Movie List

Primary Keys
aew1756: Ari Wisenburn
txb9274: Milo Berry
ggb6130: Gunnar Bachmann
np7437: Noah Pelletier

April 18, 2022

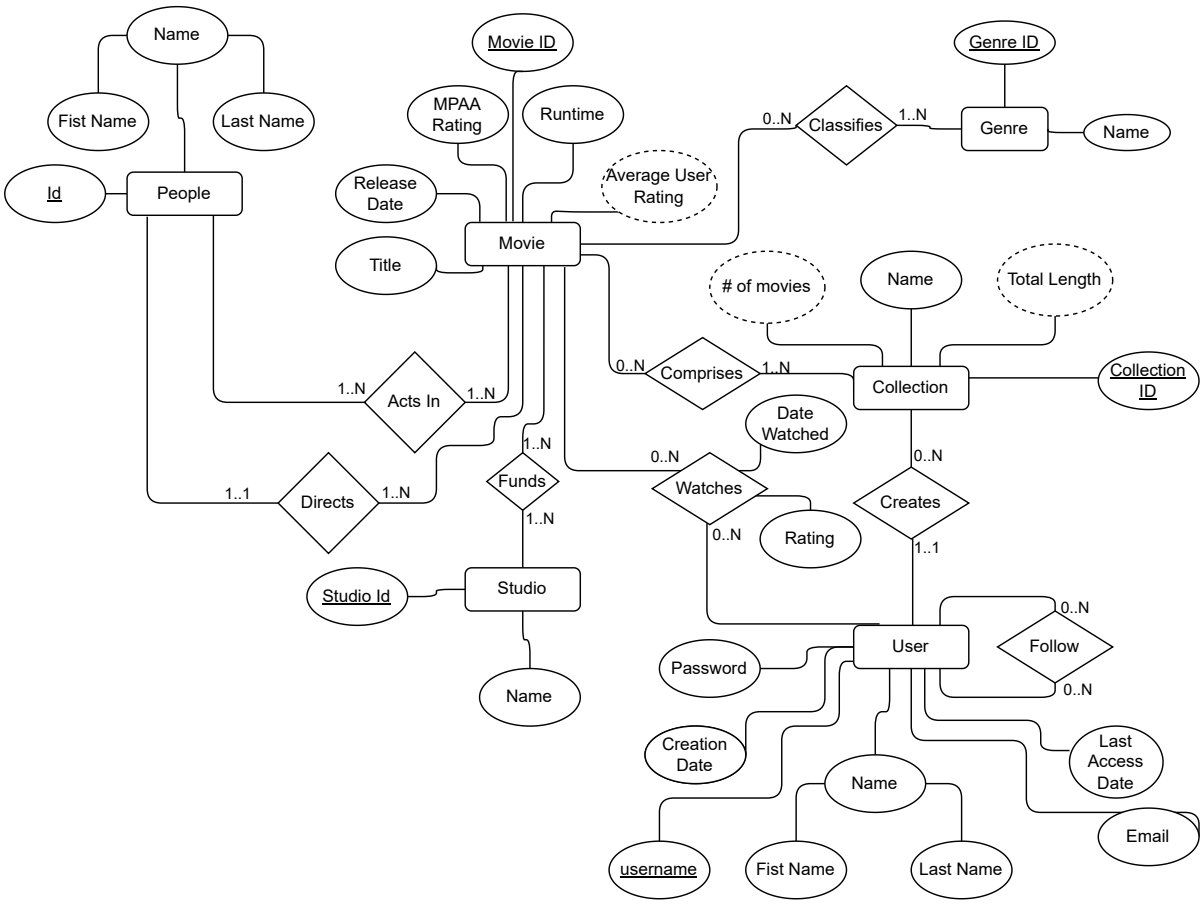
1 Introduction

Our domain is the Movie Domain which will follow a very similar outline as the Music domain. Rather than users creating collections of music, it will be of movies instead. Users will be able to create collections of movies and search for movies based off of name, release date, cast members, studio, and genre. Users will also be able to sort movies, add and delete movies from their collection, rate movies, and even watch movies individually (or play an entire collection). Following and unfollowing friends will function the same as our primary domain.

Our program will be primarily written in Python and we will be using a PostgreSQL database. User interaction will be driven through a Python-based GUI.

2 Design

2.1 Conceptual Model



Most of our ER Diagram is created directly from the requirements.

The biggest consideration made was the inclusion of the *Genre* entity. This allows us to re-use genres between movies without re-creating on that already exists keeping our database in 3NF.

Another consideration of the table, was the inclusion of the *Date Watched* attribute in the *Watches* relationship. The requirements mention that in Phase 4, we need to be able to find the most popular movies in the past 90

days as well as allow users to mark every time they play a movie, we decided to include the date they marked the movie as watched to allow us to easily perform these queries as well as have multiple entries for the same movie from the same user.

2.2 Reduction to tables

Person(id, *name_id*)
Movie(movie_id, title, mpaa_rating, runtime, release_date, *director_id*)
Genre(genre_id, name)
MovieGenre(*movie_id*, *genre_id*)
ActsIn(*movie_id*, *person_id*)
Studio(studio_id, name)
Funds(*movie_id*, *studio_id*)
Collection(collection_id, name, *username*)
MoviesInCollection(*collection_id*, *movie_id*)
User(username, *name_id*, creation_date, password, email, last_access_date)
Following(*username*, *following_username*)
Name(Id, first_name, last_name)
Watched(*username*, *movie_id*, date_watched, star_rating)

For the *Person* entity, this comprises only of a *Person Id* and a *Name Id* for an individual with the *Person Id* as the primary key and *Name Id* as a foreign key to the *Name* table. Looking at the relationships with the *Person* entity, *Acts In* became its own table since it is an N:M relationship while *Directs* turned into the field *Director Id* within the *Movie* table since movies can only have a single director. The primary key for the *ActsIn* table is a combination of the *Movie Id* and *Person Id* since these two elements will always be unique together.

