

## **SQL Basics – Assignment Questions**

**Name:** Darshan Panchal

**Email ID:** [panchaldarshan177@gmail.com](mailto: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.



The screenshot shows a SQL editor window titled "SQL File 5\*". The interface includes a toolbar with various icons for file operations, search, and help. Below the toolbar, the SQL code is displayed in a text area:

```
1 • CREATE DATABASE Company;
2 • USE Company;
3
4 • CREATE TABLE employee (
5     emp_id INT(10) PRIMARY KEY NOT NULL,
6     emp_name VARCHAR(30) NOT NULL,
7     age INT CHECK (age >= 18),
8     email TEXT UNIQUE,
9     salary DECIMAL(10, 2) DEFAULT 30000.00
10 );
11
```

## 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](mailto: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.**



```

4 • CREATE TABLE classes (
5     class_id INT PRIMARY KEY,
6     class_name VARCHAR(100) NOT NULL
7 );
8 • INSERT INTO classes (class_id, class_name) VALUES
9     (1, 'Mathematics'),
10    (2, 'Science'),
11    (3, 'History');
12
13 • CREATE TABLE students (
14     student_id INT PRIMARY KEY,
15     student_name VARCHAR(100) NOT NULL,
16     class_id INT,
17     FOREIGN KEY (class_id) REFERENCES classes(class_id)
18 );
19 • INSERT INTO students (student_id, student_name, class_id) VALUES
20     (1, 'John Doe', 1),
21     (2, 'Jane Smith', 2),
22     (3, 'Sam Brown', 3),
23     (4, 'Emily Johnson', 1);
-- 
24
25 • INSERT INTO students (student_id, student_name, class_id) VALUES
26     (1, 'John Doe', 1),
27     (2, 'Jane Smith', 2),
28     (3, 'Sam Brown', 3),
29     (4, 'Emily Johnson', 1);
30
31
32

```

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\* x

3

4 • CREATE TABLE Customers (

5     customer\_id INT PRIMARY KEY,

6     customer\_name VARCHAR(50)

7 );

8 • INSERT INTO Customers (customer\_id, customer\_name) VALUES

9     (101, 'Alice'),

10    (102, 'Bob');

11

12 • CREATE TABLE Orders (

13     order\_id INT PRIMARY KEY,

14     order\_date DATE,

15     customer\_id INT,

16     FOREIGN KEY (customer\_id) REFERENCES Customers(customer\_id)

17 );

18 • INSERT INTO Orders (order\_id, order\_date, customer\_id) VALUES

19     (1, '2024-01-01', 101),

20    (2, '2024-01-03', 102);

21

22 • CREATE TABLE Products (

23     product\_id INT PRIMARY KEY,

SQL File 5\* x

15     customer\_id INT,

16     FOREIGN KEY (customer\_id) REFERENCES Customers(customer\_id)

17 );

18 • INSERT INTO Orders (order\_id, order\_date, customer\_id) VALUES

19     (1, '2024-01-01', 101),

20    (2, '2024-01-03', 102);

21

22 • CREATE TABLE Products (

23     product\_id INT PRIMARY KEY,

24     product\_name VARCHAR(50),

25     order\_id INT,

26     FOREIGN KEY (order\_id) REFERENCES Orders(order\_id)

27 );

28 • INSERT INTO Products (product\_id, product\_name, order\_id) VALUES

29     (1, 'Laptop', 1),

30    (2, 'Phone', NULL);

31

32 |

33

SQL File 5\*

```

29     (1, 'Laptop', 1),
30     (2, 'Phone', NULL);
31
32 • SELECT
33     o.order_id,
34     c.customer_name,
35     p.product_name
36   FROM
37     Products p
38   LEFT JOIN Orders o ON p.order_id = o.order_id
39   LEFT JOIN Customers c ON o.customer_id = c.customer_id;
40

```

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

order_id	customer_name	product_name
1	Alice	Laptop
NULL	NULL	Phone

## 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.

```
1 • CREATE TABLE Products (
2     product_id INT PRIMARY KEY,
3     product_name VARCHAR(50)
4 );
5 • INSERT INTO Products (product_id, product_name) VALUES
6     (101, 'Laptop'),
7     (102, 'Phone');
8
9 • CREATE TABLE Sales (
10    sale_id INT PRIMARY KEY,
11    product_id INT,
12    amount INT,
13    FOREIGN KEY (product_id) REFERENCES Products(product_id)
14 );
15 • INSERT INTO Sales (sale_id, product_id, amount) VALUES
16     (1, 101, 500),
17     (2, 102, 300),
18     (3, 101, 700);
19
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
```

```
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

