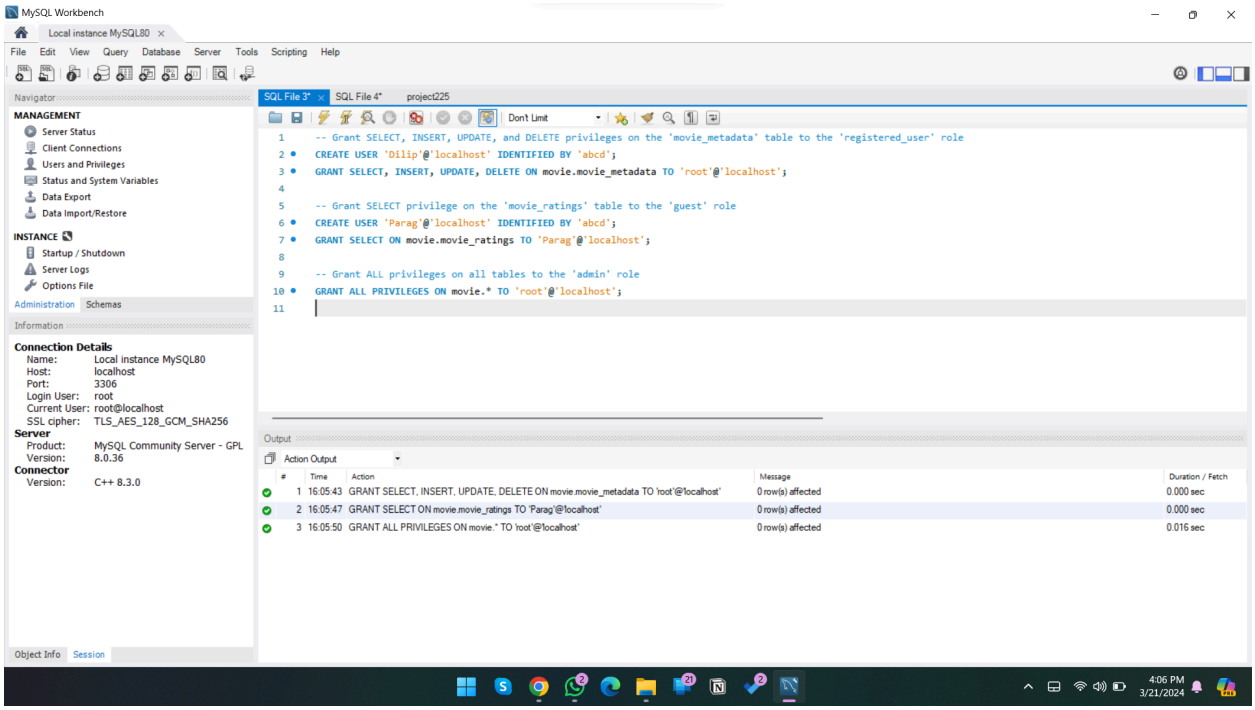
SQL Queries And Output
Lab - 1

- Access Privileges:

```
-- Grant SELECT, INSERT, UPDATE, and DELETE privileges on the 'movie_metadata' table to 'Dilip'
CREATE USER 'Dilip'@'localhost' IDENTIFIED BY 'abcd';
GRANT SELECT, INSERT, UPDATE, DELETE ON movie.movie_metadata TO 'Dilip'@'localhost';

-- Grant SELECT privilege on the 'movie_ratings' table to user 'Parag'
CREATE USER 'Parag'@'localhost' IDENTIFIED BY 'abcd';
GRANT SELECT ON movie.movie_ratings TO 'Parag'@'localhost';

-- Grant ALL privileges on all tables to the 'admin' role
GRANT ALL PRIVILEGES ON movie.* TO 'root'@'localhost';
```

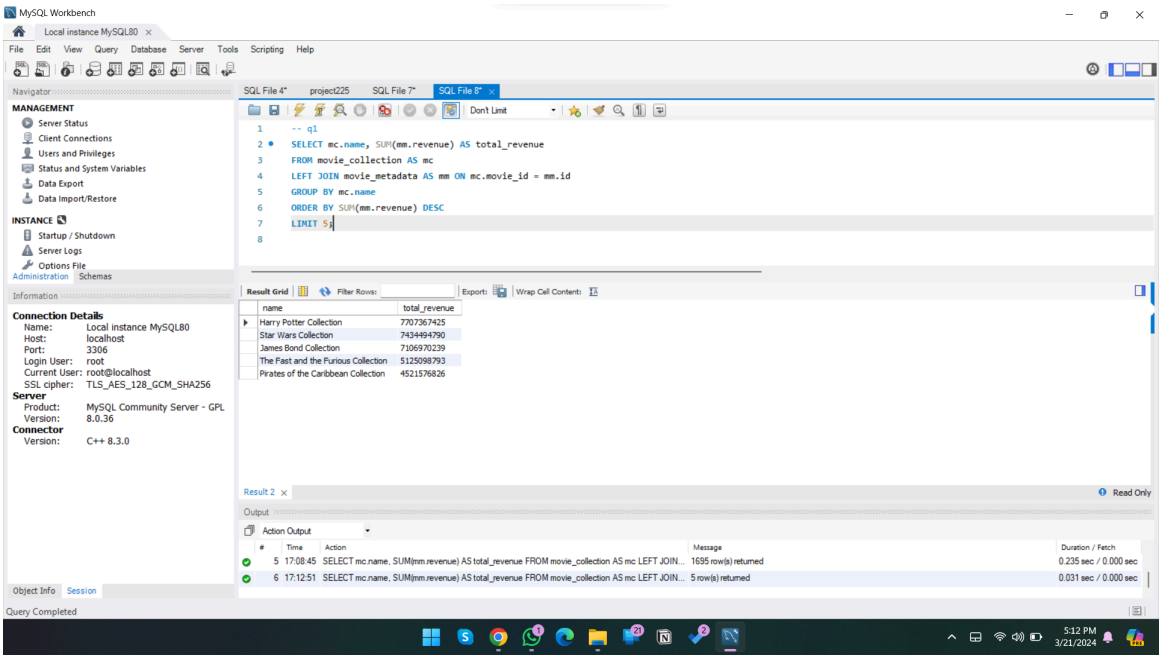
Output:



- SQL Code Queries:
    1. Get the top 5 movies with total revenue for each movie

```
SELECT mc.name, SUM(mm.revenue) AS total_revenue
FROM movie_collection AS mc
LEFT JOIN movie_metadata AS mm ON mc.movie_id = mm.id
GROUP BY mc.name
ORDER BY SUM(mm.revenue) DESC
LIMIT 5;
```