Regarding the *Name* table, since *Name* is a composite attribute made up of *First Name* and *Last Name*, it was split off into its own table.

Moving forward to the *Movie* entity itself, the *Average User Rating* was disregarded in the table since it is a derived attribute and will be calculated when needed. *Genre* has also been given its own entity type since multiple movies can have the same genre. The *Classifies* relationship connects *Genre* and *Movie* with its primary key comprising of *Genre Id* and *Movie Id*.

Then looking at the *Studio* entity, the entity itself was converted into a simple table with all its attributes. But the *Funds* relationship between the *Movie* entity and the *Studio* entity was moved into its own separate table called *Funds* since they have an N:M relationship.

Next looking at the *User* entity, the entity was mapped to its own table with all its attributes except the composite *Name* attribute. Since we already have a *Name* table, the *User* table gets the attribute *Name Id* as a reference to this table. Then translating the relationship *Watches* between the *User* entity and the *Movie* entity, this was moved into its own separate table named *Watched* with the relationship attributes as well as foreign keys to the *Movie Id*, *Date Watched*, and *Username* which make up the primary key of the table. The *User* entity also has the relationship *Follow* that was converted to the table *Following* since the relationship is M:N and with the primary key comprised of the *Collection Id* and *Movie Id*

Lastly, discussing the *Collection* entity, this was transformed into its own table with only the attributes *Collection Id* and *Name* since the other 2 attributes are derived attributes. This table also includes a foreign key *Username* to reference the user that created it since collections can only be created by a single user. Then its relationship *Comprises* was turned into its own table *MoviesInCollection* since the relationship is N:M which has the *Collection Id* and *Movie Id* as its primary key.

2.3 Data Requirements/Constraints

Within the *Person* entity, the *Name* and *Id* for each person is required and unique. There is also a foreign-key constraint on *Name Id* which connects to the *Name* entity.

For the *Studio* entity, the *Studio Id* is required and unique. The name is also required. The *Funds* relationship has two foreign key constraints on *Movie Id* for the *Movie* entity and *Studio Id* for the *Studio* entity.

For the *Movie* entity, the attribute *Movie Id* is required and unique. The other required attributes are *MPAA Rating*, *Release Date*, *Runtime*, and *Genre*. The relationships *Funds*. *Acts In* and *Directs* all have total par-

ticipation meaning each *Movie* entity must have at least one entry in these relationship tables. The *MPAA Rating* must be one of the following: G, PG, PG-13, R, NR, or NC-17. There is also a foreign-key constraint for the *Director Id* that connects to the *Person* entity. The *ActsIn* relationship has two foreign keys, *Movie Id* and *Person Id*, to connect it to the *Movie* and *Person* entities correspondingly.

For the *Genre* entity, the attribute *Genre Id* is required and unique. *Name* is also required. The *MovieGenre* relationship has 2 foreign key constraints: one for *Movie Id* to relate it to the *Movie* entity and one for the *Genre Id* to relate it to the *Genre* entity.

For the *Watched* relationship, the *Rating* attribute must be one of the following: 1, 2, 3, 4, or 5. There is a foreign key, *Username*, to the *User* entity and another, *Movie Id*, to the *Movie* entity.

For the *Collection* entity, the *Collection Id* is required and unique. *Name* is also a required attribute. The reduction to table for this entity includes a required *Username* attribute as well that is a foreign key constraint to the *User* entity.The relationship *MoviesInCollection* has two foreign key constraints to *Collection* entity, *Collection Id*, and the *Movie* entity, *Movie Id*.

Lastly, for the *User* entity, the *Username* and *Email* attributes are required and unique. The *Password*, *Name*, *Creation Date*, and *Last Access Date* are also required. The relationship *Following* has a foreign key constraint on *Username* and *Following Username* relating to the *User* entity.

2.4 Sample instance data

2.4.1 Person Entity

1, Milo, Berry
2, Theo, Cat
3, Ari, Wisenburn
4, Gunnar, Bachmann
5, Noah, Pelletier

2.4.2 Acts In Relationship

Person *Milo Berry* acts in Movie *1917*
Person *Ari Wisenburn* acts in Movie *1917*
Person *Gunnar Bachmann* acts in Movie *Spider-Man: Far From Home*
Person *Gunnar Bachmann* acts in Movie *Uncharted*
Person *Theo Cat* acts in Movie *How to be a Cat*

2.4.3 Directs Relationship

Person *Milo Berry* directs Movie *The Cursed*
Person *Ari Wisenburn* directs Movie *Scream*
Person *Noah Pelletier* directs Movie *Studio 666*
Person *Noah Pelletier* directs Movie *Dog*
Person *Theo Cat* directs Movie *How to be a Cat: The Sequel*

2.4.4 Movie Entity

1, Uncharted, PG-13, 2/18/2022, 123 min, NULL
2, 1917, R, 1/10/2020, 142 min, 4.84
3, Spider-Man, PG-13, 4/1/2002, 94 min, 3.42
4, Scream, R, 10/31/1993, 132 min, 4.56
5, Make Happy, PG-13, 2/18/2016, 83 min, 4.10

2.4.5 Genre Entity

1, Western
2, Crime
3, Dark
4, Superhero
5, Comedy

2.4.6 MovieGenre Relationship

Genre *Horror* classifies Movie *Scream*
Genre *Action* classifies Movie *Uncharted*
Genre *Comedy* classifies Movie *Make Happy*
Genre *War* classifies Movie *1917*

Genre *Superhero* classifies Movie *Spider-Man*

2.4.7 Studio Entity

1, Fake Studio
2, Another Fake Studio
3, Studio 666
4, Theo's Studio
5, Final Studio