```
1 • CREATE TABLE Customers (
2     customer_id INT PRIMARY KEY,
3     customer_name VARCHAR(50)
4 );
5 • INSERT INTO Customers (customer_id, customer_name) VALUES
6     (1, 'Alice'),
7     (2, 'Bob');
8
9 • CREATE TABLE Orders (
10    order_id INT PRIMARY KEY,
11    order_date DATE,
12    customer_id INT,
13    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
14 );
15 • INSERT INTO Orders (order_id, order_date, customer_id) VALUES
16     (1, '2024-01-02', 1),
17     (2, '2024-01-05', 2);
18
19 • CREATE TABLE Order_Details (
20    order_id INT,
21    product_id INT,
22    quantity INT,
23    FOREIGN KEY (order_id) REFERENCES Orders(order_id)
24 );
25 • INSERT INTO Order_Details (order_id, product_id, quantity) VALUES
26     (1, 101, 2),
27     (1, 102, 1),
28     (2, 101, 3);
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;
```

```
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.

The screenshot shows the MySQL Workbench interface with a query editor and a result grid. The query is:

```
1 • USE mavenmovies;
2 • SELECT * FROM actor;
```

The result grid displays 191 rows of actor data:

actor_id	first_name	last_name	last_update
177	GENE	MCKELLEN	2006-02-15 04:34:33
178	LISA	MONROE	2006-02-15 04:34:33
179	ED	GUINNESS	2006-02-15 04:34:33
180	JEFF	SILVERSTONE	2006-02-15 04:34:33
181	MATTHEW	CARREY	2006-02-15 04:34:33
182	DEBBIE	AKROYD	2006-02-15 04:34:33
183	RUSSELL	CLOSE	2006-02-15 04:34:33
184	HUMPHREY	GARLAND	2006-02-15 04:34:33
185	MICHAEL	BOLGER	2006-02-15 04:34:33
186	JULIA	ZELLWEGER	2006-02-15 04:34:33
187	RENEE	BALL	2006-02-15 04:34:33
188	ROCK	DUKAKIS	2006-02-15 04:34:33
189	CUBA	BIRCH	2006-02-15 04:34:33
190	AUDREY	BAILEY	2006-02-15 04:34:33
191	GREGORY	GOODING	2006-02-15 04:34:33

## 3. List all customer information from DB.

The screenshot shows the MySQL Workbench interface with a query editor and a result grid. The query is:

```
1 • USE mavenmovies;
2 • SELECT * FROM customer;
```

The result grid displays 194 rows of customer data:

customer_id	store_id	first_name	last_name	email	address_id	active	create_date	last_update
180	2	STACY	CUNNINGHAM	STACY.CUNNINGHAM@sakilacustomer.org	184	1	2006-02-14 22:04:36	2006-02-15 04:57:20
181	2	ANA	BRADLEY	ANA.BRADLEY@sakilacustomer.org	185	1	2006-02-14 22:04:36	2006-02-15 04:57:20
182	1	RENEE	LANE	RENEE.LANE@sakilacustomer.org	186	1	2006-02-14 22:04:36	2006-02-15 04:57:20
183	2	IDA	ANDREWS	IDA.ANDREWS@sakilacustomer.org	187	1	2006-02-14 22:04:36	2006-02-15 04:57:20
184	1	VIVIAN	RUIZ	VIVIAN.RUIZ@sakilacustomer.org	188	1	2006-02-14 22:04:36	2006-02-15 04:57:20
185	1	ROBERTA	HARPER	ROBERTA.HARPER@sakilacustomer.org	189	1	2006-02-14 22:04:36	2006-02-15 04:57:20
186	2	HOLLY	FOX	HOLLY.FOX@sakilacustomer.org	190	1	2006-02-14 22:04:36	2006-02-15 04:57:20
187	2	BRITTANY	RILEY	BRITTANY.RILEY@sakilacustomer.org	191	1	2006-02-14 22:04:36	2006-02-15 04:57:20
188	1	MELANIE	ARMSTRONG	MELANIE.ARMSTRONG@sakilacustomer.org	192	1	2006-02-14 22:04:36	2006-02-15 04:57:20
189	1	LORETTA	CARPENTER	LORETTA.CARPENTER@sakilacustomer.org	193	1	2006-02-14 22:04:36	2006-02-15 04:57:20
190	2	YOLANDA	WEAVER	YOLANDA.WEAVER@sakilacustomer.org	194	1	2006-02-14 22:04:36	2006-02-15 04:57:20
191	1	JEANETTE	GREENE	JEANETTE.GREENE@sakilacustomer.org	195	1	2006-02-14 22:04:36	2006-02-15 04:57:20
192	1	Laurie	LAWRENCE	Laurie.LAWRENCE@sakilacustomer.org	196	1	2006-02-14 22:04:36	2006-02-15 04:57:20
193	2	KATIE	ELLIOTT	KATIE.ELLIOTT@sakilacustomer.org	197	1	2006-02-14 22:04:36	2006-02-15 04:57:20
194	2	KRISTEN	CHAVEZ	KRISTEN.CHAVEZ@sakilacustomer.org	198	1	2006-02-14 22:04:36	2006-02-15 04:57:20

