Sean Harmon, Kaiming Zhang

Selected Domain: Movie

Language: Python

Interface: Command line

A command line based application that stores information about movies which includes information such as release date, genre, main actors, directors, and rating. The application will be written in python, and we will use either PyQt5, Tkinter, or Kivy for our text-based UI. It is currently decided Julia will be making the poster and Kaiming(+ someone else undecided) will be doing the presentation towards the end of the project.

Movie was created as an entity with movie_id as its primary key. Users are given the ability to edit collections which contain movies by adding and deleting movies or creating collections. Collections was created as a strong entity with its primary key as its collection_id. **Actor and director were changed in this phase to represent specialized entities of a person entity. The person entity has a primary key that represents the person_id and also contains attributes defining the name of the person, further broken down into first and last. This was decided because an actor can act in multiple movies and a director can also be an actor. Genre was also changed into an entity in this phase and was given a genre_id as a primary key as well as a type attribute that describes the genre such as horror or romance. The length attribute of a movie was further broken down into hours and minutes for easier storage of data.** A user uses user_id as its primary key and includes a name which has both first_name and last_name as a part of it. Each collection has a name and primary key of collection_id. The name attribute of the collection is associated with a user_id. It is a 1 to N relationship. Contains produces a separate table because it is an M..N relationship, and takes in the key attributes of Movie and Collections. Watches produces a separate table because it is an M..N relationship, and takes in the key attributes of User and Movie. Watches also contains the user's rating and a flag for whether the user has watched the movie or not.

The reduction to tables was changed in this phase as the nested values have been eliminated to properly adhere to the format.

Sample SQL statements for the creation of tables:

-
  ```
  CREATE TABLE users(
          username text NOT NULL,
          user_id int NOT NULL PRIMARY KEY,
          password text NOT NULL,
          firstname text NOT NULL,
          lastname text NOT NULL,
          access_date date NOT NULL,
          email text NOT NULL,
          CONSTRAINT UC1 UNIQUE(email),
          creation_date date NOT NULL,
          access_time time NOT NULL,
          creation_time time NOT NULL
  );
  ```

-
  ```
  CREATE TABLE genres(
          genre text NOT NULL,
          genre_id int NOT NULL PRIMARY KEY
  );
  ```

-
  ```
  CREATE TABLE collections(
          collection_id int NOT NULL PRIMARY KEY,
          name text NOT NULL
  );
  ```

-
  ```
  CREATE TABLE collectionuser(
          collection_id int NOT NULL,
          user_id int NOT NULL,
  ```

```
        FOREIGN KEY (collection_id) REFERENCES collections (collection_id) ON
        DELETE CASCADE ON UPDATE CASCADE,
        FOREIGN KEY (user_id) REFERENCES users (user_id) ON DELETE
        CASCADE ON UPDATE CASCADE
    );
```

Populating data:

- sql = "INSERT INTO users (username, user_id, password, firstname, lastname, access_date, email, creation_date, access_time, creation_time) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s);"
  curs.execute("SELECT COUNT(username) FROM users")
- sql = "INSERT INTO collections(collection_id, name) VALUES (%s, %s);"
  curs.execute("SELECT COUNT(collection_id) FROM collections")
  collection_ID = curs.fetchone()[0]
  curs.execute(sql, (collection_ID, user_input[1]))
- sql2 = "INSERT INTO collectionuser(collection_id, user_id) VALUES (%s, %s);"
  curs.execute(sql2, (collection_ID, user_Id))
- sql3 = "INSERT INTO collectionmovie(collection_id, movie_id) VALUES (%s, NULL);"
  curs.execute(sql3, (collection_ID,))
- sql = "INSERT INTO followerfollowee (follower_UID, followee_UID) VALUES (%s, (SELECT user_id FROM users WHERE email = %s));"
  curs.execute(sql, (user_Id, user_input[1]))
- curs.execute("INSERT INTO movieuser(user_rating, flag, movie_id, user_id) VALUES (%s, %s, (SELECT movie_id FROM movies WHERE title = %s), %s)",(None, True, user_input[1], user_Id))

The data for movies, actors, directors, and genres were loaded into the database through turning a google sheet with the data into a csv file then importing the data from the file to the already created tables with defined data types and keys.