2.4.8 Funds Relationship

Studio *Fake Studio* funds Movie *Uncharted*
Studio *Another Fake Studio* funds Movie *Uncharted*
Studio *Fake Studio* funds Movie *1917*
Studio *Studio 123* funds Movie *Dog*
Studio *Theo's Studio* funds Movie *Cat Food: The Movie*

2.4.9 User Entity

miloblueberry, Milo, Berry, 1/2/2019, 2/18/2022, fakepassword, txb9274@rit.edu
theo-is-perfect, Theo, Cat, 1/23/2020, 2/10/2021, wetfoodisbest, cat@cat.com
ari_wiserthanyou, Ari, Wisenburn, 10/2/2017, 1/1/2022, heylookapassword, ari@rit.edu
gunnarwasinuncharted, Gunnar, Bachmann, 10/21/2020, 2/18/2022, gunnarwasinuncharted, gunnar@rit.edu
noah-in-the-boat, Noah, Pelletier, 7/2/2021, 2/1/2022, fakepassword, noah@rit.edu

2.4.10 Watches Relationship

Person *Milo Berry* watches Movie *1917*: *Rating*: 5, *Date Watched*: 2/3/2019
Person *Ari Wisenburn* watches Movie *Scream*: *Rating*: 3, *Date Watched*: 2/13/2021
Person *Milo Berry* watches *1917*: *Rating*: 5, *Date Watched*: 12/5/2020
Person *Gunnar Bachmann* watches *Scream*: *Rating*: 4, *Date Watched*: 10/31/2022
Person *Gunnar Bachmann* watches *Uncharted* : *Rating*: 2, *Date Watched*: 2/3/2022

2.4.11 Follow Relationship

User *miloblueberry* follows User *ari_wiserthanyou*
User *ari_wiserthanyou* follows User *noah-in-the-boat*
User *noah-in-the-boat* follows User *ari_wiserthanyou*
User *miloblueberry* follows User *theo-is-perfect*
User *theo-is-perfect* follows User *cat-food-is-the-best*

2.4.12 Collection Entity

1, Horror Movies, 120 min, 2 movies
2, Slasher Films, 5001 min, 12 movies
3, Comedies, 290 min, 5 movies
4, Stand-up, 67 min, 1 movie
5, Empty Collection, 0 min, 0 movies

2.4.13 Creates Relationship

User *miloblueberry* creates Collection *Horror Films*
User *ari_wiserthanyou* creates Collection *Horror Films*
User *noah-in-the-boat* creates Collection *Funny Comedies*
User *miloblueberry* creates Collection *Unfunny Comedies*
User *theo-is-perfect* creates Collection *Movies about Cats*

2.4.14 Comprises Relationship

Movie *Scream* comprises Collection *Slasher Films*
Movie *Make Happy* comprises Collection *Funny Comedies*
Movie *The Addams Family* comprises Collection *Funny Comedies*
Movie *Homeward Bound* comprises Collection *Dog Movies*
Movie *Scream* comprises Collection *Horror Movies*

3 Implementation

3.1 Table Creation and Data Addition

```
create table p320_21.movie
(
```

```
    movie_id    int not null
        constraint movie_pk
            primary key,
    title       varchar,
    mpaa_rating varchar,
    runtime     int,
    release_date date,
    director_id int
        constraint movie_person_id_fk
            references p320_21.person
        constraint mpaa_rating check(mpaa_rating in ('G', 'PG',
            'PG-13', 'R', 'NC-17', 'NR'))
);

create unique index movie_movie_id_uindex
    on p320_21.movie (movie_id);
```

```
create table p320_21."user"
(
    username      varchar not null
        constraint user_pk
            primary key,
    name_id        int not null
        constraint user_name_id_fk
            references p320_21.name,
    creation_date  date not null,
    password       varchar not null,
    email          varchar not null,
    last_access_date date not null
);

create unique index user_username_uindex
    on p320_21."user" (username);

create table p320_21.name
(
    id        SERIAL
        constraint name_pk
            primary key,
```

```
    first_name varchar not null,
    last_name varchar not null
);

create unique index name_id_uindex
    on p320_21.name (id);
```

For the 2 databases, *Name* and *Collection*, which would be added to in the application with unique ids, we added the *serial* attribute allowing us to easily add to these tables without finding the previously added id and increment it to insert new data.

To upload data to our tables, we first sanitized and processed the data set we selected as well as creating dummy data for values that did not exist within that dataset. We then downloaded this sanitized data into separate CSV files for each table. We then used DataGrip's built in import CSV file to table functionality. We started with the tables that had no foreign keys: *Name*, *Genre*, and *Studio*. Then we moved to the two entities with foreign keys *Person* and *Movie*. Then, we populated the relationship tables *ActsIn*, *Funds*, and *MovieGenre*. If we had used insert statements, they would be similar to:

```
insert into p320_21.name(name_id, first_name, last_name) values
    (1, 'First', 'Last');
insert into p320_21.name(name_id, first_name, last_name) values
    (2, 'Second', 'Super Final');
insert into p320_21.genre(genre_id, name) values (1, 'Crime');
insert into p320_21.genre(genre_id, name) values (2, 'Horror');
insert into p320_21.studio(studio_id, name) values (1, 'Test
    Studio');
insert into p320_21.studio(studio_id, name) values (2, 'Test
    Studio 2');
insert into p320_21.person(person_id, name_id) values (1, 1);
insert into p320_21.person(person_id, name_id) values (2, 2);
insert into p320_21.movie(movie_id, title, mpaa_rating, runtime,
    release_date, director_id) values (1, '1917', 'R', 230,
    '2020-04-06', 1);
insert into p320_21.movie(movie_id, title, mpaa_rating, runtime,
    release_date, director_id) values (2, 'Spider-Man', 'PG-13',
    183, '2021-12-17', 2);
insert into p320_21.acts_in(movie_id, person_id) values (1, 1);
```

```
insert into p320_21.acts_in(movie_id, person_id) values (2, 2);
insert into p320_21.funds(movie_id, studio_id) values (1, 1);
insert into p320_21.funds(movie_id, studio_id) values (2, 2);
insert into p320_21.movie_genre(movie_id, genre_id) values (1, 1);
insert into p320_21.movie_genre(movie_id, genre_id) values (2, 2);
```

