MILESTONE 1

R1. Application high level description

We will be using the Sakila Movies Database to present an analysis dashboard for movies. Some features we look to implement include searching and recommendation of different movies. This would rely on setting up proper foreign key relationships in our database and determining a way to aggregate movie data to make recommendations. In addition, we would like to have a flexible interface for users to compare different movies based on genre, revenue, year released and other attributes. These pieces of information can be sorted and output tuple set size should be adjustable(limiting output to x values).

As per the outline recommendations, we will be using a locally hosted SQL based DB. Specifically, we believe DuckDB integrates well with our application written in Python. Additionally the total size of the database is well within the storage capacities of our laptops so using a remote server would introduce needless complexity.

We will be presenting the application to users via a web frontend. We are still currently exploring the specific technologies for the frontend but are leaning towards Streamlit due to them being written in Python which is the language most familiar to most members of the group. Our team will start from Streamlit and explore if there are additional libraries needed such as Plotly for dashboard development.

R2. System Support Description

The project we are implementing will be hosted locally and version control would use GitHub. This includes the database which will be DuckDB and our application which will be written in Python. Some key frameworks that we will work with include Streamlit and Plotly for data tables/UI display and a Python DB connector provided by DuckDB for performing queries against our schema. In order to collaborate on application development, we have set up a GitHub repository so group members can contribute from their local machines. Specifically we would all have Python 10.x or later installed locally along with DuckDB latest stable release of 1.2.0.

R3. Database with Sample Dataset

Sample Data

Our application is a movie review site (similar to IMDb) that draws on the Sakila sample SQL database provided by MySQL. Sakila simulates a DVD rental store containing tables for films, actors, languages, customer data, and rental transactions. It provides realistic foreign key relationships and built-in triggers (for events like returning a movie). Based on our application description, we will remove the data for store operation and focus on movies. We extend Sakila's schema with additional functionality for user reviews and ratings. We are using this DB as our base, but we are modifying it, adding extra tables and removing unnecessary tables.

Data Generation/Cleaning/Importing

- **Generation**: The Sakila dataset is already curated by MySQL, so no additional data cleaning is required.
- **Scripts**: Two SQL files are provided—one for creating the schema (`schema.sql`) and another for inserting sample rows (`data.sql`).
- **Import Process(DuckDB)**:
    1. `schema.sql` to define tables, views, and triggers.
    2. `data.sql` to populate the tables with sample records.

Using the Data to Populate the Database

1. Clone GitHub Repository for group project
2. Install DuckDB's Python API
3. Execute `schema.sql` and then `data.sql` (see test.py).
4. Confirm successful import by querying tables on either basic features or other genetic tests.