Insert statements we could have used instead for large amounts of data

INSERT INTO movies (title, hours,minutes,movie_id,studio,release_date,mpaa) VALUES ('Batman',2,30,123,'DC Universe','1999-01-01','PG-13'), ('Charlie Brown Christmas Special',1,15,345,'Disney','2000-01-01','NC-17'), ('Cowboy', 1, 26, 1, 'Muller-Windler', '1981-01-04', 'NR')

INSERT INTO genres (genre, genre_id) VALUES ('Action', 1), ('Animated', 2), ('Children', 3)
INSERT INTO actors (actor, actor_id) VALUES ('Ade Boat', 1), ('Adria Mudle', 2), ('Agnesse Winders', 3)
INSERT INTO actormovie(movie_id, person_id) VALUES (1, 143), (1, 282), (2, 80)
INSERT INTO directors(director, director_id) VALUES ('Amandi Liddy', 1), ('Lynda Savins', 2), ('Georgena Edgecombe', 3)
INSERT INTO directormovie(movie_id, person_id) VALUES (1, 497), (2, 498), (3, 499)

The data was analyzed through excel. The data from the already generated movies were retrieved from DataGrip as a csv file and imported into excel. The data was then analyzed alongside randomly generated data and put into a graph to visualize the relationships between the data analyses we chose to perform. The user ratings we utilized were randomly generated using the RAND function in excel. Primary keys for movie, user, genre, and collections were created as indexes to improve program's performance. The primary keys are labeled as the id for each of these entities. These indexes allowed our program to identify between each movie, genre, user, and collection so that if something were named the same thing, we would still be able to differentiate between them.

Appendix for all SQL statements used in this phase:

User profile SQL statements:

- sql_get_col = "SELECT collection_id FROM collectionuser WHERE user_id = %s;"
  - Gets all collections from user
- sql_get_followers = "SELECT follower_uid FROM following WHERE followee_uid = %s;"
  - Gets all followers for a user
- sql_get_following = "SELECT followee_uid FROM following WHERE follower_uid = %s;"
  - Gets all following for a user
- sql_get_movies = "SELECT m.title FROM movieuser mu INNER JOIN movies m ON mu.movie_id = m.movie_id WHERE user_id = %s ORDER BY user_rating DESC;"
  - Gets all watched movies for a user

User recommendation SQL statements:

- "SELECT genre_id FROM genremovies gm INNER JOIN movieuser mu ON gm.movie_id = mu.movie_id WHERE mu.user_id = %s GROUP BY gm.genre_id ORDER BY count(*) DESC;"
  - Gets a list of genre id's sorted in descending order from the most popular genre the user has watched to the least popular genre the user has watched.
- "SELECT m.title FROM movies m LEFT OUTER JOIN movieuser mu ON m.movie_id = mu.movie_id INNER JOIN genremovies gm ON m.movie_id = gm.movie_id WHERE (mu.user_id != %s OR mu.user_id IS NULL) AND (gm.genre_id = %s OR gm.genre_id = %s OR gm.genre_id = %s) GROUP BY m.movie_id ORDER BY RANDOM() LIMIT 10;"
  - Gets a random list of movies a user has not watched yet where the genre of the movies are the three most popular genres they have watched in the past.
- SELECT title FROM public.movies t ORDER BY release_date DESC limit 5;
  - Most recent 5 releases
- SELECT m.title FROM movies m INNER JOIN movieuser mu ON m.movie_id = mu.movie_id WHERE mu.date BETWEEN '%s' AND '9999-01-01' GROUP BY mu.movie_id ORDER BY count(*) DESC
  - Gets most watched movies from inputted date to current date