Christa Abou-Arraje 40226631
Nour Hassoun 40233077
GitHub: https://github.com/christa-ux/Assignment2-Soen363

Assignment 2-Soen 363

DDL Queries

Part 1.

- The database uses internal integer primary keys so we can uniquely identify each record and ensure independence from external systems like TMDB/IMDB id. Core entities like Movie, Actor, and Genre, Director, Country, Language, Keyword, and contentRating, are structured with distinct primary keys. We also used constraints like CHECK and UNIQUE to validate data integrity. and constraints to maintain data validity. For example, CHECK constraints ensure realistic age ranges for actors, while UNIQUE prevents duplicate actor names.

- To model relationships, we used foreign keys and join tables. Many-to-many relationships (e.g., between movies and genres) are represented with composite primary keys in join tables. One-to-many relationships (e.g., movies and content ratings) use direct foreign keys in the dependent table. We included and maintained referential integrity through foreign key constraints. We also used cascading deletes to make sure that related data is cleaned up automatically when necessary.

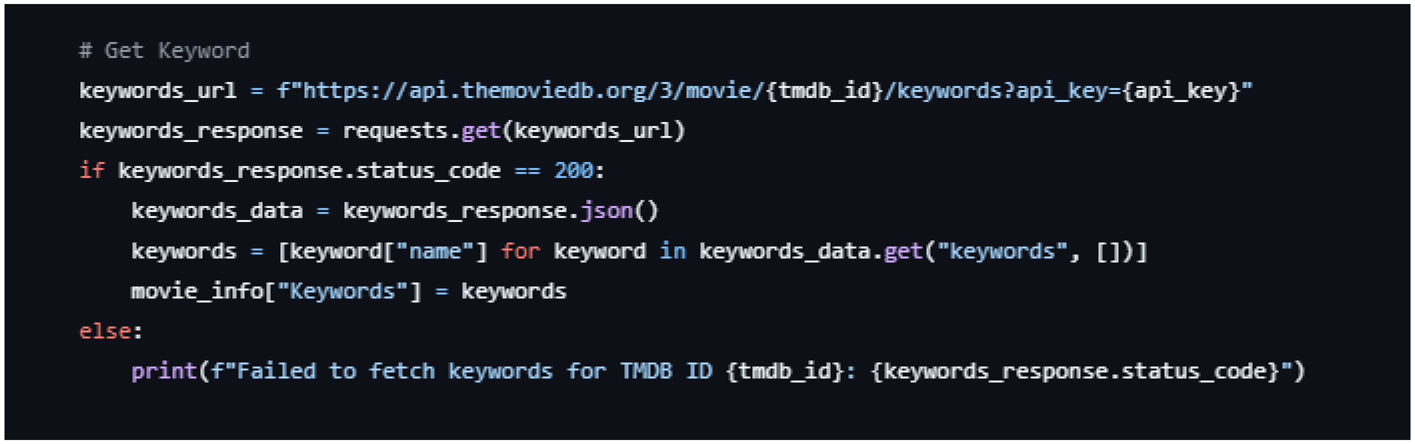
- To standardize and clarify data, we included both ISO country codes as primary keys and full country names in the Country table. Foreign tables for genres, content ratings, and keywords were also created to make the database extensible and easy to manage.

- The database allows for missing IMDB ids by making the *imdbID* field optional while requiring *tmdbID* for unique identification. We decided to do this to make sure that all movies are trackable, even if external data is incomplete.

Data Population

Part 2.