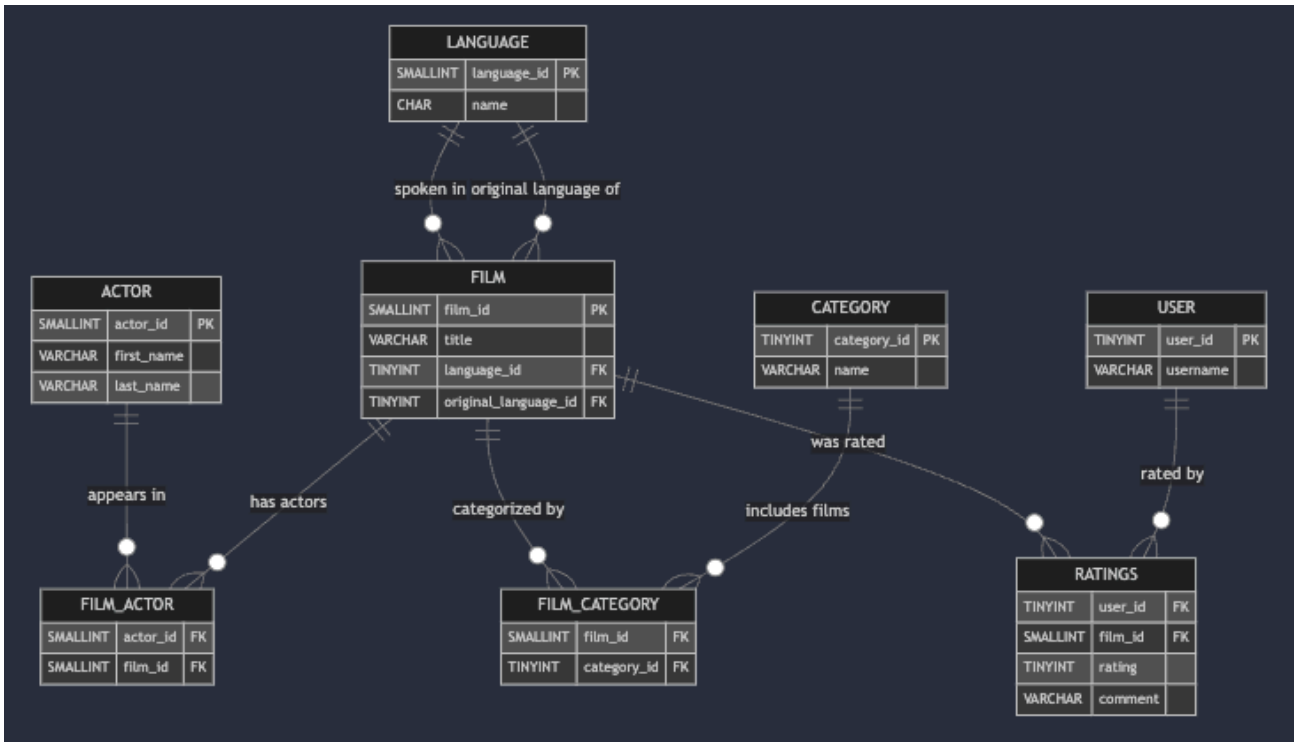
R5. Schema Design

R5a Assumptions

Many of the assumptions are defined in the schema with certain IDs being described as primary keys in their respective tables. We assume that there can be no duplicate movies (but can have the same name and language) as the movie id is a primary key.

The same can be found for the other tables where there can be no duplicate users, languages, actors, and ratings.

R5b E/R Diagram



This ER diagram was created using Mermaid.js to render the connections. If the quality of the image is poor, the raw file can be found on github.

R5c Relational Data Model

ACTOR
- actor_id (PK)

CATEGORY
- category_id (PK)

FILM
- film_id (PK)
- language_id (FK → LANGUAGE.language_id)
- original_language_id (FK → LANGUAGE.language_id)

FILM_ACTOR
- actor_id (PK, FK → ACTOR.actor_id)
- film_id (PK, FK → FILM.film_id)

FILM_CATEGORY
- film_id (PK, FK → FILM.film_id)
- category_id (PK, FK → CATEGORY.category_id)

LANGUAGE
- language_id (PK)

RATING
- user_id (PK, FK → USER.user_id)
- film_id (PK, FK → FILM.film_id)

USER
- user_id (PK)

R6. Basic Feature/Functionality 1

R-a Interface Design

There will be a button to display the average ratings of each movie and it will display the title of the movie along with the average rating.

R-b SQL Query, testing with sample data

The query can be found under tests/basic_feature_1.sql.

```
SELECT f.title, AVG(r.rating) AS average_rating
FROM FILM AS f
JOIN RATINGS AS r ON f.film_id = r.film_id
GROUP BY f.film_id, f.title;
```

The expected output should be:

Title	average_rating
Movie A	4.67
Movie B	4.52
…	…

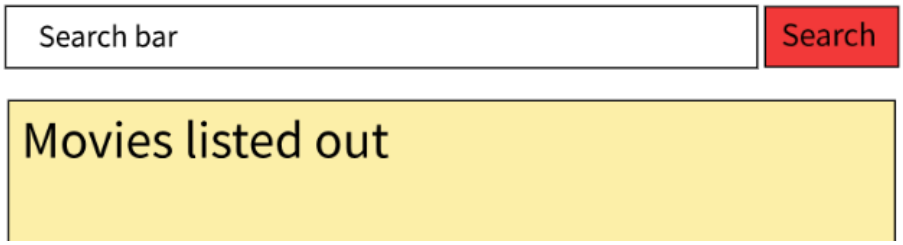
R7. Basic Feature/Functionality 2

This feature would work with the general search functionality to allow users to match their input with movies or actors. The data that we want to select:
For movies:

1) Name of the movies and their description
2) Language the movie is in
3) Other facts (Age Rating, release year, length)

R-a

This would be an input and button widget combination for users to search up movies by name

