CSC 370 – Database Systems

Dr. Sean Chester

Movie Rating System

Project Final Submission

July 27th, 2024

Team27J

Alyssa Taylor V00987477

Karan Gosal V00979752

During final submission, we addressed all limitations that were present in Sprint 4 and began considering future actions that would be necessary for the operation of this database system.

4NF was ensured. Since during the previous sprints it was determined that the database was already in BCNF and 3NF, only multi-valued dependencies needed to be removed to put the database into 4NF. Analysis of dependencies and tables allowed us to prove that there were already no multi-valued dependencies in the system, therefore it is in 4NF and takes the most rigorous form possible.

Triggers and Constraints were discussed, added and edited as needed. Constraints such as the uniqueness and default values were updated to work better with the current database design. Some Triggers were added to be fired when Insert, Delete or Update commands occur, allowing the database to be more secure and have greater integrity.

Null Value Support was added to the system. Constraints on attributes to disallow or allow null values were implemented to ensure data integrity. This will allow the system to protect key information so that issues are less likely to occur during the running of this system.

Unit tests were discussed and added to validate various queries and system interactions under both good weather and bad weather conditions in order to ensure a robust testing scheme. Each table has its own set of tests where the interaction performed falls into either the “valid” or “invalid” category. This allows the system to protect the various operations and performance by having a clear pass/fail condition to check when updates/changes are implemented.

Finally, queries were analyzed and simplified as needed. This optimization allows the system to run with increased performance and therefore handle higher numbers of active users and concurrent system use.

The database by now is fairly robust, but in the future we may want to look into various query algorithms as discussed in class in order to ensure that our system runs as efficiently as possible.

In terms of levels, we have fully covered **level-3** for **data-modelling** and **data-analytics**, and touched a bit of level-4 for both.

For **back-end engineering** we covered mostly the **level-2**.

After doing some research, our future plan for the **Movie Rating System** project includes using the following tech stack:

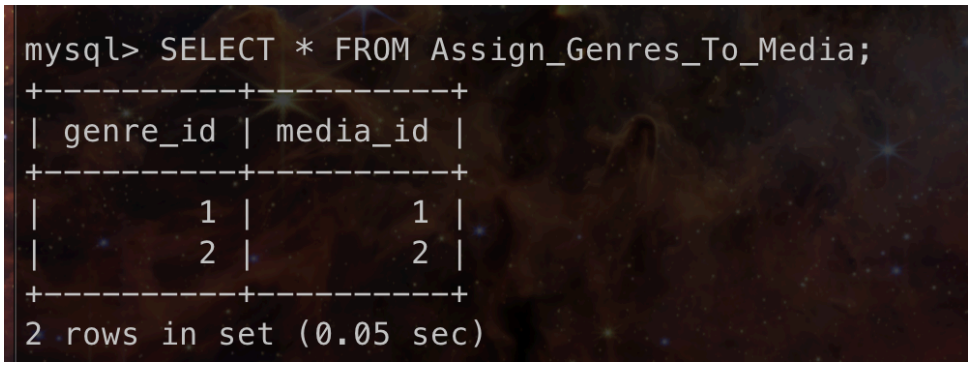
- For the **front-end**, we will use React for its great UI, Bootstrap for styling to ensure a responsive design, Context API for state management, and React Router for efficient navigation.

- On the **back-end**, we will implement Node.js with Express.js for server-side logic, Postgres as our database for its simplicity and ease of use in small projects, and Sequelize as our ORM to interact with the database. The API will follow RESTful principles to ensure smooth communication between the front-end and back-end.

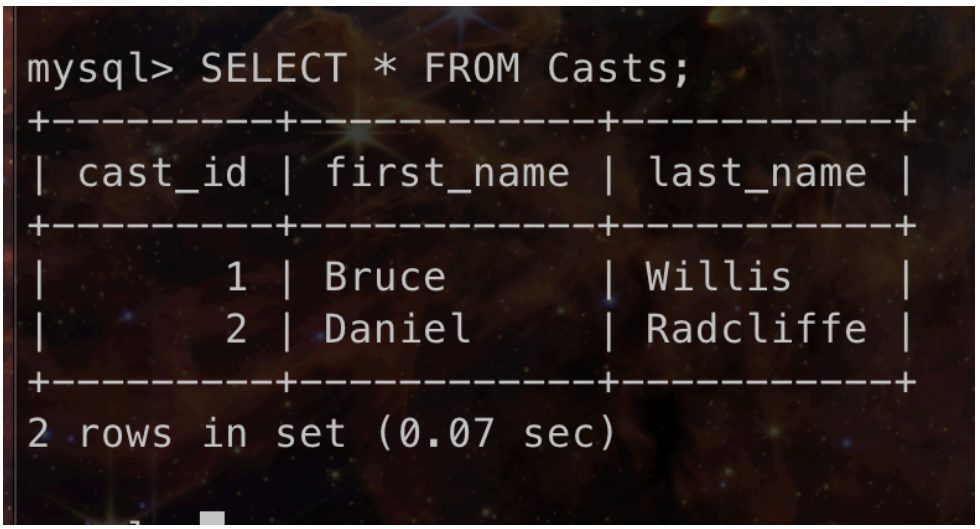
- For **DevOps**, we will deploy the application on Heroku and use GitHub for version control to manage our codebase efficiently.

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



```
mysql> SELECT * FROM Assign_Genres_To_Media;
+----------+----------+
| genre_id | media_id |
+----------+----------+
|        1 |        1 |
|        2 |        2 |
+----------+----------+
2 rows in set (0.05 sec)
```



```
mysql> SELECT * FROM Casts;
+---------+------------+-----------+
| cast_id | first_name | last_name |
+---------+------------+-----------+
|       1 | Bruce      | Willis    |
|       2 | Daniel     | Radcliffe |
+---------+------------+-----------+
2 rows in set (0.07 sec)
```

```
mysql> SELECT * FROM Episodes;
+----------+----------------+---------------+
| media_id | episode_number | episode_title |
+----------+----------------+---------------+
|        2 |              1 | Pilot         |
+----------+----------------+---------------+
1 row in set (0.00 sec)
```

```
mysql> SELECT * FROM Genres;
+----------+-------------+
| genre_id | genre_name  |
+----------+-------------+
|        1 | Action      |
|        2 | Adventure   |
+----------+-------------+
2 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM Media;
+----------+--------------------+------------------------------+--------------+
| media_id | title              | synopsis                     | release_year |
+----------+--------------------+------------------------------+--------------+
|        1 | Die Hard           | Movie about a police officer.  |         2012 |
|        2 | The Big Bang Theory | The show about the scientists. |         2007 |
+----------+--------------------+------------------------------+--------------+
2 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM Movies;
+----------+----------+
| media_id | duration |
+----------+----------+
|        1 |      120 |
+----------+----------+
1 row in set (0.00 sec)
```