For the tables that involve user interaction, *User*; *Collection*; *Following*; *MoviesInCollection*; and *Watched*, we used the application insert statements to add data in these databases instead of bulk-loading all the data into the tables. These statements looked like:

```
INSERT INTO p320_21.collection(name, username) VALUES
   ('test_name', 'test_username');
INSERT INTO p320_21.collection(name, username) VALUES ('new
   collection', 'test_username_2');
INSERT INTO p320_21.movies_in_collection(collection_id, movie_id)
   VALUES (1, 1);
INSERT INTO p320_21.movies_in_collection(collection_id, movie_id)
   VALUES (2, 2);
INSERT INTO p320_21."user" (username, name_id, creation_date,
   password, email, last_access_date) VALUES ('test_user', 1,
   '2022-03-18', 'password', 'test@test.com', '2022-03-18');
INSERT INTO p320_21."user" (username, name_id, creation_date,
   password, email, last_access_date) VALUES ('test_user_2', 2,
   '2022-03-19', 'password_2', 'test2@test2.com', '2022-03-19');
INSERT INTO p320_21.following (username, following_username)
   VALUES ('test_user', 'test_user_2');
INSERT INTO p320_21.following (username, following_username)
   VALUES ('test_user_2', 'test_user');
INSERT INTO p320_21.watched(username, movie_id, date_watched)
   VALUES ('test_user', 1, '2022-03-19');
INSERT INTO p320_21.watched(username, movie_id, date_watched)
   VALUES ('test_user_2', 2, '2022-03-20');
INSERT INTO p320_21.name(first_name, last_name) VALUES ('Milo',
   'Berry') RETURNING id;
INSERT INTO p320_21.name(first_name, last_name) VALUES ('Gunnar',
   'Bachman') RETURNING id;
```

3.2 Example Queries

Here are a list of example queries we made to populate data in our application:

```
-- Name
SELECT first_name, last_name FROM p320_21.name WHERE id = 1;
SELECT name_id FROM p320_21.name WHERE first_name = 'Sample' and
    last_name = 'Name';
-- Studio
SELECT name FROM p320_21.studio WHERE id = 1;
SELECT studio_id, name FROM p320_21.studio WHERE name LIKE
    '%sample%';
-- Genre
SELECT name FROM p320_21.genre WHERE id = 1;
SELECT genre_id, name FROM p320_21.studio WHERE name LIKE
    '%sample%';
-- Person
SELECT name.first_name, name.last_name FROM p320_21.person LEFT
    JOIN p320_21.name on person.name_id = name.id WHERE
    person.person_id = 1;
SELECT person.id FROM p320_21.person LEFT JOIN p320_21.name on
    person.name_id = name.id WHERE name.first_name LIKE '%test%';
-- Movie
SELECT title, mpaa_rating, runtime / 60 AS hours, runtime % 60 AS
    minutes, release_date, date_watched, star_rating FROM
    p320_21.movie LEFT JOIN p320_21.watched ON movie.movie_id =
    watched.movie_id WHERE username = 'test_user' AND
    movie.movie_id = 1;
SELECT id FROM p320_21.movie WHERE title LIKE '%test%';
-- ActsIn
SELECT person)id FROM p320_21.acts_in WHERE movie_id = 1;
SELECT movie_id FROM p320_21.acts_in WHERE person_id = 1;
-- MovieGenre
SELECT movie_id FROM p320_21.movie_genre WHERE genre_id = 1;
SELECT genre.name FROM p320_21.movie_genre LEFT JOIN p320_21.genre
    ON movie_genre.genre_id = genre.genre_id WHERE
    movie_genre.movie_id = 1;
-- Funds
SELECT movie_id FROM p320_21.funds WHERE studio_id = 1;
SELECT studio.name FROM p320_21.funds LEFT JOIN p320_21.studio ON
```

```
    funds.studio_id = studio.studio_id WHERE funds.movie_id = 1;
-- User
SELECT * FROM p320_21."user" WHERE username = 'sample_user' LIMIT
    1;
SELECT * FROM p320_21."user" WHERE username = 'sample_user' AND
    password = 'password' LIMIT 1;
-- Collection
SELECT collection_id FROM p320_21.collection WHERE name =
    'collection' AND username = 'test_user';
SELECT collection_id FROM p320_21.collection WHERE username =
    'test_user';
-- Following
SELECT following.following_username, name.first_name,
    name.last_name FROM p320_21.following LEFT JOIN p320_21."user"
    ON following.following_username = "user".username LEFT JOIN
    p320_21.name ON "user".name_id = name.id WHERE
    following.username = 'test_user';
SELECT following_username FROM p320_21.following WHERE username =
    'test_user';
-- MoviesInCollection
SELECT movie_id FROM p320_21.movies_in_collection WHERE
    collection_id = 1;
SELECT movie.title FROM p320_21.movies_in_collection LEFT JOIN
    p320_21.movie ON movies_in_collection.movie_id = movie.movie_id
    WHERE movies_in_collection.collection_id = 1;
-- Watched
SELECT movie_id FROM p320_21.watched WHERE username = 'test_user';
SELECT rating FROM p320_21.watched WHERE username = 'test user'
    and movie_id = 1;
```

3.3 Indices

Besides the indexes that are automatically added to the primary keys of a table by PostgreSQL, we added indices to *title* and *release date* on *Movie*, *name Id* on *Person*, *name* on *Studio*, *email* on *User*, *name* on *Genre*, and *star rating* on *Watched*.

