SQL Basics – Assignment Questions

Name: Darshan Panchal

Email ID: panchaldarshan177@gmail.com

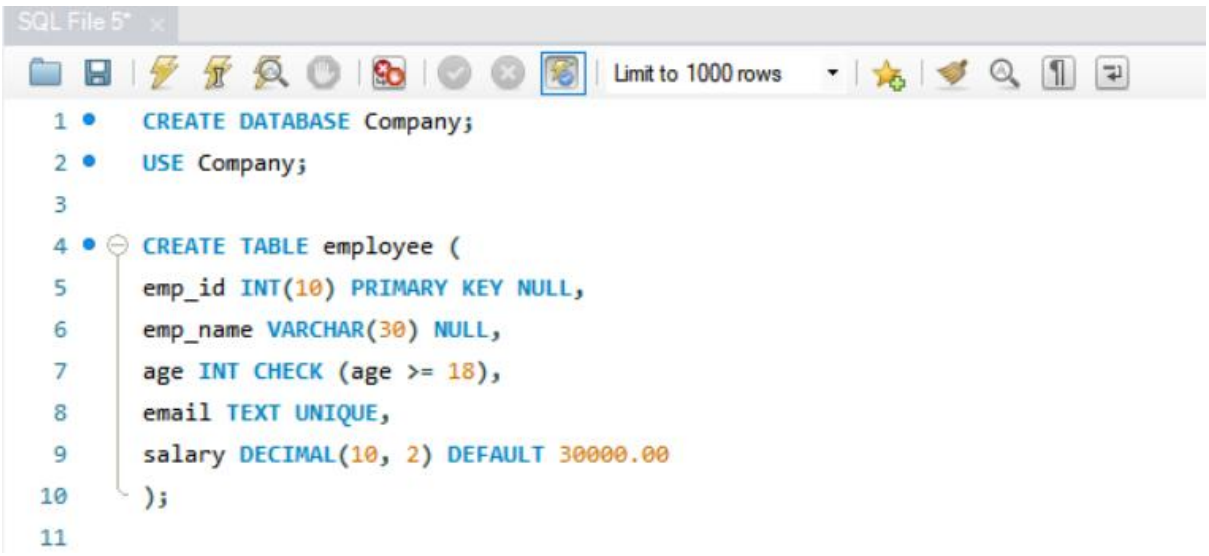
Assignment Name: SQL BASICS – Database – Module-1

Q1.

1. Create a table called employees with the following structure:

- **emp_id (integer, should not be NULL and should be a primary key).**
- **emp_name (text, should not be NULL).**
- **age (integer, should have a check constraint to ensure the age is at least 18).**
- **email (text, should be unique for each employee).**
- **salary (decimal, with a default value of 30,000).**

Write the SQL query to create the above table with all constraints.



Q2.

2. Explain the purpose of constraints and how they help maintain data integrity in a database. Provide examples of common types of constraints.

- **Purpose of Constraints =** Prevent invalid data from being entered, ensure relationships between tables remain consistent, provide default values when none are supplied.
- Examples = NOT NULL = Ensures a column cannot have NULL values
  name VARCHAR(100) NOT NULL
  UNIQUE = Ensures all values in a column are different
  email TEXT UNIQUE
  PRIMARY KEY = Uniquely identifies each record in a table
  id INT PRIMARY KEY
  FOREIGN KEY = Enforces link between records in two tables
  dept_id INT REFERENCES departments(id)
  CHECK = Ensures values meet a specific condition
  age INT CHECK (age >= 18)

Q3.

3.Why would you apply the NOT NULL constraint to a column? Can a primary key contain NULL values? Justify your answer.

- **NOT NULL** = The NOT NULL constraint is applied to a column to ensure that a value must always be provided for that field. It helps maintain data integrity by preventing missing or incomplete information in important columns, such as names, emails, or user IDs. This is particularly useful when certain fields are essential for business logic or application functionality. Without this constraint, someone might accidentally insert a row with missing data, which could cause errors or inconsistencies in processing or reporting.
- **PRIMARY KEY** = A PRIMARY KEY cannot contain NULL values. This is because a primary key's main purpose is to uniquely identify each record in a table. Allowing NULL would violate this uniqueness since NULL represents an unknown or undefined value and cannot be reliably used for identification. In fact, when a primary key is defined, most database systems automatically apply both the NOT NULL and UNIQUE constraints to enforce this rule. Therefore, every table that uses a primary key must ensure that this column is always filled with a unique, non-null value.

Q4.

4. Explain the steps and SQL commands used to add or remove constraints on an existing table. Provide an example for both adding and removing a constraint.

- **Steps to Add a Constraint =**
    1. Identify the table and the column where you want the constraint.
    2. Use ALTER TABLE with the appropriate ADD CONSTRAINT clause.
    3. Give the constraint a name (optional but recommended).
    4. Specify the type of constraint (e.g., CHECK, UNIQUE, FOREIGN KEY, etc.).
    5. Example = Add a CHECK constraint to ensure age is at least 18

```
ALTER TABLE employees
ADD CONSTRAINT chk_age CHECK (age >= 18);
```

- **Steps to Remove a Constraint =**
    1. Find the name of the constraint you want to drop.
    2. Use ALTER TABLE with the DROP CONSTRAINT clause.
    3. Example: Remove the chk_age constraint

```
ALTER TABLE employees
DROP CONSTRAINT chk_age;
```

Q5.

5. Explain the consequences of attempting to insert, update, or delete data in a way that violates constraints. Provide an example of an error message that might occur when violating a constraint.

- Constraints in a database are designed to protect data integrity, so when you attempt to **insert, update, or delete data** in a way that **violates a constraint**, the database will **reject the operation** and return an error. These errors prevent invalid, inconsistent, or harmful data from being saved. For example, if you try to insert a NULL into a column defined with a NOT NULL constraint, the database will stop the operation because it violates the rule that a value must always be present.
- Similarly, trying to insert a duplicate value into a column with a UNIQUE constraint, or an age below 18 in a column with a CHECK (age >= 18) constraint, will cause the operation to fail. If you attempt to delete a record that is referenced by a foreign key in another table, it will also fail unless you've set specific cascading rules.
- Here's a typical error message example when violating a CHECK constraint in PostgreSQL:
  ERROR: new row for relation "employees" violates check constraint "chk_age"
  DETAIL: Failing row contains (7, John, 16, john@example.com).

Q6.

6. You created a products table without constraints as follows:

```
CREATE TABLE products (
    product_id INT,
    product_name VARCHAR(50),
    price DECIMAL(10, 2));
```

Now, you realise that:

- **The product_id should be a primary key.**
- **The price should have a default value of 50.00**

```
4
5  CREATE TABLE products (
6      product_id INT PRIMARY KEY,
7      product_name VARCHAR(50),
8      price DECIMAL(10, 2) DEFAULT 50.00
9  );
10
11 ALTER TABLE products
12 DROP PRIMARY KEY;
13
14 ALTER TABLE products
15 ADD CONSTRAINT pk_product_id PRIMARY KEY (product_id);
16
17 ALTER TABLE products
18 MODIFY price DECIMAL(10, 2) DEFAULT 50.00;
19
```

Q7.

7. You have two tables:

- Students:

student_id	student_name	class_id
1	Alice	101
2	Bob	102
3	Charlie	101

- Classes:

class_id	class_name
101	Math
102	Science
103	History

Write a query to fetch the student_name and class_name for each student using an INNER JOIN.

```sql
CREATE TABLE classes (
    class_id INT PRIMARY KEY,
    class_name VARCHAR(100) NOT NULL
);
INSERT INTO classes (class_id, class_name) VALUES
(1, 'Mathematics'),
(2, 'Science'),
(3, 'History');

CREATE TABLE students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(100) NOT NULL,
    class_id INT,
    FOREIGN KEY (class_id) REFERENCES classes(class_id)
);
INSERT INTO students (student_id, student_name, class_id) VALUES
(1, 'John Doe', 1),
(2, 'Jane Smith', 2),
(3, 'Sam Brown', 3),
(4, 'Emily Johnson', 1);
```

```sql
INSERT INTO students (student_id, student_name, class_id) VALUES
(1, 'John Doe', 1),
(2, 'Jane Smith', 2),
(3, 'Sam Brown', 3),
(4, 'Emily Johnson', 1);

SELECT
  s.student_name,
  c.class_name
FROM
  students AS s
  INNER JOIN classes AS c
    ON s.class_id = c.class_id;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

student_name	class_name
John Doe	Mathematics
Jane Smith	Science
Sam Brown	History
Emily Johnson	Mathematics

Q8.

8. Consider the following three tables:

- Orders:

```
+------------+------------+------------+
| order_id   | order_date | customer_id|
+------------+------------+------------+
| 1          | 2024-01-01 | 101        |
| 2          | 2024-01-03 | 102        |
+------------+------------+------------+
```

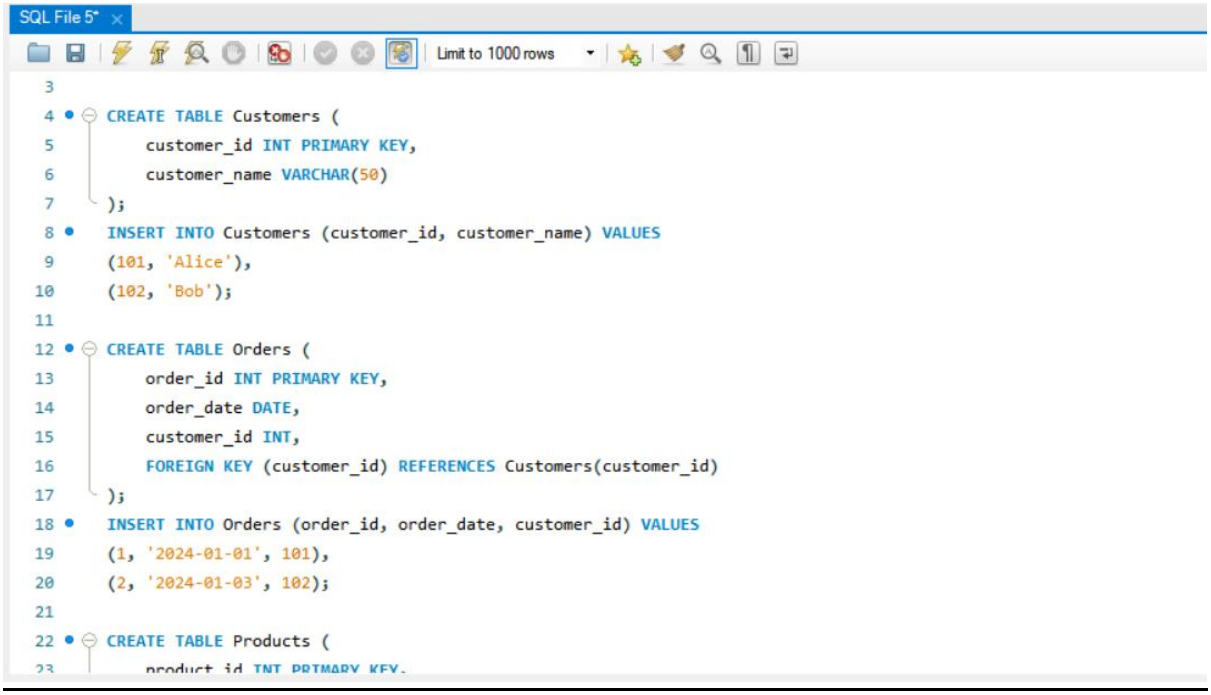
- Customers:

```
+------------+------------+
| customer_id| customer_name|
+------------+------------+
| 101        | Alice        |
| 102        | Bob          |
+------------+------------+
```

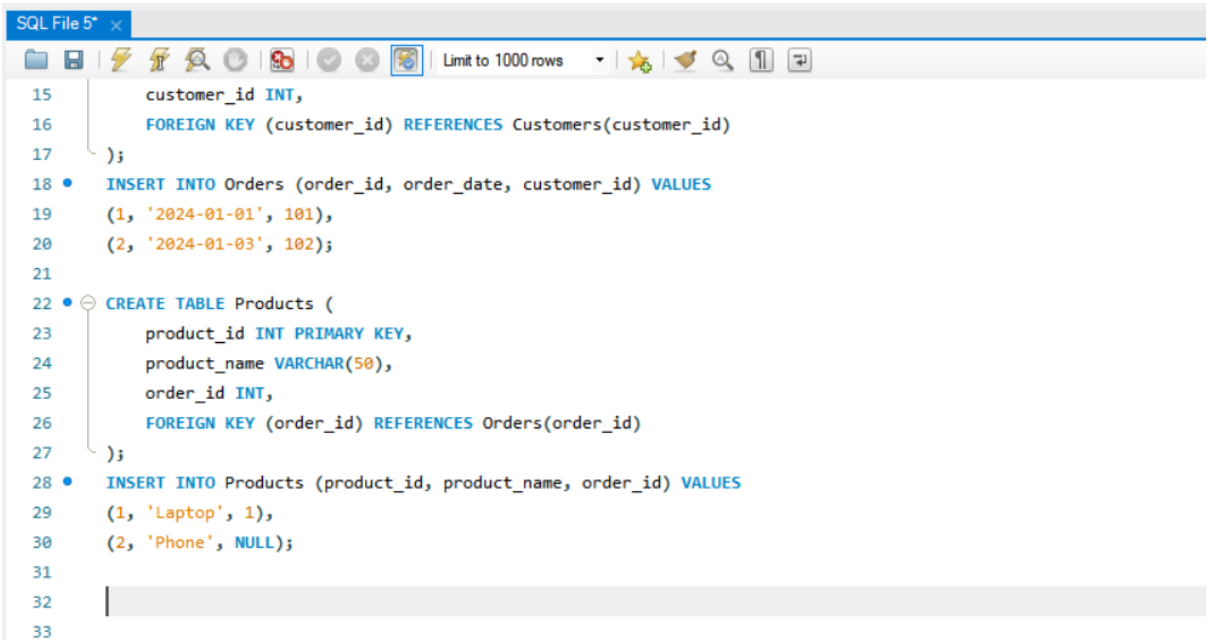
- Products:

```
+------------+------------+------------+
| product_id | product_name | order_id |
+------------+------------+------------+
| 1          | Laptop       | 1        |
| 2          | Phone        | NULL     |
+------------+------------+------------+
```

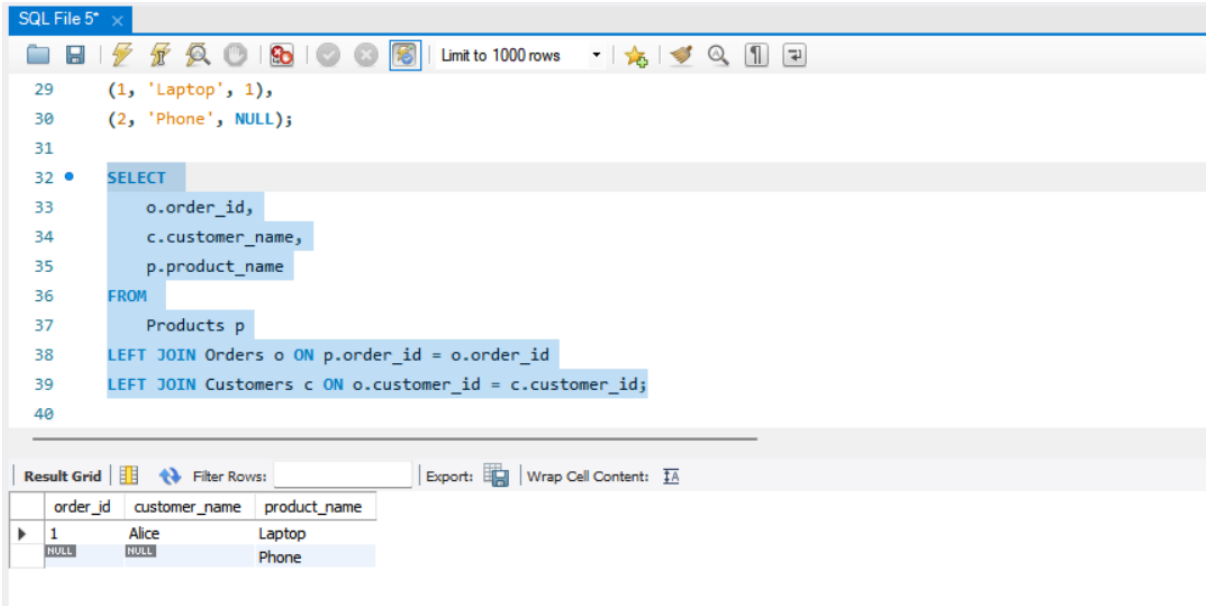
Write a query that shows all order_id, customer_name, and product_name, ensuring that all products are listed even if they are not associated with an order
Hint: (use INNER JOIN and LEFT JOIN).



```
SQL File 5*
Limit to 1000 rows

CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(50)
);
INSERT INTO Customers (customer_id, customer_name) VALUES
(101, 'Alice'),
(102, 'Bob');

CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    order_date DATE,
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);
INSERT INTO Orders (order_id, order_date, customer_id) VALUES
(1, '2024-01-01', 101),
(2, '2024-01-03', 102);

CREATE TABLE Products (
    product_id INT PRIMARY KEY,
```



```
SQL File 5*
Limit to 1000 rows

    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);
INSERT INTO Orders (order_id, order_date, customer_id) VALUES
(1, '2024-01-01', 101),
(2, '2024-01-03', 102);

CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50),
    order_id INT,
    FOREIGN KEY (order_id) REFERENCES Orders(order_id)
);
INSERT INTO Products (product_id, product_name, order_id) VALUES
(1, 'Laptop', 1),
(2, 'Phone', NULL);
```



Q9.

9. Given the following tables:

Sales:

```
+------------+------------+------------+
| sale_id    | product_id | amount     |
+------------+------------+------------+
| 1          | 101        | 500        |
| 2          | 102        | 300        |
| 3          | 101        | 700        |
+------------+------------+------------+
```

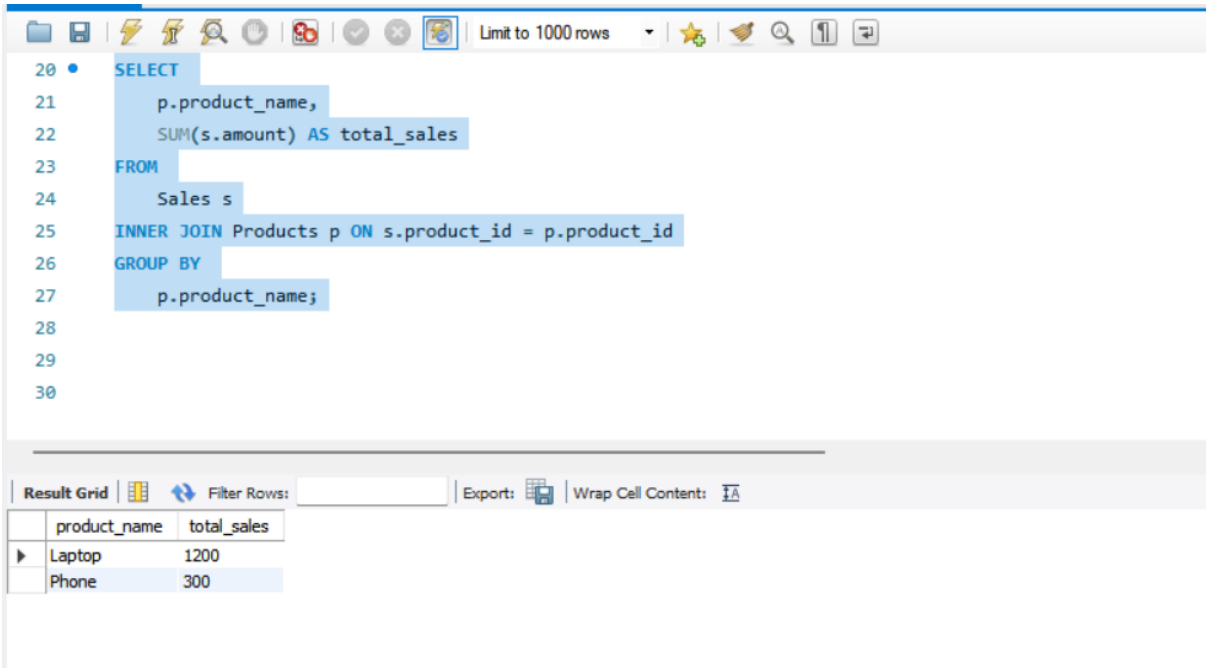
Products:

```
+------------+------------+
| product_id | product_name|
+------------+------------+
| 101        | Laptop      |
| 102        | Phone       |
+------------+------------+
```

Write a query to find the total sales amount for each product using an INNER JOIN and the SUM() function.