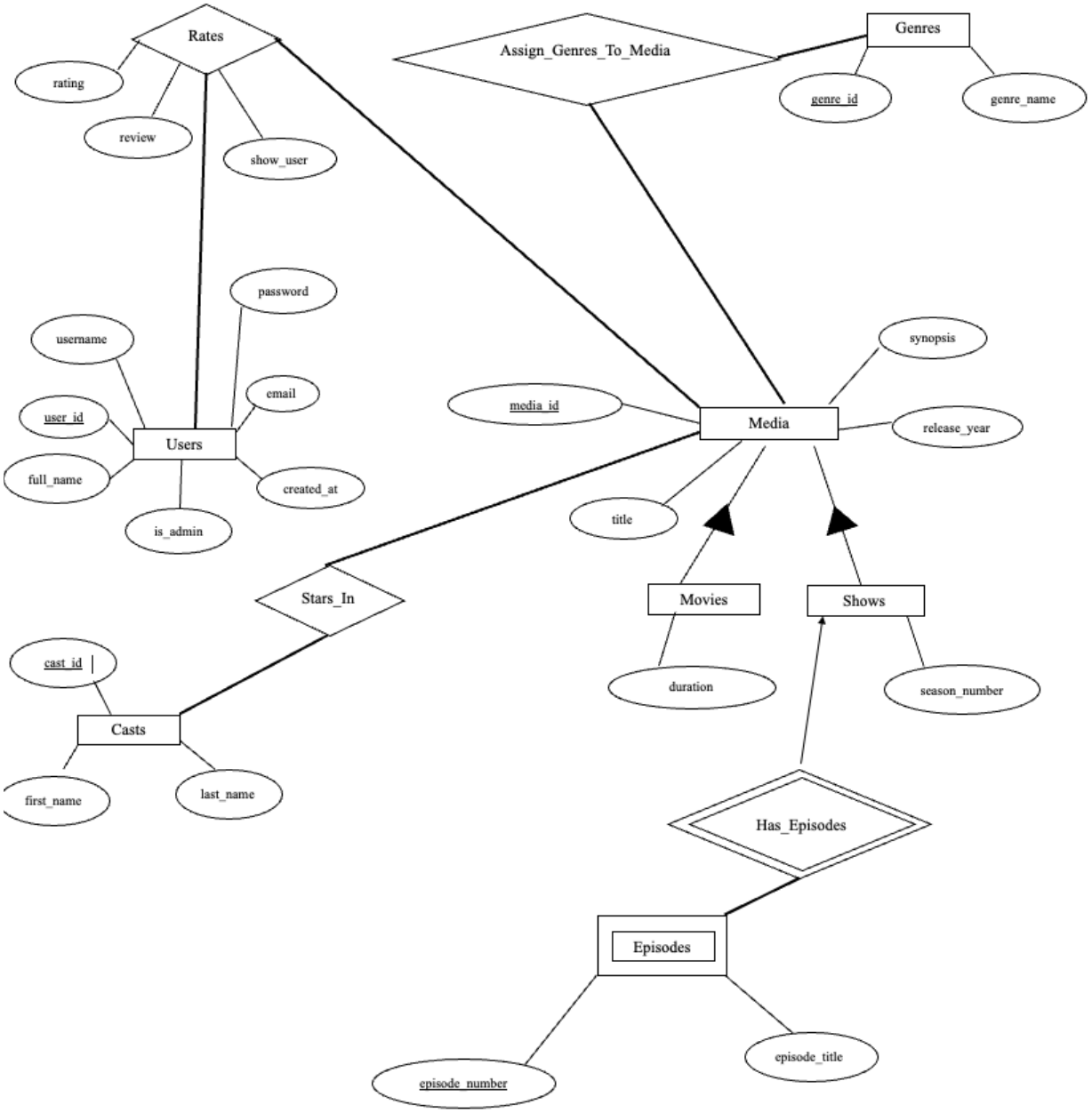
```
mysql> SELECT * FROM Rates;
+---------+----------+--------+-----------------+-----------+
| user_id | media_id | rating | review          | show_user |
+---------+----------+--------+-----------------+-----------+
|       1 |        1 |      5 | Great movie!    |         1 |
|       2 |        2 |      4 | Enjoyable show. |         0 |
+---------+----------+--------+-----------------+-----------+
2 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM Shows;
+----------+---------------+
| media_id | season_number |
+----------+---------------+
|        2 |             1 |
+----------+---------------+
1 row in set (0.00 sec)
```

```
mysql> SELECT * FROM Stars_In;
+---------+----------+
| cast_id | media_id |
+---------+----------+
|       1 |        1 |
|       2 |        2 |
+---------+----------+
2 rows in set (0.00 sec)
```

All the tables above represent the tables that follow our updated ERD schema.

Entity-Relationship Diagram (ERD):



FOURTH NORMAL FORM (4NF)

Fourth Normal Form (4NF) is a level of database normalization designed to reduce redundancy and dependency in a relational database. Achieving 4NF involves addressing multi-valued dependencies, a more complex type of dependency than those addressed in previous normal forms.

A table is in 4NF if:

- It is in Boyce-Codd Normal Form (BCNF).
- It has no multi-valued dependencies.

Multi-Valued Dependency: It occurs when one attribute in a table uniquely determines another attribute, independently of all other attributes. Formally, a multi-valued dependency X->>Y exists in a table if for each X value, there is a well-defined set of Y values, independent of other attributes.

Our tables are already in BCNF and we know if a table is in BCNF, it is already in 3NF and the forms below that. We need to confirm if there are any multi-valued dependencies in our tables.

We have the following relations:

Users(user_id, username, password, full_name, email, is_admin, created_at)
Media(media_id, title, synopsis, release_year)
Movies(media_id, duration)
Shows(media_id, season_number)
Genres(genre_id, genre_name)
Casts(cast_id, first_name, last_name)
Episodes(media_id, episode_number , episode_title)

We have these functional dependencies:

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

media_id -> title, synopsis, release_year

media_id -> duration

media_id -> season_number

genre_id -> genre_name

cast_id -> first_name, last_name

media_id, episode_number -> episode_title

Looking at Normalization through independence:

FD	**Join Key (J)**	**Dependent D = $(J^+ \setminus J)$**	**Independent D = $(A \setminus J^+)$**
user_id -> username, password, full_name, email, is_admin, created_at	user_id	username, password, full_name, email, is_admin, created_at	φ
media_id -> title, synopsis, release_year	media_id	title, synopsis, release_year	φ
media_id -> duration	media_id	duration	φ
media_id -> season_number	media_id	season_number	φ
genre_id -> genre_name	genre_id	genre_name	φ
cast_id -> first_name, last_name	cast_id	first_name, last_name	φ
media_id, episode_number -> episode_title	media_id, episode_number	episode_title	φ

From the table, we can observe there are **no independent attribute** and every key uniquely determines all the attributes of its relation. No attributes exhibit a multi-valued relationship independent of the key.

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

Primary Key: user_id

Dependent Attributes: username, password, full_name, email, is_admin, created_at

Here, we have username, email as unique which is covered in constraints and table creation.

No MVDs. This table is in 4NF.

media_id -> title, synopsis, release_year

Primary Key: media_id

Dependent Attributes: title, synopsis, release_year

No MVDs. This table is in 4NF.

media_id -> duration

Primary Key: media_id

Dependent Attributes: duration

No MVDs. This table is in 4NF.

media_id -> season_number

Primary Key: media_id

Dependent Attributes: season_number

No MVDs. This table is in 4NF.

genre_id -> genre_name

Primary Key: genre_id
Dependent Attributes: genre_name

No MVDs. This table is in 4NF.

cast_id -> first_name, last_name

Primary Key: cast_id
Dependent Attributes: first_name, last_name

No MVDs. This table is in 4NF.

media_id, episode_number -> episode_title

Primary Key: media_id, episode_number
Dependent Attributes: episode_title

No MVDs. This table is in 4NF.

As we can see from above, there are no independent attributes. As from prior sprint, we know our tables were **in BCNF** and we had just checked **there are no multi-valued dependencies**. **Therefore, our tables are in 4NF as proved.**

CHECK CONSTRAINTS AND TRIGGERS

Constraints in a database can be seen as the rules applied to the columns of a table to enforce data integrity and ensure the accuracy and reliability of the data within the database. These constraints help maintain the consistency of data by restricting the types of data that can be stored in a column, the relationships between tables, and the operations that can be performed on the data. In our DBMS, we have added constraints for the following things: Primary Key, Foreign Key, Unique, Default, Cascade and Check.

Brief summary of changes made:

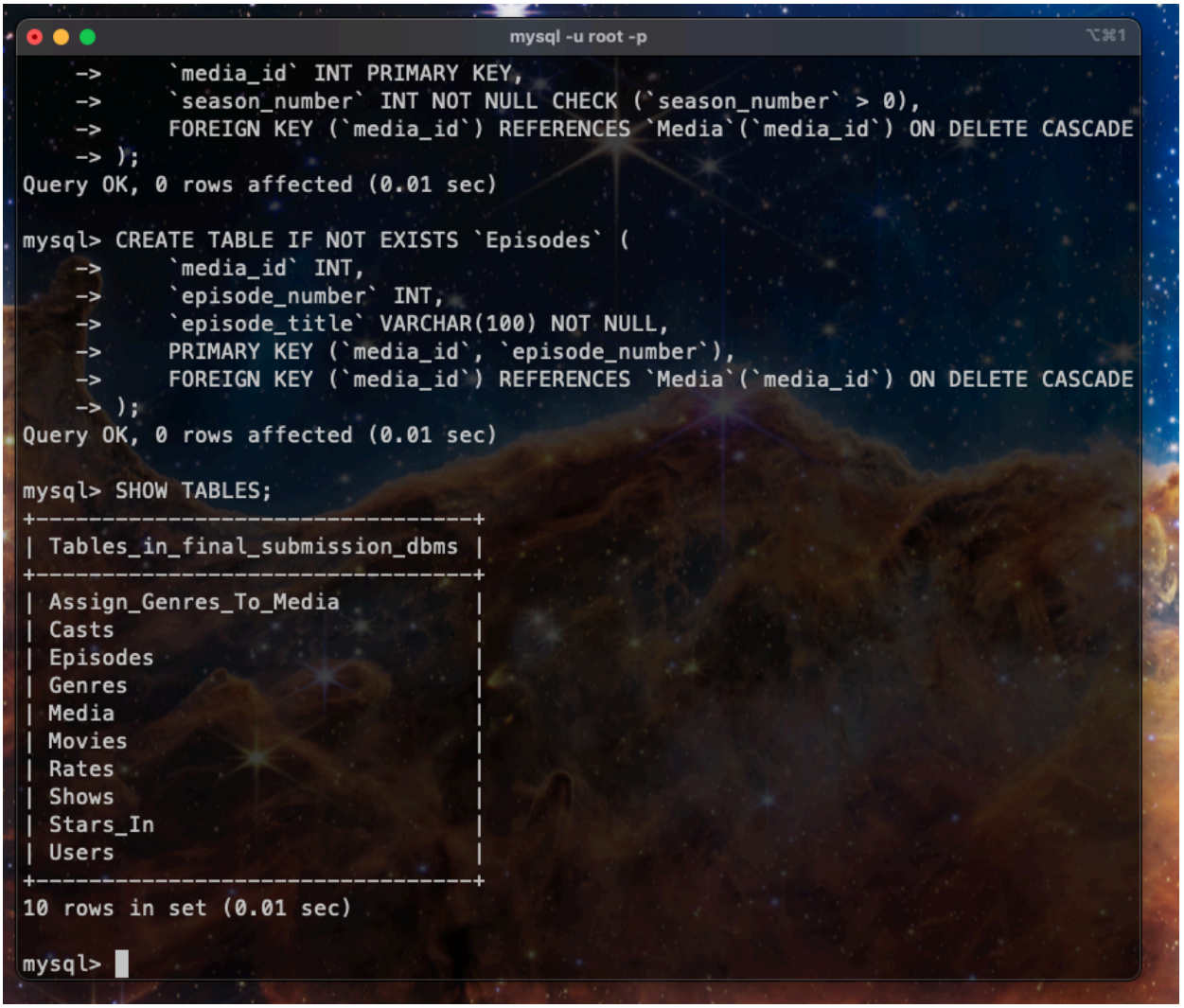
- UNIQUE: For the *username* and *email* in the **Users** table and *genre_name* in the **Genres** table is unique.

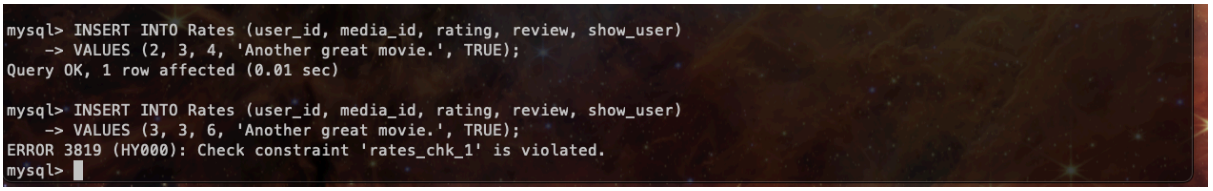
- DEFAULT: Adds default values for columns such as *is_admin* which should be false by default and *created_at*.

- CHECK: Ensures *release_year* is valid, *rating* is within 0 to 5, and *duration* and *season_number* are greater than 0.

- ON DELETE CASCADE: Ensures that related entries are automatically deleted when a referenced entry is deleted, maintaining referential integrity.

We have modified the create table statement as per the new constraints and the sql commands can be found in the “PS_5.sql” file in the repository. Here is just a proof of working with the new statements:





Triggers are special stored procedures in a database that automatically execute or fire when certain events occur. They are associated with a table or view and can be activated by actions such as INSERT, UPDATE, or DELETE.

Trigger types covered in the class:

- BEFORE: Execute before an insert, update, or delete operation.

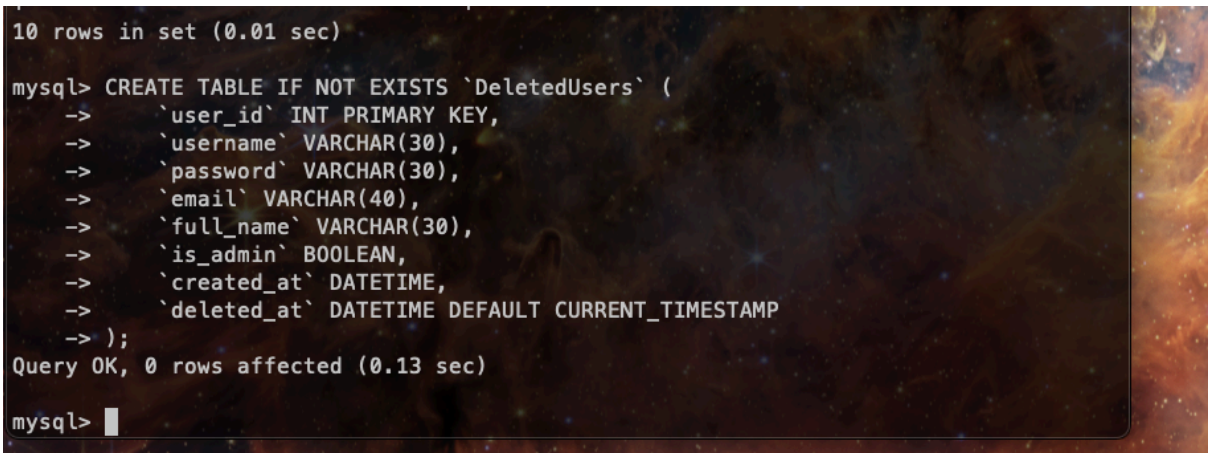
- AFTER: Execute after an insert, update, or delete operation.

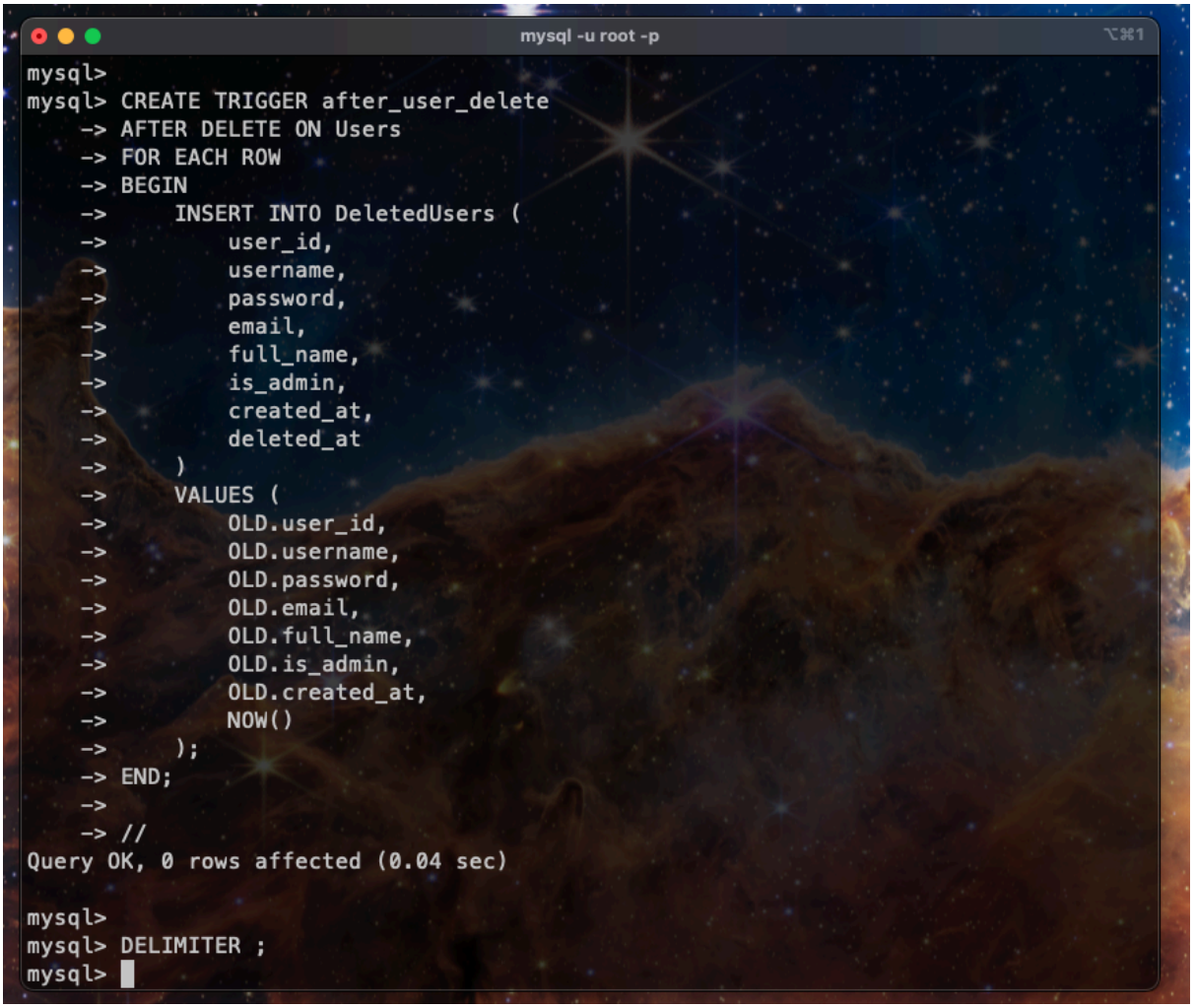
Few use cases that can be applied on our Movie Rating System:

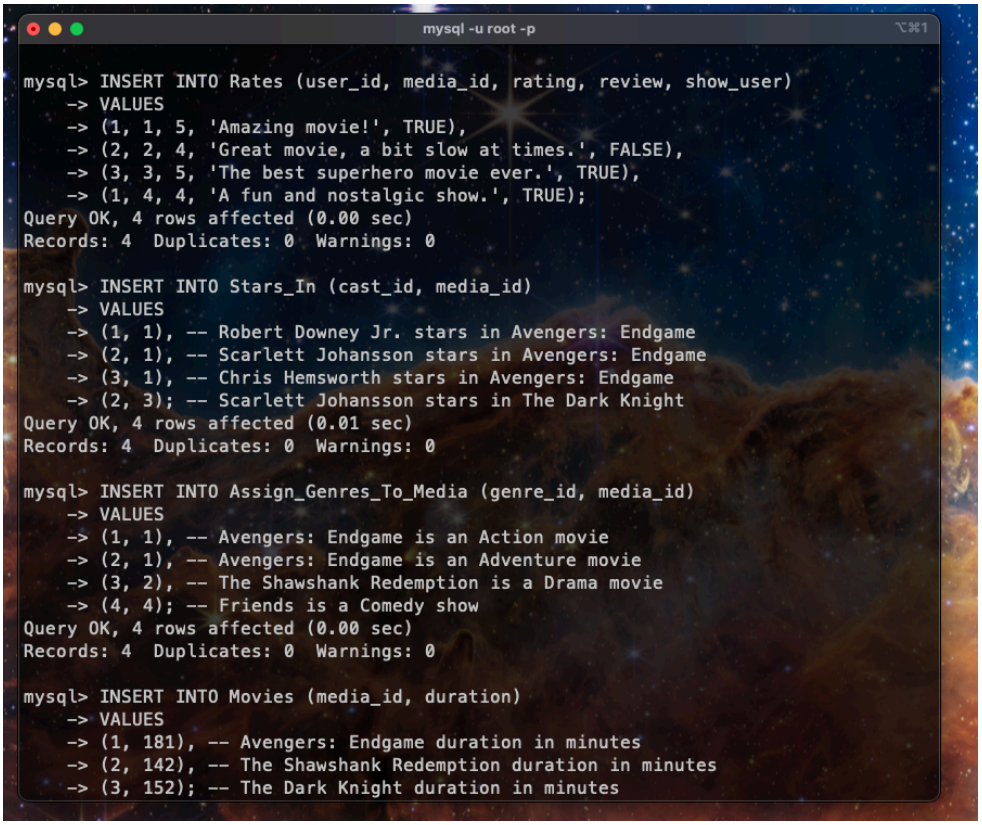
- Whenever a movie, show, or user is deleted, a trigger can be used to copy the deleted record into a backup table. This ensures that we have a record of deletions for auditing or recovery purposes. This can be considered as **Backup use case**.
- When a particular event occurs, such as a high-rating movie being added, trigger can be used to insert notification records into some notifications table. This can then be used to alert users or administrators notifying them about the new media. It can be a **notification** use case. One thing to be taken into account is that this might also be implemented by some other method.
- When a user updates a rating for a movie or show, a trigger can log the old and new ratings along with the timestamp and user information. This helps in tracking changes to ratings over time and can be used for analytics or dispute resolution. This can be useful for **auditing**.

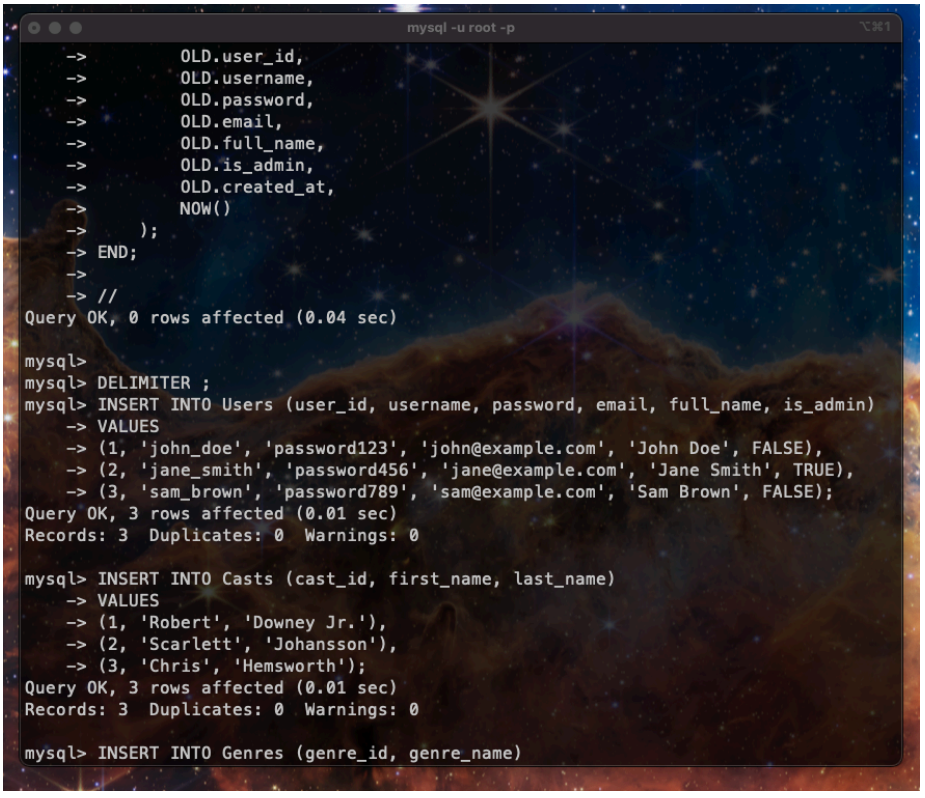
Below are the screenshots showcasing the running of triggers on the Users table to back up the deleted users.