We decided to add the *title*, *release date*, *Person name id*, *Genre name*, and *Studio name* in order to speed up the search functionality. Each of these fields where used to search movies: title of the movie, release date of the

film, actor's names that were in the film, genre name, and studio name. In a similar vein, we added the *email* index on the *User* table because users are able to search for other users by their email address.

The last index we added was for the *star rating* on the *Watched* table. This was added to speed up the user profile functionality. Since the user has an option to see their highest rated films, we added an index to the ratings.

3.4 Appendix of SQL Statements Used in Phase 4

3.4.1 Indices

```
CREATE INDEX collection_username_index ON p320_21.collection
    (username);
CREATE INDEX genre_name_index ON p320_21.genre (name);
CREATE INDEX movie_title_index ON p320_21.movie (title);
CREATE INDEX movie_release_date_index ON p320_21.movie
    (release_date);
CREATE INDEX person_name_id_index ON p320_21.person (name_id);
CREATE INDEX studio_name_index ON p320_21.studio (name);
CREATE INDEX user_email_index ON p320_21."user" (email);
CREATE INDEX watched_star_rating_index ON p320_21.watched
    (star_rating);
```

3.4.2 Recommendation System

```
-- View for the users top watched genres
CREATE VIEW p320_21.user_top_genre AS
with genre_rank as(
SELECT username,
    name as Genre,
    count(name) as count,
    row_number() over (partition by username order by count(name)
        desc) as row_num
FROM p320_21.watched
LEFT JOIN p320_21.movie_genre ON watched.movie_id =
    movie_genre.movie_id
LEFT JOIN p320_21.genre ON movie_genre.genre_id = genre.genre_id
group by username, name
```

```
order by count desc)

-- Top 20 most popular movies in the last 90 days (rolling)
SELECT watched.movie_id,
    title,
    mpaa_rating,
    runtime / 60 AS hours,
    runtime % 60 AS minutes,
    to_char(release_date, 'yyyy-MM-dd'),
    avg(star_rating),
    to_char(MAX(date_watched), 'yyyy-MM-dd'),
    COUNT(watched.movie_id)
FROM p320_21.watched
LEFT JOIN p320_21.movie ON watched.movie_id = movie.movie_id
WHERE date_watched > current_date - interval '90' day
GROUP BY watched.movie_id, title, mpaa_rating, hours, minutes,
    release_date
ORDER BY COUNT(watched.movie_id) DESC
LIMIT 20;

-- Top 20 most popular movies among my friends
SELECT DISTINCT watched.movie_id,
    title,
    mpaa_rating,
    runtime / 60 AS hours,
    runtime % 60 AS minutes,
    to_char(release_date, 'yyyy-MM-dd'),
    avg(star_rating),
    to_char(MAX(date_watched), 'yyyy-MM-dd'),
    COUNT(watched.movie_id)
FROM p320_21.following
LEFT JOIN p320_21.watched ON following.following_username =
    watched.username
LEFT JOIN p320_21.movie ON watched.movie_id = movie.movie_id
WHERE following.username = 'username'
GROUP BY watched.movie_id, title, mpaa_rating, hours, minutes,
    release_date, star_rating
ORDER BY COUNT(watched.movie_id) DESC
LIMIT 20;
```

```
-- Top 5 new releases of the month
SELECT watched.movie_id,
    title,
    mpaa_rating,
    runtime / 60 AS hours,
    runtime % 60 AS minutes,
    to_char(release_date, 'yyyy-MM-dd'),
    avg(star_rating),
    to_char(MAX(date_watched), 'yyyy-MM-dd')
FROM p320_21.watched
LEFT JOIN p320_21.movie ON watched.movie_id = movie.movie_id
WHERE release_date > current_date - interval '1' month
GROUP BY watched.movie_id, title, mpaa_rating, runtime / 60,
    runtime % 60, to_char(release_date, 'yyyy-MM-dd'), star_rating
ORDER BY COUNT(watched.movie_id) DESC, star_rating DESC
LIMIT 5;

-- For you: Recommend movies to watch based on your play history
    (e.g. genre, -- cast member, rating) and the play history of
    similar users
WITH user_genre AS (SELECT username, genre_id FROM
    p320_21.user_top_genre WHERE username = 'username' OR
username IN (SELECT following_username
FROM p320_21.following
WHERE username = '{username}'))
SELECT DISTINCT movie.movie_id,
    title,
    mpaa_rating,
    runtime / 60 AS hours,
    runtime % 60 AS minutes,
    to_char(release_date, 'yyyy-MM-dd'),
    avg(star_rating),
    to_char(MAX(date_watched), 'yyyy-MM-dd'),
    COUNT(watched.movie_id)
FROM user_genre
LEFT JOIN p320_21.movie_genre ON user_genre.genre_id =
    movie_genre.genre_id
LEFT JOIN p320_21.movie ON movie_genre.movie_id = movie.movie_id
LEFT JOIN p320_21.watched ON movie.movie_id = watched.movie_id
```

```
WHERE movie_genre.genre_id = user_genre.genre_id and star_rating
    is not null
GROUP BY movie.movie_id, title, mpaa_rating, runtime / 60, runtime
    % 60, to_char(release_date, 'yyyy-MM-dd')
ORDER BY COUNT(watched.movie_id) DESC
LIMIT 20;
```

