CSC 370 – Database Systems

Dr. Sean Chester

Movie Rating System

Project Sprint: 2

June 12th, 2024

Team27J

Alyssa Taylor V00987477

Karan Gosal V00979752

For the Course Level Competency, we are currently at ***Level-2*** for both *Data Modelling* and *Data Analytics* as we have mostly accomplished these.

Considering the information from the competencies from lecture, in detail:

We have achieved Level 2 proficiency in Data Analytics. In SQL, we can transform data for visualisation using slicing, dicing, pivoting, and aggregation. We have mastered expressing complex logic through single SQL queries, understanding how dependencies and integrity affect query semantics, and effectively managing operators in nested queries. Additionally, we have also achieved Level 2 data modelling, constructing well-normalized conceptual and relational schemas that accurately capture requirements without redundancy. By effectively normalizing, we eliminate data anomalies. We identify dependencies among attributes and select appropriate identifiers/keys for entity sets and relations. We justify the quality of a schema through a theoretical lens. We map requirements onto schemas and vice versa to ensure designs are minimal and complete.

By the **end of the next Sprint** i.e., Sprint 3, We will cover some parts of ***Level-3*** for *Data Analytics* and ***Level-1*** for Back-end Engineering.

From Level-3 Data Analytics, we will look into **indexes**, which are essential for enhancing data retrieval efficiency and query performance by reducing the time and resources required to locate specific data.

From Level-1 for Back-end Engineering, we will also explore **views** in SQL, understanding how to create, modify, and query views to control data access and simplify complex database operations. This includes using views as subqueries, performing joins, and managing data modifications through views.

We will also explore the **ACID properties**—Atomicity, Consistency, Isolation, and Durability—essential for ensuring data integrity and reliability in transactional database systems. We will learn how these properties support the execution of reliable and consistent **transactions**, allowing us to maintain data integrity and recover from failures effectively.

These are some **current limitations**. We might discover more as we move forward.

- The current system might suffer from inefficiencies in data retrieval times as it grows in size in terms of data, which impacts operational performance. Slow data retrieval not only affects user experience but also affects decision-making processes.
- Missing the GRANT and REVOKE commands features as admins may grant superuser privileges to someone but can also revoke them if necessary to avoid these issues.
- The system might experience issues with data integrity, resulting in inconsistencies or incomplete transactions. The system lacks Atomicity, Consistency, Integrity and Durability.
- The system lacks proper views, potentially exposing sensitive information like passwords to unauthorized access. This poses a significant security risk and undermines user trust.

Users

user_id	username	password	email	full_name	is_admin	created_at
1	johndoe	securepassword	johndoe@example.com	John Doe	FALSE	2024-06-01 12:34
2	sidhum	testpassword	sidhumoose@example.com	Sidhu Moose	TRUE	2024-06-01 12:50

Stars_In

cast_id	movie_episode_id	episode_number
4	1	0
5	1	0
1	2	1
2	2	1
3	2	1
1	2	2
3	2	2

Rates

user_id	movie_episode_id	episode_number	rating	review	show_username
1	1	0	5	Great Movie	TRUE
2	2	1	4	Excellent pilot episode	FALSE
2	2	2	5	Enjoyable show	TRUE

Movie_Episodes

movie_episode_id	episode_number	title	release_year	synopsis	has_episodes
1	0	Die Hard	2014	John McClane and a young hacker join forces to take down master cyber-terrorist in Washington D.C.	0
2	0	Big Bang Theory	2005	A woman who moves into an apartment across the hall from two brilliant scientists.	1
2	1	S01: Pilot	2005	Meet all the characters first time.	0
2	2	S02: Leonard meets Penny	2005	In this Leonard meets Penny for the first time.	0