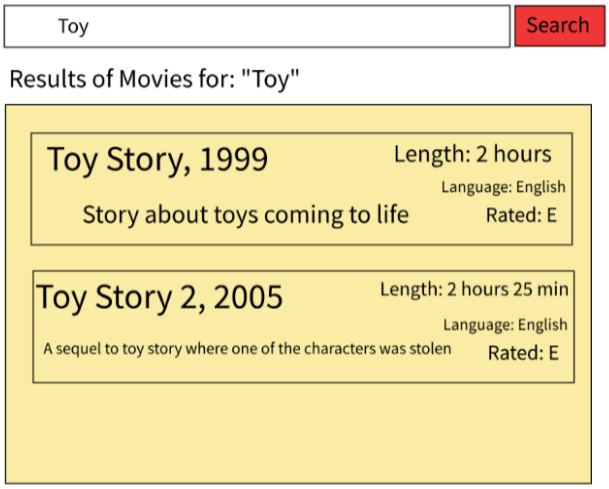


R-b

Sample Data:
FILM table:
{
(1,Toy Story,Story about toys coming to life,1995,1,120,E),
(2,Avatar,Story of the Na'vi on Pandora,2009,2,190,PG-13),
(3,Toy Story 2, A sequel to toy story where one of the characters was stolen ,2005,1,142,E)
}
LANGUAGE Data:
{
(1,English),
(2,Italian),
(3,Japanese)
}

```
SELECT FILM.title, FILM.description,FILM.release_year, FILM.age_rating,
FILM.length, LANGUAGE.name, from FILM
INNER JOIN LANGUAGE ON FILM.language_id = LANGUAGE.language_id
where FILM.title like '%{user input}%';
```

The expected output for a search string "Toy" would be:
{
('Toy Story', 'Story about toys coming to life',1995, `E`, 120, 'English'),
('Toy Story 2', 'A sequel to toy story where one of the characters was stolen', 2005, 'E', 142, 'English')
}



R8. Basic Feature/Functionality 3

The third feature we will implement is updating the specific user's original rating value into a new rating value from 1 (bad) to 5 (great).

R-a

The user selects a specific film from the film menu, clicks on the 5 stars for rating value from 1 (bad) to 5 (good), and the new rating value will replace the original rating values made by this user before.

R-b

1. Suppose user 001 wants to rate the film 106 with a new rating value 5

```
UPDATE RATINGS
SET rating = 5  -- Change this to any rating value between 1 and 5
WHERE user_id = 111  -- Replace with the specific user's ID
  AND film_id = 106;  -- Replace with the specific film's ID
```

2. Suppose user 002 wants to rate the film 369 with a new rating value 1

```
UPDATE RATINGS
SET rating = 1  -- Change this to any rating value between 1 and 5
WHERE user_id = 222  -- Replace with the specific user's ID
  AND film_id = 369;  -- Replace with the specific film's ID
```

Sample Data

- **FILM**: (film_id=106, title="Movie A"), (film_id=369, title="Movie B")
- **RATINGS**:
    - (user_id=111, film_id=106, rating=3)
    - (user_id=222, film_id=369, rating=5)

Expected Results

- User 111's rating for film_id=106 → returns 5
- User 222's rating for film_id=369 → returns 1

R9. Basic Feature/Functionality 4

R9-a

Description: Users enter a number **N**, then select either "Exactly N" or "At Least N." After clicking a "Search" button, the application displays a list of films meeting the chosen criterion along with the total number of users who rated each film.

Users: General site visitors or administrators who want to filter films based on engagement.
Interaction:

1. Enter **N** (e.g., 5).
2. Choose **"Exactly"** or **"At Least"**.
3. Click **"Search"**.
4. A results table shows film titles and the count of ratings each has received.

R9-b

1. **Exactly N**:

```sql
SELECT f.title, COUNT(r.user_id) AS rating_count
FROM FILM f
JOIN RATINGS r ON f.film_id = r.film_id
GROUP BY f.film_id
HAVING COUNT(r.user_id) = ?;
```

2. **At Least N**:

```sql
SELECT f.title, COUNT(r.user_id) AS rating_count
FROM FILM f
JOIN RATINGS r ON f.film_id = r.film_id
GROUP BY f.film_id
HAVING COUNT(r.user_id) >= ?;
```

Sample Data

- **FILM**: (film_id=1, title="Movie A"), (film_id=2, title="Movie B")
- **RATINGS**:
    - (user_id=1, film_id=1, rating=5), (user_id=2, film_id=1, rating=4)
    - (user_id=1, film_id=2, rating=3)

      Expected Results
- **Exactly 2** for film_id=1 → returns "Movie A."
- **At Least 1** → returns "Movie A" and "Movie B."

R9-c

```
CREATE INDEX idx_ratings_film_user ON RATINGS (film_id, user_id);
```

With large rating data, this index speeds up the `GROUP BY` operation. Testing involves comparing query times before/after creating the index.

R9-d

Implementation: The front end has an input for **N** and a toggle for “Exactly” vs. “At Least.” A backend service runs the corresponding SQL query and returns results in a table.
Snapshot: We then show a webpage with the input form and results table.
Testing: Verify that for a sample input, only those films rated by exactly or at least 2 users appear.
We can also check both sample and production databases to ensure correctness and performance.

R16. Members.txt

See GitHub repository.

C1. ReadMe

Listed in Github with general setup instructions for Python Project.

C2. SQL Code (Create table, etc)

See SQL files under the queries folder of GitHub.

C3. SQL Queries for features, test-sample.sql test-sample.out

See GitHub Repository

C5. Application Code

See GitHub Repository