3.4.3 User Profile Functionality

```
-- Number of collections a user has
SELECT COUNT(DISTINCT collection_id)
FROM p320_21.collection
WHERE username = 'username';

-- Number of followers
SELECT COUNT(DISTINCT username)
FROM p320_21.following
WHERE following_username = 'username';

-- Number of following
SELECT COUNT(DISTINCT following_username)
FROM p320_21.following
WHERE username = 'username';

-- Top 10 movies (by highest rating)
SELECT DISTINCT watched.movie_id,
    title,
    mpaa_rating,
    runtime / 60 AS hours,
    runtime % 60 AS minutes,
    to_char(release_date, 'yyyy-MM-dd'),
    star_rating,
    to_char(MAX(date_watched), 'yyyy-MM-dd')
FROM p320_21.watched
LEFT JOIN p320_21.movie ON watched.movie_id = movie.movie_id
WHERE username = 'username' AND star_rating IS NOT NULL
GROUP BY watched.movie_id, title, mpaa_rating, hours, minutes,
    release_date, star_rating
ORDER BY star_rating DESC
```

```
LIMIT 10;

-- Top 10 movies (most plays)
SELECT DISTINCT watched.movie_id,
    title,
    mpaa_rating,
    runtime / 60 AS hours,
    runtime % 60 AS minutes,
    to_char(release_date, 'yyyy-MM-dd'),
    star_rating,
    to_char(MAX(date_watched), 'yyyy-MM-dd'),
    COUNT(watched.movie_id)
FROM p320_21.watched
LEFT JOIN p320_21.movie ON watched.movie_id = movie.movie_id
WHERE username = 'username' AND star_rating IS NOT NULL
GROUP BY watched.movie_id, title, mpaa_rating, hours, minutes,
    release_date, star_rating
ORDER BY COUNT(watched.movie_id) DESC
LIMIT 10;

-- Top 10 movies (Combination of highest rating and most plays)
SELECT DISTINCT watched.movie_id,
    title,
    mpaa_rating,
    runtime / 60 AS hours,
    runtime % 60 AS minutes,
    to_char(release_date, 'yyyy-MM-dd'),
    star_rating,
    to_char(MAX(date_watched), 'yyyy-MM-dd'),
    COUNT(watched.movie_id)
FROM p320_21.watched
LEFT JOIN p320_21.movie ON watched.movie_id = movie.movie_id
WHERE username = 'username' AND star_rating IS NOT NULL
GROUP BY watched.movie_id, title, mpaa_rating, hours, minutes,
    release_date, star_rating
ORDER BY COUNT(watched.movie_id) DESC, star_rating DESC
LIMIT 10;
```

3.4.4 Data Analysis

```
-- average rating per genre of movies in collections
select g.name, ROUND(CAST(avg(star_rating) as numeric), 2) as
    average_rating
from p320_21.movie
left join p320_21.movies_in_collection as mc on movie.movie_id =
    mc.movie_id
left join p320_21.movie_genre as mg on mg.movie_id = movie.movie_id
left join p320_21.genre as g on g.genre_id = mg.genre_id
left join p320_21.watched as w on w.movie_id = movie.movie_id
where collection_id is not null
and w.star_rating is not null
group by g.name;

--number of movies in each genre that have been added to
    collections
select g.name, count(mc.movie_id)
from p320_21.movie
left join p320_21.movies_in_collection as mc on movie.movie_id =
    mc.movie_id
left join p320_21.movie_genre as mg on mg.movie_id = movie.movie_id
left join p320_21.genre as g on g.genre_id = mg.genre_id
where collection_id is not null
group by g.name;

--average rating of top 10 actor (that have been in the most
    movies)'s movie
select concat(n.first_name, ' ', n.last_name), (
    select ROUND(cast(avg(w.star_rating) as numeric), 2)
    from p320_21.movie
    left join p320_21.watched as w on movie.movie_id = w.movie_id
    left join p320_21.acts_in as a on movie.movie_id = a.movie_id
    where w.movie_id is not null
    and a.person_id = acts_in.person_id
    group by person_id
) as star_avg
from p320_21.acts_in
left join p320_21.movie m on m.movie_id = acts_in.movie_id
left join p320_21.person p on acts_in.person_id = p.id
```

```
left join p320_21.name n on p.name_id = n.id
group by person_id, n.first_name, n.last_name
order by count(person_id) desc
limit 10;
```

4 Data Analysis

4.1 Hypothesis

We wanted to know the most popular genres and actors within the application. This would allow us to determine which films to add to the application in the future and guide some decisions on which movies to feature or promote. Since the genres were taken from the Kaggle data, we except the most popular genres to be drama, comedy, and romance.

4.2 Data Preprocessing

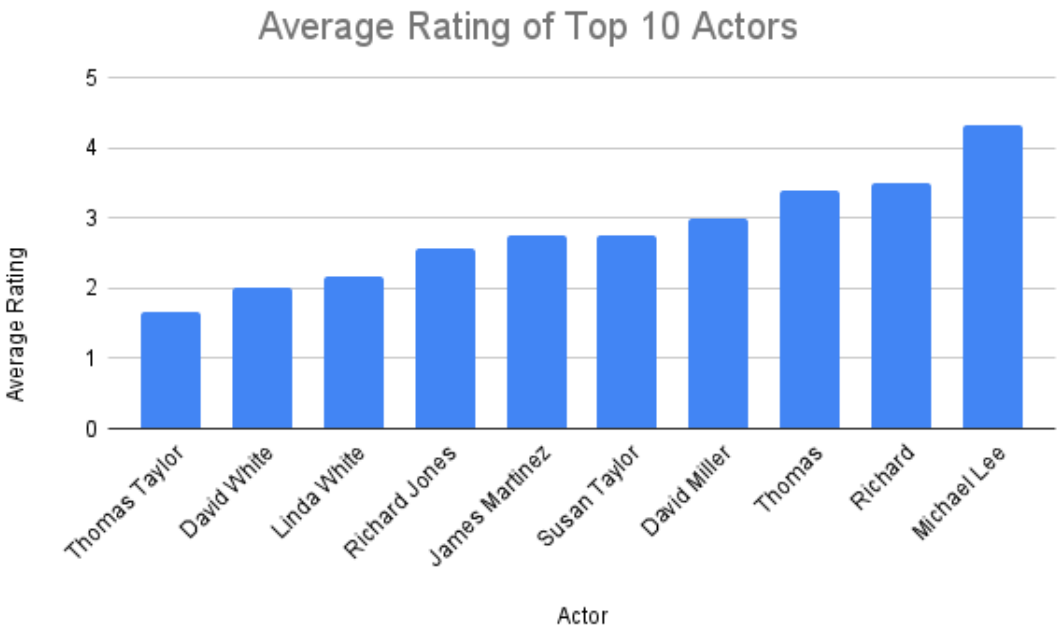
Most of our data preprocessing involved filling in missing data or adding more data. Since most of our analytics depended on user functionality, we need to add more instances of users watching and rating movies. This was especially important for our average movie rating of top ten actors since 4 of them lacked any rating data.