Genres

genre_id	genre_name
0	Scary
1	Comedy
2	Romance
3	Action
4	Thriller

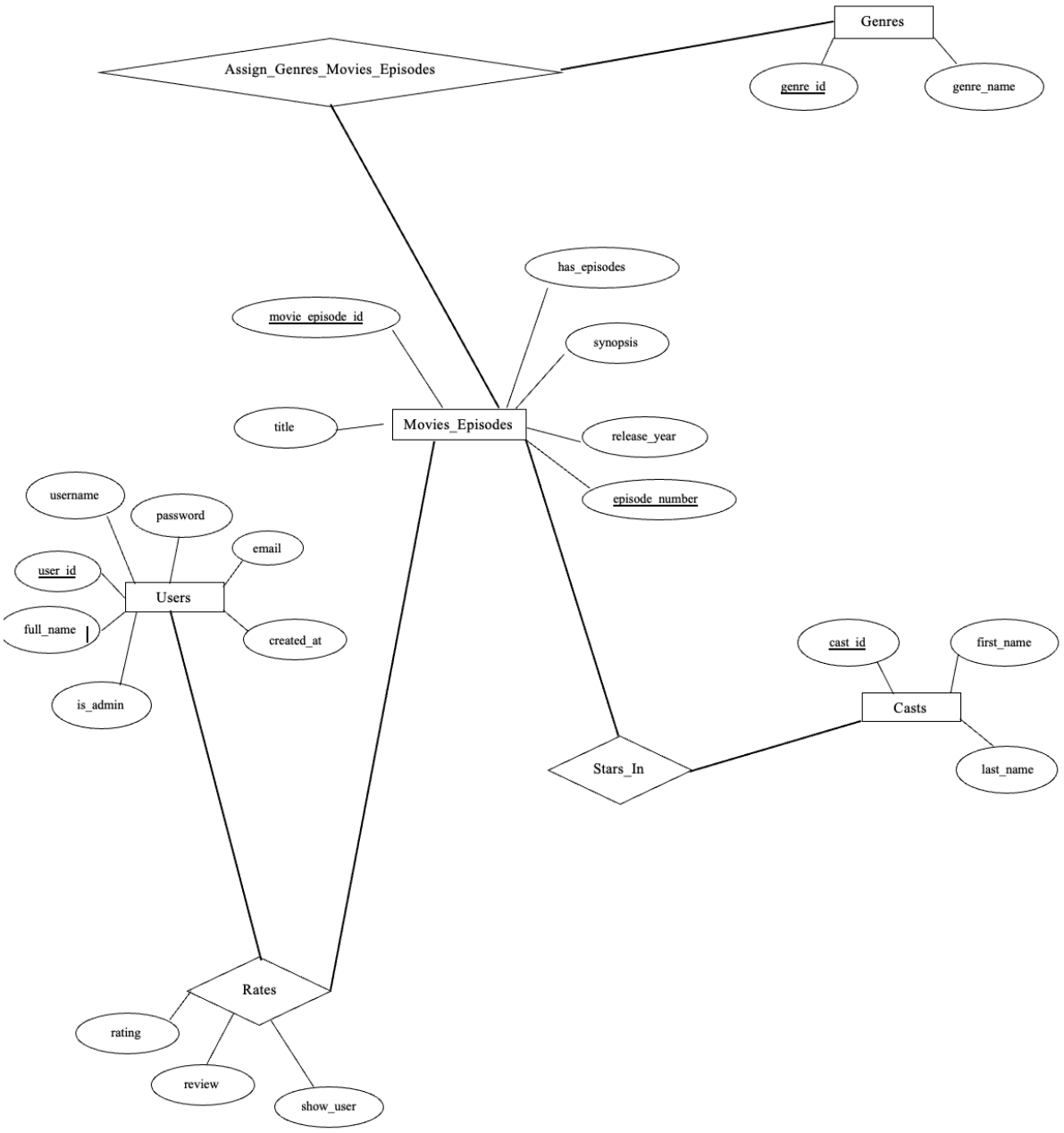
Casts

cast_id	first_name	last_name
1	Johnny	Galecki
2	Jim	Parsons
3	Kaley	Cuoco
4	Bruce	Willis
5	Justin	Long

Assign_Genres_to_Movie_Episodes

genre_id	movie_episode_id	episode_number
3	1	0
4	1	0
1	2	1
1	2	0
2	2	0

All the tables above represent the tables that follow our ERD schema and in the past sprint we have covered the basics for creating tables and creating tuples using the information from the CSV files or manually using INSERT SQL commands. Here is our ERD just in case.



DATA ANOMALIES:

1) **Redundancy -** Redundancy occurs when information is needlessly duplicated. In our case, we have a design that is redundancy free as evident by the example tuples from above.

2) **Insertion Anomaly** - Insertion anomaly refers to the possibility of introducing data errors or violating database design when inserting new tuples.
   Currently, we don't have any foreign key references. So, there is a chance of insertion anomaly in our tables. For example, if we add some entry inside the **Rates** table with some random ***user_id*** which does not exist in the **Users** table.

3) **Updation Anomaly** - Update anomaly occurs when modifying information could lead to inconsistencies in the data.
   Currently, we don't have any foreign key relations. So, there is a chance of updation anomaly in our tables. For example, if we update some entry inside the **Rates** table for ***user_id***, then we have different values in the **Rates** table compared to the **Users** table.

4) **Deletion Anomaly** - Deletion anomaly happens when removing information could result in losing other related information as well.
   Here in our case as we don't have foreign key references yet, if we delete some entry from the **Users** table which was used in some other table, we would lose all the information about the user as we are just using ***user_id*** in other tables for example **Rates** Table.

FUNCTIONAL DEPENDENCIES:

Functional dependencies are constraints between attributes in a relation (or table) that describe the relationships between those attributes. We have four Entities - **Users, Movies_Episodes, Genres, Casts**. So, we need to consider the Functional Dependencies in these. The main purpose of identifying functional dependencies is to ensure that the database is normalized and that it adheres to certain design principles.