Adding some data as well creating DeletedUsers table and Trigger:



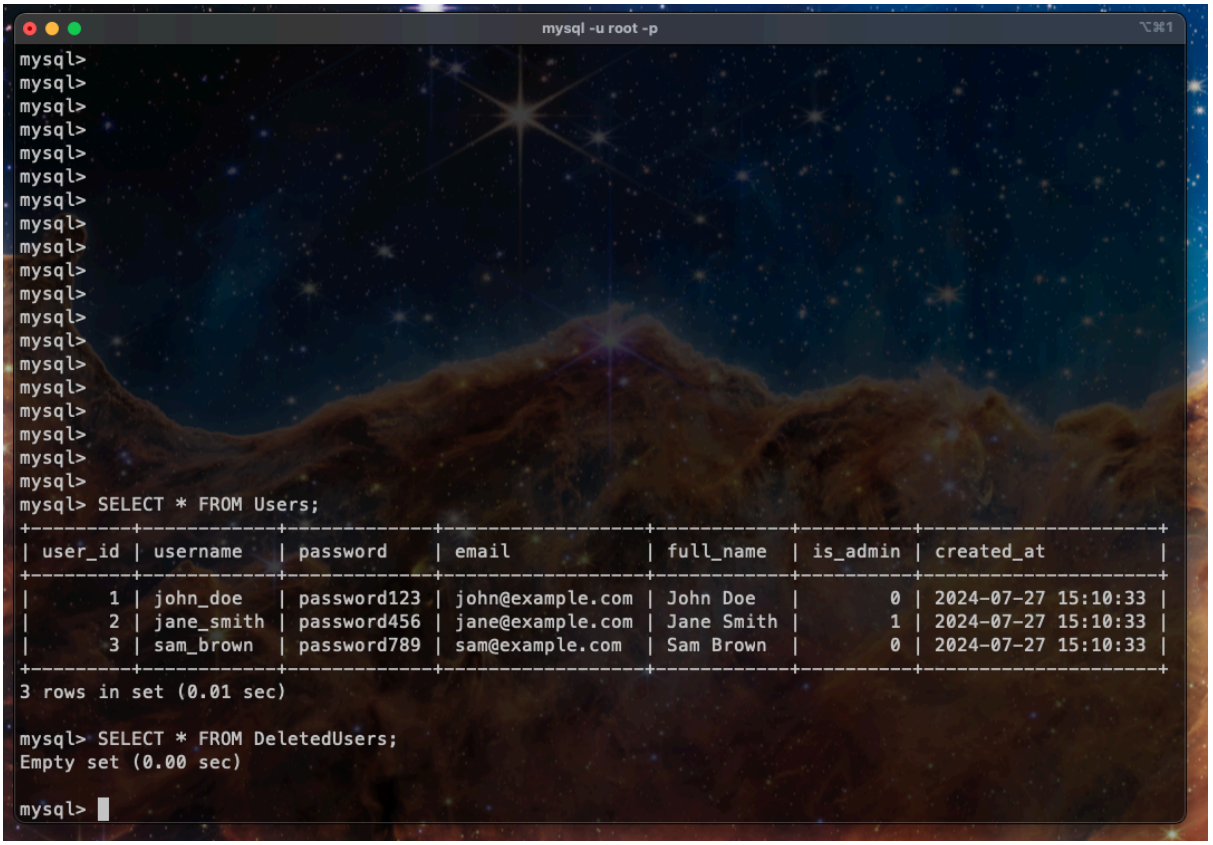

mysql -u root -p
⌥⌘1
mysql>
mysql> CREATE TRIGGER after_user_delete
-> AFTER DELETE ON Users
-> FOR EACH ROW
-> BEGIN
-> INSERT INTO DeletedUsers (
-> user_id,
-> username,
-> password,
-> email,
-> full_name,
-> is_admin,
-> created_at,
-> deleted_at
->)
-> VALUES (
-> OLD.user_id,
-> OLD.username,
-> OLD.password,
-> OLD.email,
-> OLD.full_name,
-> OLD.is_admin,
-> OLD.created_at,
-> NOW()
->);
-> END;
->
-> //
Query OK, 0 rows affected (0.04 sec)
mysql>
mysql> DELIMITER ;
mysql>



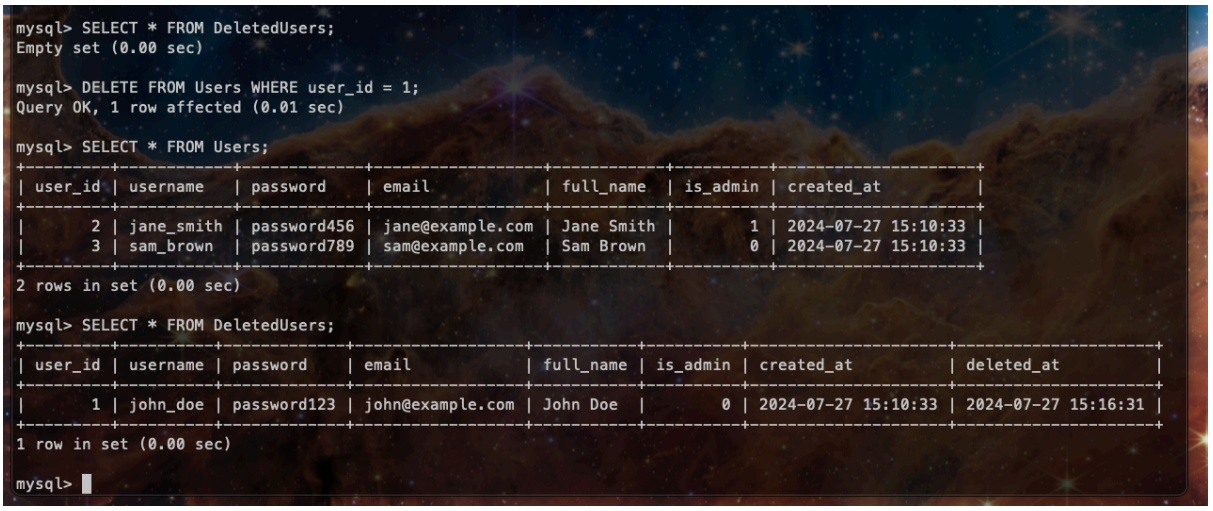


Running of Trigger:

Before –

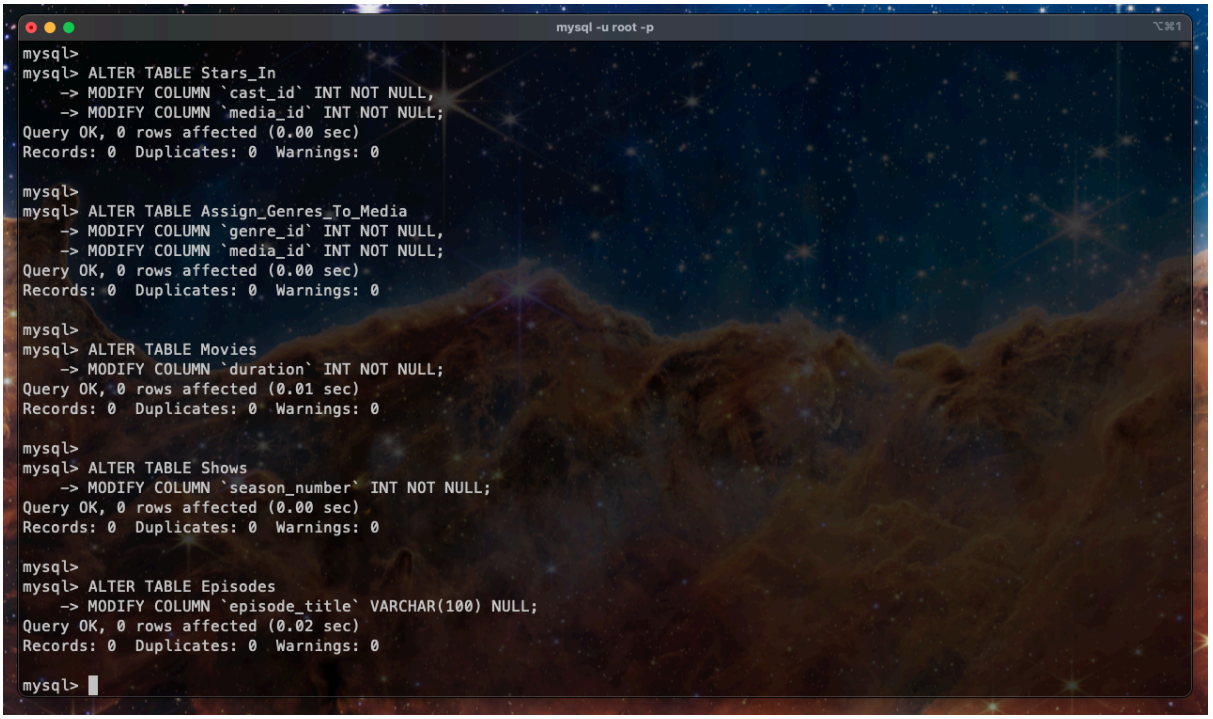

mysql -u root -p
mysql> SELECT * FROM Users;
+---------+------------+-------------+-------------------+------------+----------+---------------------+
| user_id | username | password | email | full_name | is_admin | created_at |
+---------+------------+-------------+-------------------+------------+----------+---------------------+
1	john_doe	password123	john@example.com	John Doe	0	2024-07-27 15:10:33
2	jane_smith	password456	jane@example.com	Jane Smith	1	2024-07-27 15:10:33
3	sam_brown	password789	sam@example.com	Sam Brown	0	2024-07-27 15:10:33
+---------+------------+-------------+-------------------+------------+----------+---------------------+
3 rows in set (0.01 sec)
mysql> SELECT * FROM DeletedUsers;
Empty set (0.00 sec)
mysql>