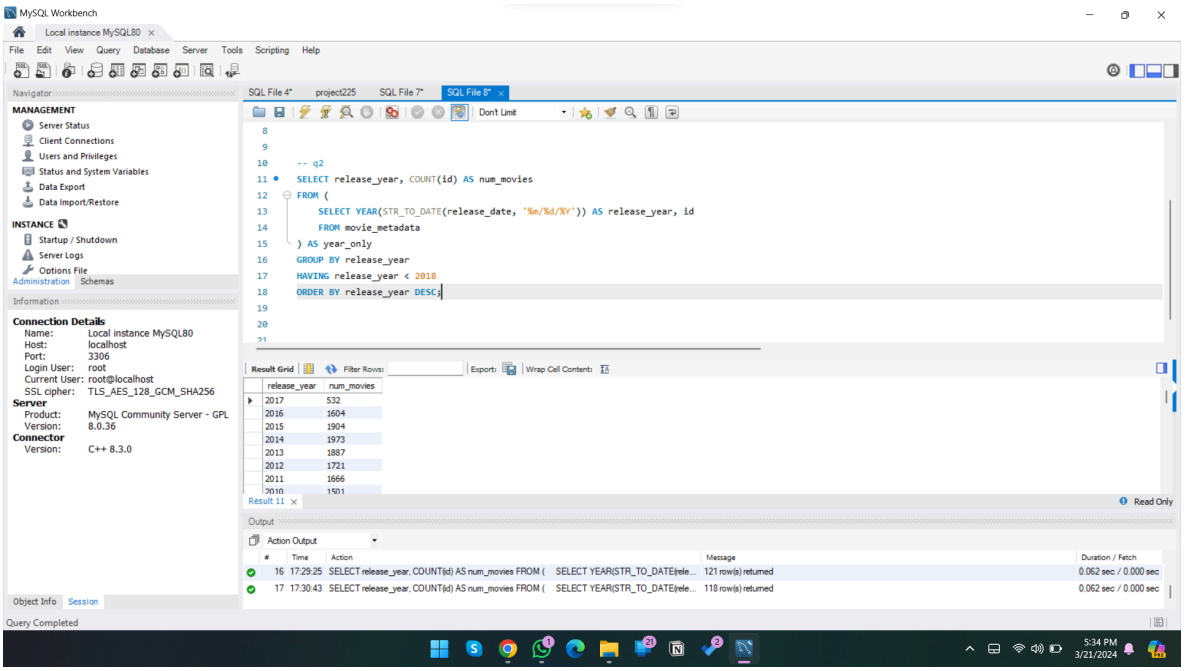
Output:



2. Count the number of movies released each year before 2018 and order by release year in descending.

```
SELECT release_year, COUNT(id) AS num_movies
FROM (
    SELECT YEAR(STR_TO_DATE(release_date, '%m/%d/%Y')) AS
release_year, id
    FROM movie_metadata
) AS year_only
GROUP BY release_year
HAVING release_year < 2018
ORDER BY release_year DESC;
```

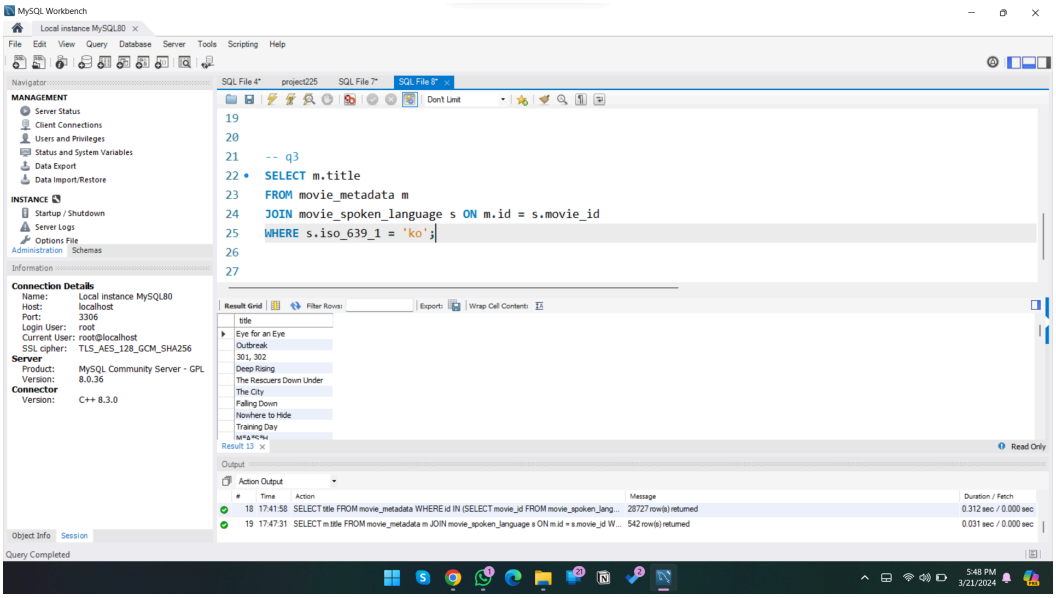
Output:



3. Get the movies in which the spoken languages is korean

```
SELECT m.title
FROM movie_metadata m
JOIN movie_spoken_language s ON m.id = s.movie_id
WHERE s.iso_639_1 = 'ko';
```

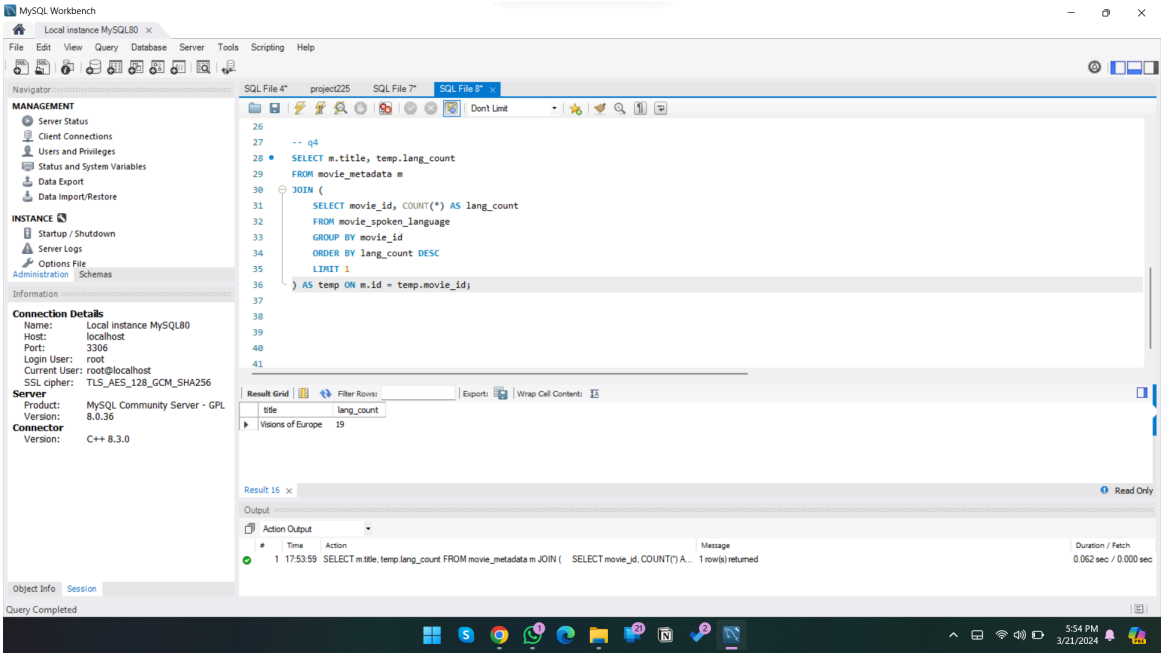
Output:



4. Select the movie which has highest number of spoken languages

```
SELECT m.title, temp.lang_count
FROM movie_metadata m
JOIN (
    SELECT movie_id, COUNT(*) AS lang_count
    FROM movie_spoken_language
    GROUP BY movie_id
    ORDER BY lang_count DESC
    LIMIT 1
) AS temp ON m.id = temp.movie_id;
```

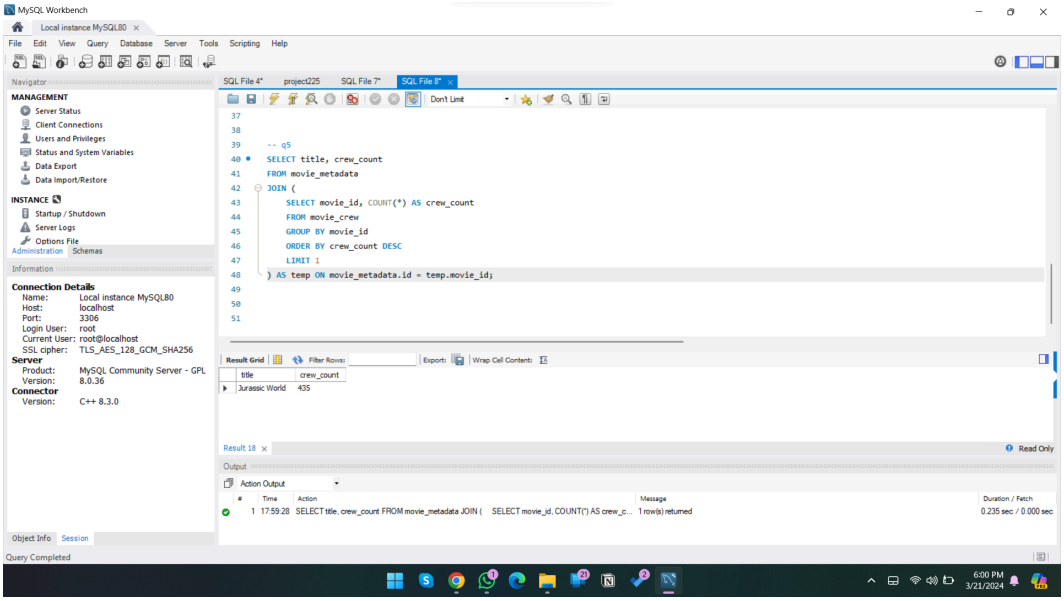
Output:



5. Select movie with highest number of cast members

```
SELECT title, crew_count
FROM movie_metadata
JOIN (
    SELECT movie_id, COUNT(*) AS crew_count
    FROM crew
    GROUP BY movie_id
    ORDER BY crew_count DESC
    LIMIT 1
) AS temp ON movie_metadata.id = temp.movie_id;
```

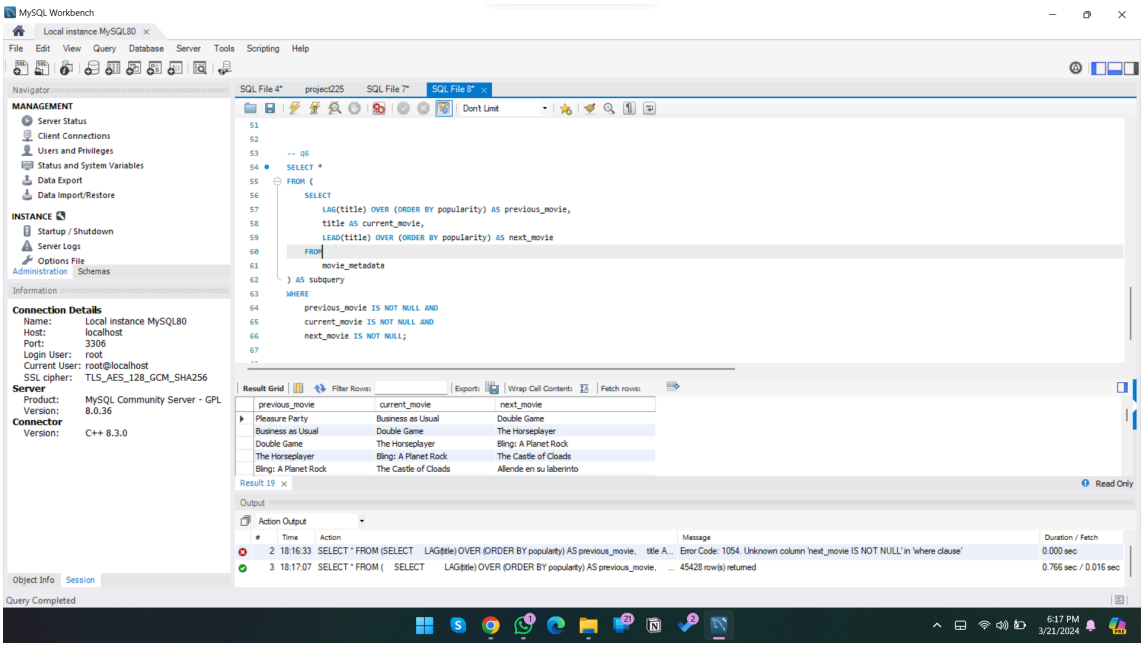
Output:



6. Retrieve the previous, current, and next movie titles based on their popularity.

```
SELECT * FROM (SELECT
    LAG(title) OVER (ORDER BY popularity) AS previous_movie,
    title AS current_movie,
    LEAD(title) OVER (ORDER BY popularity) AS next_movie
FROM
    movie_metadata) AS subquery
WHERE previous_movie IS NOT NULL AND current_movie IS NOT NULL
AND next_movie IS NOT NULL;
```