## 4. List different countries.

Mavenmovies\*

```

1 • USE mavenmovies;
2 • SELECT * FROM customer;
3
4
5

```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content: | SQLAdditions | < | > | Jump to |

Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.

customer_id	store_id	first_name	last_name	email	address_id	active	create_date	last_update
384	2	ERNEST	STEPP	ERNEST.STEPP@sakila.customer.org	389	1	2006-02-14 22:04:37	2006-02-15 04:57:20
385	1	PHILIP	HOLM	PHILIP.HOLM@sakila.customer.org	390	1	2006-02-14 22:04:37	2006-02-15 04:57:20
386	1	TOC	PHILLIP	TODD.TAN@sakila.customer.org	391	1	2006-02-14 22:04:37	2006-02-15 04:57:20
387	2	JESSE	SCHILLING	JESSE.SCHILLING@sakila.customer.org	392	1	2006-02-14 22:04:37	2006-02-15 04:57:20
388	2	CRAIG	MORRELL	CRAIG.MORRELL@sakila.customer.org	393	1	2006-02-14 22:04:37	2006-02-15 04:57:20
389	1	ALAN	KAHN	ALAN.KAHN@sakila.customer.org	394	1	2006-02-14 22:04:37	2006-02-15 04:57:20
390	1	SHAWN	HEATON	SHAWN.HEATON@sakila.customer.org	395	1	2006-02-14 22:04:37	2006-02-15 04:57:20
391	1	CLARENCE	GAMEZ	CLARENCE.GAMEZ@sakila.customer.org	396	1	2006-02-14 22:04:37	2006-02-15 04:57:20
392	2	SEAN	DOUGLASS	SEAN.DOUGLAS@sakila.customer.org	397	1	2006-02-14 22:04:37	2006-02-15 04:57:20
393	1	PHILIP	CAUSEY	PHILIP.CAUSEY@sakila.customer.org	398	1	2006-02-14 22:04:37	2006-02-15 04:57:20
394	2	OHRIS	BROTHERS	OHRIS.BROTHERS@sakila.customer.org	399	1	2006-02-14 22:04:37	2006-02-15 04:57:20
395	2	JOHNNY	TURPIN	JOHNNY.TURPIN@sakila.customer.org	400	1	2006-02-14 22:04:37	2006-02-15 04:57:20
396	1	EARL	SHANKS	EARL.SHANKS@sakila.customer.org	401	1	2006-02-14 22:04:37	2006-02-15 04:57:20
397	1	JIMMY	SCHRADER	JIMMY.SCHRADER@sakila.customer.org	402	1	2006-02-14 22:04:37	2006-02-15 04:57:20
398	1	ANTONIO	MEEK	ANTONIO.MEEK@sakila.customer.org	403	1	2006-02-14 22:04:37	2006-02-15 04:57:20

customer 5 ×

Output:

Action Output

#	Time	Action	Message	Duration / Fetch
120	12:22:01	USE mavenmovies	0 row(s) affected	0.000 sec
121	12:22:01	SELECT * FROM actor LIMIT 0, 1000	200 row(s) returned	0.000 sec / 0.000 sec
122	12:27:31	USE mavenmovies	0 row(s) affected	0.000 sec
123	12:27:31	SELECT * FROM customer LIMIT 0, 1000	599 row(s) returned	0.016 sec / 0.000 sec
124	12:28:50	USE mavenmovies	0 row(s) affected	0.015 sec
125	12:28:50	SELECT * FROM customer LIMIT 0, 1000	599 row(s) returned	0.000 sec / 0.000 sec