- We designed a script to fetch detailed information about movies from the TMDB (The Movie Database) API. The script performs several GET requests to retrieve specific movie data. That way, we make sure there's a comprehensive dataset for each movie. The main API call retrieves core information about the movie such as its title, release year, plot, genre, viewer rating, original language, country of production, IMDB id, and content rating. A separate API call retrieves the movie's cast and crew. Another API call fetches related keywords. Finally, there's an API call to retrieve the movie's translations and alternate titles. We also have included error handling by making sure each API call the *status_code* to confirm success and logs errors if the call fails.
- To summarize, the code asks for the user to input the TMDb-id of any movie, and it will return all its information (listed above).



```python
# Get Keyword
keywords_url = f"https://api.themoviedb.org/3/movie/{tmdb_id}/keywords?api_key={api_key}"
keywords_response = requests.get(keywords_url)
if keywords_response.status_code == 200:
    keywords_data = keywords_response.json()
    keywords = [keyword["name"] for keyword in keywords_data.get("keywords", [])]
    movie_info["Keywords"] = keywords
else:
    print(f"Failed to fetch keywords for TMDB ID {tmdb_id}: {keywords_response.status_code}")
```

In the above picture we're using GET for keyword, using the appropriate API

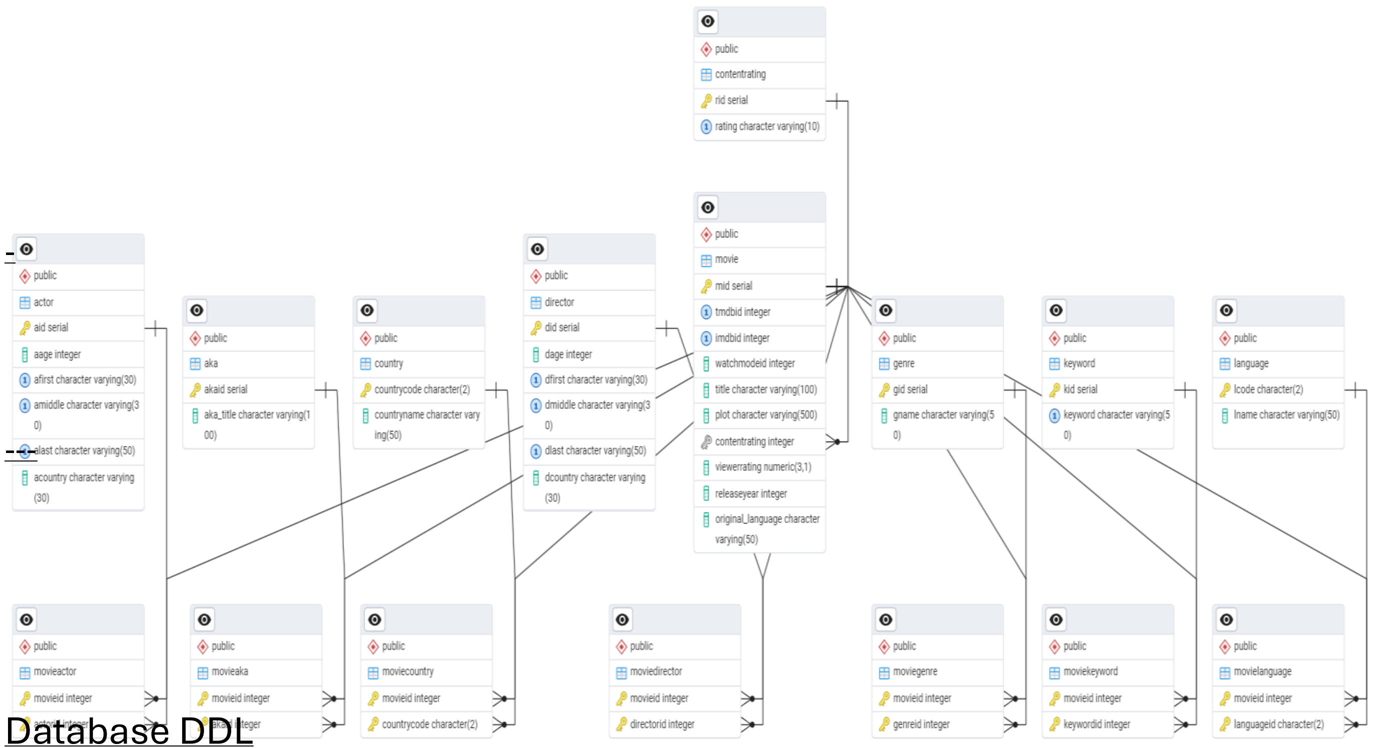
Part 3.