Users

user_id -> username, password, full_name, email, is_admin, created_at
username -> user_id, password, full_name, email, is_admin, created_at
email -> username, password, full_name, user_id, is_admin, created_at

Candidate keys: {user_id}, {username}, {email}.
Primary key: {user_id}

Closures:

$\{user\_id\}^{+}$ = {username, password, full_name, email, is_admin, created_at, user_id}

$\{username\}^{+}$ = {username, password, full_name, email, is_admin, created_at, user_id}

$\{email\}^{+}$ = {username, password, full_name, email, is_admin, created_at, user_id}

Movies_Episodes

movie_episode_id , episode_number -> title, has_episodes, synopsis, release_year

Closures:

$\{movie\_episode\_id , episode\_number\}^{+}$ = {movie_episode_id , episode_number, title, has_episodes, synopsis, release_year}

Candidate keys: {movie_episode_id, episode_number}

Primary key: {movie_episode_id, episode_number}

Genres

genre_id -> genre_name

genre_name -> genre_id

Closures:

$\{genre\_id\}^{+}$ = {genre_id, genre_name}

$\{genre\_name\}^{+}$ = {genre_id, genre_name}

Candidate keys: {genre_id}, {genre_name}

Primary key: {genre_id}

Casts

cast_id -> first_name, last_name

Closures:

$\{cast\_id\}^{+}$ = {cast_id, first_name, last_name}

Candidate keys: {cast_id}

Primary key: {cast_id}

By checking all the functional dependencies and computing closures(the maximum set of attributes that can be identified from one attribute), we can say all of our functional dependencies fulfil the conditions for the BCNF. **Every determinant must be a candidate key**: A determinant is an attribute (or a set of attributes) on which some other attribute is fully functionally dependent. In BCNF, for every non-trivial functional dependency X→Y, X must be a superkey (a candidate key). This is true for all of our functional dependencies. Therefore, we can say **all of our tables are in**

BCNF and we don't need any decomposition(breaking up the table into more than one table due to BCNF violation) of the table.

We have incorporated some rules out of these few logical rules for the closures:

1) **Combining rule**: For example

   user_id -> username

   user_id -> password

   After combining:

   user_id -> username, password

2) **Splitting rule**: For example

   user_id -> username, password

   After splitting:

   user_id -> username

   user_id -> password

3) **Transitivity rule**: For example

   user_id -> username

   username -> password

   Using transitivity:

   user_id -> username, password

4) **Triviality rule**: For example

   username -> username, password

5) **Semi-triviality rule**: For example

   username -> username, password

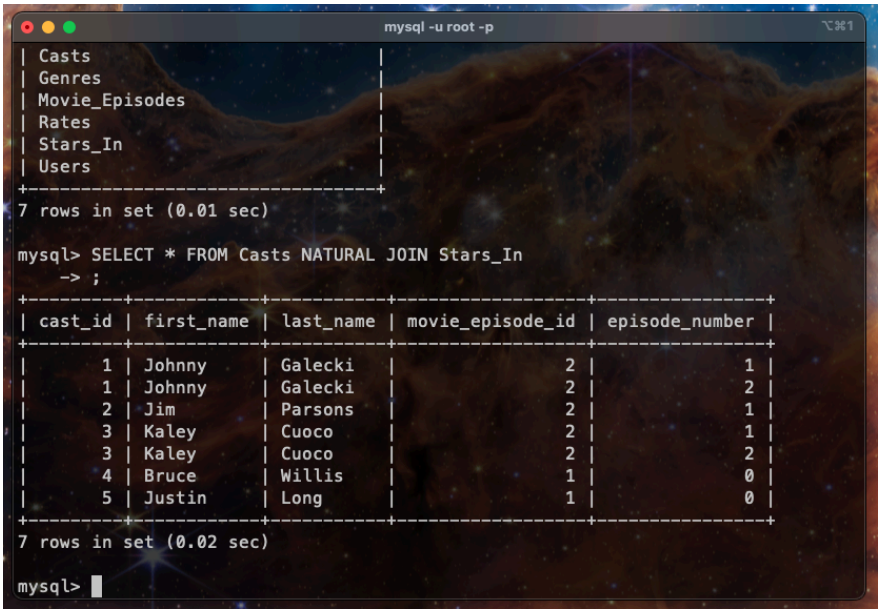
   Simplified as:

   username -> password

LOSSLESS DECOMPOSITION AND NATURAL JOIN:

Since our tables are in BCNF and we already have a lossless decomposition. We can perform the natural join on the tables to get the whole picture of the tables if they were combined.

```
SELECT * FROM Casts NATURAL JOIN Stars_In;
```



CONSTRAINTS AND REFERENTIAL INTEGRITY:

CONSTRAINTS

Since this is a complex database system with personal user data and complex possible queries, many constraints need to be included in order to keep data anomalies and running errors to a minimum. We have decided on some basic constraints below and have written them in their equivalent relational algebra definition:

- Every account has a Unique email
- Every account has a Unique username
- Every genre has a Unique name
- Ratings are Bound between 0 and 5

Relational Algebra Constraint Definitions

Rating between 0 & 5

$$\sigma_{Rating < 0 \vee Rating > 5}(Ratcs) = \emptyset$$

Unique Email

$$\rho_{u1}(users) \bowtie_{u1.email = u_2.email \wedge u_1.uswid != u_2.userid} \rho_{u2}(usrs) = \emptyset$$

Unique username

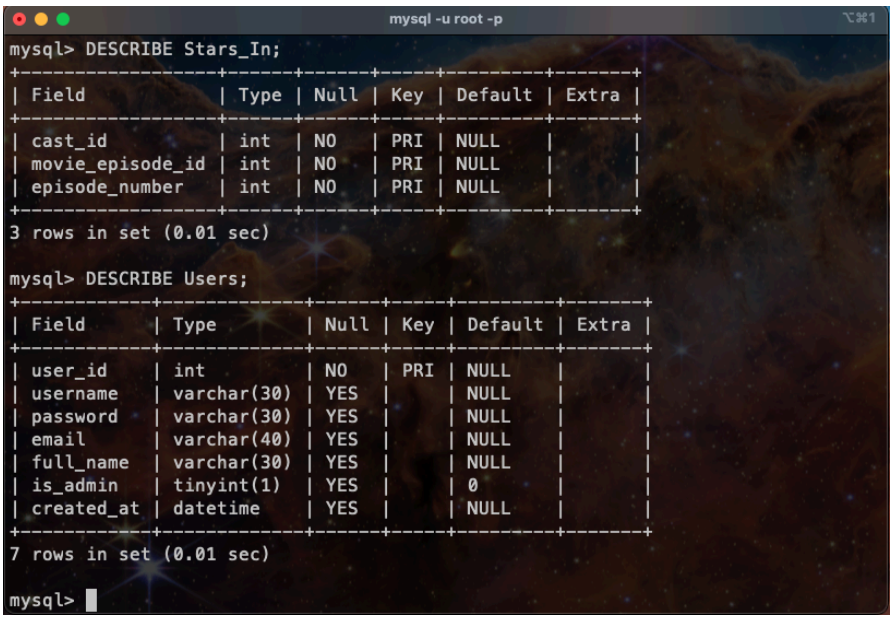
$$\rho_{u1}(users) \bowtie_{u1.\underset{name}{user} = u_2.\underset{name}{user} \wedge u_1.uswid != u_2.userid} \rho_{u2}(usrs) = \emptyset$$

unique genre names

$$\rho_{g1}(genres) \bowtie_{u1.\underset{name}{genre} = u_2.\underset{name}{genre} \wedge u_1.\underset{id}{genre} != u_2.\underset{id}{genre}} \rho_{g2}(genres) = \emptyset$$

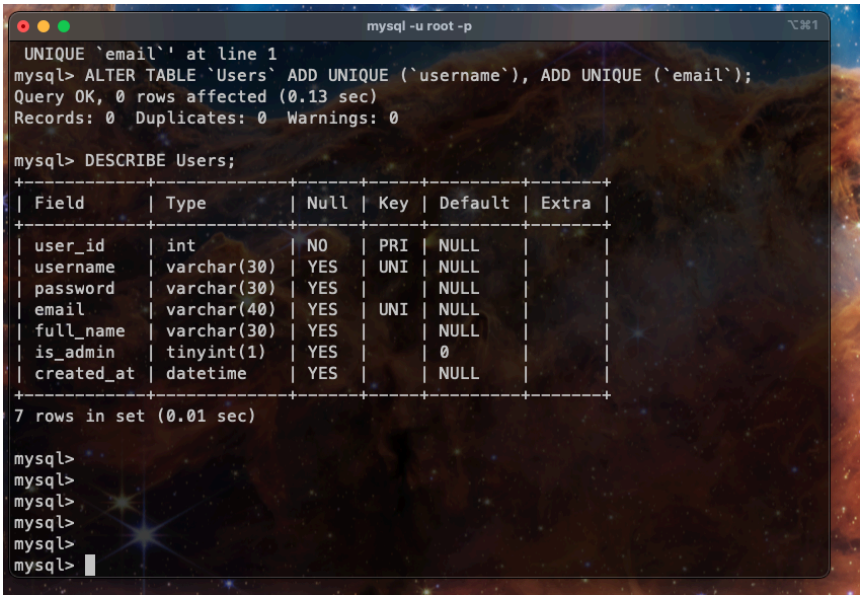
ALTER Table making attributes unique:

Before -




```
ALTER TABLE `Users` ADD UNIQUE (`username`), ADD UNIQUE (`email`);
```