```sql
CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50)
);
INSERT INTO Products (product_id, product_name) VALUES
(101, 'Laptop'),
(102, 'Phone');

CREATE TABLE Sales (
    sale_id INT PRIMARY KEY,
    product_id INT,
    amount INT,
    FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
INSERT INTO Sales (sale_id, product_id, amount) VALUES
(1, 101, 500),
(2, 102, 300),
(3, 101, 700);
```

```sql
);
INSERT INTO Sales (sale_id, product_id, amount) VALUES
(1, 101, 500),
(2, 102, 300),
(3, 101, 700);

SELECT
    p.product_name,
    SUM(s.amount) AS total_sales
FROM
    Sales s
INNER JOIN Products p ON s.product_id = p.product_id
GROUP BY
    p.product_name;
```


Limit to 1000 rows
20 SELECT
21 p.product_name,
22 SUM(s.amount) AS total_sales
23 FROM
24 Sales s
25 INNER JOIN Products p ON s.product_id = p.product_id
26 GROUP BY
27 p.product_name;
28
29
30
Result Grid
Filter Rows:
Export:
Wrap Cell Content:
product_name
total_sales
Laptop
1200
Phone
300

Q10.

10. You are given three tables:

Orders:

order_id	order_date	customer_id
1	2024-01-02	1
2	2024-01-05	2

Customers:

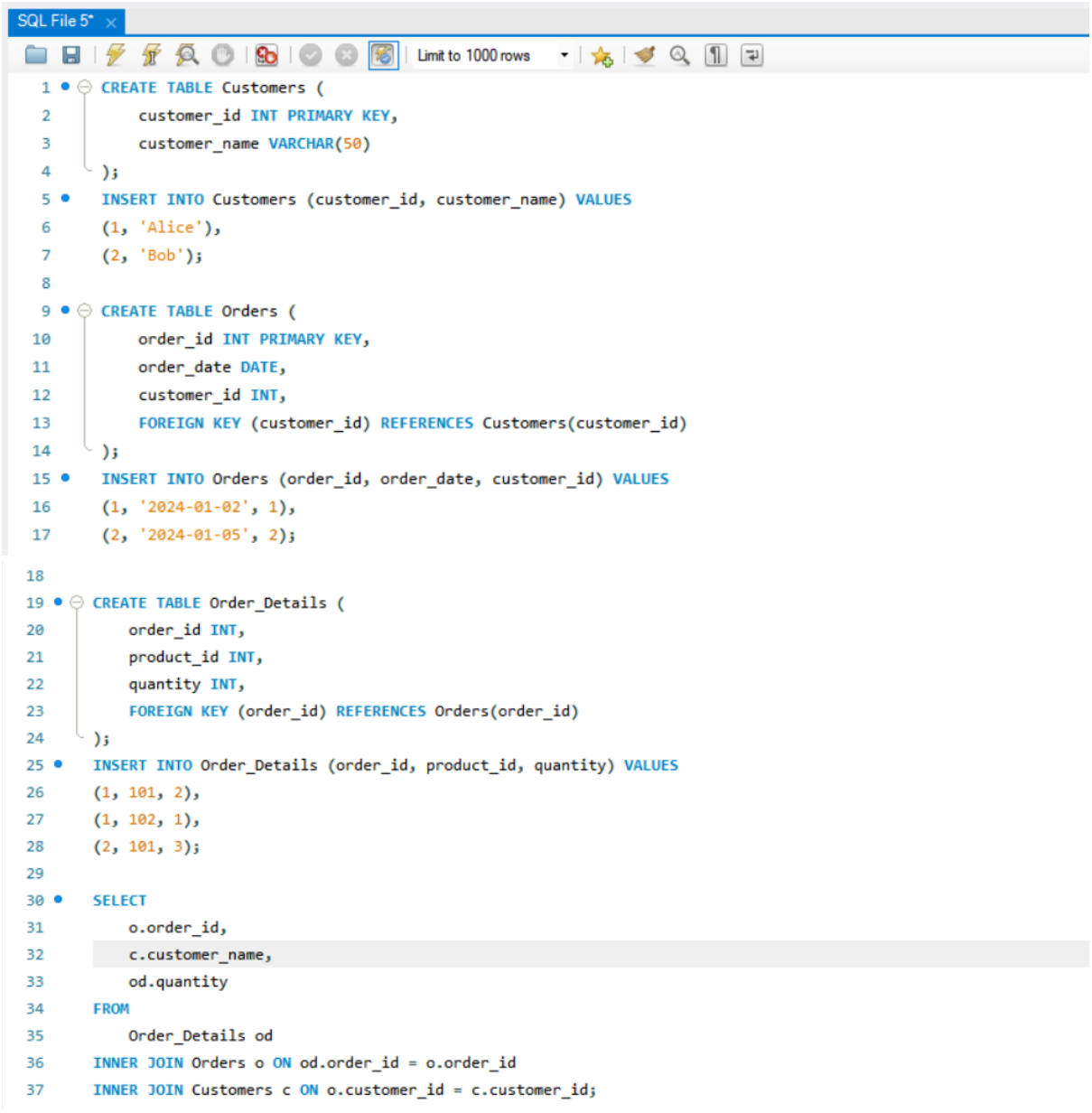
customer_id	customer_name
1	Alice
2	Bob

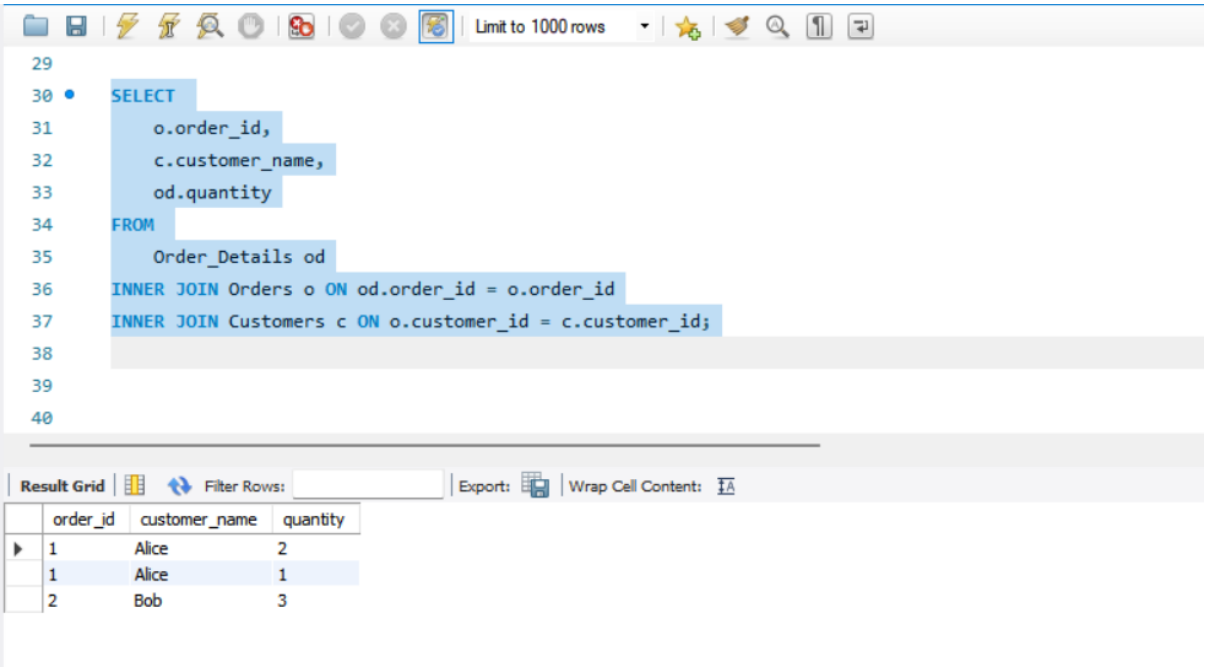
Order_Details:

order_id	product_id	quantity
1	101	2
1	102	1
2	101	3

Write a query to display the order_id, customer_name, and the quantity of products ordered by each customer using an INNER JOIN between all three tables.

Note - The above-mentioned questions don't require any dataset.


SQL File 5*
Limit to 1000 rows
CREATE TABLE Customers (
customer_id INT PRIMARY KEY,
customer_name VARCHAR(50)
);
INSERT INTO Customers (customer_id, customer_name) VALUES
(1, 'Alice'),
(2, 'Bob');
CREATE TABLE Orders (
order_id INT PRIMARY KEY,
order_date DATE,
customer_id INT,
FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);
INSERT INTO Orders (order_id, order_date, customer_id) VALUES
(1, '2024-01-02', 1),
(2, '2024-01-05', 2);
CREATE TABLE Order_Details (
order_id INT,
product_id INT,
quantity INT,
FOREIGN KEY (order_id) REFERENCES Orders(order_id)
);
INSERT INTO Order_Details (order_id, product_id, quantity) VALUES
(1, 101, 2),
(1, 102, 1),
(2, 101, 3);
SELECT
o.order_id,
c.customer_name,
od.quantity
FROM
Order_Details od
INNER JOIN Orders o ON od.order_id = o.order_id
INNER JOIN Customers c ON o.customer_id = c.customer_id;

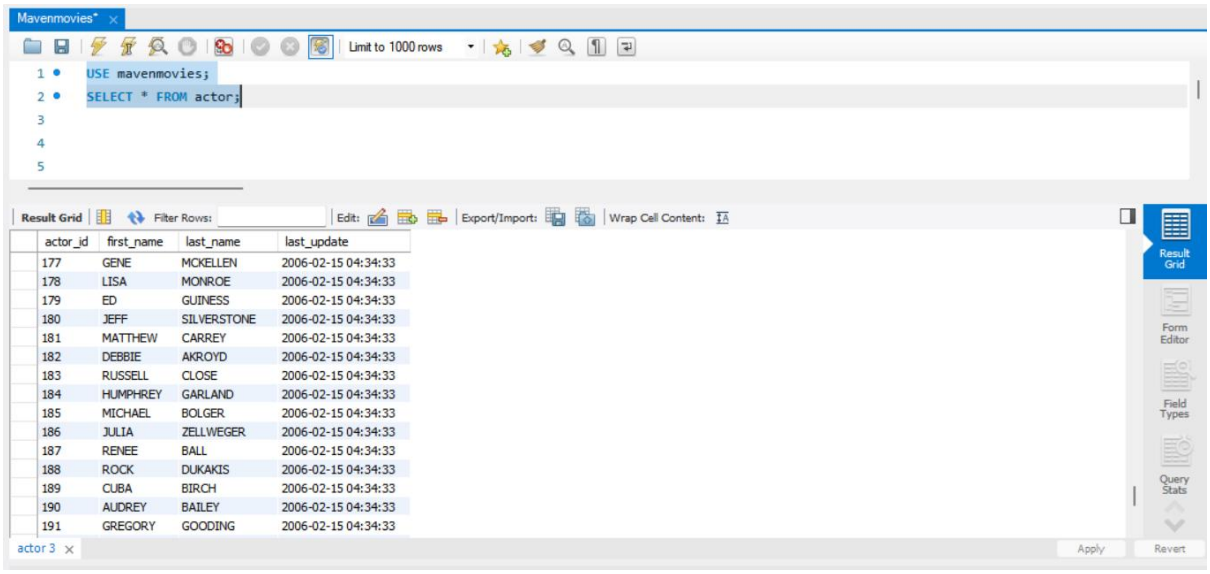

Limit to 1000 rows
29
30 SELECT
31 o.order_id,
32 c.customer_name,
33 od.quantity
34 FROM
35 Order_Details od
36 INNER JOIN Orders o ON od.order_id = o.order_id
37 INNER JOIN Customers c ON o.customer_id = c.customer_id;
38
39
40
Result Grid
Filter Rows:
Export:
Wrap Cell Content:
order_id customer_name quantity
1 Alice 2
1 Alice 1
2 Bob 3

SQL Commands - Questions

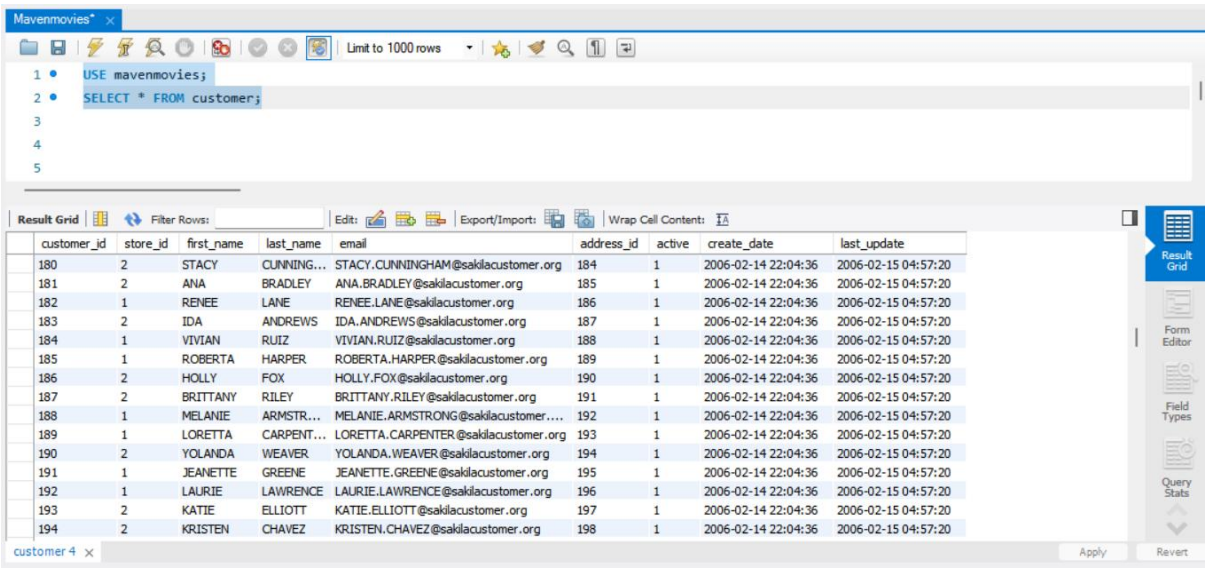
1. Identify the primary keys and foreign keys in maven movies db. Discuss the differences.

- **Primary Keys in Maven Movies DB:** In the Maven Movies database, a Primary Key is a unique identifier for each record in a table. For example, in the customer table, the customer_id is the primary key. It ensures that each customer has a unique identifier. Similarly, film_id in the film table and rental_id in the rental table are also primary keys. A primary key cannot contain NULL values and must be unique for each row.
- **Foreign Keys in Maven Movies DB:** A Foreign Key is a field (or collection of fields) in one table that refers to the primary key in another table. In Maven Movies, for example, the customer_id in the rental table is a foreign key that references the customer_id in the customer table. Similarly, the film_id in the inventory table references the film_id in the film table. Foreign keys are used to create relationships between tables and enforce referential integrity.
- **Difference 1 – Uniqueness and Role:** A primary key uniquely identifies each record in its own table and ensures there are no duplicates. It is the main point of reference for accessing a record. In contrast, a foreign key may have duplicate values and is used to link rows between two tables — it does not uniquely identify a record in its own table but points to a unique record in another.
- **Difference 2 – Data Integrity and Relationship:** Primary keys ensure data integrity within a table by preventing NULLs and duplicates. Foreign keys ensure data consistency between related tables by enforcing that values in the child table must exist in the parent table. For instance, a rental record cannot exist for a customer who doesn't exist in the customer table — this is enforced via a foreign key.

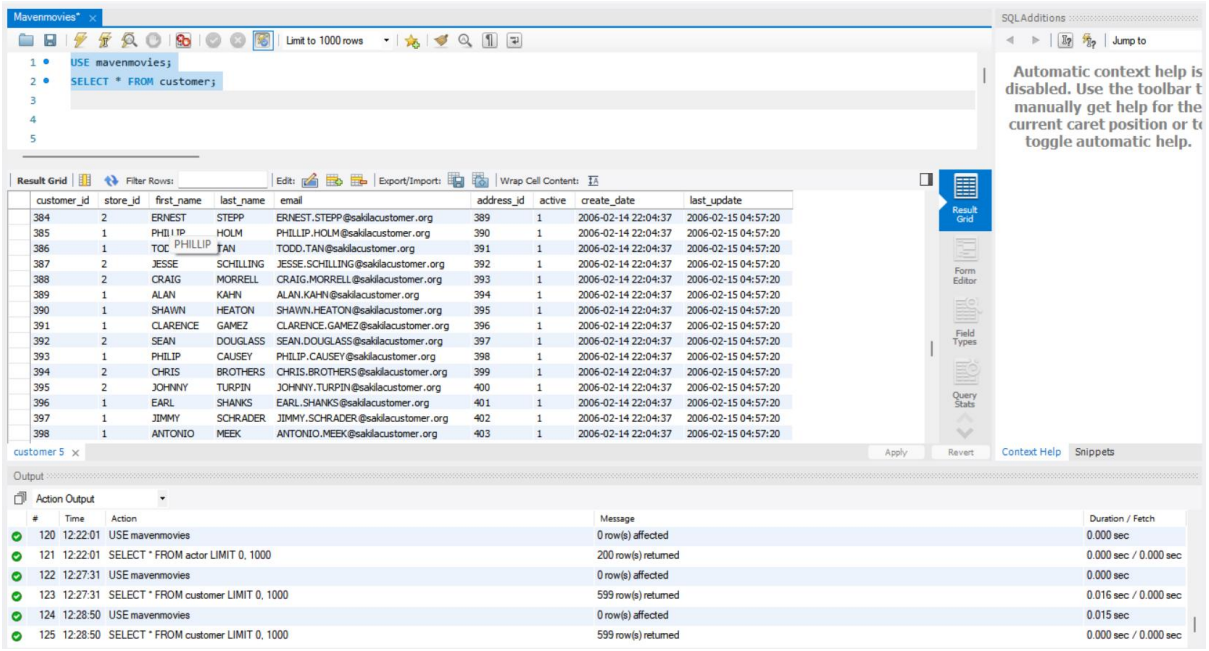
2. List all details of actors.



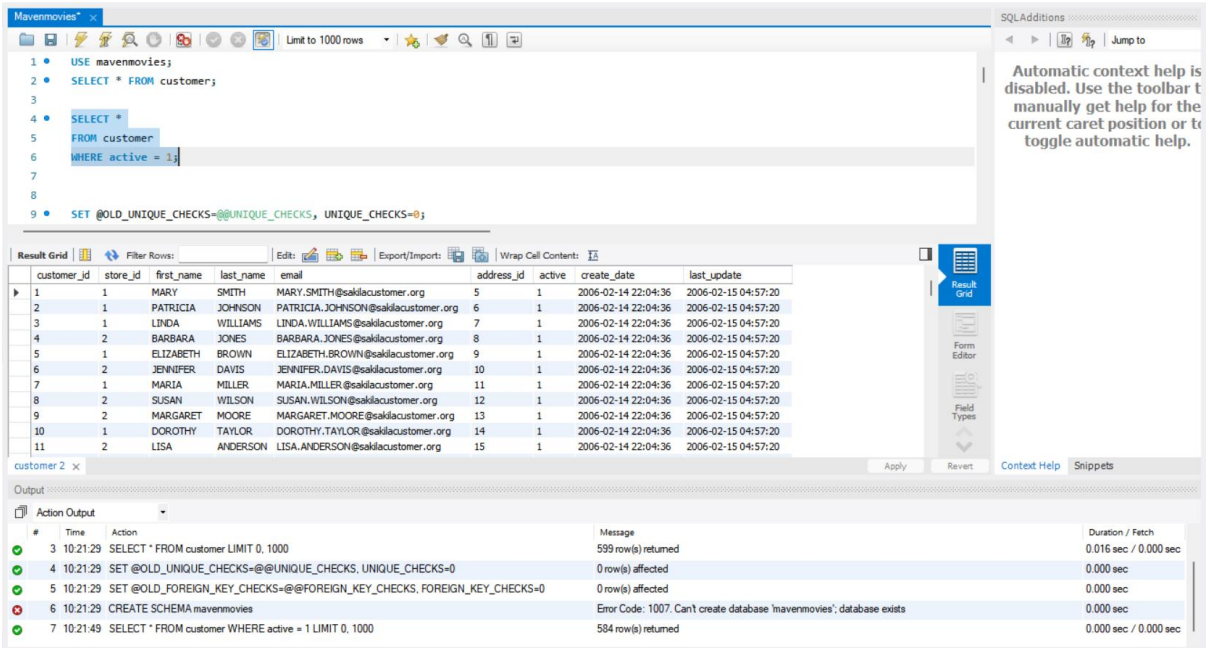
3. List all customer information from DB.



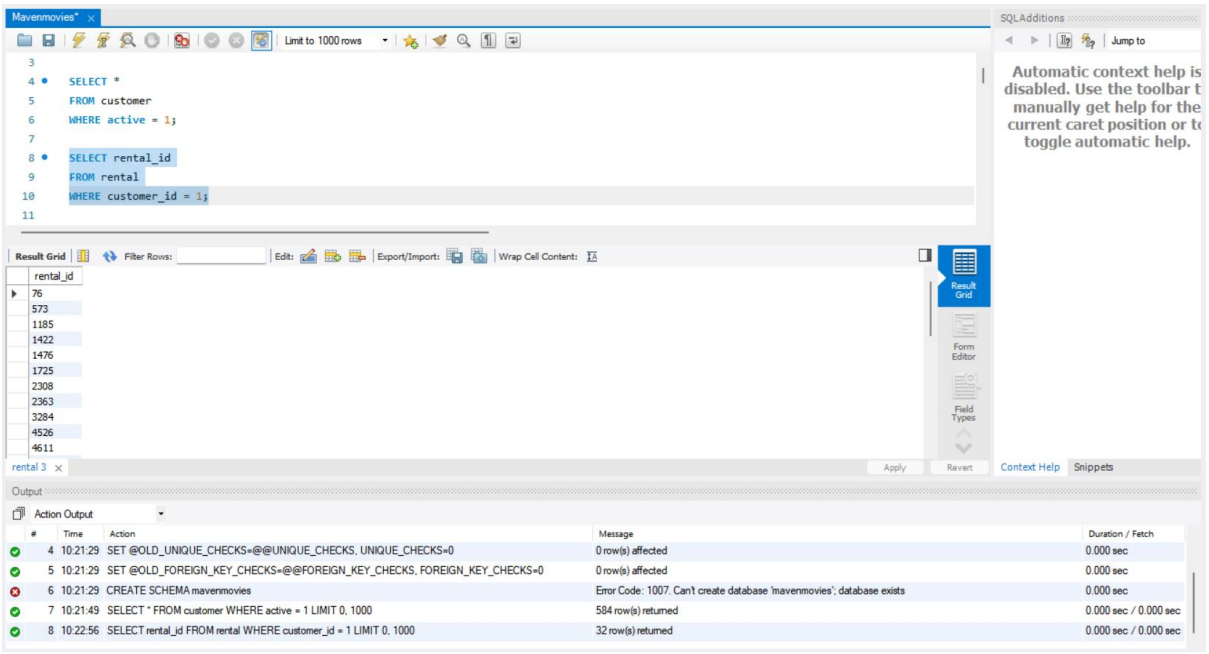
4. List different countries.



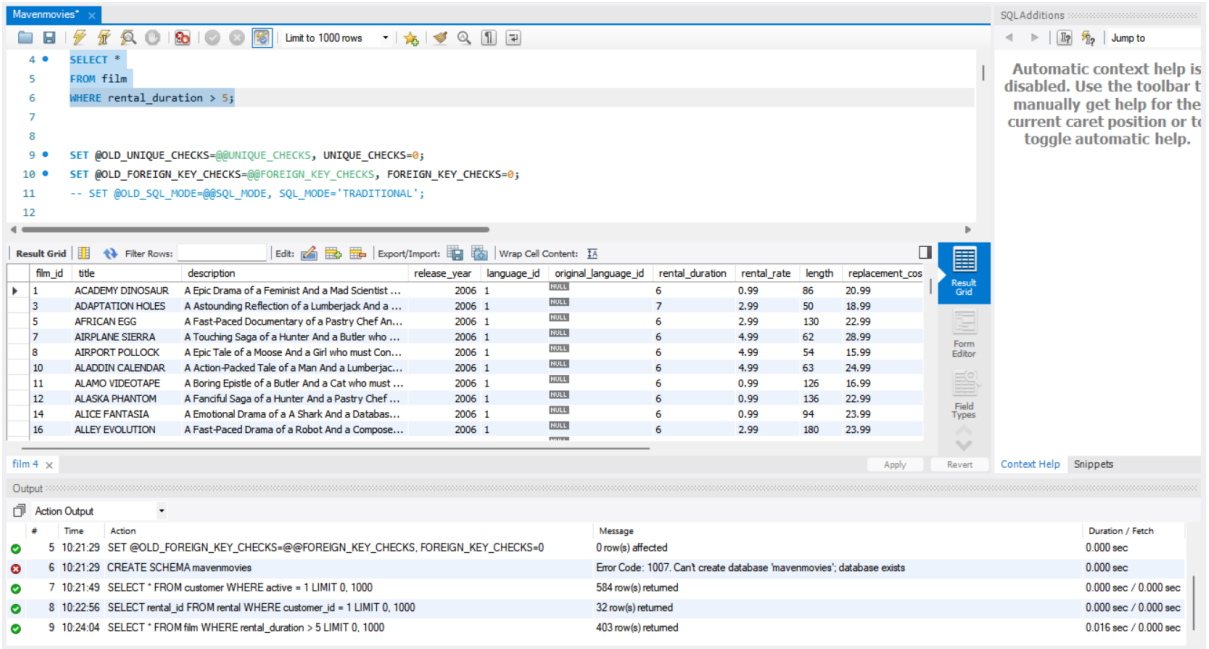
5. Display all active customers.



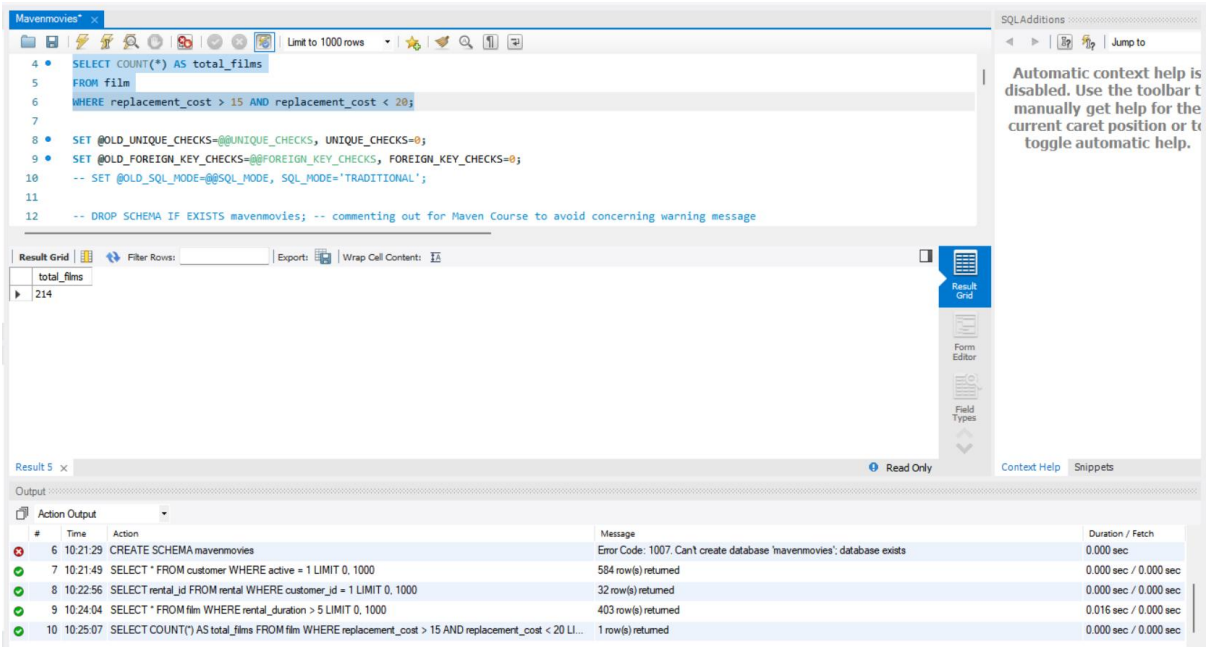
6. List of all rental IDs for customer with ID 1.



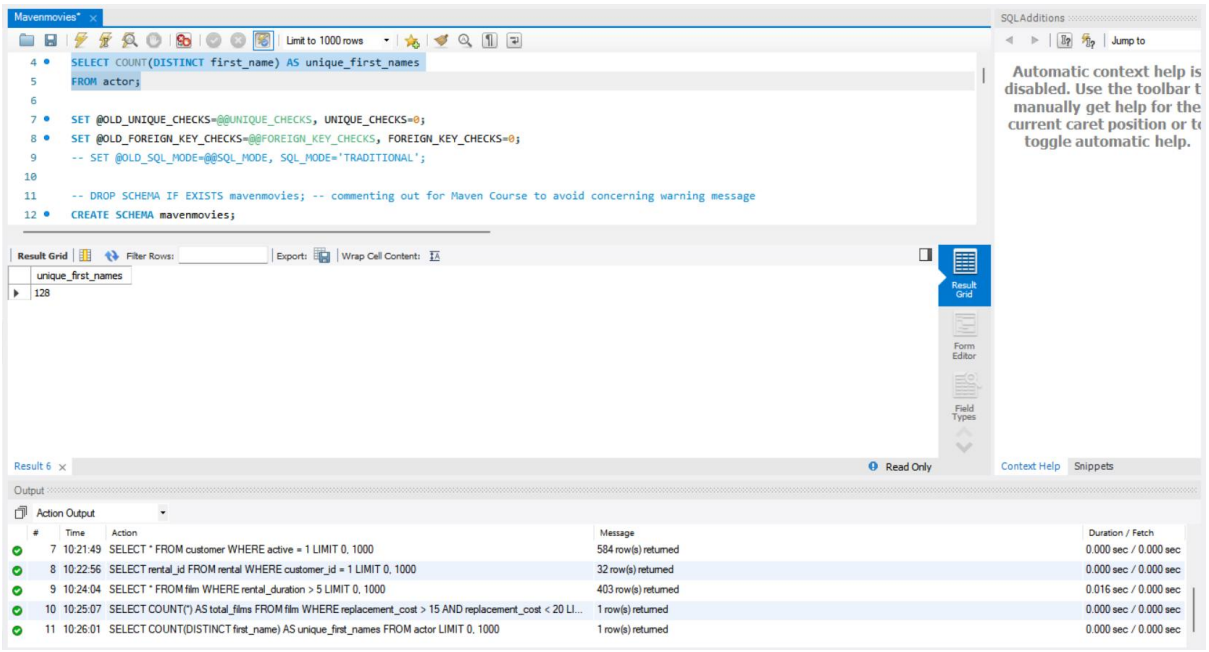
7. Display all the films whose rental duration is greater than 5.



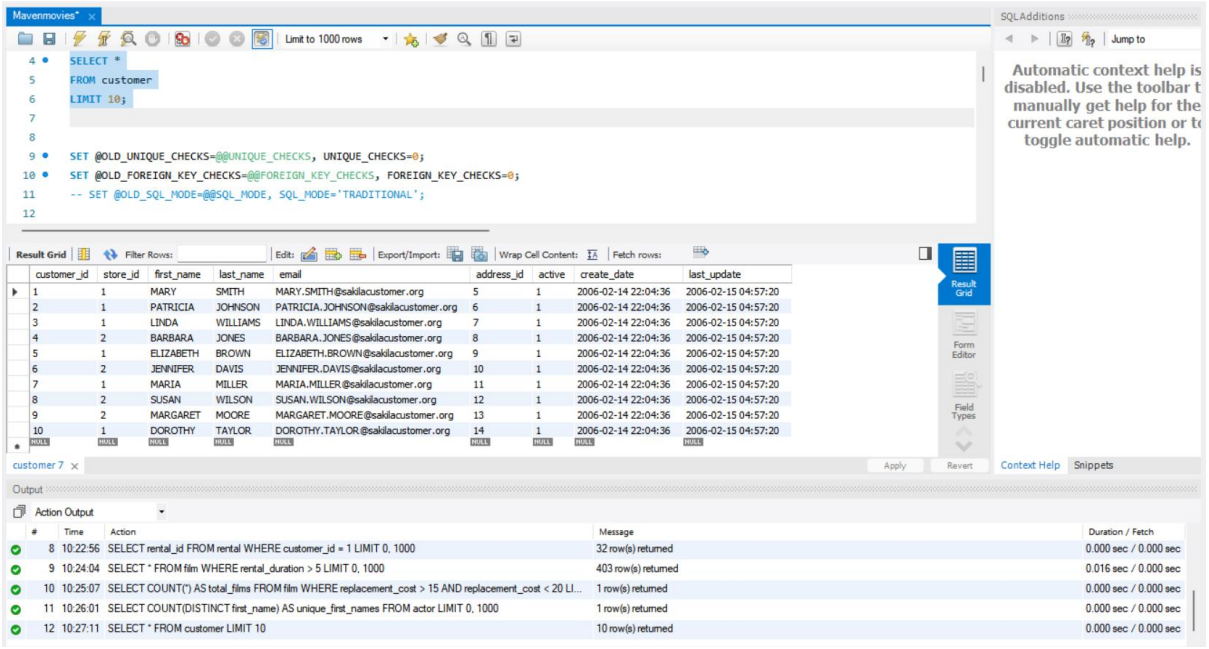
8. List the total number of films whose replacement cost is greater than $15 and less than $20.



9. Display the count of unique first names of actors.



10. Display the first 10 records from the customer table.

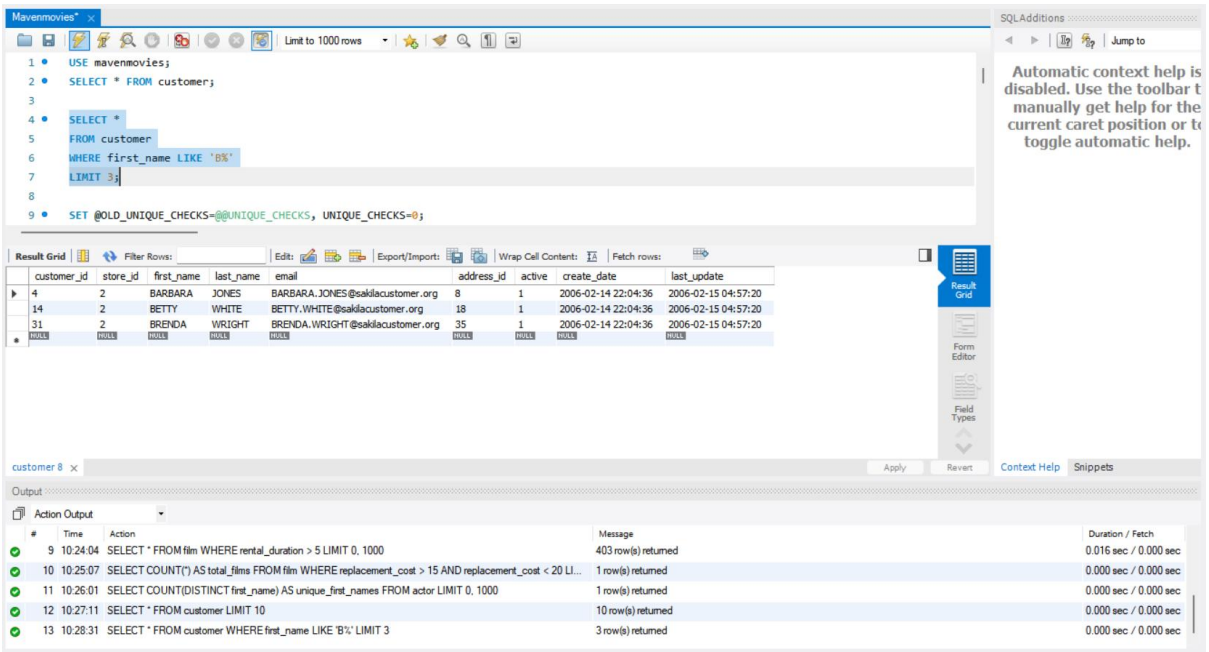