After –


mysql> SELECT * FROM DeletedUsers;
Empty set (0.00 sec)
mysql> DELETE FROM Users WHERE user_id = 1;
Query OK, 1 row affected (0.01 sec)
mysql> SELECT * FROM Users;
+---------+------------+-------------+-------------------+------------+----------+---------------------+
| user_id | username | password | email | full_name | is_admin | created_at |
+---------+------------+-------------+-------------------+------------+----------+---------------------+
| 2 | jane_smith | password456 | jane@example.com | Jane Smith | 1 | 2024-07-27 15:10:33 |
| 3 | sam_brown | password789 | sam@example.com | Sam Brown | 0 | 2024-07-27 15:10:33 |
+---------+------------+-------------+-------------------+------------+----------+---------------------+
2 rows in set (0.00 sec)
mysql> SELECT * FROM DeletedUsers;
+---------+----------+-------------+-------------------+-----------+----------+---------------------+---------------------+
| user_id | username | password | email | full_name | is_admin | created_at | deleted_at |
+---------+----------+-------------+-------------------+-----------+----------+---------------------+---------------------+
| 1 | john_doe | password123 | john@example.com | John Doe | 0 | 2024-07-27 15:10:33 | 2024-07-27 15:16:31 |
+---------+----------+-------------+-------------------+-----------+----------+---------------------+---------------------+
1 row in set (0.00 sec)
mysql>

NULL VALUES

When designing a relational database, handling NULL values is essential for maintaining data integrity and ensuring accurate query results. NULL values represent missing or unknown data and are distinct from zero or empty strings. Properly defining columns to allow or disallow NULL values, using constraints like NOT NULL, ensures that critical fields are always populated.



Some use cases for our Move Rating System:

- Valid User Data: *username, password, email, full_name* fields in the Users table must not be NULL to ensure each user profile is complete and functional.

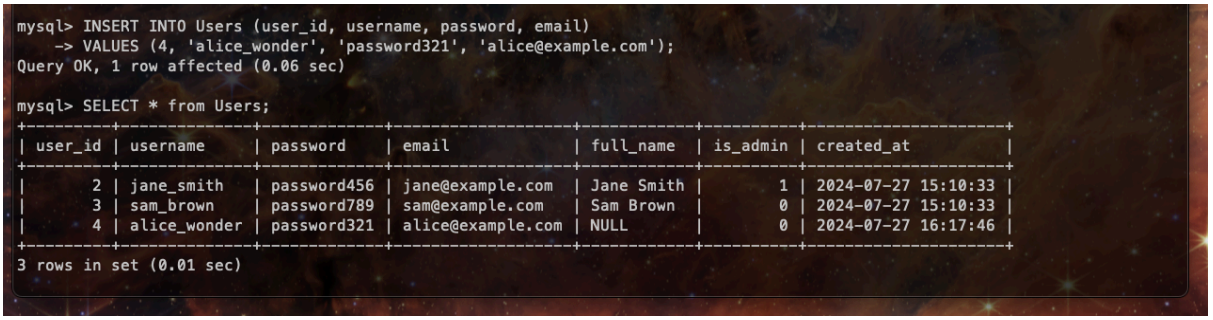
- Optional Review Content: The review field in the Rates table can be NULL to indicate that a user provided a rating but did not leave a review.

- Optional Media Details: The synopsis field in the Media table can be NULL to accommodate entries where the synopsis is not available or needed.

- Differentiating Between No Entry and Empty Entry: Using NULL for fields like review versus an empty string can differentiate between users who didn't provide a review and those who left a blank review.

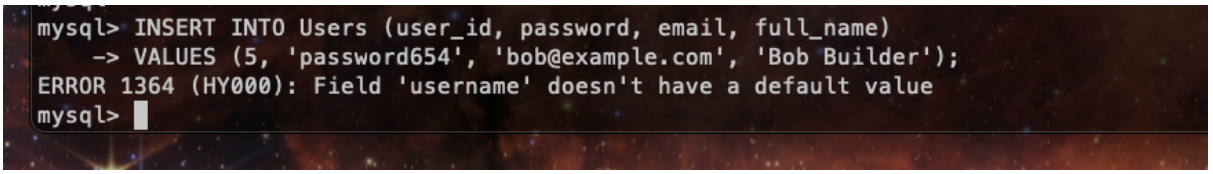
Working example on the Users table:

When Null value **is** allowed:



```
mysql> INSERT INTO Users (user_id, username, password, email)
    -> VALUES (4, 'alice_wonder', 'password321', 'alice@example.com');
Query OK, 1 row affected (0.06 sec)

mysql> SELECT * from Users;
+---------+--------------+-------------+-------------------+------------+----------+---------------------+
| user_id | username     | password    | email             | full_name  | is_admin | created_at          |
+---------+--------------+-------------+-------------------+------------+----------+---------------------+
|       2 | jane_smith   | password456 | jane@example.com  | Jane Smith |        1 | 2024-07-27 15:10:33 |
|       3 | sam_brown    | password789 | sam@example.com   | Sam Brown  |        0 | 2024-07-27 15:10:33 |
|       4 | alice_wonder | password321 | alice@example.com | NULL       |        0 | 2024-07-27 16:17:46 |
+---------+--------------+-------------+-------------------+------------+----------+---------------------+
3 rows in set (0.01 sec)
```

When Null value **is not** allowed and we get an error:



```
mysql> INSERT INTO Users (user_id, password, email, full_name)
    -> VALUES (5, 'password654', 'bob@example.com', 'Bob Builder');
ERROR 1364 (HY000): Field 'username' doesn't have a default value
mysql>
```

Again, all the sql commands can be found in the “PS_5.sql” file in the repository.

TESTING SQL

SQL testing involves validating SQL queries and database interactions to ensure they function correctly.

As learned from the lectures, we are using equivalence partitioning, the testing technique that divides input data into partitions, with each partition containing values expected to be treated similarly by the SQL queries. By selecting representative values from each partition, we can create effective test cases that cover a wide range of input scenarios without the need to test every possible value. Moreover, we must believe in the fact that testing with all inputs is never possible and this is the smart way to make covers most of the tests.

Based on the approach, we have come up with some test cases and will continue to add more as our system grows.

Test Cases

- Users Table

    Valid:

        Insert a user with all required fields.
        Insert a user with *is_admin* set to TRUE.
        Insert a user with *is_admin* set to FALSE.

    Invalid:

        Insert a user with a duplicate *username*.
        Insert a user with a duplicate *email*.
        Insert a user with *username* as NULL.
        Insert a user with *password* as NULL.
        Insert a user with *email* as NULL.
        Insert a user with *full_name* as NULL.

- Casts Table

Valid:

Insert a cast member with all required fields.

Invalid:

Insert a cast member with *first_name* as NULL.

Insert a cast member with *last_name* as NULL.

- Genres Table

Valid:

Insert a genre with a unique *genre_name*.

Invalid:

Insert a genre with a duplicate *genre_name*.

Insert a genre with *genre_name* as NULL.

- Media Table

Valid:

Insert media with all required fields.

Insert media with *synopsis* as NULL.

Invalid:

Insert media with *title* as NULL.

Insert media with *release_year* as NULL.

Insert media with *release_year* less than 1888.

- Rates Table

Valid:

Insert a rate with a valid *rating* between 0 and 5.

Insert a rate with *review* as NULL.

Invalid:

Insert a rate with *rating* less than 0.

Insert a rate with *rating* greater than 5.

Insert a rate with a *user_id* not existing in the Users table.

Insert a rate with a *media_id* not existing in the Media table.

- Stars_In Table

Valid:

Insert a cast-member association with valid *cast_id* and *media_id*.

Invalid:

Insert a cast-member association with a *cast_id* not existing in the Casts table.

Insert a cast-member association with a *media_id* not existing in the Media table.

- Assign_Genres_To_Media Table

Valid:

Assign a genre to media with valid *genre_id* and *media_id*.

Invalid:

Assign a genre to media with a *genre_id* not existing in the Genres table.

Assign a genre to media with a *media_id* not existing in the Media table.

- Movies Table

Valid:

Insert a movie with valid *media_id* and positive *duration*.