After -

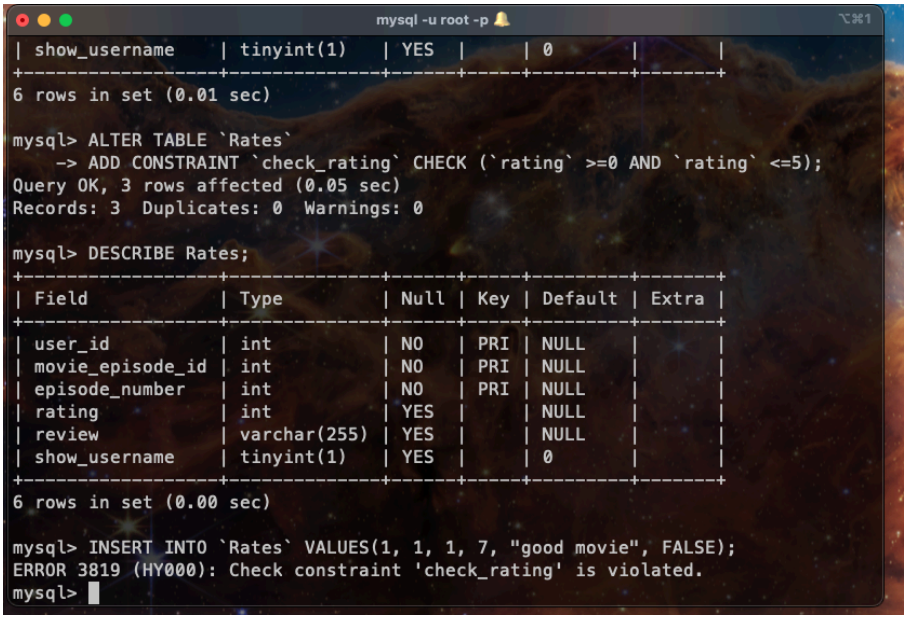


Similarly, this can be applied on the **Genres** table as we want ***genre_name*** to be unique.

Adding Constraint to the Rating table, so the rating must be between 0 to 5 included.

```
ALTER TABLE `Rates` ADD CONSTRAINT `check_rating` CHECK (`rating` >=0
AND `rating` <=5);
```

Proof of its working:

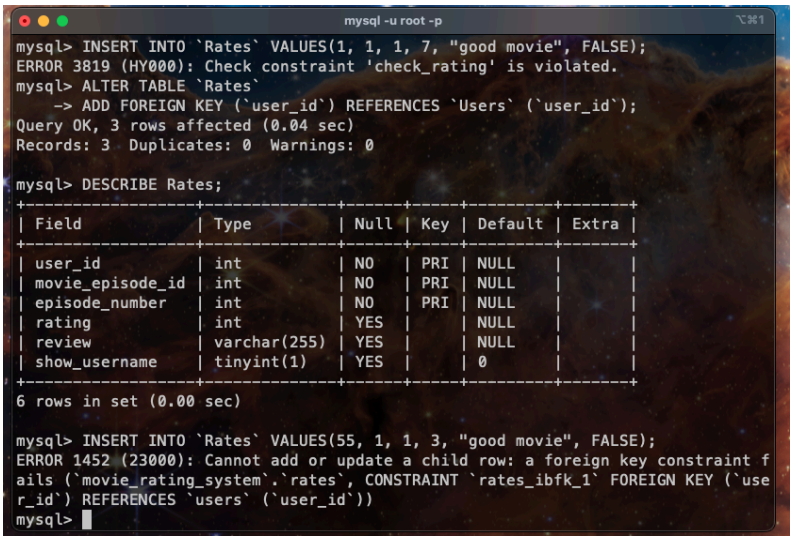


FOREIGN KEYS:

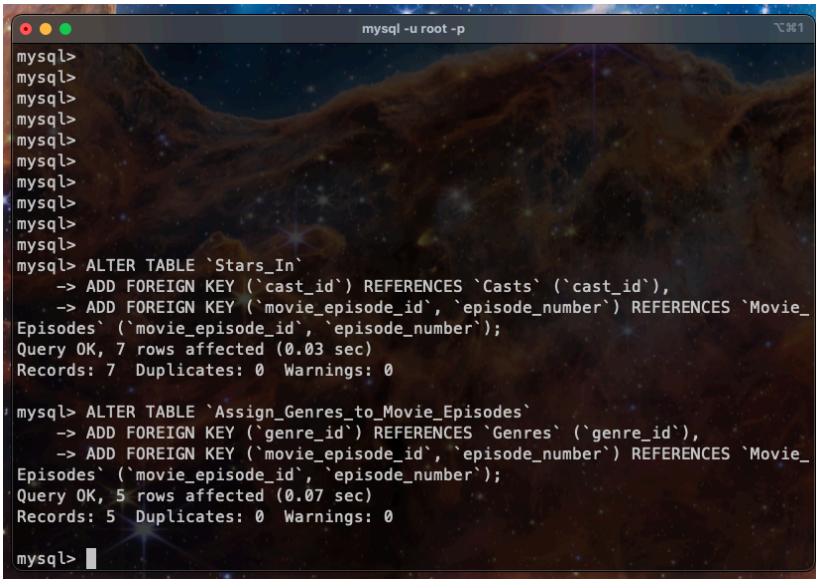
Adding the foreign key to **Rates** table referencing ***user_id*** from **Users** table.

```
ALTER TABLE `Rates` ADD FOREIGN KEY (`user_id`) REFERENCES `Users` (`user_id`);
```

```
ALTER TABLE `Rates` ADD FOREIGN KEY (`movie_episode_id`, `episode_number`) REFERENCES `Movie_Episodes` (`movie_episode_id`, `episode_number`);
```