In this part, we created a Python script using the psycopg2 library to allow users to input details about a movie and insert the data into the Movie database table.

- This script first establishes a connection to the database with the required credentials.
- We used a function insert_movie( ) to gather user inputs such as *tmdbI_D, imdb_ID*, *title* and other movie attributes. That way, we can make sure that optional fields like *imdb_ID* and *watchmode_id* handle null values properly.
- The data is inserted into the database using an SQL INSERT statement with placeholders (%s) to prevent SQL injection.

ERD

Part 4.

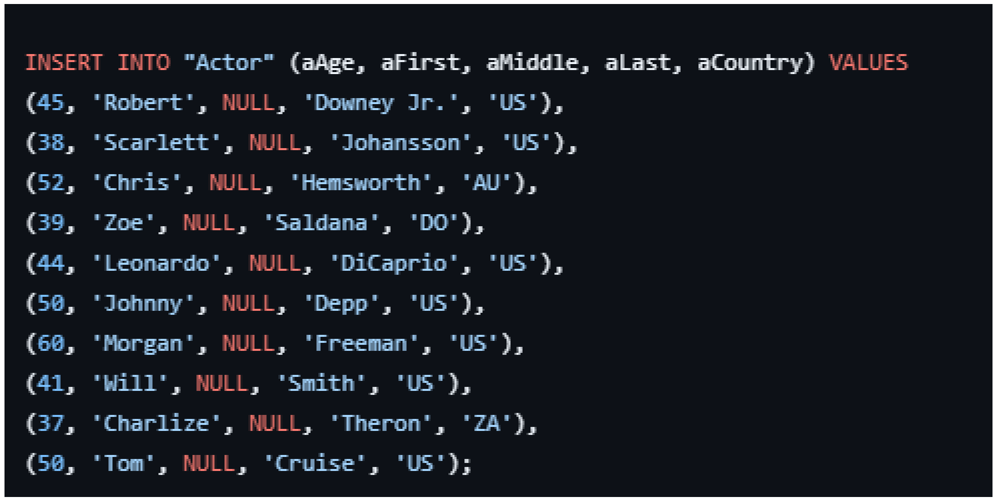


Database DDL

Part 5.

In this part, we structured the INSERT statements to populate the database systematically, ensuring logical consistency and completeness.

- **Genres, Actors, Directors,** and **Other Base Tables:** These were populated with diverse data to represent a wide range of movies. For example, Genre and Actor tables include commonly known genres and actors to make sure we have realistic associations.
- **Relationships and Associations:** The many-to-many relationships, such as movies with genres (Moviegenre) and actors (Movieactor), were defined to showcase the complexity of real-world data. We're showing how a single movie can belong to multiple genres and feature multiple actors.