Invalid:

Insert a movie with *duration* less than or equal to 0.

Insert a movie with *a media_id* not existing in the Media table.

- Shows Table

    Valid:

        Insert a show with valid *media_id* and positive *season_number*.

    Invalid:

        Insert a show with *season_number* less than or equal to 0.

        Insert a show with a *media_id* not existing in the Media table.

- Episodes Table

    Valid:

        Insert an episode with valid *media_id* and positive *episode_number*.

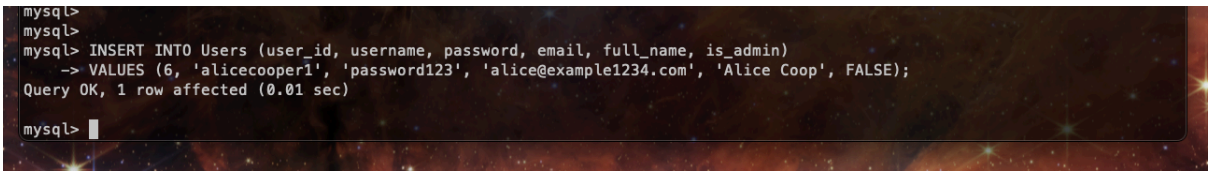
        Insert an episode with *episode_title* as NULL.

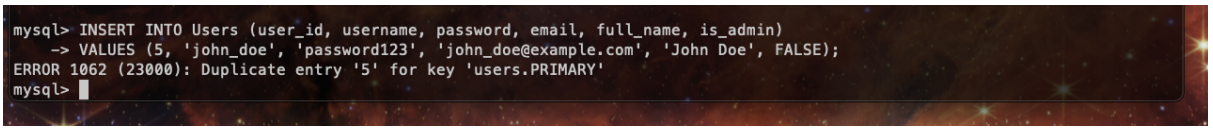
    Invalid:

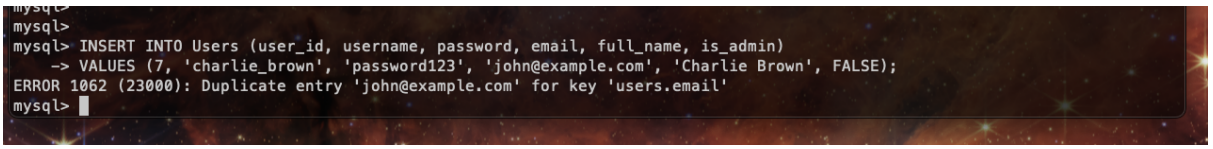
        Insert an episode with *media_id* not existing in the Media table.

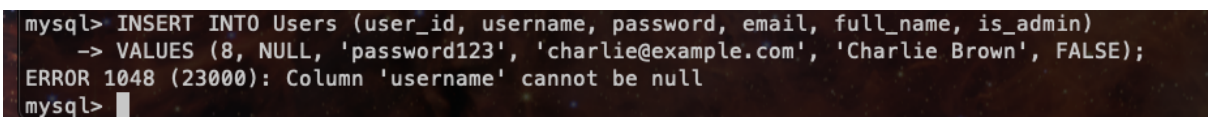
        Insert an episode with negative *episode_number*.

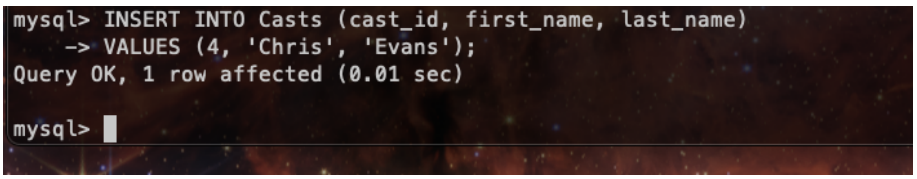
The commands for all these can be found in the sql commands document in the repo. Below are some working examples.

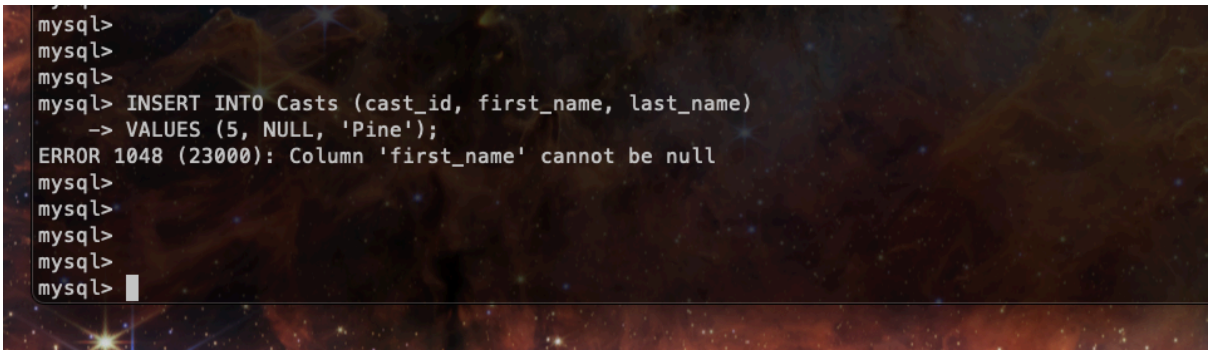












Considering time for this project, these are just few testing cases covered by us. There can be a lot more testing that can be done in the future taking into account the time.

SIMPLIFYING QUERIES

Simplifying and optimizing SQL queries in our project helps our database run faster and more efficiently. It makes sure the queries use less memory and processing power, which means our application can handle more users and data without slowing down.

Simpler queries are easier to understand and maintain, making it easier to fix bugs and add new features.

As learned from lectures, we can use several operators to simplify the queries.

- Composing Queries

  This query retrieves the titles and ratings of media that have a rating greater than 4 and were released after the year 2000. It joins the Media and Rates tables based on media_id.

  Original Query:

Composed Query:

```
mysql> SELECT recent_media.title, high_ratings.rating
    -> FROM (
    ->     SELECT media_id, rating
    ->     FROM Rates
    ->     WHERE rating > 4
    -> ) AS high_ratings
    -> JOIN (
    ->     SELECT media_id, title
    ->     FROM Media
    ->     WHERE release_year > 2000
    -> ) AS recent_media ON high_ratings.media_id = recent_media.media_id;
+-----------------+--------+
| title           | rating |
+-----------------+--------+
| The Dark Knight |      5 |
+-----------------+--------+
1 row in set (0.00 sec)

mysql>
```

This final query joins the results of the two sub-queries to get the titles and ratings of media that have high ratings and are recent. Overall, composing queries involves breaking down a complex SQL query into simpler sub-queries and then combining their results making the overall query easier to understand and optimize.

- Associativity

This query retrieves the titles of media, their ratings, and the first and last names of the cast members for media that have a rating greater than 4. It performs multiple joins between Media, Rates, Stars_In, and Casts.

Original Query:

```
mysql>
mysql>
mysql>
mysql> SELECT M.title, R.rating, C.first_name, C.last_name
    -> FROM Media M
    -> JOIN Rates R ON M.media_id = R.media_id
    -> JOIN Stars_In S ON M.media_id = S.media_id
    -> JOIN Casts C ON S.cast_id = C.cast_id
    -> WHERE R.rating > 4;
+-----------------+--------+------------+-----------+
| title           | rating | first_name | last_name |
+-----------------+--------+------------+-----------+
| The Dark Knight |      5 | Scarlett   | Johansson |
+-----------------+--------+------------+-----------+
1 row in set (0.07 sec)

mysql>
mysql>
mysql>
mysql>
mysql>
```

Optimized:

```
mysql>
mysql>
mysql> SELECT M.title, R.rating, C.first_name, C.last_name
    -> FROM Rates R
    -> JOIN Media M ON R.media_id = M.media_id
    -> JOIN Stars_In S ON M.media_id = S.media_id
    -> JOIN Casts C ON S.cast_id = C.cast_id
    -> WHERE R.rating > 4;
+-----------------+--------+------------+-----------+
| title           | rating | first_name | last_name |
+-----------------+--------+------------+-----------+
| The Dark Knight |      5 | Scarlett   | Johansson |
+-----------------+--------+------------+-----------+
1 row in set (0.00 sec)

mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
```

This rearranged query starts with the Rates table, and then joins it with Media, Stars_In, and Casts. It can help reduce the number of rows earlier in the query and improve performance. Overall, associativity of relational operators allows us to

rearrange joins in a way that optimizes query performance by starting with the most filtered tables.