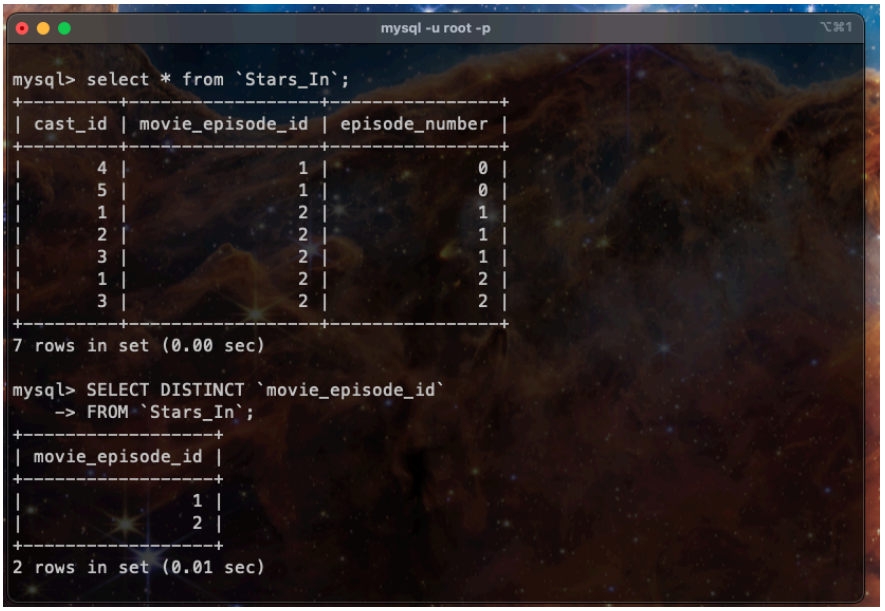


Similarly adding reference for the other tables. The queries can be found in the sql files in the repo folder.



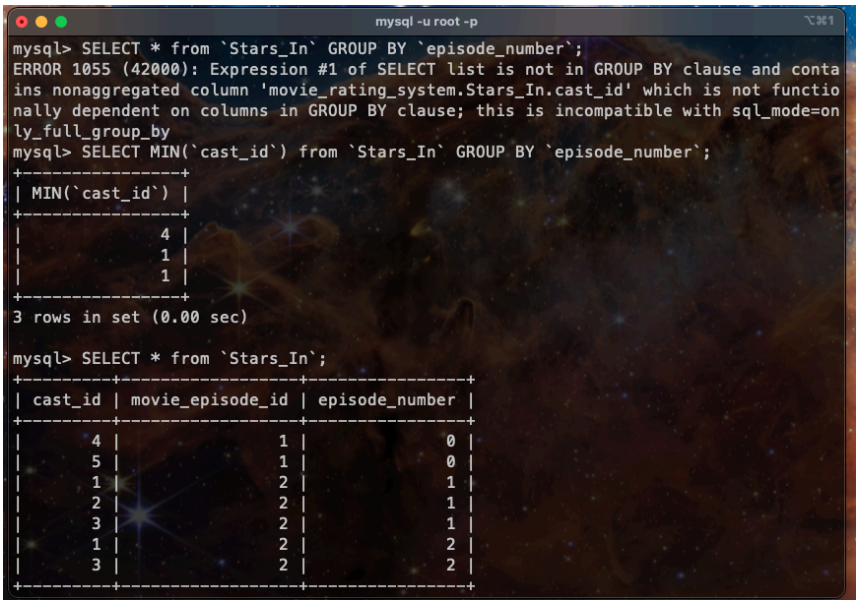
GROUPING AND AGGREGATION:

Equivalence Classes - DISTINCT, GROUP BY



AGGREGATE FUNCTION:

Convert values to scalar - MIN, MAX, AVG, SUM, COUNT, ANY

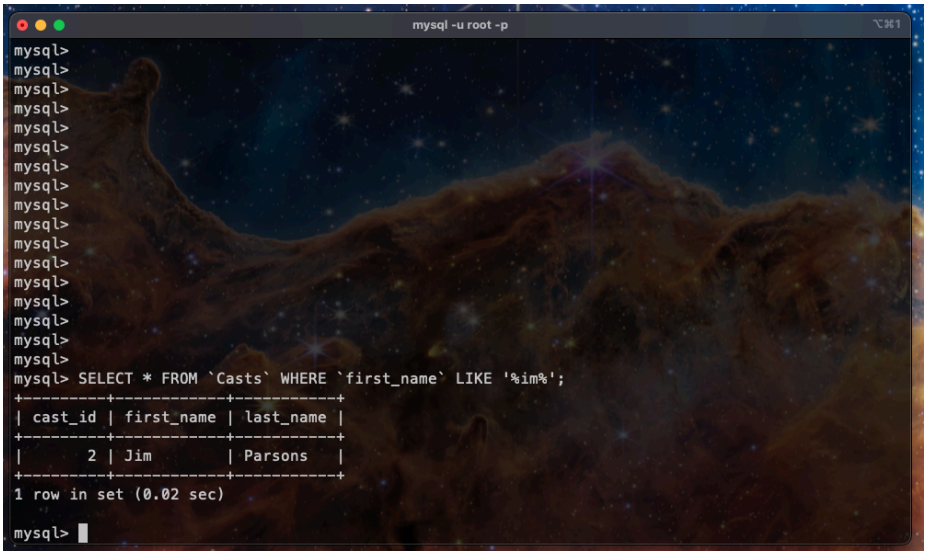


More queries can be found in the sql file in the repo.

HAVING CLAUSE:

Casts that have performed more than once in any show.

LIKE:



USE CASES AND COMPLEX QUERIES:

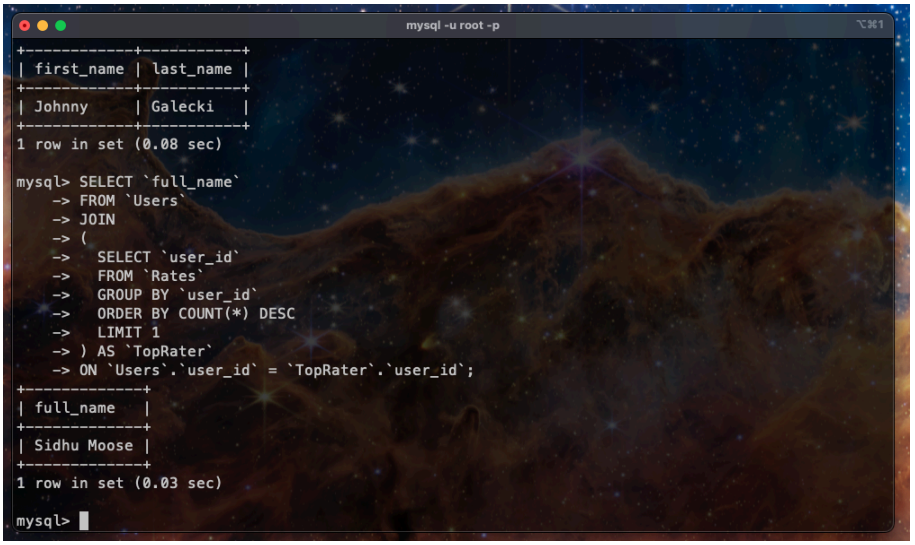
As a public movie rating system, this web app is usable by many different types of users, such as administrators, new users (no account), and full users (has an account).

Below are some of the basic level actions that would need to be included in order for this application to work as intended. The important information from each use case statement such as user-type, action, and impacted data are all bolded to easier understand each scenario. The highlighted points are complex queries demonstrated below:

1. **Admin** can see which **Users** have **rated** the most **films**
2. **User** and **Admin** can **see** which **actors** have starred in the most **films**
3. **User** and **Admin** can **see** the highest **rating** a **user** has given
4. **User** and **Admin** can **see** the **films** an **actor** has **starred in** (in one show)
5. **User** and **Admin** can **see** the **title** of **films** with specific **ratings**
6. **User** and **Admin** can **see** the highest **rating** a **user** has given
7. **User** and **Admin** can **add** a **review**
8. **Admin** can **change** a **film title**
9. **User** and **Admin** can **search** for a **specific film**
10. **Admin** can **add** a new **film**
11. **Admin** can **delete** a **film/TV show** and all its **data**