```
SELECT *
FROM customer
LIMIT 10;

SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
-- SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';
```

customer_id	store_id	first_name	last_name	email	address_id	active	create_date	last_update
1	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	5	1	2006-02-14 22:04:36	2006-02-15 04:57:20
2	1	PATRICIA	JOHNSON	PATRICIA.JOHNSON@sakilacustomer.org	6	1	2006-02-14 22:04:36	2006-02-15 04:57:20
3	1	LINDA	WILLIAMS	LINDA.WILLIAMS@sakilacustomer.org	7	1	2006-02-14 22:04:36	2006-02-15 04:57:20
4	2	BARBARA	JONES	BARBARA.JONES@sakilacustomer.org	8	1	2006-02-14 22:04:36	2006-02-15 04:57:20
5	1	ELIZABETH	BROWN	ELIZABETH.BROWN@sakilacustomer.org	9	1	2006-02-14 22:04:36	2006-02-15 04:57:20
6	2	JENNIFER	DAVIS	JENNIFER.DAVIS@sakilacustomer.org	10	1	2006-02-14 22:04:36	2006-02-15 04:57:20
7	1	MARIA	MILLER	MARIA.MILLER@sakilacustomer.org	11	1	2006-02-14 22:04:36	2006-02-15 04:57:20
8	2	SUSAN	WILSON	SUSAN.WILSON@sakilacustomer.org	12	1	2006-02-14 22:04:36	2006-02-15 04:57:20
9	2	MARGARET	MOORE	MARGARET.MOORE@sakilacustomer.org	13	1	2006-02-14 22:04:36	2006-02-15 04:57:20
10	1	DOROTHY	TAYLOR	DOROTHY.TAYLOR@sakilacustomer.org	14	1	2006-02-14 22:04:36	2006-02-15 04:57:20

#	Time	Action	Message	Duration / Fetch
8	10:22:56	SELECT rental_id FROM rental WHERE customer_id = 1 LIMIT 0, 1000	32 row(s) returned	0.000 sec / 0.000 sec
9	10:24:04	SELECT * FROM film WHERE rental_duration > 5 LIMIT 0, 1000	403 row(s) returned	0.016 sec / 0.000 sec
10	10:25:07	SELECT COUNT(*) AS total_films FROM film WHERE replacement_cost > 15 AND replacement_cost < 20 LI...	1 row(s) returned	0.000 sec / 0.000 sec
11	10:26:01	SELECT COUNT(DISTINCT first_name) AS unique_first_names FROM actor LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec
12	10:27:11	SELECT * FROM customer LIMIT 10	10 row(s) returned	0.000 sec / 0.000 sec

11. Display the first 3 records from the customer table whose first name starts with ‘b’.

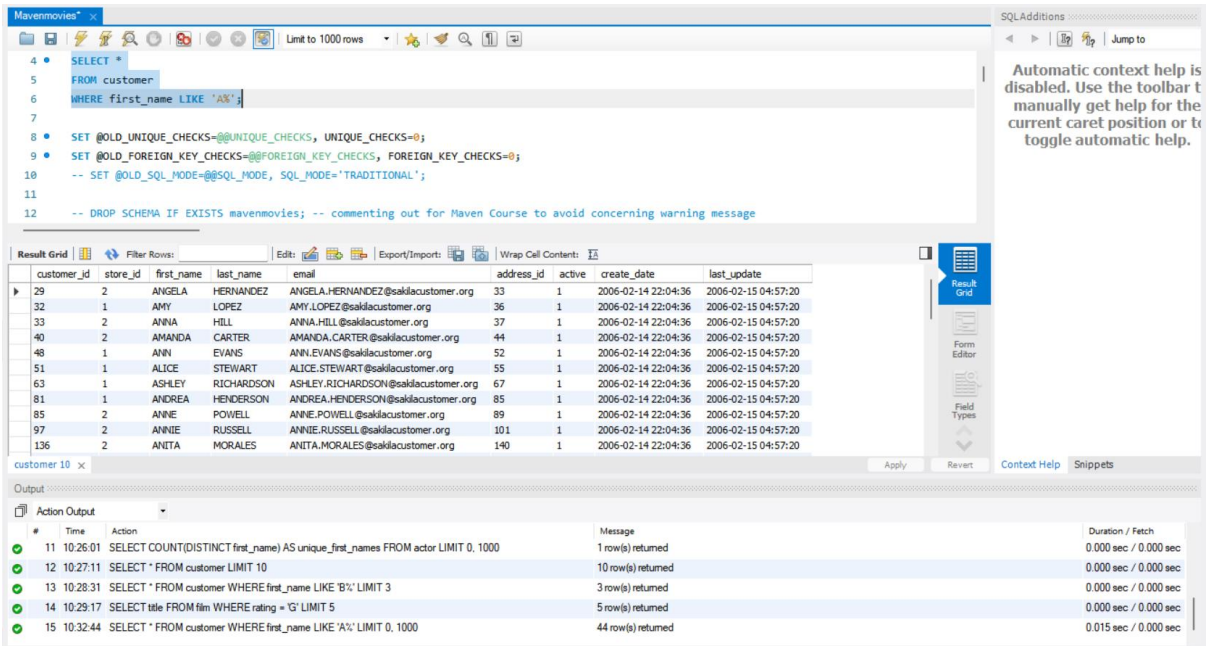


```
USE mavenmovies;
SELECT * FROM customer;

SELECT *
FROM customer
WHERE first_name LIKE 'B%'
LIMIT 3;

SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
```

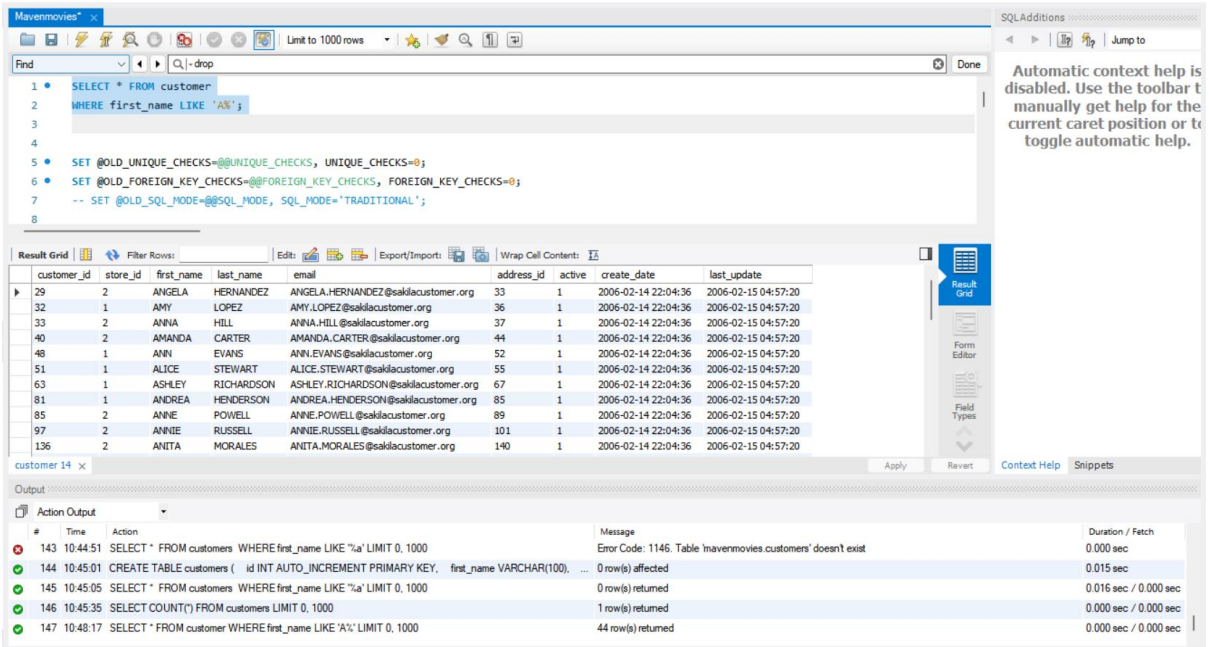
customer_id	store_id	first_name	last_name	email	address_id	active	create_date	last_update
4	2	BARBARA	JONES	BARBARA.JONES@sakilacustomer.org	8	1	2006-02-14 22:04:36	2006-02-15 04:57:20
14	2	BETTY	WHITE	BETTY.WHITE@sakilacustomer.org	18	1	2006-02-14 22:04:36	2006-02-15 04:57:20
31	2	BRENDA	WRIGHT	BRENDA.WRIGHT@sakilacustomer.org	35	1	2006-02-14 22:04:36	2006-02-15 04:57:20

#	Time	Action	Message	Duration / Fetch
9	10:24:04	SELECT * FROM film WHERE rental_duration > 5 LIMIT 0, 1000	403 row(s) returned	0.016 sec / 0.000 sec
10	10:25:07	SELECT COUNT(*) AS total_films FROM film WHERE replacement_cost > 15 AND replacement_cost < 20 LI...	1 row(s) returned	0.000 sec / 0.000 sec
11	10:26:01	SELECT COUNT(DISTINCT first_name) AS unique_first_names FROM actor LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec
12	10:27:11	SELECT * FROM customer LIMIT 10	10 row(s) returned	0.000 sec / 0.000 sec
13	10:28:31	SELECT * FROM customer WHERE first_name LIKE 'B%' LIMIT 3	3 row(s) returned	0.000 sec / 0.000 sec

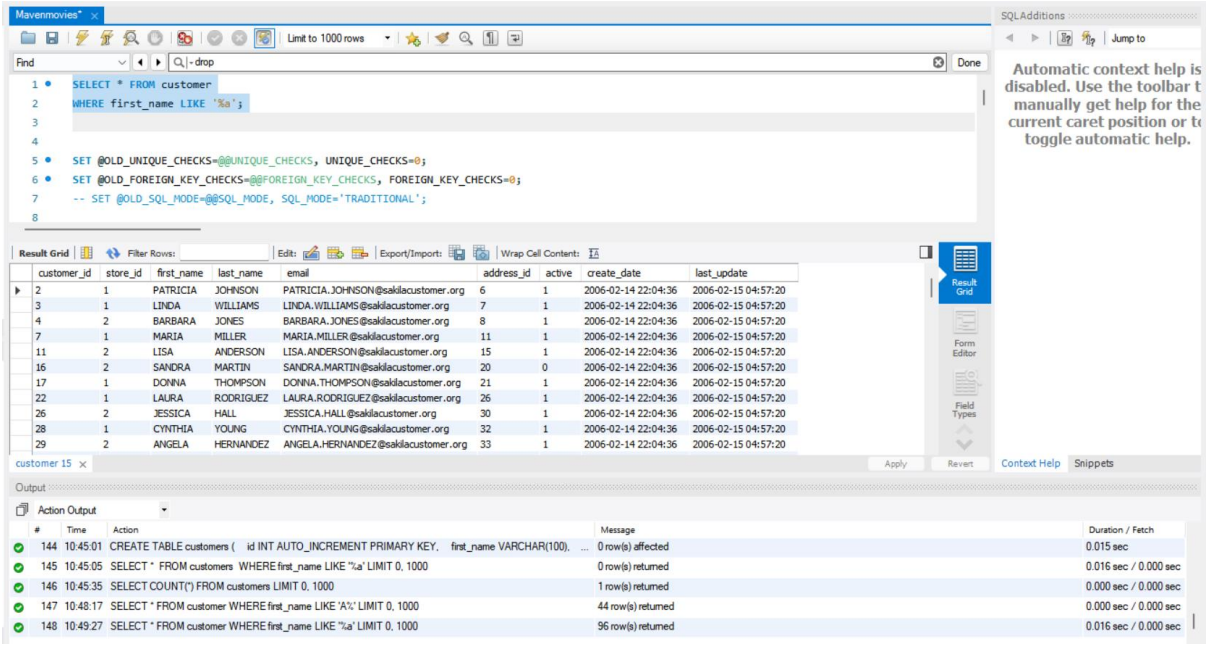
12. Display the names of the first 5 movies which are rated as ‘G.



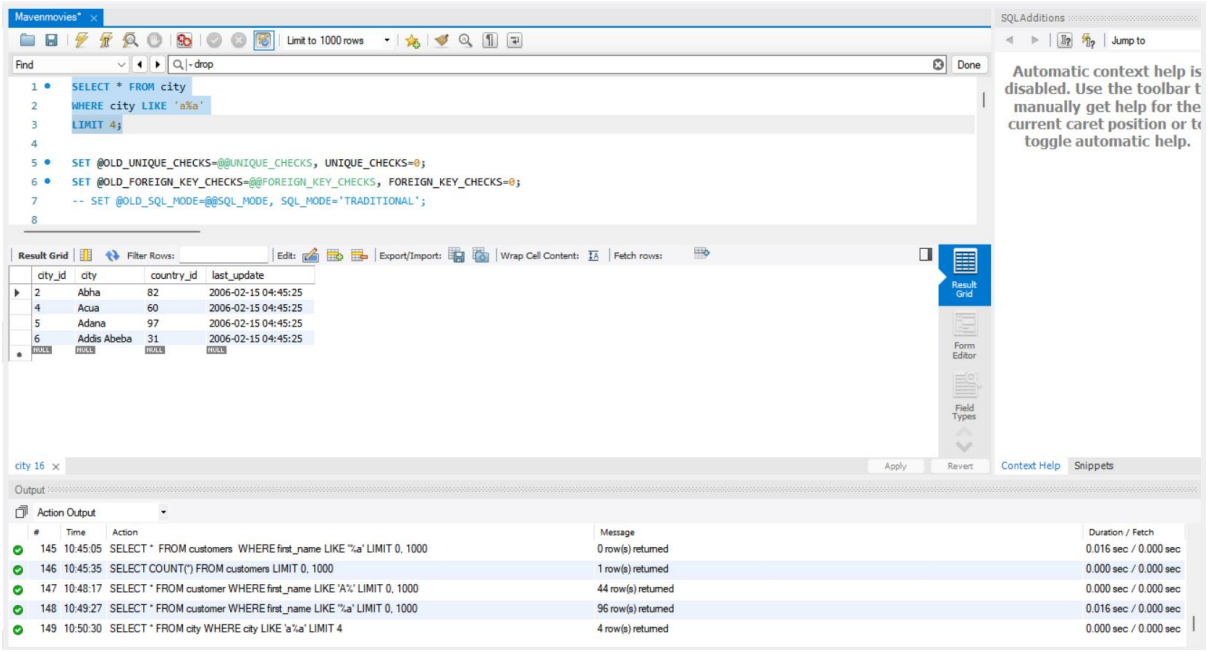
13. Find all customers whose first name starts with "a".



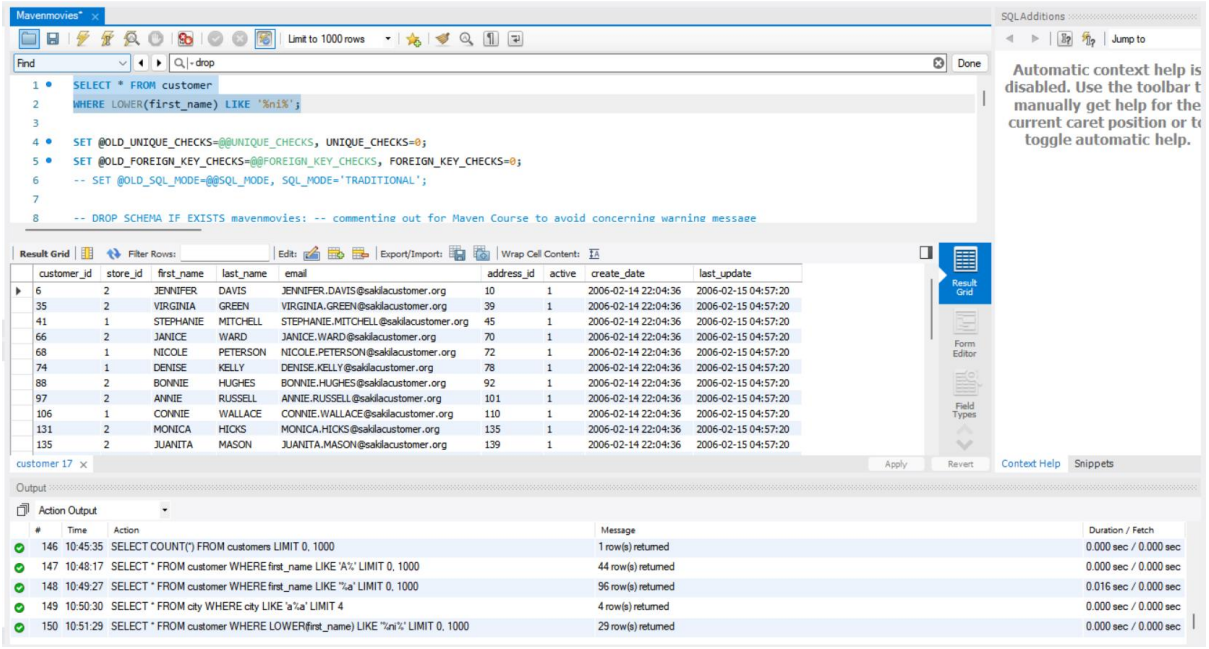
14. Find all customers whose first name ends with "a".



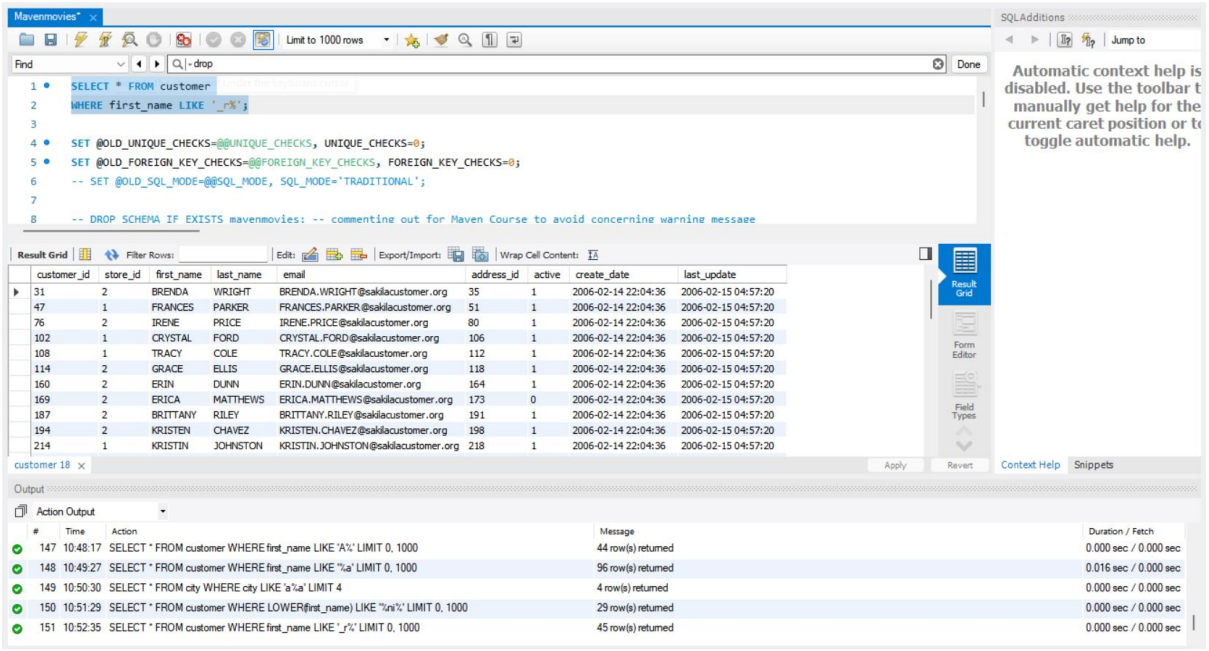
15. Display the list of first 4 cities which start and end with 'a'.



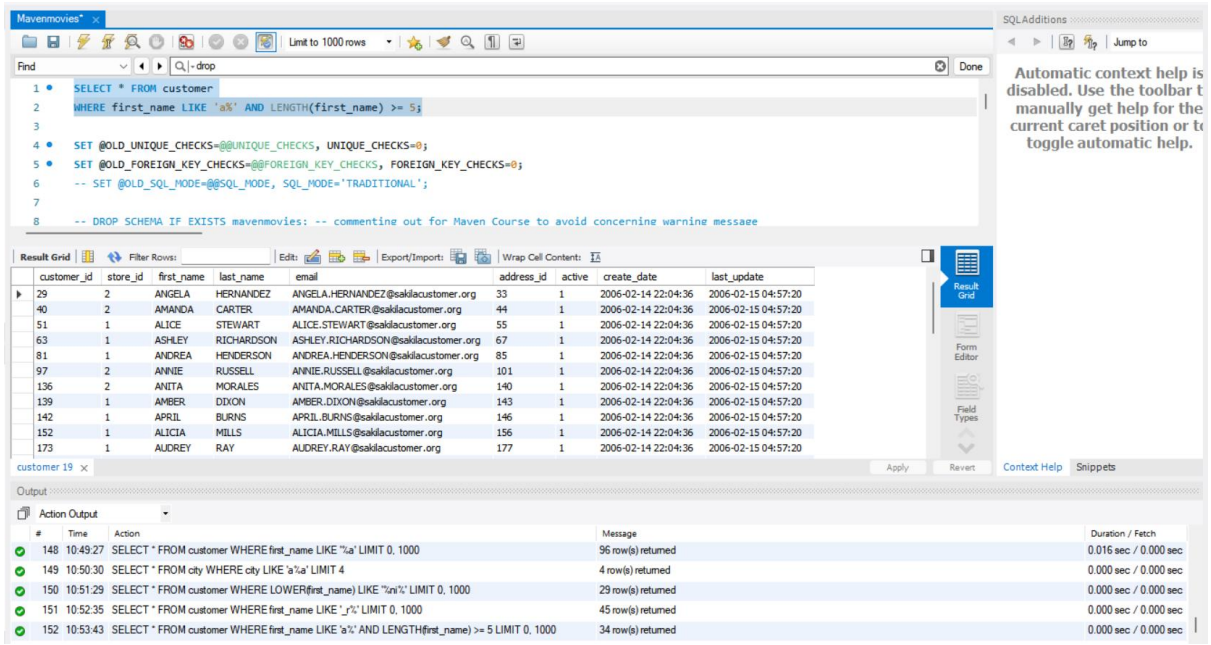
16. Find all customers whose first name have "NI" in any position.



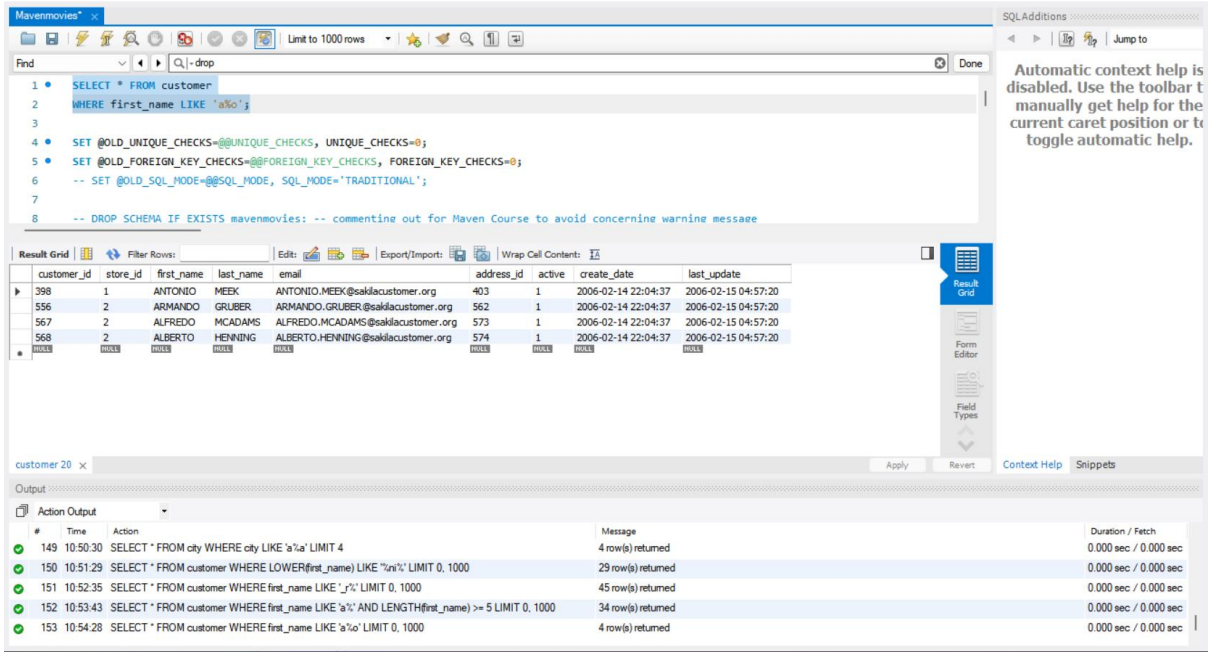
17. Find all customers whose first name have "r" in the second position.



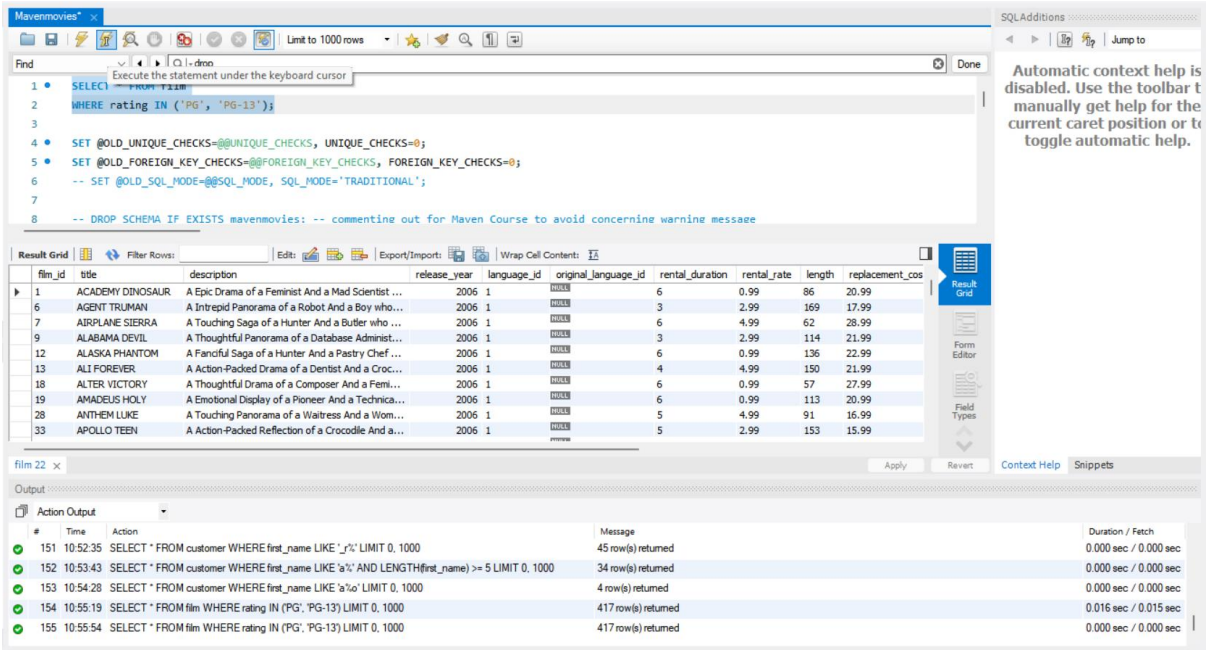
18. Find all customers whose first name starts with "a" and are at least 5 characters in length.



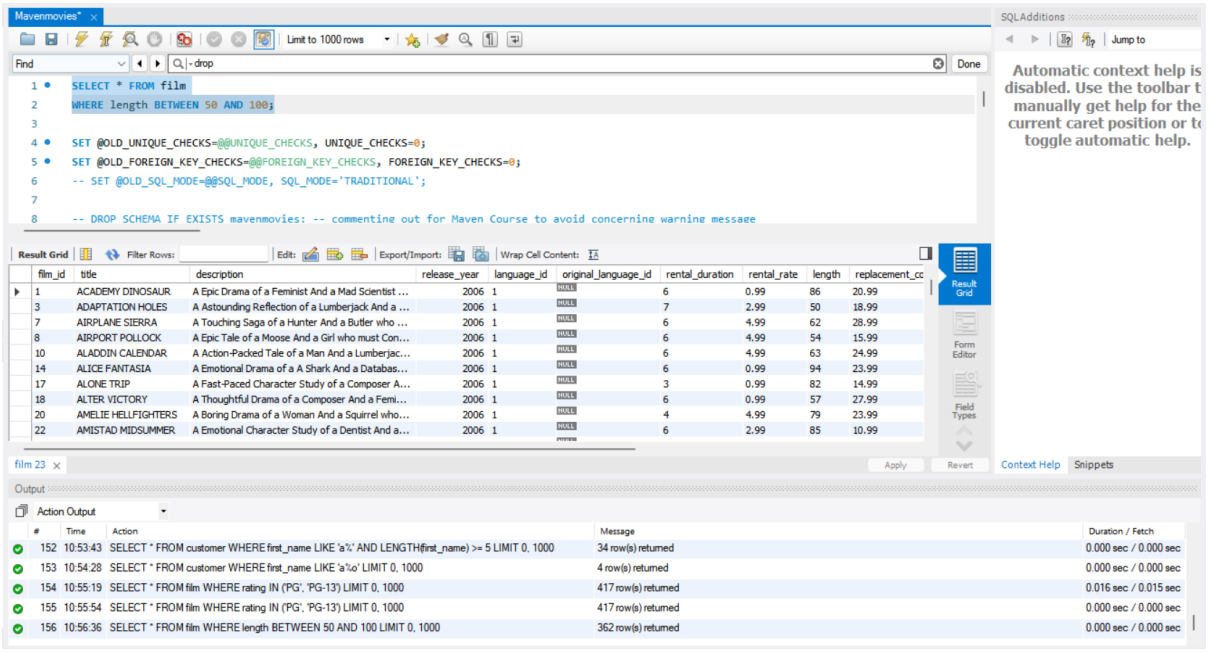
19. Find all customers whose first name starts with "a" and ends with "o".



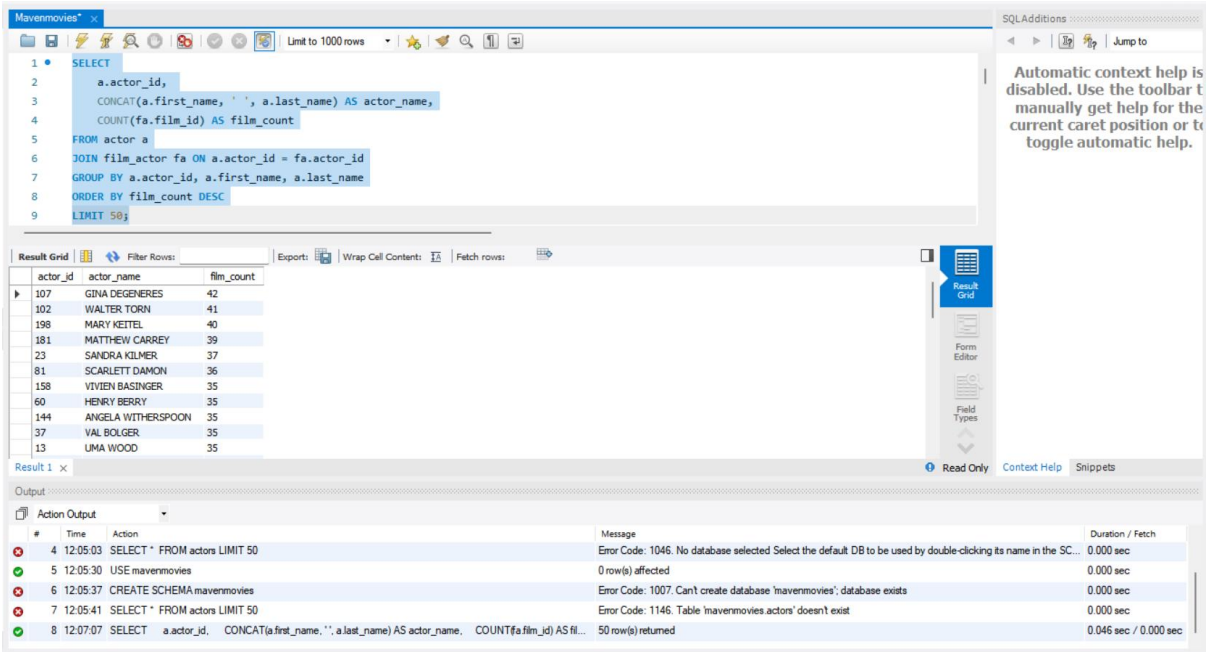
20. Get the films with pg and pg-13 rating using IN operator.



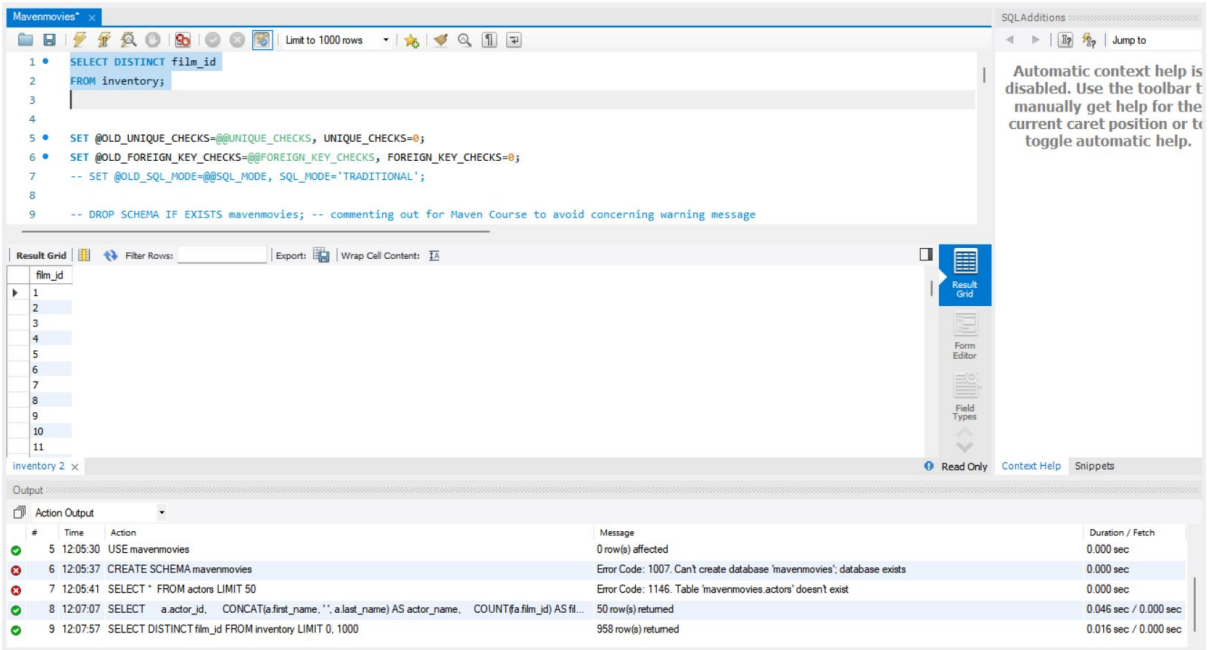
21. Get the films with length between 50 to 100 using between operator.



22. Get the top 50 actors using limit operator.



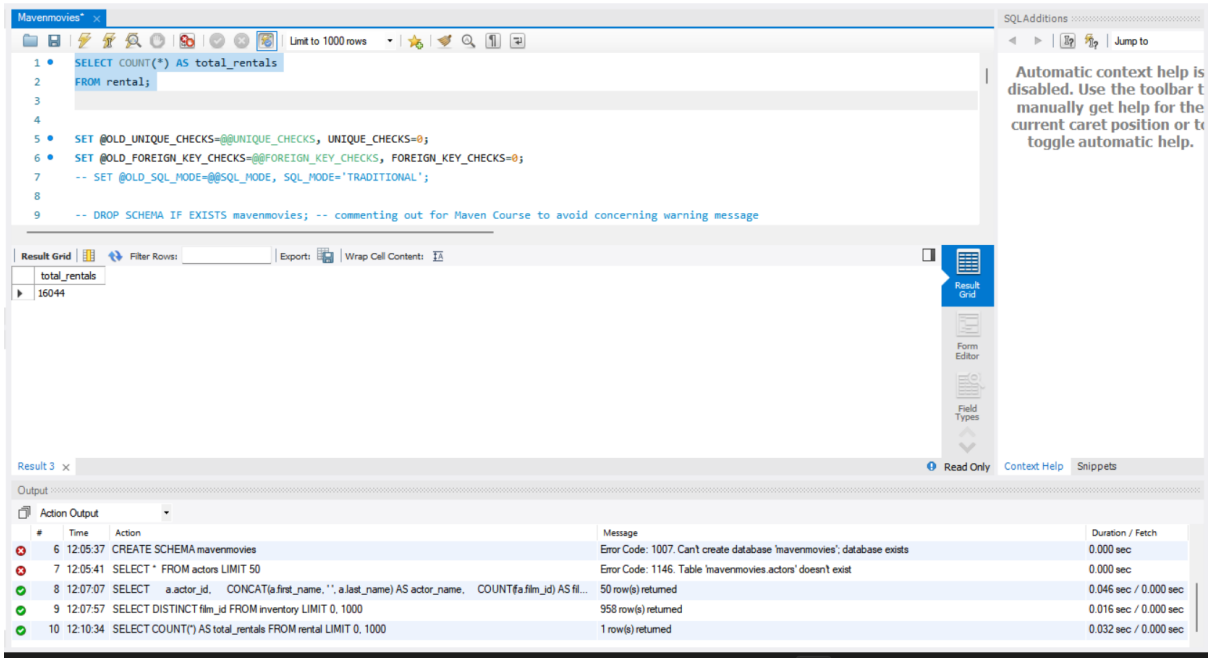
23. Get the distinct film ids from inventory table.



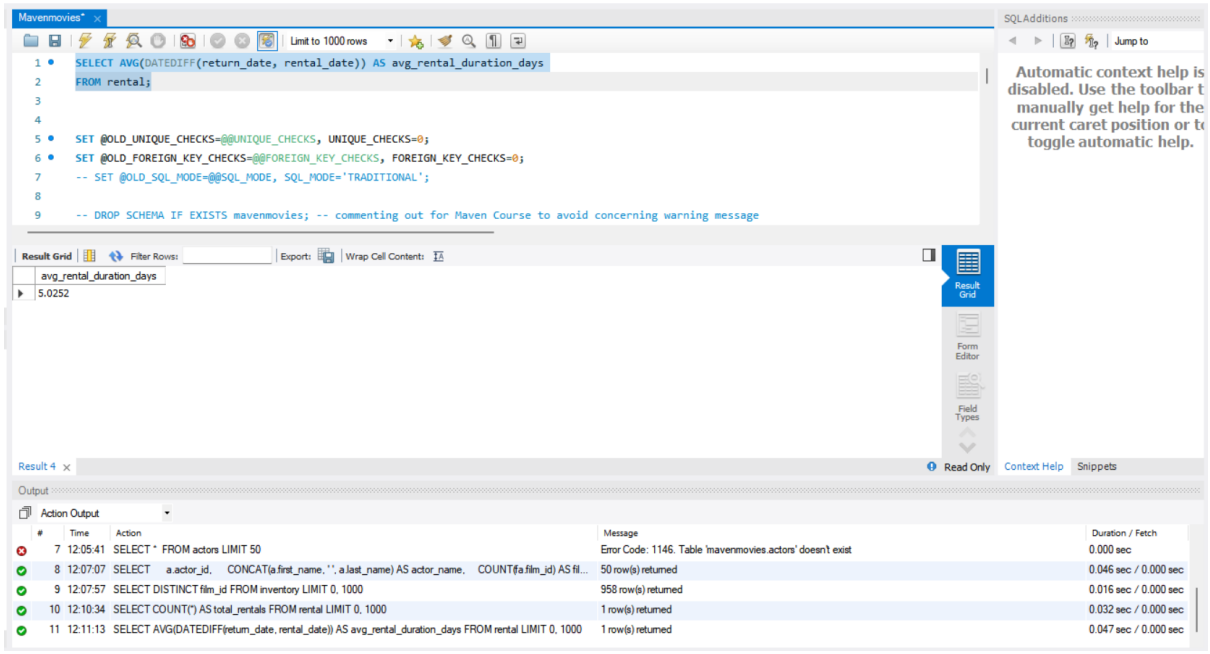
❖ ***Functions***

Basic Aggregate Functions:

Question 1: Retrieve the total number of rentals made in the Sakila database. Hint: Use the COUNT() function.

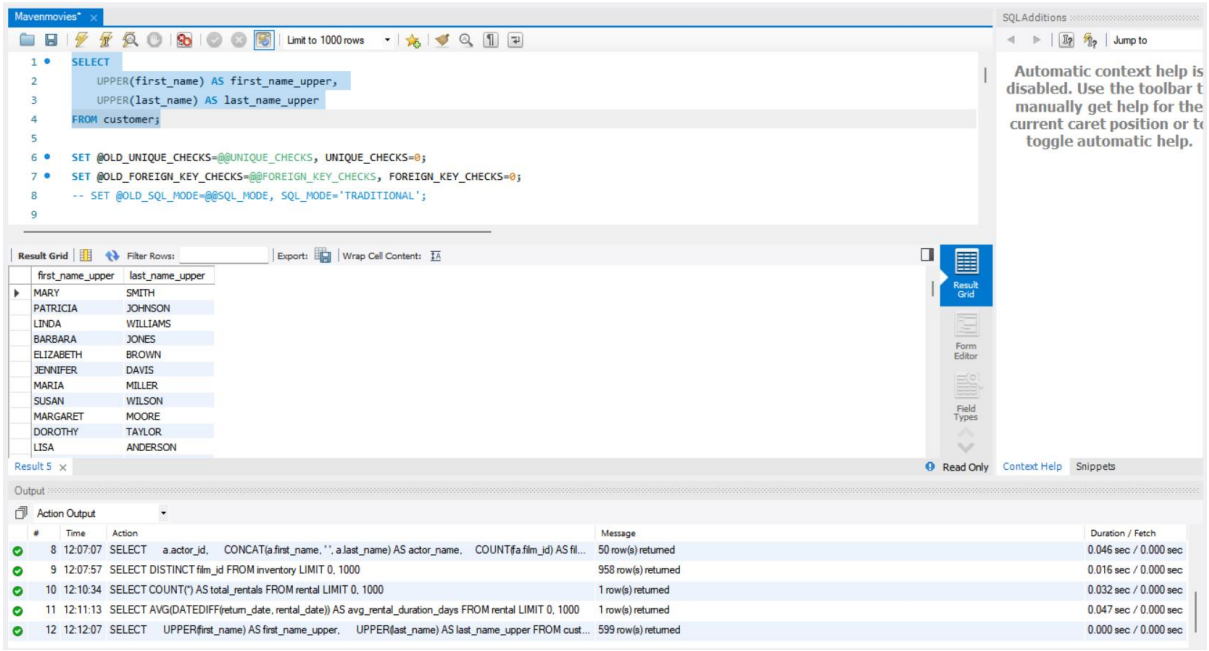


Question 2: Find the average rental duration (in days) of movies rented from the Sakila database.
Hint: Utilize the AVG() function.



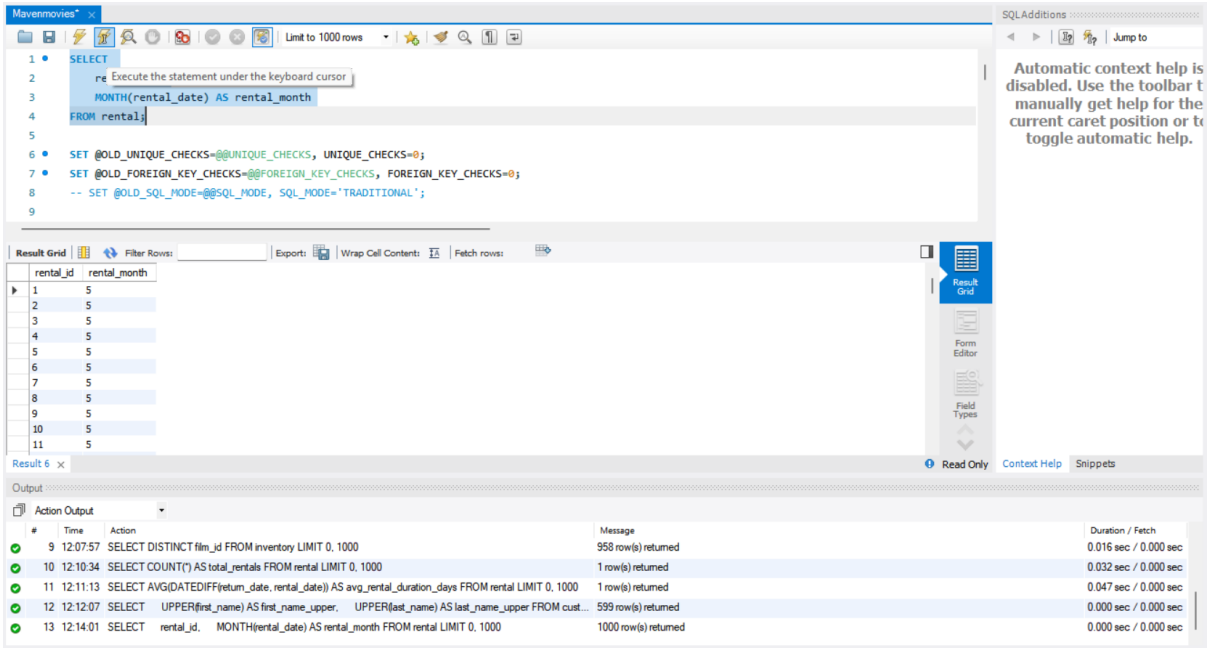
Question 3: Display the first name and last name of customers in uppercase.

Hint: Use the UPPER () function.



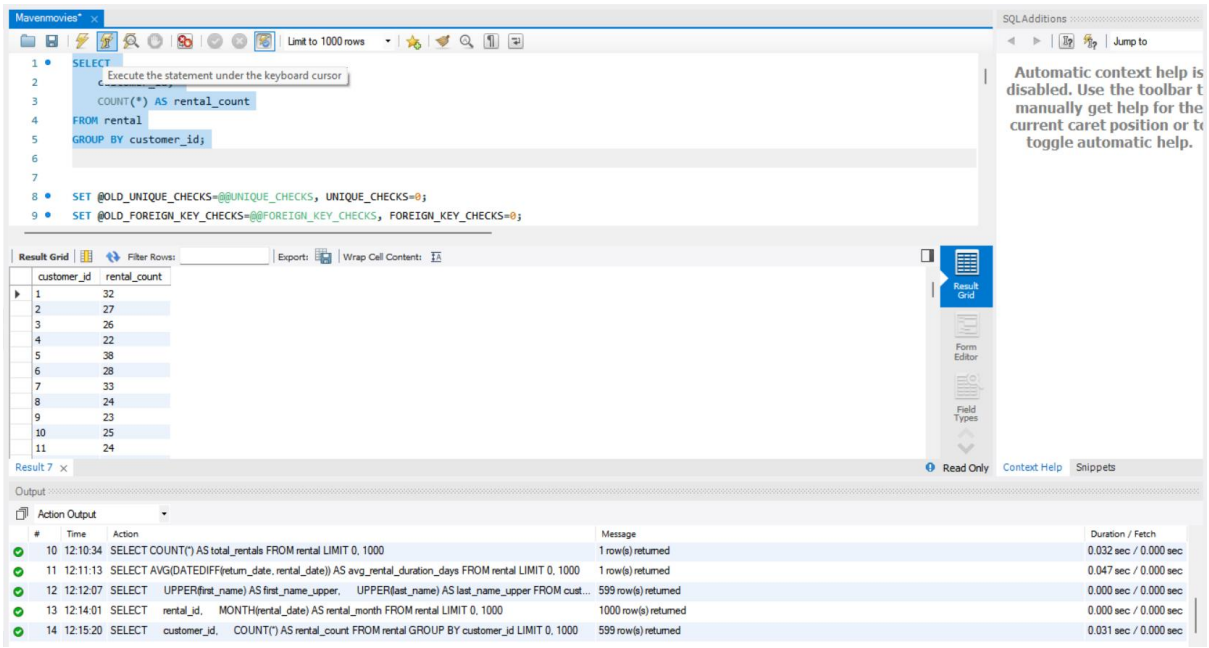
Question 4: Extract the month from the rental date and display it alongside the rental ID.

Hint: Employ the MONTH() function.



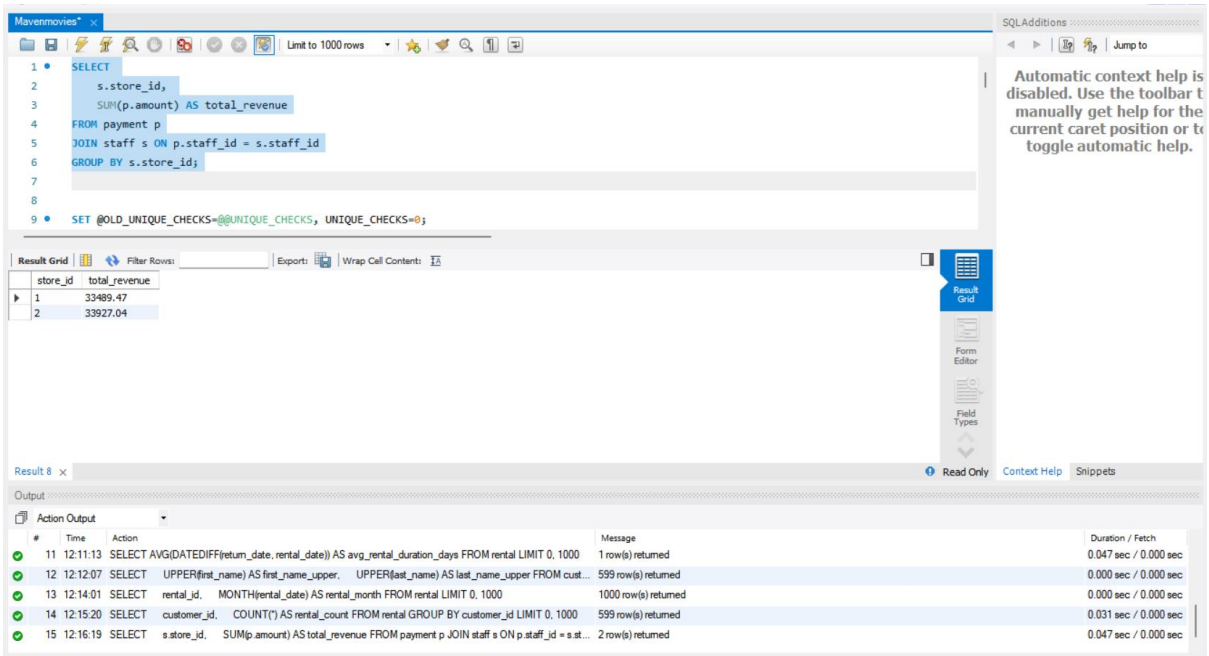
Question 5: Retrieve the count of rentals for each customer (display customer ID and the count of rentals).

Hint: Use COUNT () in conjunction with GROUP BY.



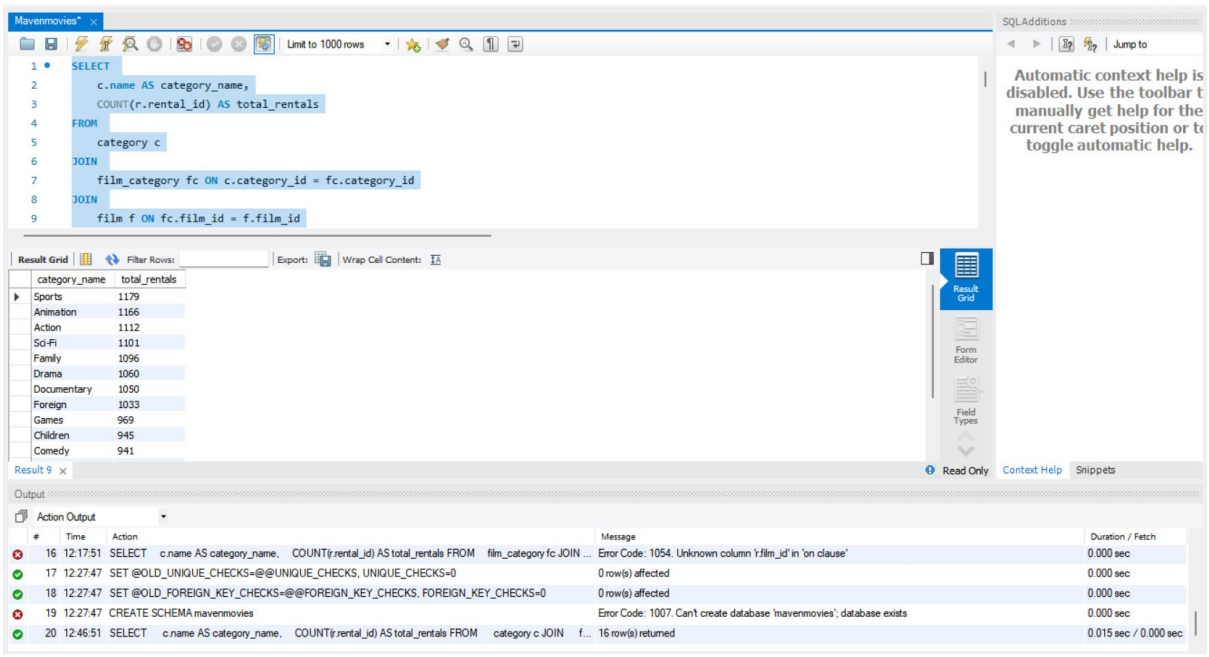
Question 6: Find the total revenue generated by each store.

Hint: Combine SUM() and GROUP BY.



Question 7: Determine the total number of rentals for each category of movies.

Hint: JOIN film_category, film, and rental tables, then use cOUNT () and GROUP BY.

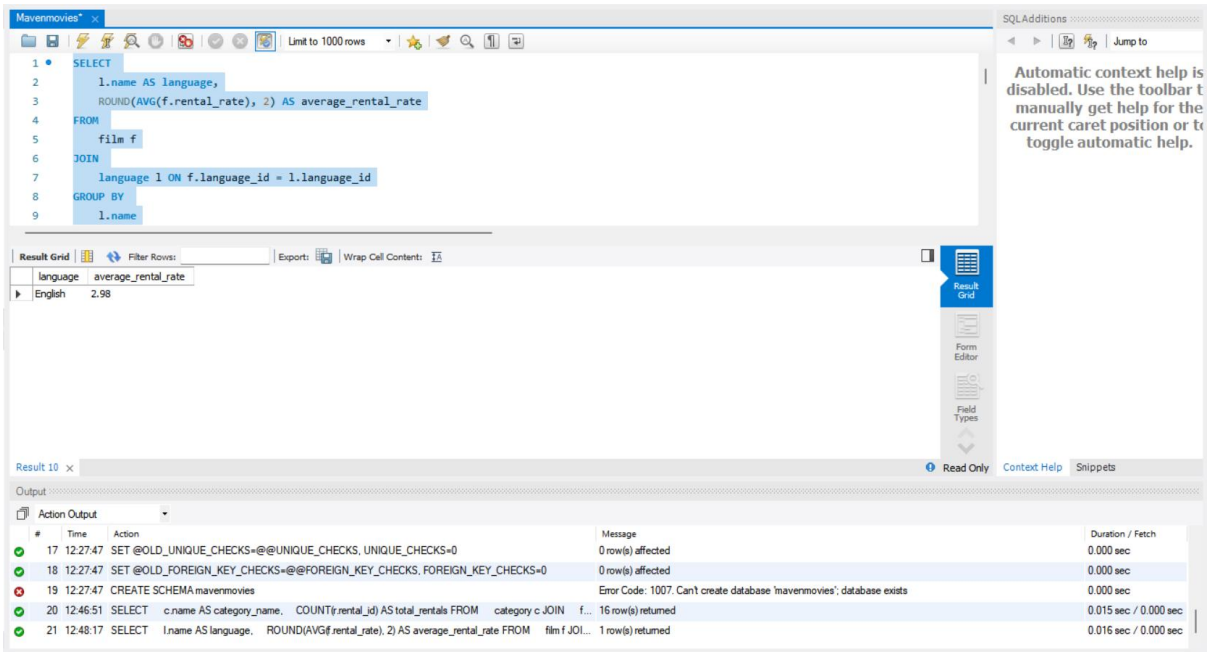