- Distributivity

This query retrieves the titles of media and their genres for media that are in the 'Action' genre and were released after the year 2000. It joins Media, Assign_Genres_To_Media, and Genres.

Original Query:

Simplified Using Distributive Law:

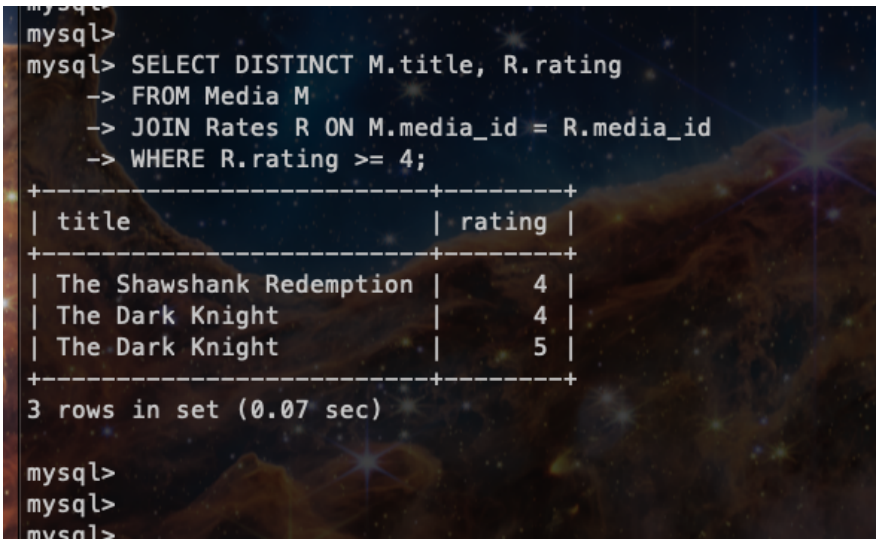
```
mysql>
mysql> SELECT M.title, G.genre_name
    -> FROM (
    ->     SELECT media_id, title, release_year
    ->     FROM Media
    ->     WHERE release_year > 2000
    -> ) M
    -> JOIN Assign_Genres_To_Media AGM ON M.media_id = AGM.media_id
    -> JOIN (
    ->     SELECT genre_id, genre_name
    ->     FROM Genres
    ->     WHERE genre_name = 'Action'
    -> ) G ON AGM.genre_id = G.genre_id;
+------------------+------------+
| title            | genre_name |
+------------------+------------+
| Avengers: Endgame | Action     |
+------------------+------------+
1 row in set (0.00 sec)

mysql>
```

This query applies the conditions in sub-queries to filter the rows before joining them. The sub-query for Media filters for release year, and the sub-query for Genres filters for genre name. Then, these sub-queries are joined. Using the distributive laws for relational operators, we can apply conditions in sub-queries to filter rows before joining, simplifying and optimizing our SQL queries.

- Equivalent

  This query retrieves distinct titles of media and their ratings where the rating is greater than or equal to 4.

  Original:



```
mysql>
mysql> SELECT DISTINCT M.title, R.rating
    -> FROM Media M
    -> JOIN Rates R ON M.media_id = R.media_id
    -> WHERE R.rating >= 4;
+--------------------------+--------+
| title                    | rating |
+--------------------------+--------+
| The Shawshank Redemption |      4 |
| The Dark Knight          |      4 |
| The Dark Knight          |      5 |
+--------------------------+--------+
3 rows in set (0.07 sec)

mysql>
mysql>
```

Using Equivalent Operators:

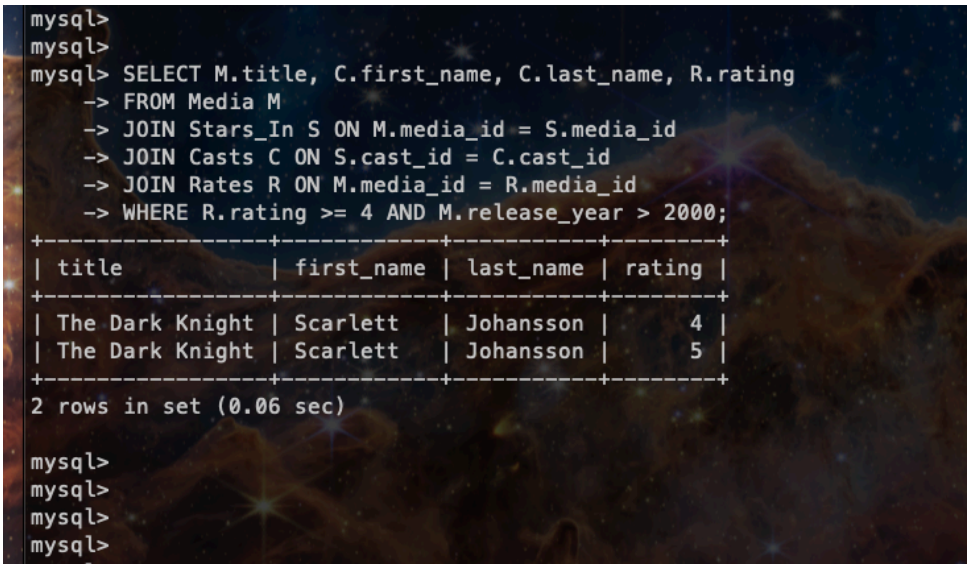
```
mysql>
mysql>
mysql>
mysql> SELECT M.title, R.rating
    -> FROM Media M
    -> JOIN Rates R ON M.media_id = R.media_id
    -> WHERE R.rating >= 4
    -> GROUP BY M.title, R.rating;
+--------------------------+--------+
| title                    | rating |
+--------------------------+--------+
| The Shawshank Redemption |      4 |
| The Dark Knight          |      4 |
| The Dark Knight          |      5 |
+--------------------------+--------+
3 rows in set (0.00 sec)

mysql>
mysql>
mysql>
```

This query replaces DISTINCT with GROUP BY, which can make use of indexes and improve performance. Therefore, equivalent operators helps us to choose alternative SQL constructs to maintain the same functionality but might have to get better performance or readability.

Using some of the above concepts, we can simplify and optimize our queries. One more example for some combined operators can be:

This query retrieves the titles of media, cast members' names, and ratings for media that have a rating of 4 or higher and were released after the year 2000.

Original:



```
mysql>
mysql>
mysql> SELECT M.title, C.first_name, C.last_name, R.rating
    -> FROM Media M
    -> JOIN Stars_In S ON M.media_id = S.media_id
    -> JOIN Casts C ON S.cast_id = C.cast_id
    -> JOIN Rates R ON M.media_id = R.media_id
    -> WHERE R.rating >= 4 AND M.release_year > 2000;
+-----------------+------------+-----------+--------+
| title           | first_name | last_name | rating |
+-----------------+------------+-----------+--------+
| The Dark Knight | Scarlett   | Johansson |      4 |
| The Dark Knight | Scarlett   | Johansson |      5 |
+-----------------+------------+-----------+--------+
2 rows in set (0.06 sec)

mysql>
mysql>
mysql>
mysql>
```

After:

```
mysql> WITH recent_media AS (
    ->     SELECT media_id, title
    ->     FROM Media
    ->     WHERE release_year > 2000
    -> ),
    -> -- Sub-query for high ratings
    -> high_ratings AS (
    ->     SELECT media_id, rating
    ->     FROM Rates
    ->     WHERE rating >= 4
    -> )
    -> -- Final query
    -> SELECT rm.title, C.first_name, C.last_name, hr.rating
    -> FROM recent_media rm
    -> JOIN high_ratings hr ON rm.media_id = hr.media_id
    -> JOIN Stars_In S ON rm.media_id = S.media_id
    -> JOIN Casts C ON S.cast_id = C.cast_id;
+-----------------+------------+-----------+--------+
| title           | first_name | last_name | rating |
+-----------------+------------+-----------+--------+
| The Dark Knight | Scarlett   | Johansson |      4 |
| The Dark Knight | Scarlett   | Johansson |      5 |
+-----------------+------------+-----------+--------+
2 rows in set (0.00 sec)

mysql>
mysql>
mysql>
mysql>
```

This optimized query first filters the media based on release year and ratings in sub-queries, then joins the filtered results with Stars_In and Casts to get the desired information.

All the sql commands can be found in the “PS_5.sql” file.