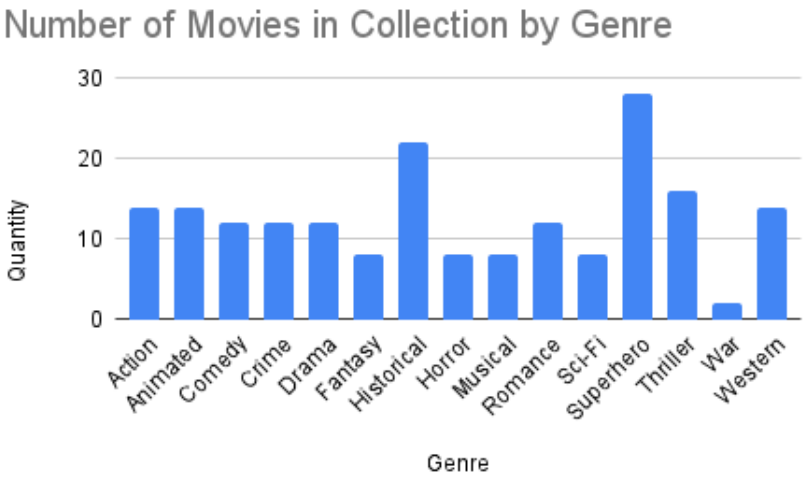
Our data was valid and consistent since we had sanitized it before entering it into the database in the previous phase. We ran into few formatting issues but these were just converting dates with timestamps to just dates.

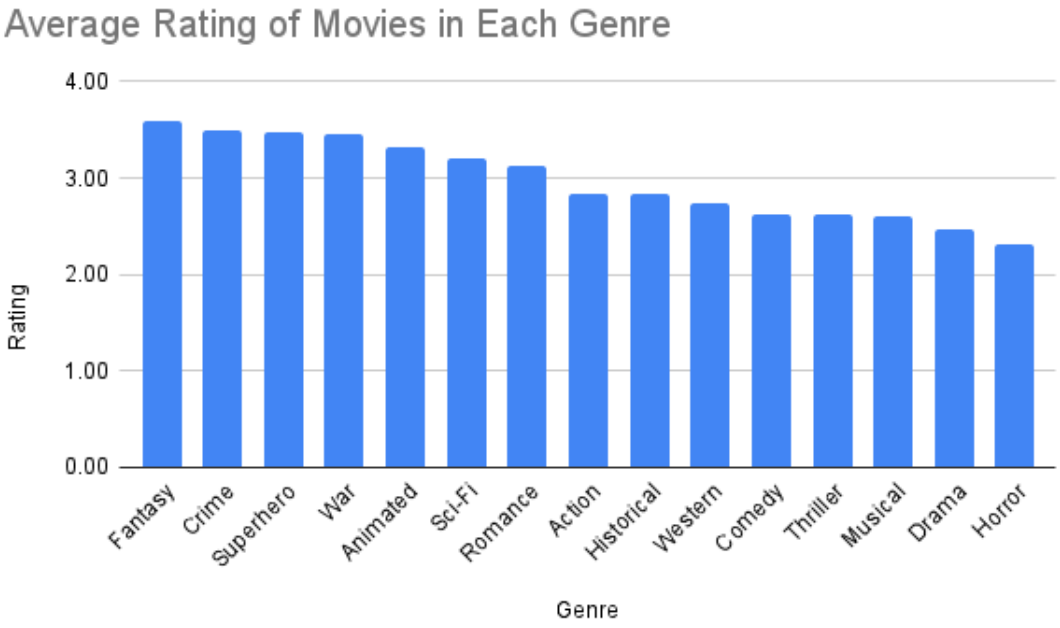
In order to get the data, we used the previous section's SQL query statements and saved the results as CSVs using DataGrip's save query result as CSV functionality. We then imported these CSVs into an Excel file to further process the data.

4.3 Data Analytics & Visualization

To analyze our data, we started by creating three separate bar charts to initially visualize the data


Average Rating of Top 10 Actors
Average Rating
5
4
3
2
1
0
Thomas Taylor
David White
Linda White
Richard Jones
James Martinez
Susan Taylor
David Miller
Thomas
Richard
Michael Lee
Actor


Number of Movies in Collection by Genre
Quantity
30
20
10
0
Action
Animated
Comedy
Crime
Drama
Fantasy
Historical
Horror
Musical
Romance
Sci-Fi
Superhero
Thriller
War
Western
Genre