```
SELECT
    c.name AS category_name,
    COUNT(r.rental_id) AS total_rentals
FROM
    category c
JOIN
    film_category fc ON c.category_id = fc.category_id
JOIN
    film f ON fc.film_id = f.film_id
```

category_name	total_rentals
Sports	1179
Animation	1166
Action	1112
Sci-Fi	1101
Family	1096
Drama	1060
Documentary	1050
Foreign	1033
Games	969
Children	945
Comedy	941

Question 8: Find the average rental rate of movies in each language.

Hint: JOIN film and language tables, then use AVG () and GROUP BY.

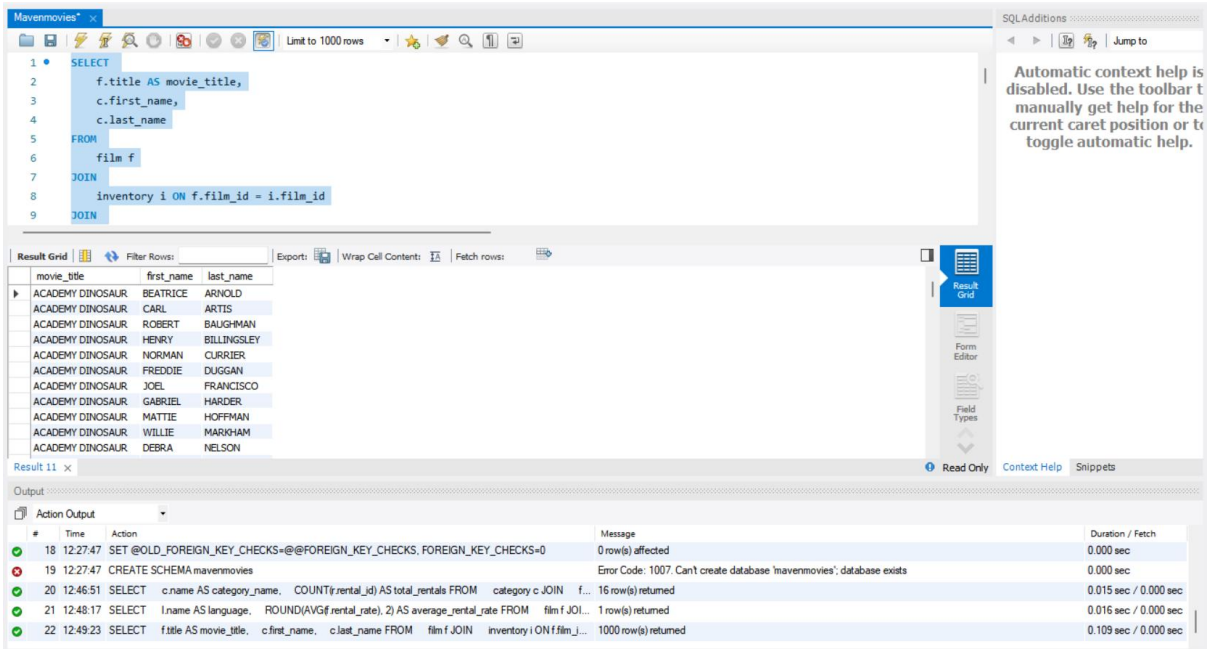