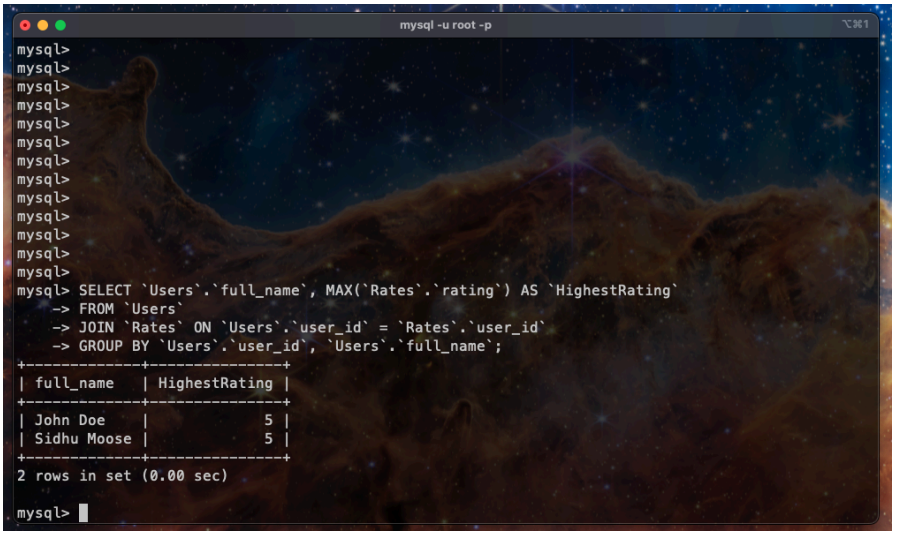
Real use cases using complex queries:

1) Get the display names of users who have rated the most episodes.

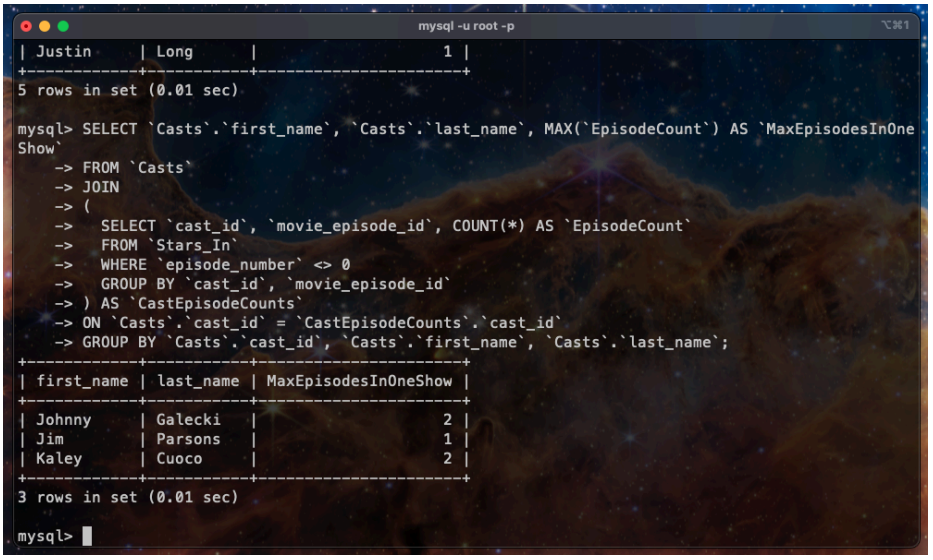


2) Get the cast members who have starred in the most episodes.

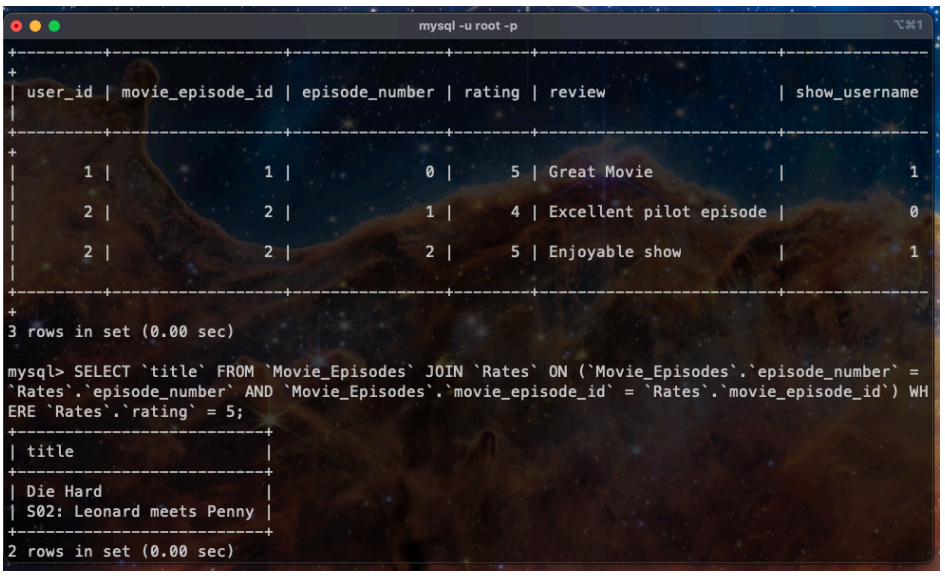
3) For each user, get the highest rating they have given.



4) For each cast member, get the maximum number of episodes they have starred in one show.



5) Get the title of the shows or movies with rating 5.



6) Get the Cast Members Not in Die Hard.