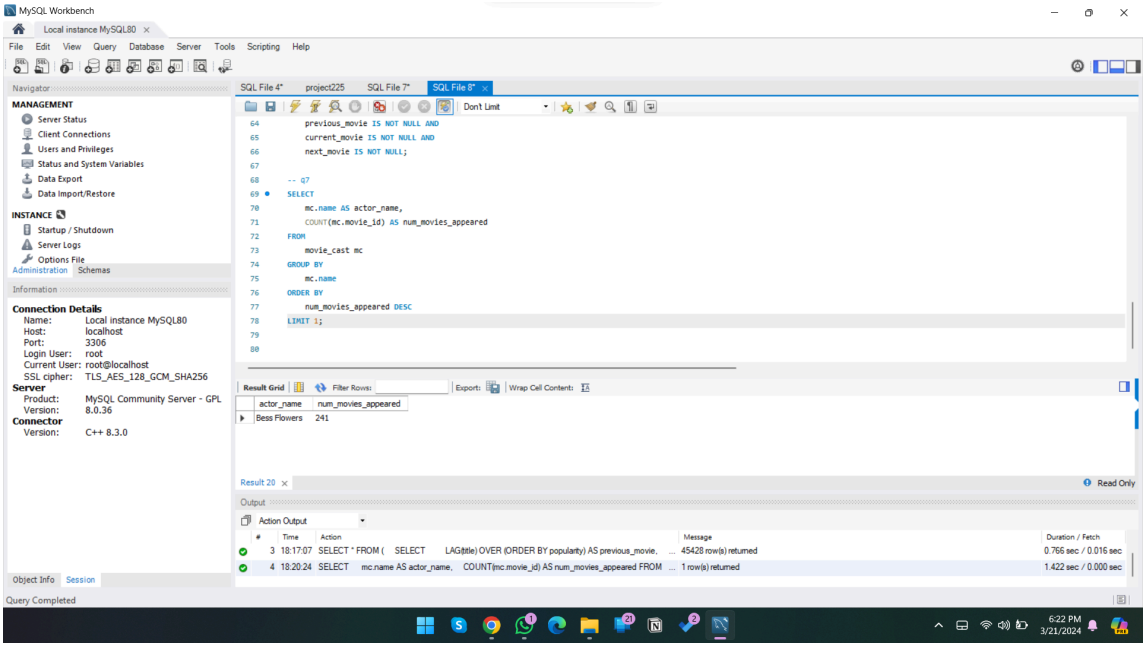
Output:



7. Identify the actor/actress who have appeared in the most movies

```
SELECT
    mc.name AS actor_name,
    COUNT(mc.movie_id) AS num_movies_appeared
FROM
    movie_cast mc
GROUP BY
    mc.name
ORDER BY
    num_movies_appeared DESC
LIMIT 1;
```

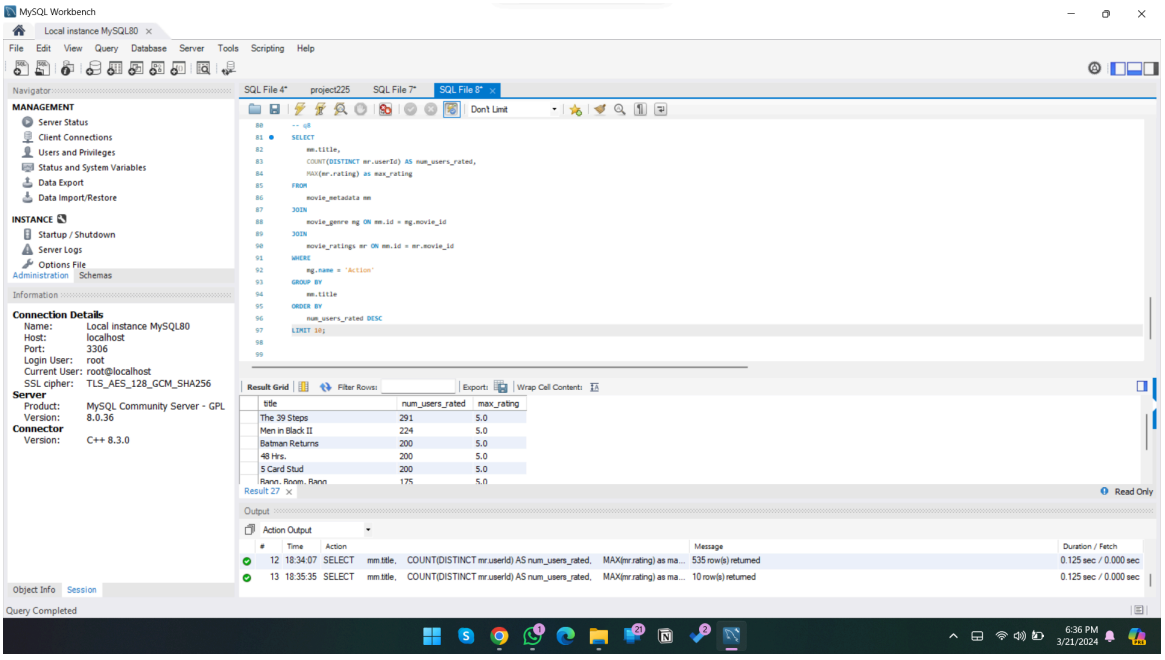
Output:



8. Top 10 movies with highest rating with most numbers of users who have rated the movie

```
SELECT
    mm.title,
    COUNT(DISTINCT mr.userId) AS num_users_rated,
    MAX(mr.rating) as max_rating
FROM
    movie_metadata mm
JOIN
    movie_genre mg ON mm.id = mg.movie_id
JOIN
    movie_ratings mr ON mm.id = mr.movie_id
WHERE
    mg.name = 'Action'
GROUP BY
    mm.title
ORDER BY
    num_users_rated DESC
```

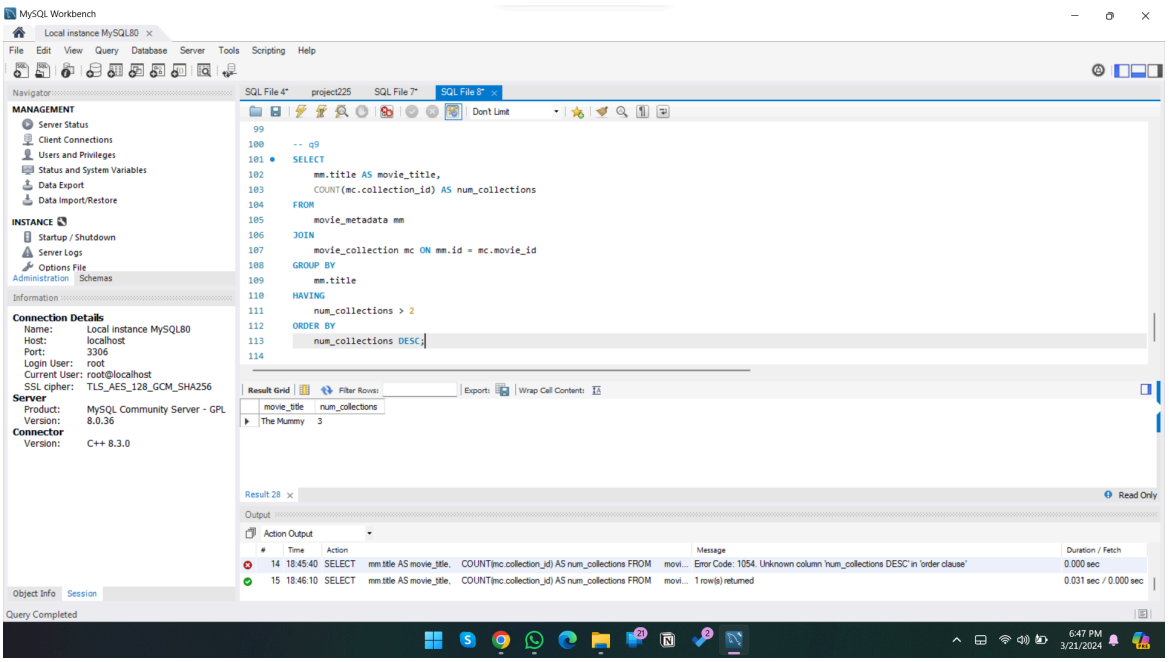
Output:



9. Display movies having greater than 2 collections

```
SELECT
    mm.title AS movie_title,
    COUNT(mc.collection_id) AS num_collections
FROM
    movie_metadata mm
JOIN
    movie_collection mc ON mm.id = mc.movie_id
GROUP BY
    mm.title
HAVING
    num_collections > 2
ORDER BY
    num_collections DESC;
```

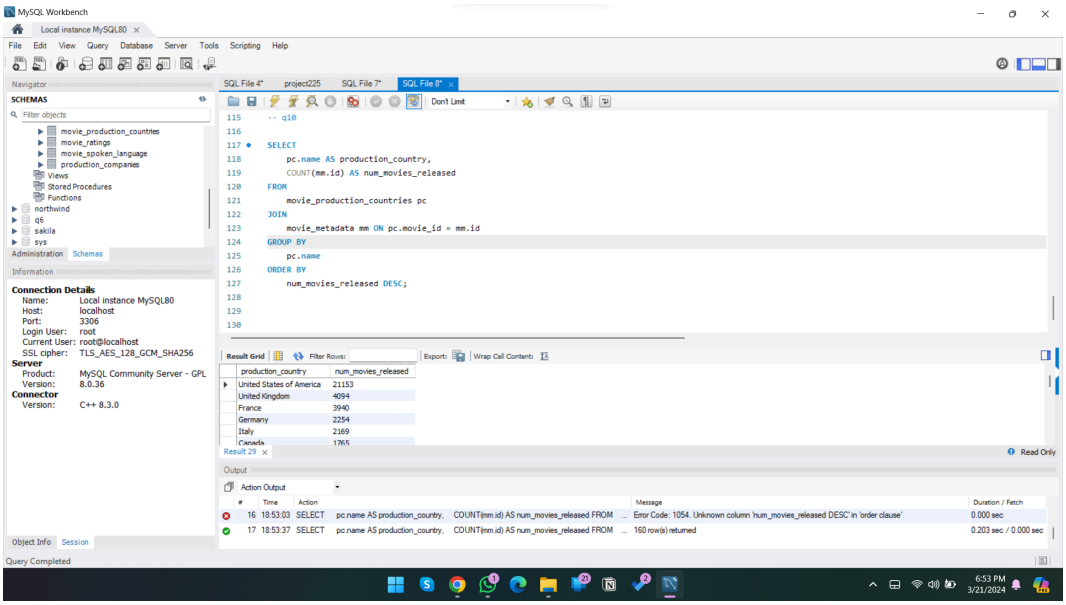
Output:



10. Display the total number of movies each country has released.

```
SELECT
    pc.name AS production_country,
    COUNT(mm.id) AS num_movies_released
FROM
    movie_production_countries pc
JOIN
    movie_metadata mm ON pc.movie_id = mm.id
GROUP BY
    pc.name
ORDER BY
    num_movies_released DESC;
```

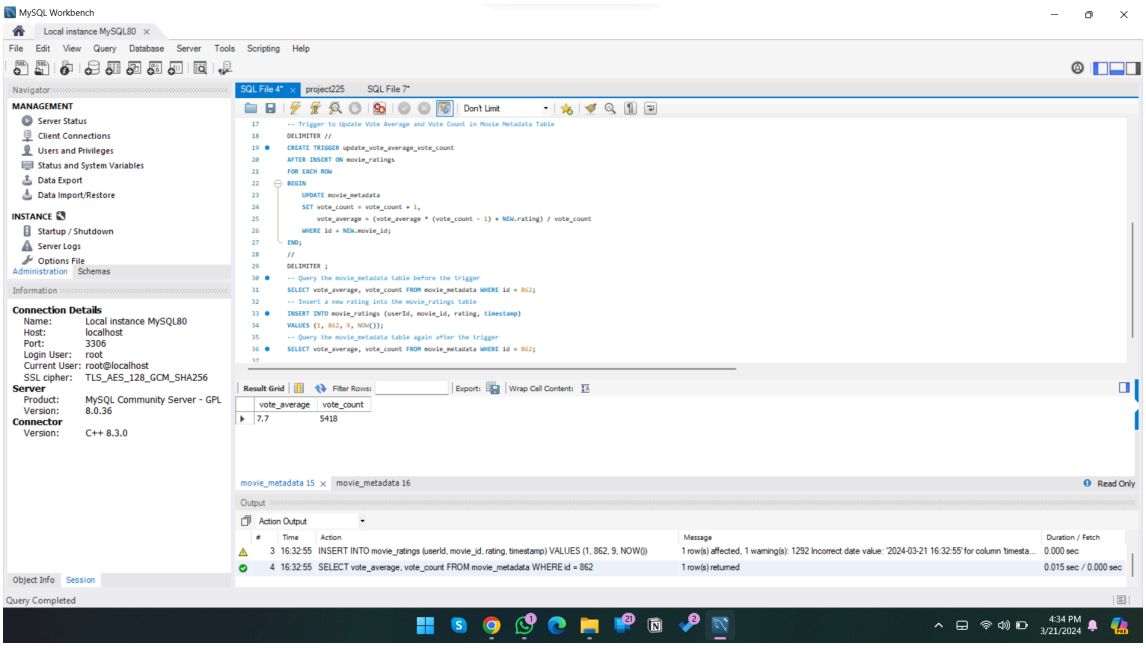
Output:



- Triggers:
    1. Trigger to Update Vote Average and Vote Count in Movie Metadata Table.

```
DELIMITER //
CREATE TRIGGER update_vote_average_vote_count
AFTER INSERT ON movie_ratings
FOR EACH ROW
BEGIN
    UPDATE movie_metadata
    SET vote_count = vote_count + 1,
        vote_average = (vote_average * (vote_count - 1) + NEW.rating) / vote_count
    WHERE id = NEW.movie_id;
END;
//
DELIMITER ;
-- Query the movie_metadata table before the trigger
SELECT vote_average, vote_count FROM movie_metadata WHERE id = 862;
-- Insert a new rating into the movie_ratings table
INSERT INTO movie_ratings (userId, movie_id, rating, timestamp)
VALUES (1, 862, 9, NOW());
-- Query the movie_metadata table again after the trigger
SELECT vote_average, vote_count FROM movie_metadata WHERE id = 862;
```