```
SELECT
    l.name AS language,
    ROUND(AVG(f.rental_rate), 2) AS average_rental_rate
FROM
    film f
JOIN
    language l ON f.language_id = l.language_id
GROUP BY
    l.name
```

language	average_rental_rate
English	2.98

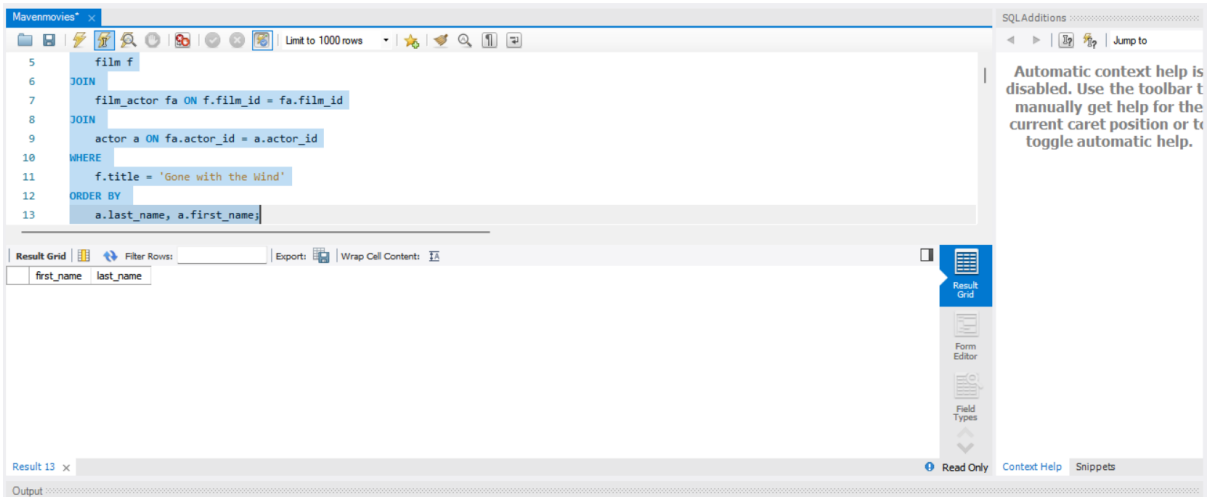
Questions 9 - Display the title of the movie, customer s first name, and last name who rented it.

Hint: Use JOIN between the film, inventory, rental, and customer tables

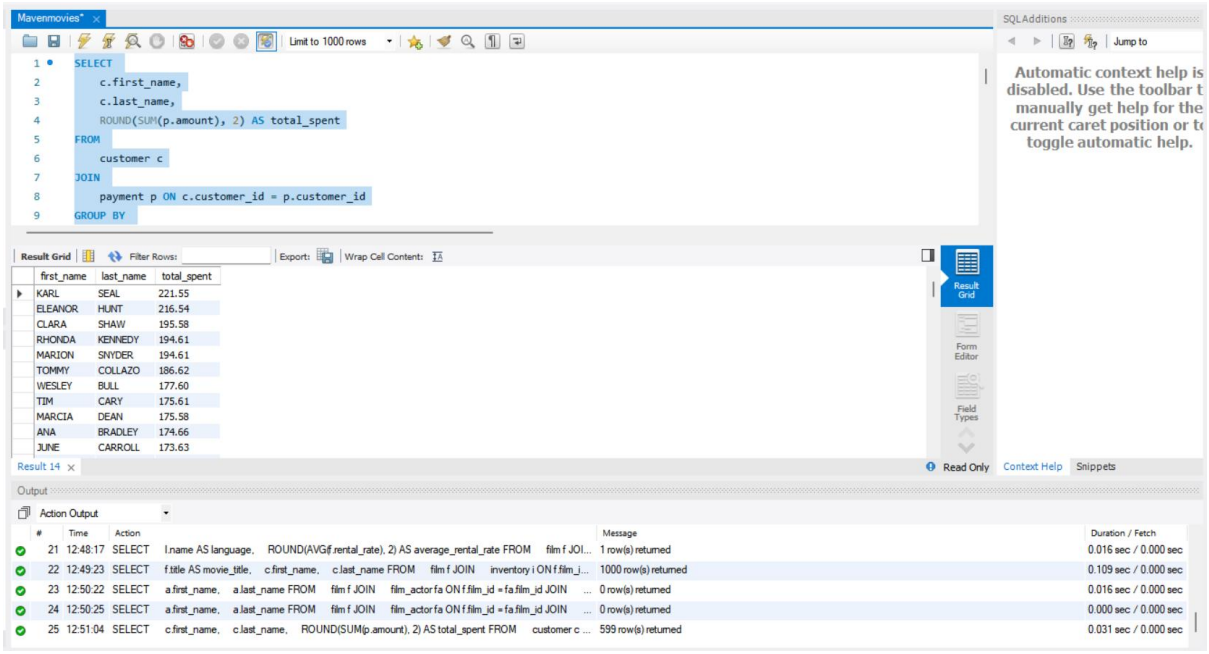


Question 10: Retrieve the names of all actors who have appeared in the film "Gone with the Wind."

Hint: Use JOIN between the film actor, film, and actor table.



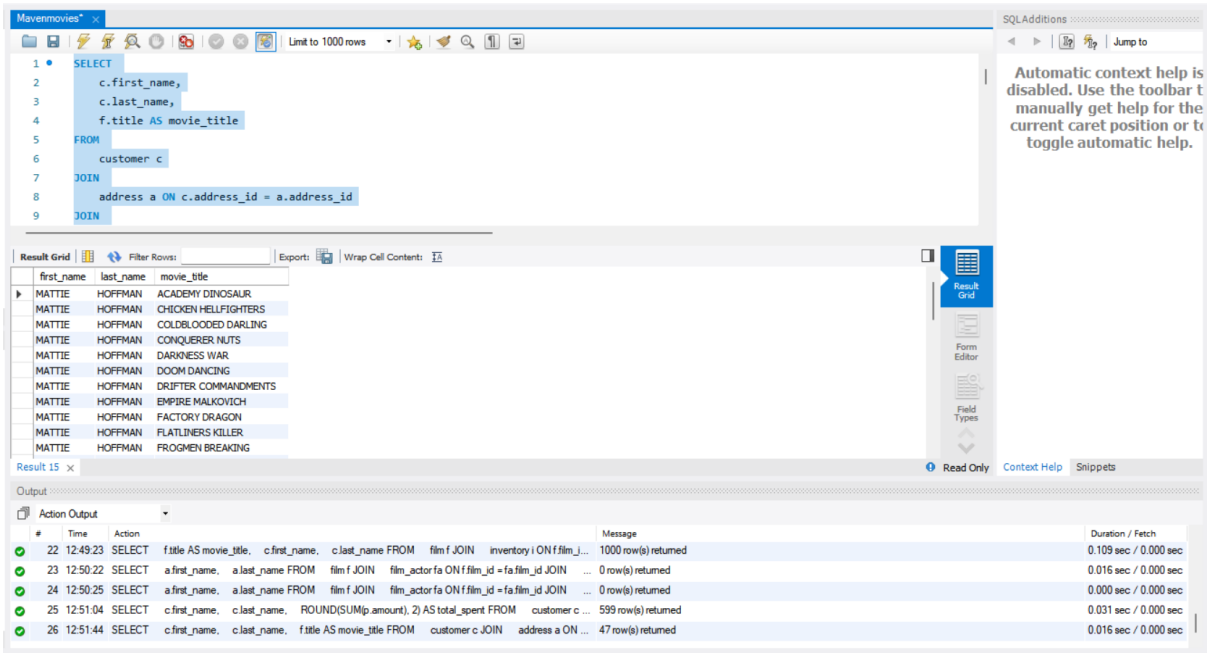
Question 11: Retrieve the customer names along with the total amount they've spent on rentals. Hint: JOIN customer, payment, and rental tables, then use SUM() and GROUP BY.