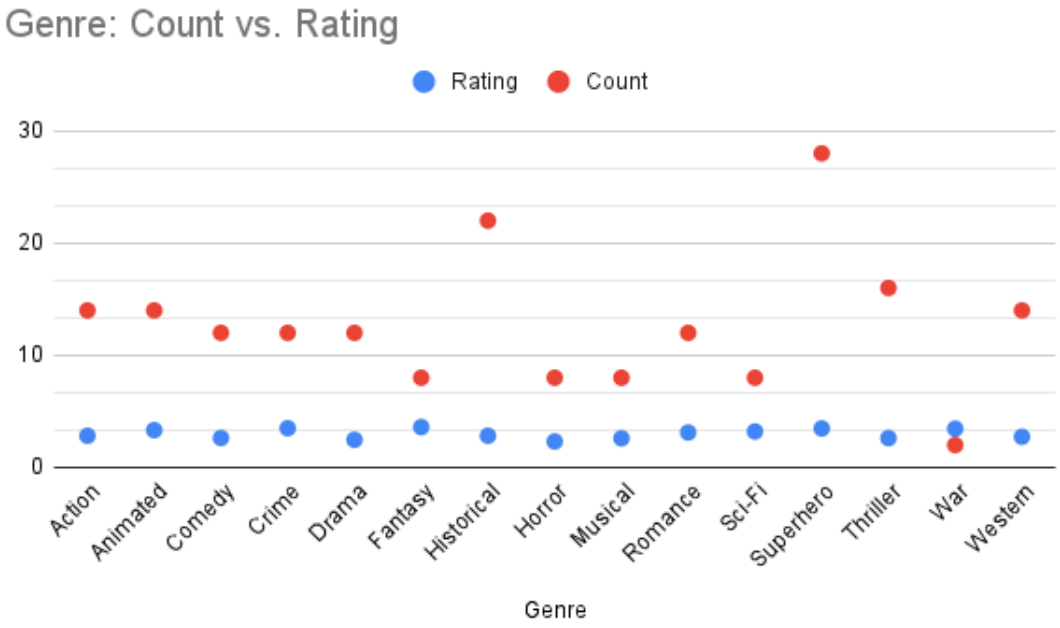


Which indicated to us that historical and superhero movies were by far the most popular with over 20 films in collections while the other genres barely broke 10. Ratings-wise, fantasy and crime were the highest rated, but these values were much closer.

So from this point we calculated the standard deviation of both datasets finding the standard deviation of the ratings (≈ 0.425) was lower than the standard deviation of the count (6.218).

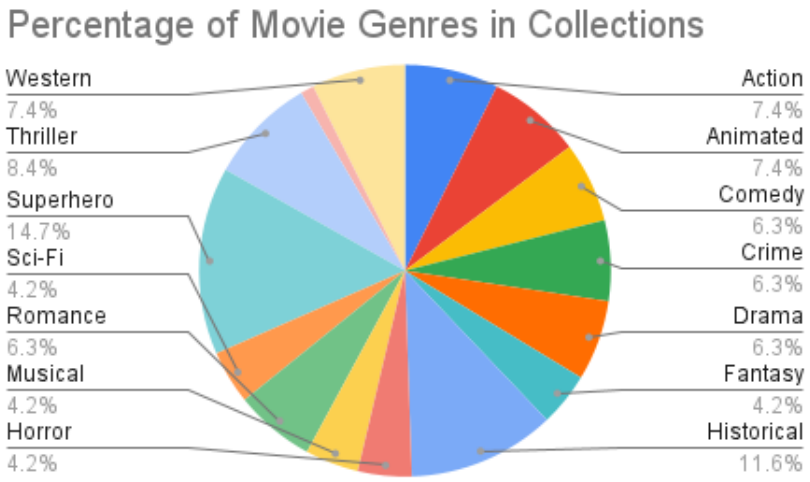
For the actor's rating chart, the results were fairly straightforward. If given more time, we could have extended the data in this to include the genres they acted in and see the correlation between their top genres and their ratings in comparison to the genre's overall rating, but we decided to focus on the genres themselves for this analysis.

These charts also generated interest in further exploring the relationship between average rating of a genre versus the number of movies for that genre in collections. So we created the following graph:



As well as calculated the correlation coefficient between the two values, getting the result of ≈ 0.025, which indicates there is a slight positive relationship between the two values (the more movies within the genre that are in collections, the higher the average rating will be) but the relationship is very weak.

From this point, we also wanted to see the percentage breakdown of movie genres within collections and created the following graph:



Which indicated that the genre breakdown was less intense than it appeared in the previous graph, but still present enough to note.

4.4 Conclusions

Overall, we found the most common genre to be superhero movies and the most well liked genre to be fantasy. The slight correlation between number of films and rating is interesting, but not particularly useful in determining what films to add to the database. If we had to add more films, I would recommend looking the number of films added within a genre rather the the ratings of a specific genre. Our predictions about which genres would be the most popular did not turn out to be true, which makes sense as we randomly added movies to collections rather than real users. In a real-world scenario, this could indicate that our users are not interested in most of the films in our database and we should add more films within their top genres.

Which leads to a large issue with our results, our lack of extensive data also skewed the analysis of this slightly; in a real-world scenario, there would probably be a stronger correlation between average rating of a genre and the number of films within a collection and the genre rating breakdown would not be as tight-knit as it is. Although, the lack of data did help us with pre-processing since we did not have to have intensive data review activities.

5 Lessons Learned

Overall, we all become much more comfortable with SQL and manipulating complex databases. Queries that would take us an hour to write in the 3rd phase, only took 15 minutes in the 4th phase.

We also learned first hand how overly dividing data can lead to unnecessary complexity. For this project specifically, dividing the actors' names into a separate table was the "correct" choice, it led to our queries to get an actor in a movie to involve joining three separate tables and a larger learning curve for explaining how our application works.

The next biggest issue we faced was the UI implementation. While it was not an explicit requirement of the application, the user (us mostly) still had to interface with the application. Trying to decide what UI issues were necessary to solve and which were not was something that required a lot of thought that we hadn't anticipated going in. The UI also brought some extra code changes that were not initially in the requirements. Because of how the UI worked, we prevented users from having multiple collections with the same names which technically falls outside of the expected application behavior. Overall, given another chance, we should have started the UI sooner in order to alleviate some of the issues we had with it from impacting the phase.

6 Resources

The dataset we used to populate our *Movie* data table was from Kaggle. All other data was randomly generated.