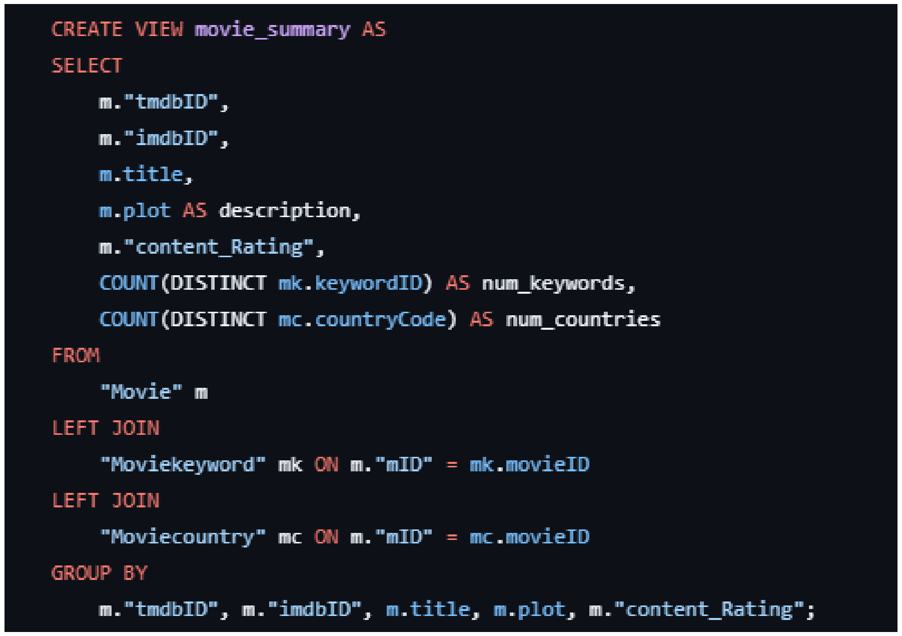
```
INSERT INTO "Actor" (aAge, aFirst, aMiddle, aLast, aCountry) VALUES
(45, 'Robert', NULL, 'Downey Jr.', 'US'),
(38, 'Scarlett', NULL, 'Johansson', 'US'),
(52, 'Chris', NULL, 'Hemsworth', 'AU'),
(39, 'Zoe', NULL, 'Saldana', 'DO'),
(44, 'Leonardo', NULL, 'DiCaprio', 'US'),
(50, 'Johnny', NULL, 'Depp', 'US'),
(60, 'Morgan', NULL, 'Freeman', 'US'),
(41, 'Will', NULL, 'Smith', 'US'),
(37, 'Charlize', NULL, 'Theron', 'ZA'),
(50, 'Tom', NULL, 'Cruise', 'US');
```

For example, in the above picture, we're populating the Actor table

Use of Views

Part 6.

We created a SQL view named create*_movie_summary* to provide a summary of movies, which includes identifiers, title, plot, content rating, and aggregated information on the number of associated keywords and countries. We used a left join to connect the Movie table (m) with Moviekeyword (mk) and Moviecountry (mc). By doing so, we make sure that all movies are included even if they have no keywords or country associations. We used GROUP BY on relevant columns to group data per movie and allow aggregation of keyword and country counts. To make sure that each count is unique, we used COUNT (DISTINCT...).



```sql
CREATE VIEW movie_summary AS
SELECT
    m."tmdbID",
    m."imdbID",
    m.title,
    m.plot AS description,
    m."content_Rating",
    COUNT(DISTINCT mk.keywordID) AS num_keywords,
    COUNT(DISTINCT mc.countryCode) AS num_countries
FROM
    "Movie" m
LEFT JOIN
    "Moviekeyword" mk ON m."mID" = mk.movieID
LEFT JOIN
    "Moviecountry" mc ON m."mID" = mc.movieID
GROUP BY
    m."tmdbID", m."imdbID", m.title, m.plot, m."content_Rating";
```

<u>Queries</u>

Part 7.

A) We use COUNT with conditional logical (CASE WHEN) to count movies with and without an *imdbID* and provide both results in one query.

B) This query joins the Movie, Movieactor, and Actor tables. It also filters them by the actor's first and last name and the release year range. It returns essential details like the *tmdbID, imdbID, title, releaseYear,* and *watchmodeID.*

C) This query joins the Movie and Reviews tables. It was important to use GROUP BY to group the results by title to count reviews per movie. To sort the *review_count* in descending order, we used ORDER BY. Finally, we used LIMIT to ensure that only the top 3 results are returned.

D) This query counts movies with more than one associated language by joining Movie and Movielanguage tables, grouping by *mID*. We used HAVING COUNT to filter for those with multiple languages (>1).

E) This query joins Language and Movielanguage, grouping by lName and counting the movies per language. We ordered the result in descending order for highest to lowest counts by using the ORDER BY clause.

F) This query filters movies that belong to the Comedy genre. Here again, we used ORDER BY to sort by *viewerRating* in descending order. We also used LIMIT to restrict the output to the top 2 results.

G) Here, the UPDATE statement uses the CEIL( ) function to round up *viewerRating* values for all movies. This ensures that ratings are stored as whole numbers.