```
SELECT
    c.first_name,
    c.last_name,
    ROUND(SUM(p.amount), 2) AS total_spent
FROM
    customer c
JOIN
    payment p ON c.customer_id = p.customer_id
GROUP BY
```

first_name	last_name	total_spent
KARL	SEAL	221.55
ELEANOR	HUNT	216.54
CLARA	SHAW	195.58
RHONDA	KENNEDY	194.61
MARION	SNYDER	194.61
TOMMY	COLLAZO	186.62
WESLEY	BULL	177.60
TIM	CARY	175.61
MARCIA	DEAN	175.58
ANA	BRADLEY	174.66
JUNE	CARROLL	173.63

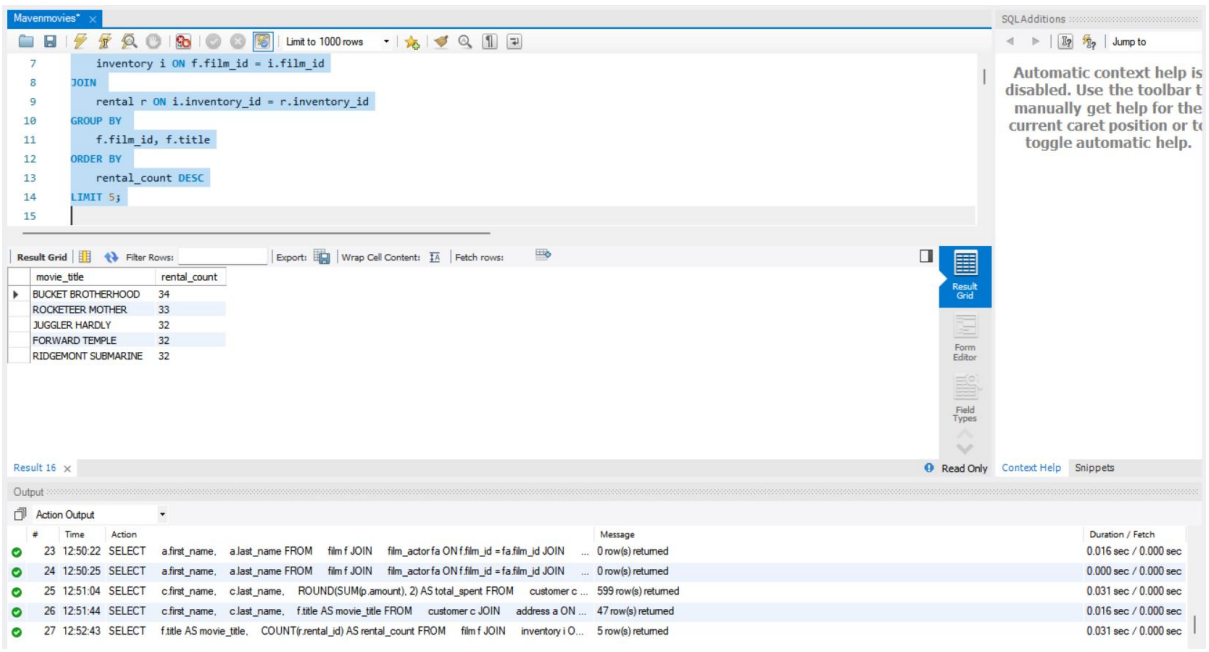
Question 12: List the titles of movies rented by each customer in a particular city (e.g., 'London'). Hint: JOIN customer, address, city, rental, inventory, and film tables, then use GROUP BY.



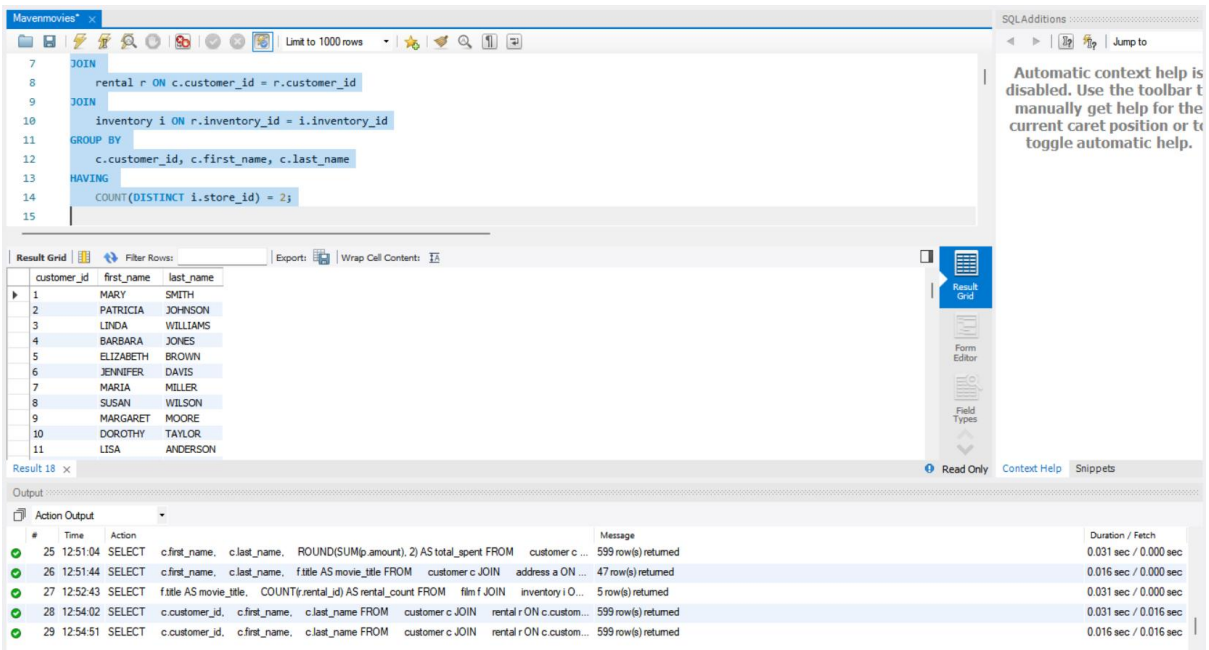
```
SELECT
    c.first_name,
    c.last_name,
    f.title AS movie_title
FROM
    customer c
JOIN
    address a ON c.address_id = a.address_id
JOIN
```

first_name	last_name	movie_title
MATTIE	HOFFMAN	ACADEMY DINOSAUR
MATTIE	HOFFMAN	CHICKEN HELLFIGHTERS
MATTIE	HOFFMAN	COLDBLOODED DARLING
MATTIE	HOFFMAN	CONQUERER NUTS
MATTIE	HOFFMAN	DARKNESS WAR
MATTIE	HOFFMAN	DOOM DANCING
MATTIE	HOFFMAN	DRIFTER COMMANDMENTS
MATTIE	HOFFMAN	EMPIRE MALKOVICH
MATTIE	HOFFMAN	FACTORY DRAGON
MATTIE	HOFFMAN	FLATLINERS KILLER
MATTIE	HOFFMAN	FROGMEN BREAKING

Question 13: Display the top 5 rented movies along with the number of times they've been rented. Hint: JOIN film, inventory, and rental tables, then use COUNT () and GROUP BY, and limit the results.

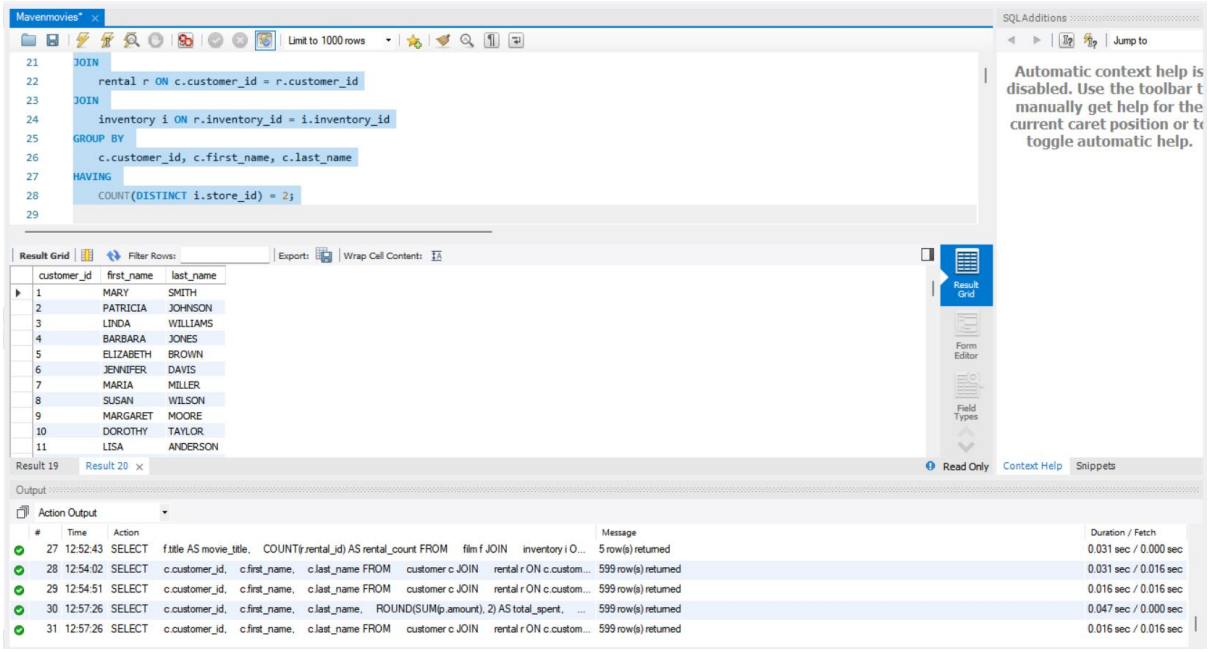


Question 14: Determine the customers who have rented movies from both stores (store ID 1 and store ID 2). Hint: Use JOINS with rental, inventory, and customer tables and consider COUNT() and GROUP BY.

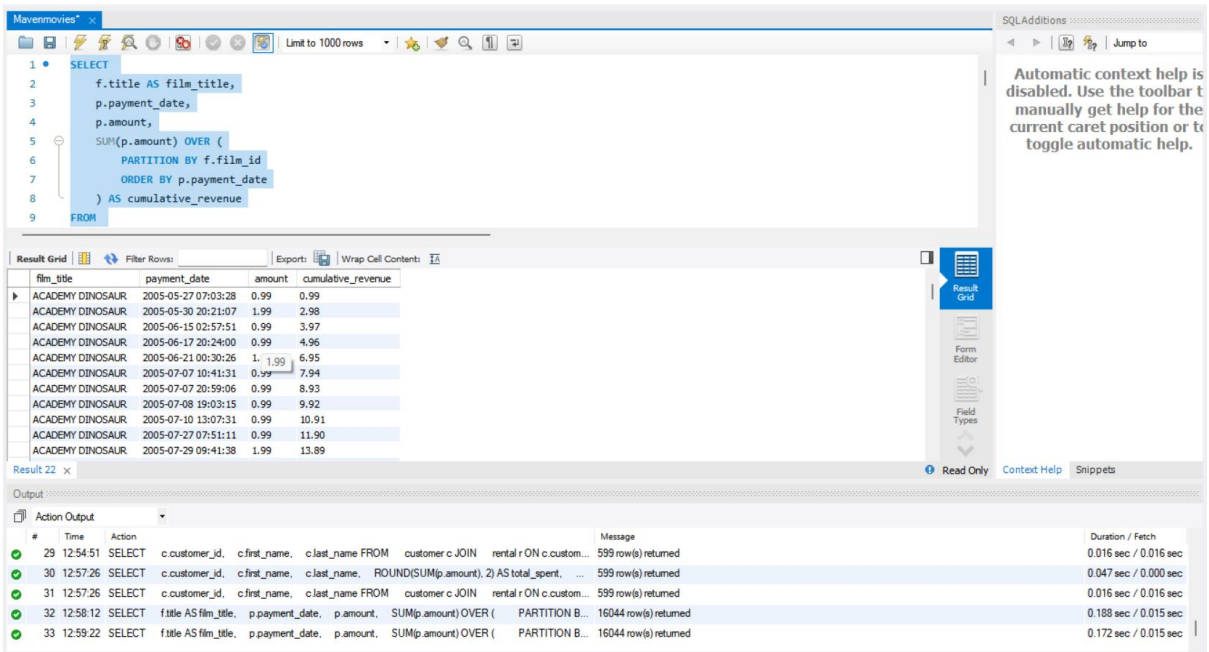


❖ *Windows Function:*

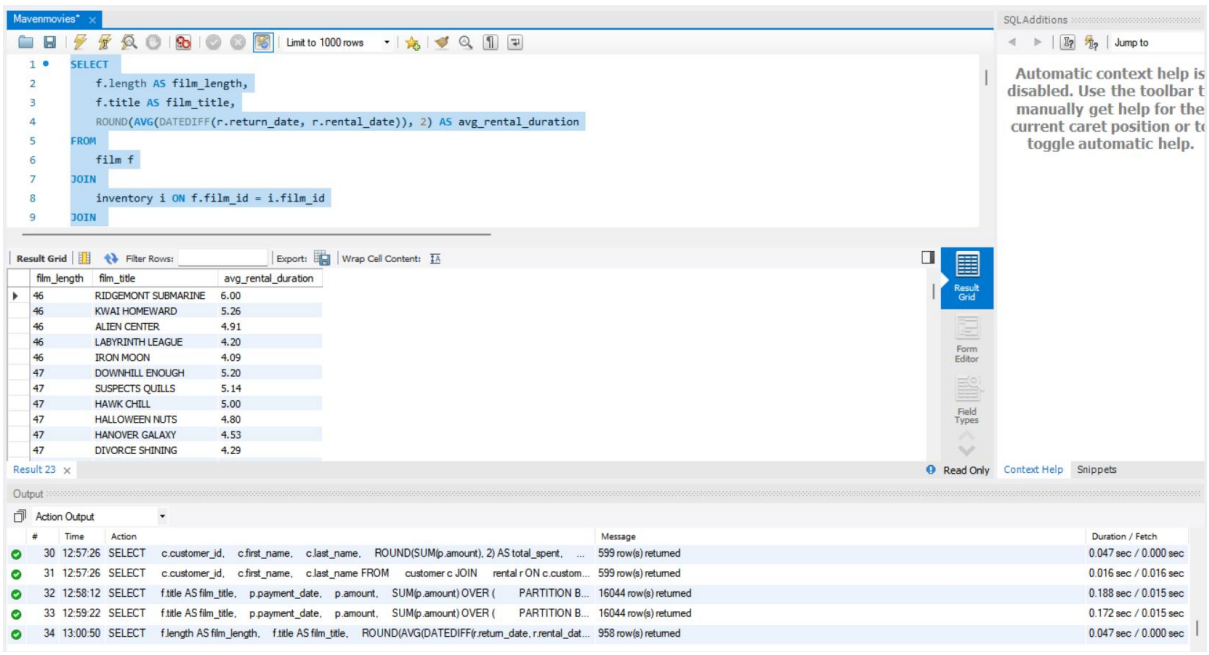
1. **Rank the customers based on the total amount they've spent on rentals.**



2. **Calculate the cumulative revenue generated by each film over time.**



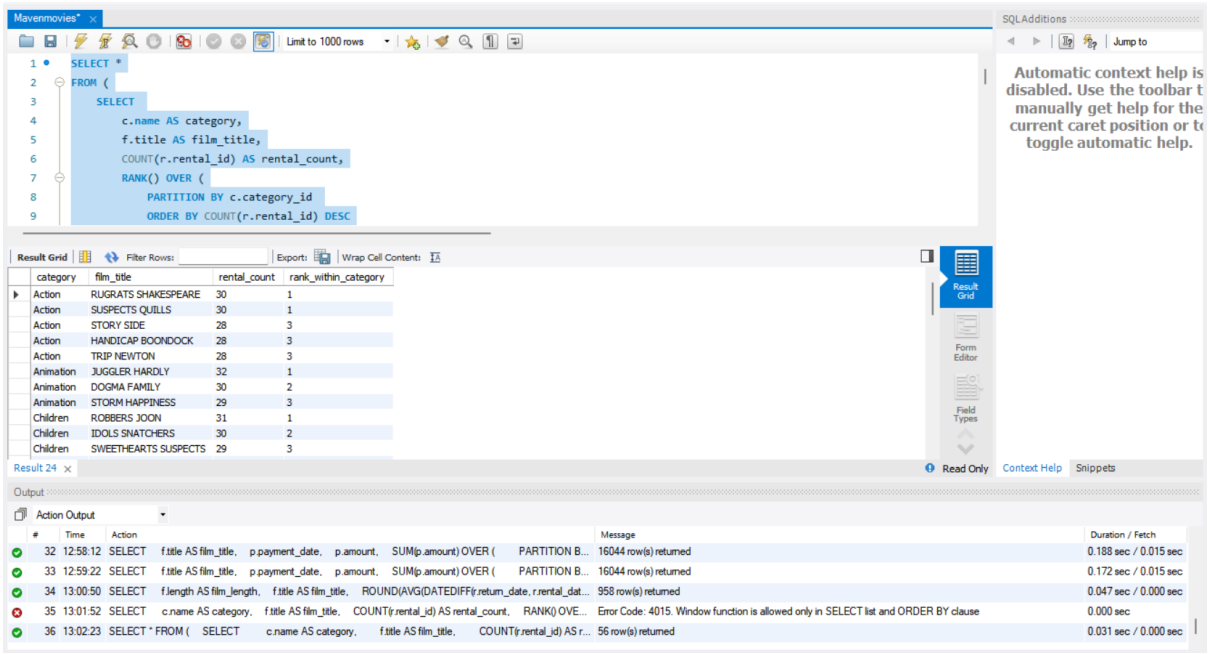
3. **Determine the average rental duration for each film, considering films with similar lengths.**



```
SELECT
    f.length AS film_length,
    f.title AS film_title,
    ROUND(AVG(DATEDIFF(r.return_date, r.rental_date)), 2) AS avg_rental_duration
FROM
    film f
JOIN
    inventory i ON f.film_id = i.film_id
JOIN
```

film_length	film_title	avg_rental_duration
46	RIDGEMONT SUBMARINE	6.00
46	KWAI HOMEWARD	5.26
46	ALIEN CENTER	4.91
46	LABYRINTH LEAGUE	4.20
46	IRON MOON	4.09
47	DOWNHILL ENOUGH	5.20
47	SUSPECTS QUILLS	5.14
47	HAWK CHILL	5.00
47	HALLOWEEN NUTS	4.80
47	HANOVER GALAXY	4.53
47	DIVORCE SHINING	4.29

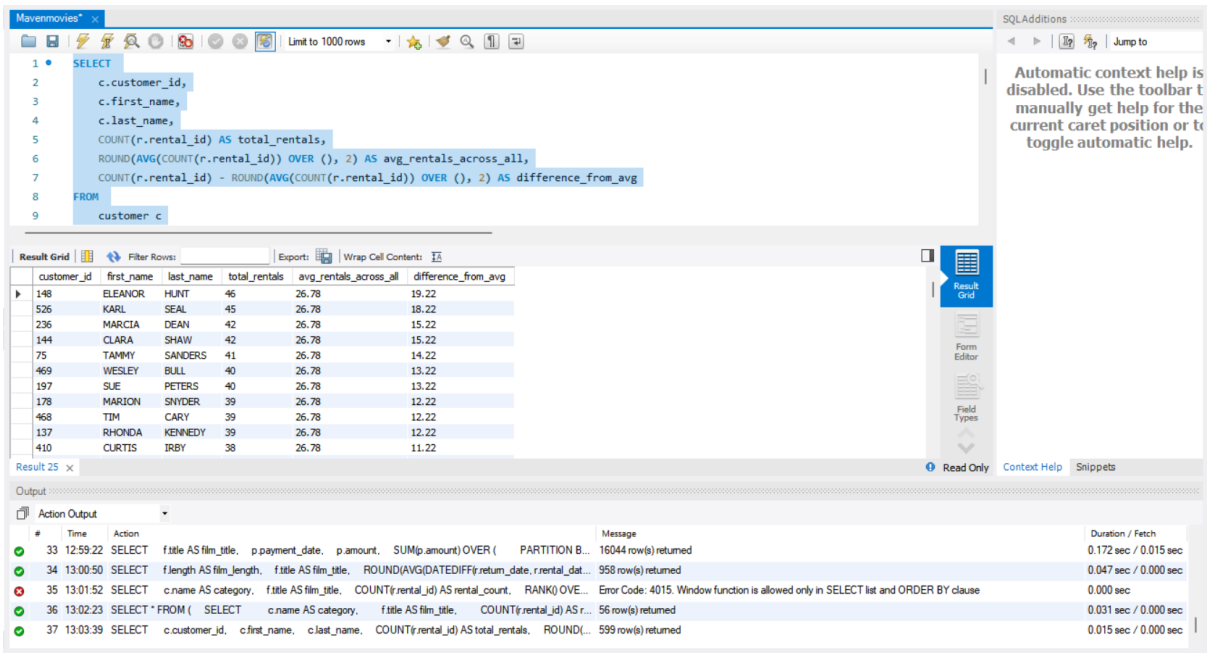
4. **Identify the top 3 films in each category based on their rental counts.**



```
SELECT *
FROM (
    SELECT
        c.name AS category,
        f.title AS film_title,
        COUNT(r.rental_id) AS rental_count,
        RANK() OVER (
            PARTITION BY c.category_id
            ORDER BY COUNT(r.rental_id) DESC
```

category	film_title	rental_count	rank_within_category
Action	RUGRATS SHAKESPEARE	30	1
Action	SUSPECTS QUILLS	30	1
Action	STORY SIDE	28	3
Action	HANDICAP BOONDOCK	28	3
Action	TRIP NEWTON	28	3
Animation	JUGGLER HARDLY	32	1
Animation	DOGMA FAMILY	30	2
Animation	STORM HAPPINESS	29	3
Children	ROBBERS JOON	31	1
Children	IDOLS SNATCHERS	30	2
Children	SWEETHEARTS SUSPECTS	29	3

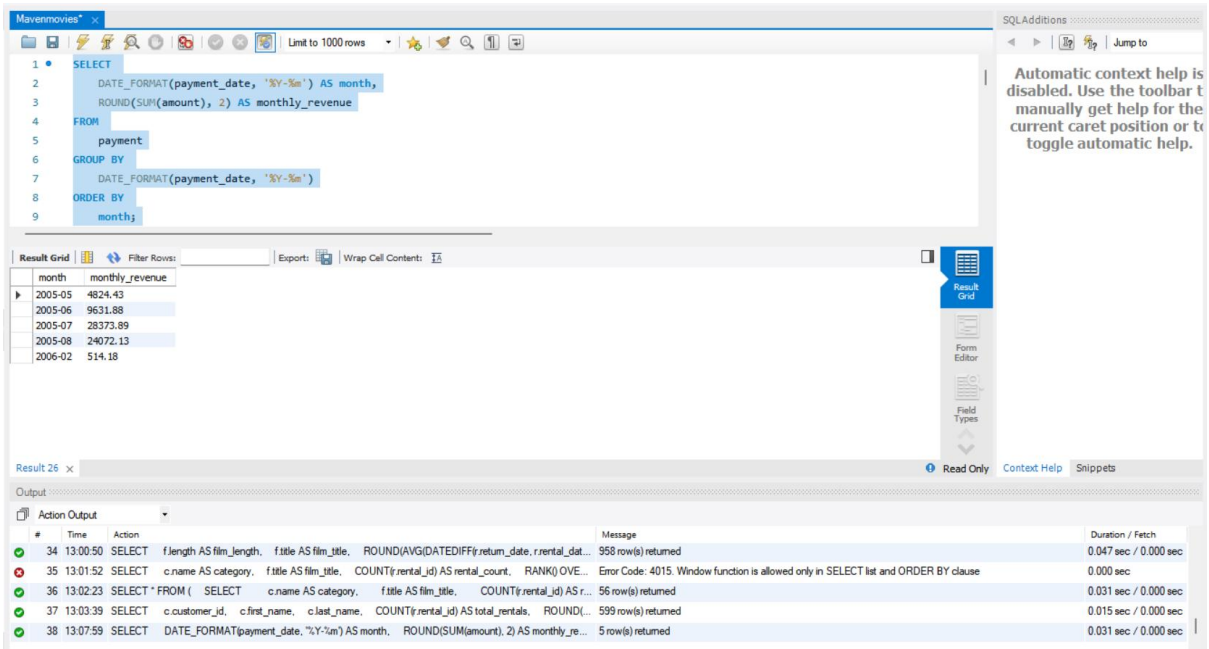
5. **Calculate the difference in rental counts between each customer's total rentals and the average rentals across all customers.**