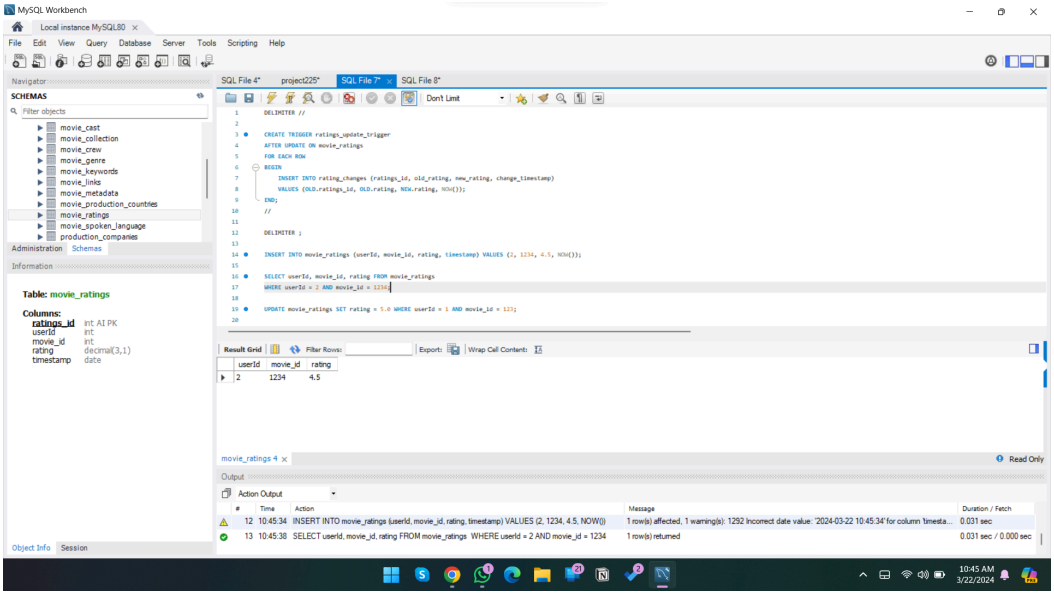
Output:



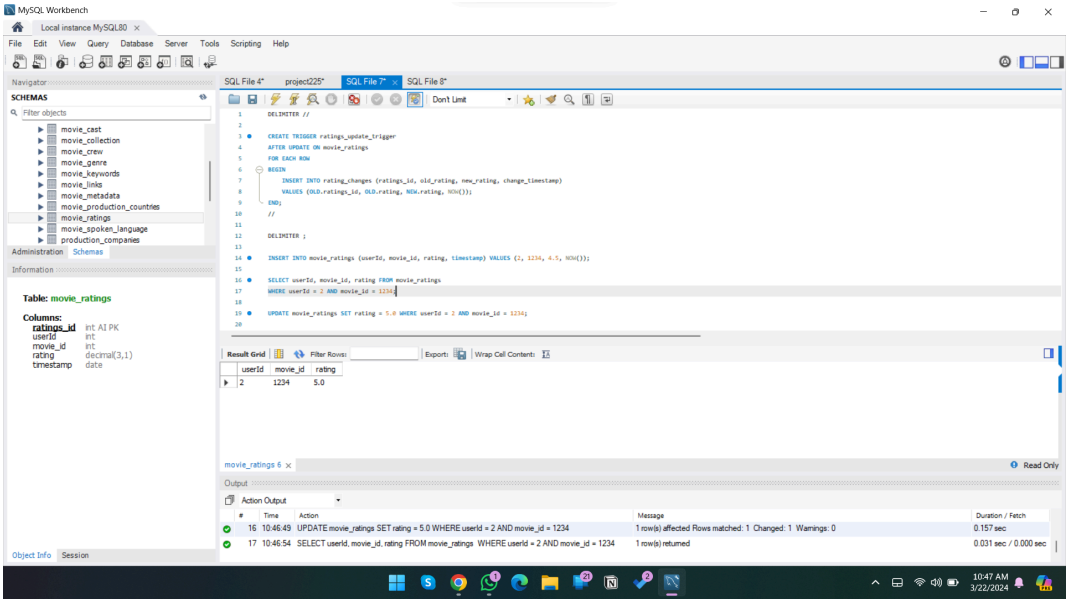
2. Design a trigger that should activate after each update operation on the movie_ratings table. Upon update of a rating record, it should print the timestamp of the update, the action performed (update), the name of the affected table (movie_ratings), and details about the updated record, including its ID, the old rating, and the new rating. Implement this trigger to directly display the changes for each update operation. (Includes Logging)

```
DELIMITER //

CREATE TRIGGER ratings_update_trigger
AFTER UPDATE ON movie_ratings
FOR EACH ROW
BEGIN
   INSERT INTO rating_changes (ratings_id, old_rating, new_rating,
change_timestamp)
   VALUES (OLD.ratings_id, OLD.rating, NEW.rating, NOW());
END;
//

DELIMITER ;

INSERT INTO movie_ratings (userId, movie_id, rating, timestamp) VALUES
(2, 1234, 4.5, NOW());

SELECT userId, movie_id, rating FROM movie_ratings
WHERE userId = 2 AND movie_id = 1234;

UPDATE movie_ratings SET rating = 5.0 WHERE userId = 2 AND movie_id
= 1234;

SELECT userId, movie_id, rating FROM movie_ratings
WHERE userId = 2 AND movie_id = 1234;
```

Output Before Update:



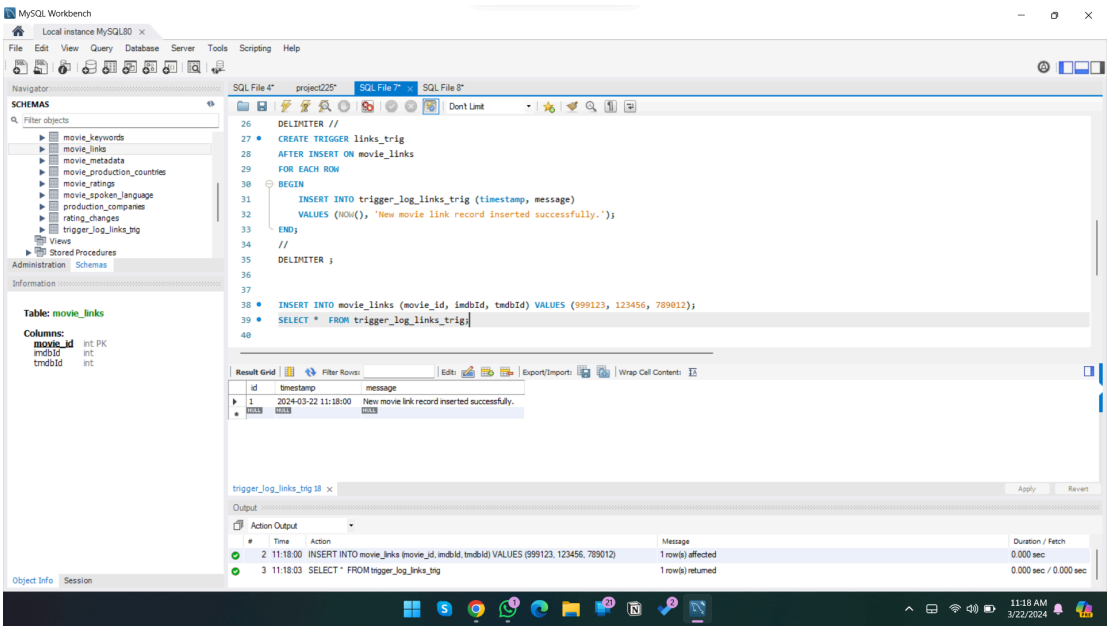
Output After Update:



3. Design a trigger in MySQL for the provided database schema to monitor the movie_links table. The trigger should activate after each insertion operation on the movie_links table. Upon insertion of a new movie link record, it should print a message indicating the successful insertion. (Includes Logging)

```
DELIMITER //
CREATE TRIGGER links_trig
AFTER INSERT ON movie_links
FOR EACH ROW
BEGIN
   INSERT INTO trigger_log_links_trig (timestamp, message)
   VALUES (NOW(), 'New movie link record inserted successfully.');
END;
//
DELIMITER ;
INSERT INTO movie_links (movie_id, imdbId, tmdbId) VALUES (999123, 123456, 789012);
SELECT *  FROM trigger_log_links_trig;
```

Output:

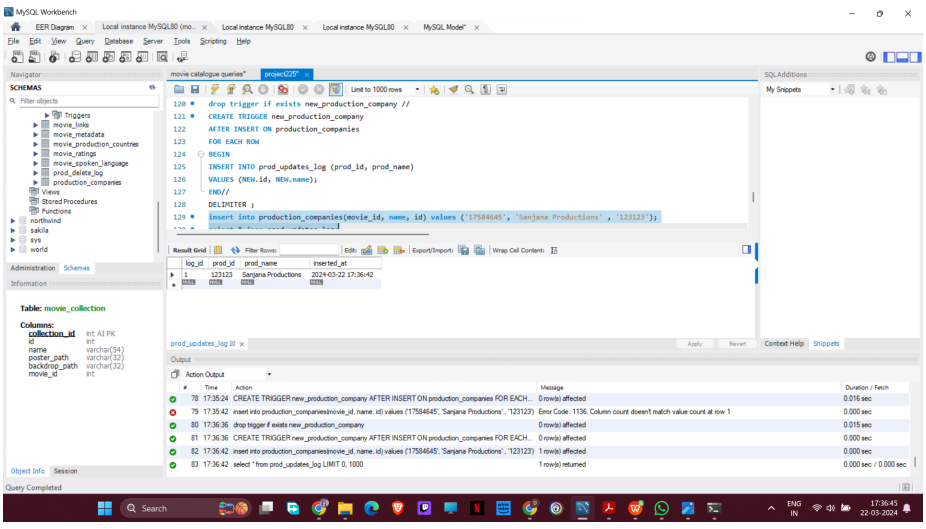


4. Design a trigger in MySql for adding a new production house into the Production_name table. A production_name_log table is created for logging a new production house and the records will be stored in the log table, An example is taken where ‘Sanjana Productions’ is the table inserted into the Production_name table. This record is stored in the production_name_log as shown in the figure.

use movie;

```
DROP TABLE IF EXISTS prod_updates_log;
CREATE TABLE IF NOT EXISTS prod_updates_log (
log_id INT AUTO_INCREMENT PRIMARY KEY,
prod_id INT,
prod_name VARCHAR(255),
inserted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
DELIMITER //
drop trigger if exists new_production_company //
CREATE TRIGGER new_production_company
AFTER INSERT ON production_companies
FOR EACH ROW
BEGIN
INSERT INTO prod_updates_log (prod_id, prod_name)
VALUES (NEW.id, NEW.name);
END//
DELIMITER ;
insert into production_companies(movie_id, name, id) values ('17584645', 'Sanjana Productions' , '123123');
select * from prod_updates_log;
```

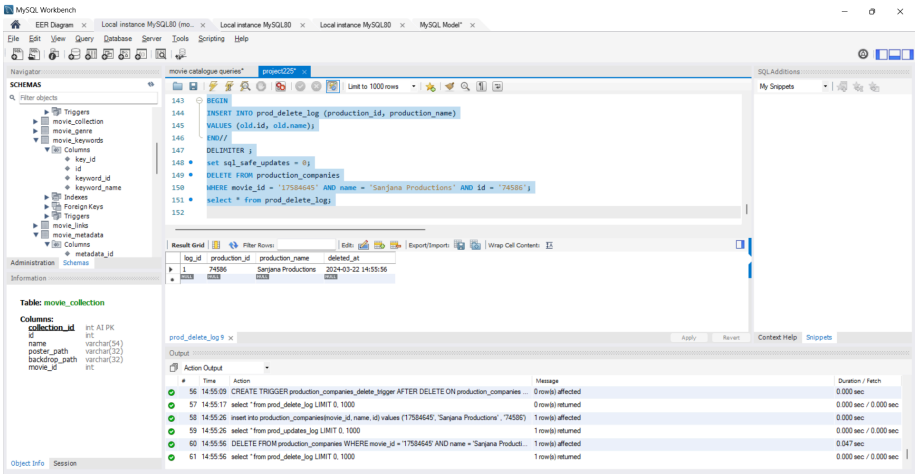
Output:



5. To keep track of deletions from the production_companies table, create the log table prod_delete_log. Additionally, it defines a trigger called production_companies_delete_trigger, which will cause deletions from production_companies to be automatically logged into the log table anytime a record is removed. Finally, it shows how to run a deletion statement to remove a particular entry from the production_companies database and read the deletion log, as well as how to deactivate SQL-safe updates.

```
CREATE TABLE IF NOT EXISTS prod_delete_log (
log_id INT AUTO_INCREMENT PRIMARY KEY,
production_id INT,
production_name VARCHAR(255),
deleted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
DELIMITER //
drop trigger if exists production_companies_delete_trigger //
CREATE TRIGGER production_companies_delete_trigger
AFTER DELETE ON production_companies
FOR EACH ROW
BEGIN
INSERT INTO prod_delete_log (production_id, production_name)
VALUES (old.id, old.name);
END//
DELIMITER ;
set sql_safe_updates = 0;
DELETE FROM production_companies
WHERE movie_id = '17584645' AND name = 'Sanjana Productions' AND id = '74586';
select * from prod_delete_log;
```