```
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    COUNT(r.rental_id) AS total_rentals,
    ROUND(AVG(COUNT(r.rental_id)) OVER (), 2) AS avg_rentals_across_all,
    COUNT(r.rental_id) - ROUND(AVG(COUNT(r.rental_id)) OVER (), 2) AS difference_from_avg
FROM
    customer c
```

customer_id	first_name	last_name	total_rentals	avg_rentals_across_all	difference_from_avg
148	ELEANOR	HUNT	46	26.78	19.22
526	KARL	SEAL	45	26.78	18.22
236	MARCIA	DEAN	42	26.78	15.22
144	CLARA	SHAW	42	26.78	15.22
75	TAMMY	SANDERS	41	26.78	14.22
469	WESLEY	BULL	40	26.78	13.22
197	SUE	PETERS	40	26.78	13.22
178	MARION	SNYDER	39	26.78	12.22
468	TIM	CARY	39	26.78	12.22
137	RHONDA	KENNEDY	39	26.78	12.22
410	CURTIS	IRBY	38	26.78	11.22

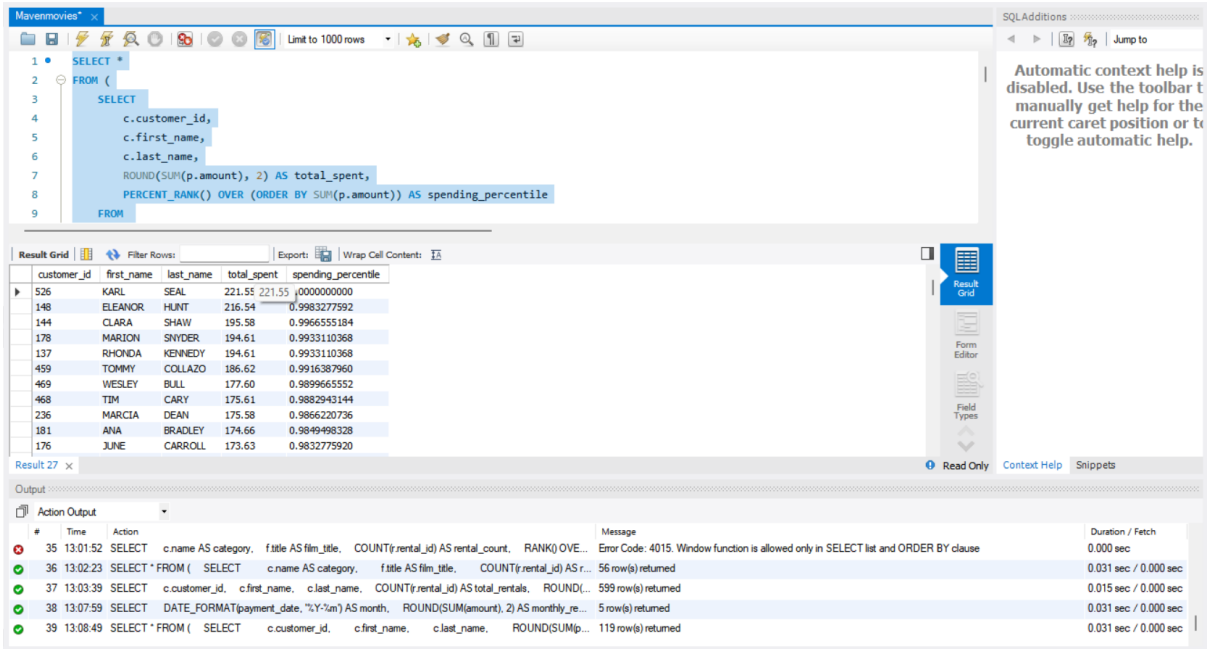
6. **Find the monthly revenue trend for the entire rental store over time.**



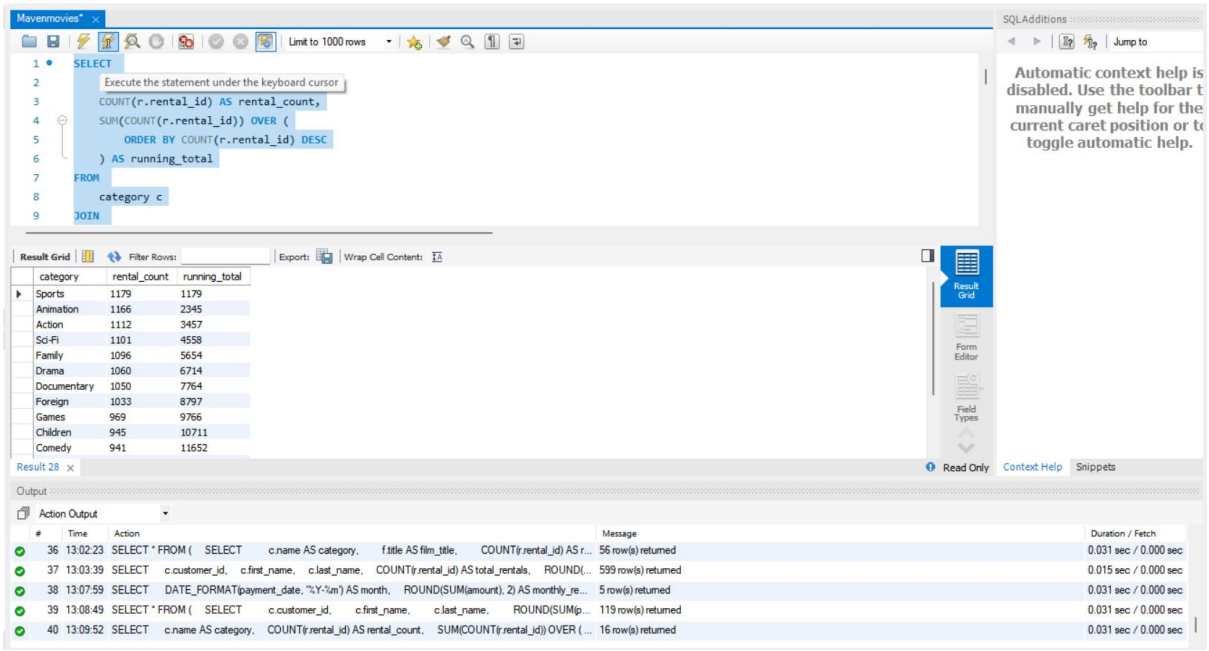
```
SELECT
    DATE_FORMAT(payment_date, '%Y-%m') AS month,
    ROUND(SUM(amount), 2) AS monthly_revenue
FROM
    payment
GROUP BY
    DATE_FORMAT(payment_date, '%Y-%m')
ORDER BY
    month;
```

month	monthly_revenue
2005-05	4824.43
2005-06	9631.88
2005-07	28373.89
2005-08	24072.13
2006-02	514.18

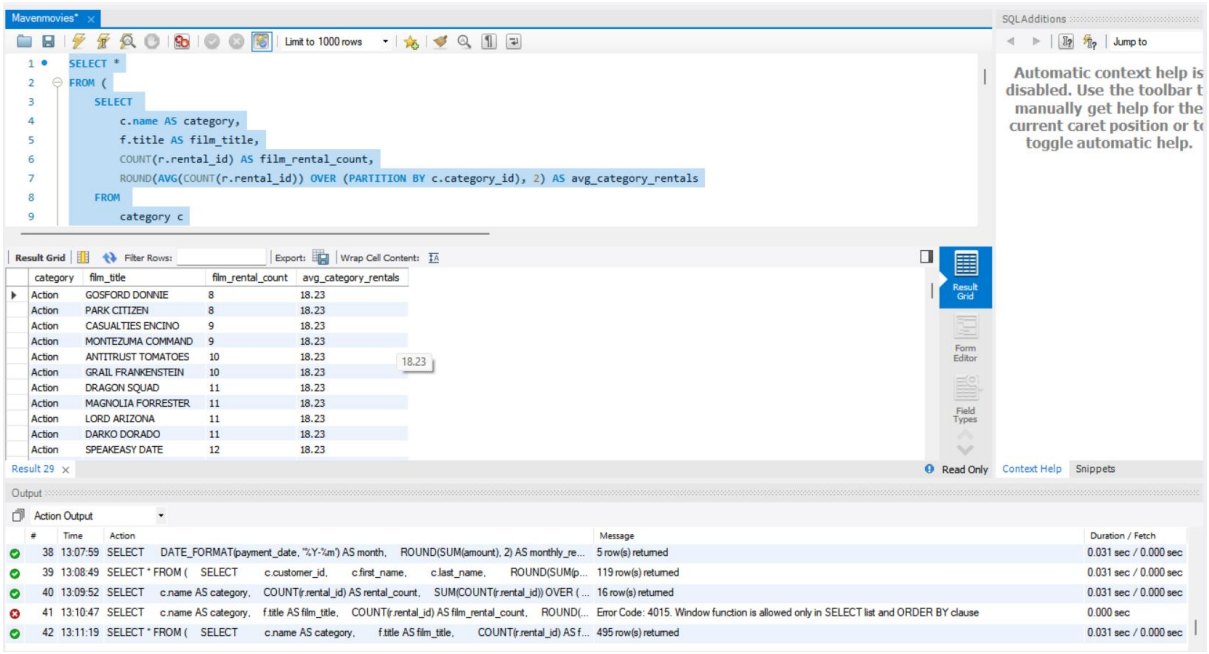
7. **Identify the customers whose total spending on rentals falls within the top 20% of all customers.**



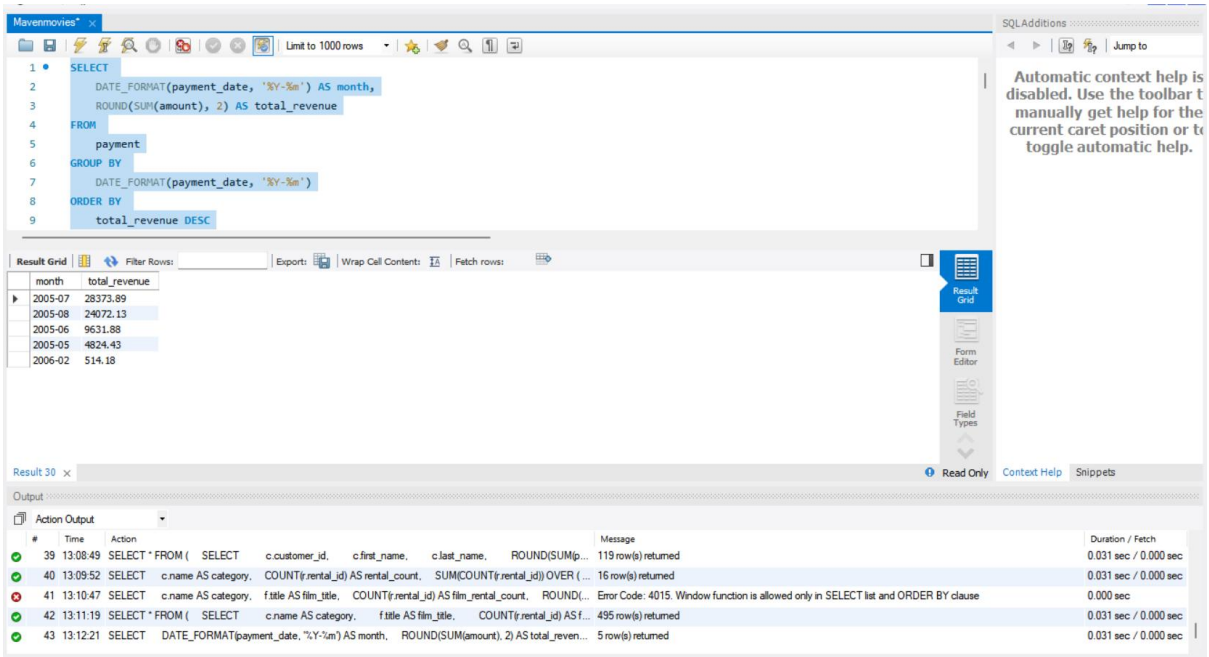
8. **Calculate the running total of rentals per category, ordered by rental count.**



9. **Find the films that have been rented less than the average rental count for their respective categories.**



10. **Identify the top 5 months with the highest revenue and display the revenue generated in each month.**



❖ ***Normalisation & CTE:***

1. **First Normal Form (1NF):**
    a. **Identify a table in the Sakila database that violates 1NF. Explain how you would normalize it to achieve 1NF.**

- In the Sakila database, a good example of a table that potentially violates First Normal Form (1NF) is the film_text table. This table includes a fulltext column, which is used for full-text search purposes. The fulltext column typically contains a concatenated string of searchable keywords related to each film, representing multiple values within a single cell. This violates the 1NF requirement that all attributes must contain only atomic (indivisible) values and that there should be no repeating groups or sets within a column.

To normalize this table and achieve 1NF, we can decompose the non-atomic fulltext column into a new related table. For instance, we can create a separate film_keywords table, where each row represents a single keyword associated with a film. This table would have two columns: film_id and keyword, establishing a one-to-many relationship between films and their associated search terms. By storing each keyword in a separate row, the data becomes atomic and satisfies the conditions of 1NF. This normalization improves data structure, making it easier to search, maintain, and expand.

2. **Second Normal Form (2NF): a. Choose a table in Sakila and describe how you would determine whether it is in 2NF. If it violates 2NF, explain the steps to normalize it.**

    - In the Sakila database, the rental table can be examined to determine if it is in Second Normal Form (2NF). To be in 2NF, a table must first satisfy 1NF, and then ensure that all non-key attributes are fully dependent on the entire primary key—not just part of it. In the rental table, the primary key is typically rental_id, so all other columns like rental_date, inventory_id, customer_id, and return_date depend entirely on this key. However, if we had a composite key (e.g., inventory_id and customer_id) and included attributes like

customer_name, that would violate 2NF because customer details depend only on customer_id, not the entire key. To normalize it, we would move customer-related data to a separate customer table and keep only references (foreign keys) in the rental table, ensuring all non-key attributes depend on the full primary key.

3. **Third Normal Form (3NF): a. Identify a table in Sakila that violates 3NF. Describe the transitive dependencies present and outline the steps to normalize the table to 3NF.**
   - **In the Sakila database, the payment table can be considered when discussing Third Normal Form (3NF). A table is in 3NF if it is already in 2NF and all non-key attributes are only dependent on the primary key, not on other non-key attributes (i.e., no transitive dependencies). If the payment table includes attributes like customer_name or staff_name, these would be transitively dependent on customer_id or staff_id, not directly on payment_id (the primary key). To normalize it to 3NF, such attributes should be moved to their respective customer and staff tables, with only their IDs referenced in the payment table. This removes transitive dependencies and ensures that all non-key attributes depend only on the primary key.**
4. **Normalization Process: a. Take a specific table in Sakila and guide through the process of normalizing it from the initial unnormalized form up.**
   - **In the Sakila database, consider a hypothetical unnormalized version of the customer table where multiple phone numbers and the full address are stored in single fields. For example, a customer might have phone numbers listed as "123456, 789101" and an address stored as one long string like "221B Baker St, London, UK". This violates First Normal Form (1NF) because the fields contain multiple values and are not atomic. To bring the table into 1NF, we split the phone numbers into individual rows and divide the address into separate fields such as street, city, and country. Next, to achieve Second Normal Form (2NF), we remove partial dependencies. If the table uses a composite key (like customer_id and phone_number), fields like name and address should depend only on customer_id, so we separate phone numbers into a**

new customer_phone table and keep personal details in the main customer table. Finally, for Third Normal Form (3NF), we eliminate transitive dependencies. If the country is determined by the city, we move city and country into a separate city table and link it using a foreign key in the customer table. This ensures that all non-key attributes depend only on the primary key, achieving 3NF.

5. **CTE Basics: a. Write a query using a CTE to retrieve the distinct list of actor names and the number of films they have acted in from the actor and film_actor tables.**

```
- WITH actor_film_counts AS (
-     SELECT
-         actor.actor_id,
-         CONCAT(actor.first_name, ' ', actor.last_name) AS actor_name,
-         COUNT(film_actor.film_id) AS film_count
-     FROM
-         actor
-     JOIN
-         film_actor ON actor.actor_id = film_actor.actor_id
-     GROUP BY
-         actor.actor_id, actor.first_name, actor.last_name
- )
- SELECT
-     actor_name, film_count
- FROM
-     actor_film_counts
- ORDER BY
-     film_count DESC;
```

6. **CTE with Joins: a. Create a CTE that combines information from the film and language tables to display the film title, language name, and rental rate.**

```
- WITH film_language_info AS (
-     SELECT
```

```
        f.film_id,
        f.title AS film_title,
        l.name AS language_name,
        f.rental_rate
    FROM
        film f
    JOIN
        language l ON f.language_id = l.language_id
)
SELECT
    film_title,
    language_name,
    rental_rate
FROM
    film_language_info
ORDER BY
    film_title;
```

7. CTE for Aggregation: a. Write a query using a CTE to find the total revenue generated by each customer (sum of payments) from the customer and payment tables.

```
WITH customer_revenue AS (
    SELECT
        c.customer_id,
        CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
        SUM(p.amount) AS total_revenue
    FROM
        customer c
    JOIN
        payment p ON c.customer_id = p.customer_id
    GROUP BY
        c.customer_id, c.first_name, c.last_name
)
SELECT
```

```
    customer_name,
    total_revenue
FROM
    customer_revenue
ORDER BY
    total_revenue DESC;
```

8. **CTE with Window Functions: a. Utilize a CTE with a window function to rank films based on their rental duration from the film table.**

```
WITH film_ranking AS (
    SELECT
        film_id,
        title,
        rental_duration,
        RANK() OVER (ORDER BY rental_duration DESC) AS rental_rank
    FROM
        film
)
SELECT
    title,
    rental_duration,
    rental_rank
FROM
    film_ranking
ORDER BY
    rental_rank;
```

9. **CTE and Filtering: a. Create a CTE to list customers who have made more than two rentals, and then join this CTE with the customer table to retrieve additional customer details.**

```
WITH frequent_renters AS (
    SELECT
        customer_id,
```

```
-        COUNT(rental_id) AS rental_count
-    FROM
-        rental
-    GROUP BY
-        customer_id
-    HAVING
-        COUNT(rental_id) > 2
- )
- SELECT
-    c.customer_id,
-    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
-    c.email,
-    fr.rental_count
- FROM
-    frequent_renters fr
- JOIN
-    customer c ON fr.customer_id = c.customer_id
- ORDER BY
-    fr.rental_count DESC;
```

10. CTE for Date Calculations: a. Write a query using a CTE to find the total number of rentals made each month, considering the rental_date from the rental table.

```
- WITH monthly_rentals AS (
-    SELECT
-        DATE_FORMAT(rental_date, '%Y-%m') AS rental_month,
-        COUNT(rental_id) AS total_rentals
-    FROM
-        rental
-    GROUP BY
-        DATE_FORMAT(rental_date, '%Y-%m')
- )
- SELECT
-    rental_month,
```

```
    total_rentals
FROM
    monthly_rentals
ORDER BY
    rental_month;
```

11. CTE and Self-Join: a. Create a CTE to generate a report showing pairs of actors who have appeared in the same film together, using the film_actor table.

```
WITH actor_pairs AS (
    SELECT
        fa1.film_id,
        fa1.actor_id AS actor1_id,
        fa2.actor_id AS actor2_id
    FROM
        film_actor fa1
    JOIN
        film_actor fa2
        ON fa1.film_id = fa2.film_id AND fa1.actor_id < fa2.actor_id
)
SELECT
    f.title AS film_title,
    CONCAT(a1.first_name, ' ', a1.last_name) AS actor_1,
    CONCAT(a2.first_name, ' ', a2.last_name) AS actor_2
FROM
    actor_pairs ap
JOIN
    film f ON ap.film_id = f.film_id
JOIN
    actor a1 ON ap.actor1_id = a1.actor_id
JOIN
    actor a2 ON ap.actor2_id = a2.actor_id
ORDER BY
    film_title, actor_1, actor_2;
```

12. CTE For recursive search: a. Implement a recursive CTE to find all employees in the staff table who report to a specific manager, considering the reports_to column.

```
- WITH RECURSIVE employee_hierarchy AS (
-     -- Anchor member: start with the manager
-     SELECT
-         staff_id,
-         first_name,
-         last_name,
-         reports_to
-     FROM
-         staff
-     WHERE
-         staff_id = 1  -- specific manager ID
-
-     UNION ALL
-
-     -- Recursive member: find employees who report to someone already in the hierarchy
-     SELECT
-         s.staff_id,
-         s.first_name,
-         s.last_name,
-         s.reports_to
-     FROM
-         staff s
-     INNER JOIN
-         employee_hierarchy eh ON s.reports_to = eh.staff_id
- )
- SELECT
-     staff_id,
-     CONCAT(first_name, ' ', last_name) AS employee_name,
-     reports_to
- FROM
-     employee_hierarchy
```

- **WHERE**
- **staff_id != 1 -- exclude the original manager from the results**
- **ORDER BY**
- **Employee_name;**

Thank You Sir / Ma’am