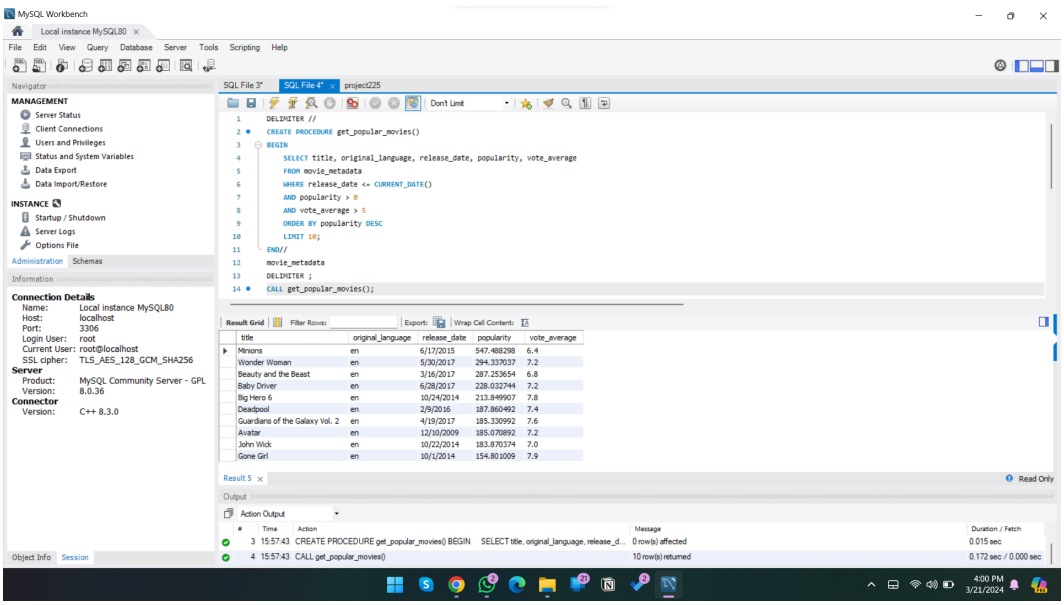
Output:



- Stored procedures:
    1. List the top 10 popular movies available in movie_metadata table, considering their release dates are on or before today, and each movie has a positive popularity rating and vote average is greater than 5?

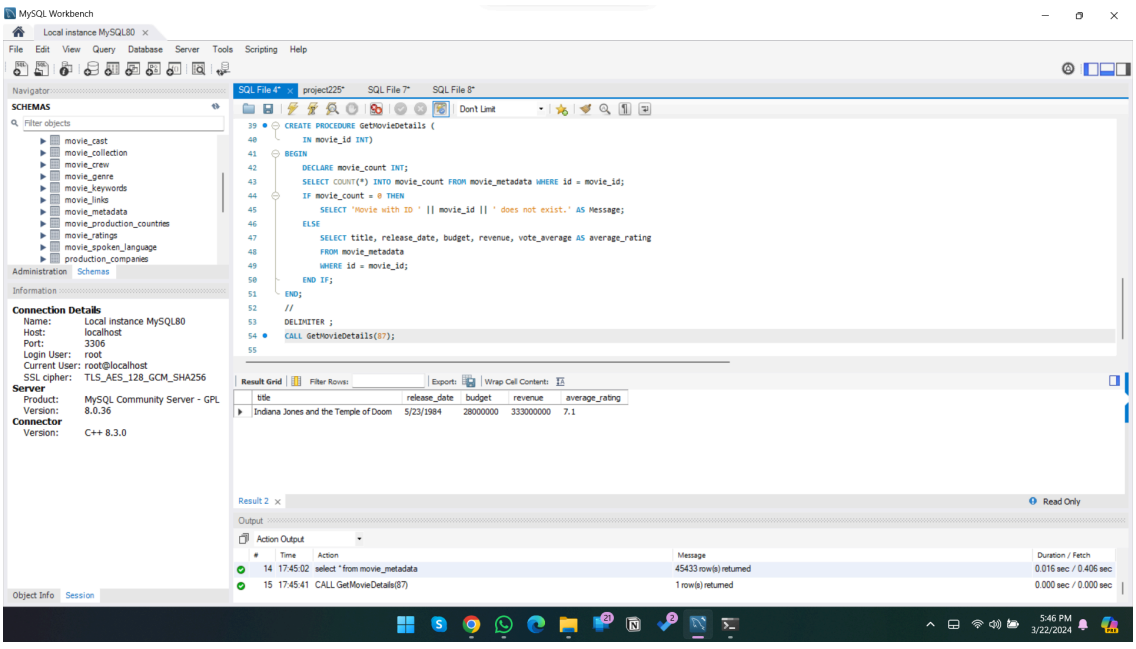
```
DELIMITER //

CREATE PROCEDURE get_popular_movies()
BEGIN
    SELECT title, original_language, release_date, popularity, vote_average
    FROM movie_metadata
    WHERE release_date <= CURRENT_DATE()
    AND popularity > 0
    AND vote_average > 5
    ORDER BY popularity DESC
    LIMIT 10;
END//
movie_metadata
DELIMITER ;
CALL get_popular_movies();
```

Output:



2. Write a stored procedure named GetMovieDetails that takes a movie ID as input and returns details about the movie, including its title, release date, budget, revenue, and average rating. If the movie ID does not exist, the procedure should return a message indicating that the movie does not exist.

```
DELIMITER //
CREATE PROCEDURE GetMovieDetails (
   IN movie_id INT)
BEGIN
   DECLARE movie_count INT;
   SELECT COUNT(*) INTO movie_count FROM movie_metadata WHERE id = movie_id;
   IF movie_count = 0 THEN
      SELECT 'Movie with ID ' || movie_id || ' does not exist.' AS Message;
   ELSE
      SELECT title, release_date, budget, revenue, vote_average AS average_rating
      FROM movie_metadata
      WHERE id = movie_id;
   END IF;
END;
//
DELIMITER ;
CALL GetMovieDetails(87);
```



- Logging Mechanism:
    1. When a user rates a movie. The user's actions are automatically logged and a log table is maintained with the information given below.
       A log table is created which takes log_id (Primary key with TIMESTAMP as unique identifier), user_id, actions performed, affect on table, and details as input.
       A trigger is created to automate the process when a user inputs rating.

```
CREATE TABLE db_log (
   log_id INT AUTO_INCREMENT PRIMARY KEY,
   timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   user_id INT,
   action VARCHAR(255),
   table_affected VARCHAR(255),
   details TEXT
);
DELIMITER //
CREATE TRIGGER log_after_insert
AFTER INSERT ON movie_ratings
FOR EACH ROW
BEGIN
   INSERT INTO db_log (user_id, action, table_affected, details)
   VALUES (NEW.user_id, 'INSERT', 'movie_ratings', CONCAT('New row inserted: ',
NEW.column1, ', ', NEW.column2));
END //
DELIMITER ;
```