Result Grid | Form Editor | Field Types | Query Stats | Context Help | Snippets

## 5. Display all active customers.

Mavenmovies\*

```

1 • USE mavenmovies;
2 • SELECT * FROM customer;
3
4 • SELECT *
5   FROM customer
6   WHERE active = 1;
7
8
9 • SET @OLD_UNIQUE_CHECKS=@UNIQUE_CHECKS, UNIQUE_CHECKS=0;

```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content: | SQLAdditions | < | > | Jump to |

Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.

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

customer 2 ×

Output:

Action Output

#	Time	Action	Message	Duration / Fetch
3	10:21:29	SELECT * FROM customer LIMIT 0, 1000	599 row(s) returned	0.016 sec / 0.000 sec
4	10:21:29	SET @OLD_UNIQUE_CHECKS=@UNIQUE_CHECKS, UNIQUE_CHECKS=0	0 row(s) affected	0.000 sec
5	10:21:29	SET @OLD_FOREIGN_KEY_CHECKS=@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0	0 row(s) affected	0.000 sec
6	10:21:29	CREATE SCHEMA mavenmovies	Error Code: 1007. Can't create database 'mavenmovies'; database exists	0.000 sec
7	10:21:49	SELECT * FROM customer WHERE active = 1 LIMIT 0, 1000	584 row(s) returned	0.000 sec / 0.000 sec

Result Grid | Form Editor | Field Types | Query Stats | Context Help | Snippets

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

The screenshot shows the MySQL Workbench interface with the following details:

- SQL Editor:** Contains the following SQL code:

```
3
4 • SELECT *
5   FROM customer
6   WHERE active = 1;
7
8 • SELECT rental_id
9   FROM rental
10  WHERE customer_id = 1;
```
- Result Grid:** Shows the results of the second query, listing rental IDs:

rental_id
76
573
1185
1422
1476
1725
2308
2363
3284
4526
4611
- Action Output:** Displays the execution log:

#	Time	Action	Message	Duration / Fetch
4	10:21:29	SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0	0 row(s) affected	0.000 sec
5	10:21:29	SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0	0 row(s) affected	0.000 sec
6	10:21:29	CREATE SCHEMA mavenmovies	Error Code: 1007. Can't create database 'mavenmovies'; database exists	0.000 sec
7	10:21:49	SELECT * FROM customer WHERE active = 1 LIMIT 0, 1000	584 row(s) returned	0.000 sec / 0.000 sec
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

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

The screenshot shows the MySQL Workbench interface with the following details:

- SQL Editor:** Contains the following SQL code:

```
4 • SELECT *
5   FROM film
6   WHERE rental_duration > 5;
7
8
9 • SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
10 • SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
11 -- SET @OLD_SQL_MODE=@$SQL_MODE, SQL_MODE='TRADITIONAL';
12
```
- Result Grid:** Shows the results of the query, listing films with rental duration greater than 5:

film_id	title	description	release_year	language_id	original_language_id	rental_duration	rental_rate	length	replacement_cost
1	ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist ...	2006	1	NULL	6	0.99	86	20.99
3	ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a ...	2006	1	NULL	7	2.99	50	18.99
5	AFRICAN EGG	A Fast-Paced Documentary of a Pastry Chef An...	2006	1	NULL	6	2.99	130	22.99
7	AIRPLANE SIERRA	A Touching Saga of a Hunter And a Butler who ...	2006	1	NULL	6	4.99	62	28.99
8	AIRPORT POLLACK	A Epic Tale of a Moose And a Girl who must Con...	2006	1	NULL	6	4.99	54	15.99
10	ALADDIN CALENDAR	A Action-Packed Tale of a Man And a Lumberjac...	2006	1	NULL	6	4.99	63	24.99
11	ALAMO VIDEOTAP	A Boring Epistle of a Butler And a Cat who must ...	2006	1	NULL	6	0.99	126	16.99
12	ALASKA PHANTOM	A Fanciful Saga of a Hunter And a Pastry Chef ...	2006	1	NULL	6	0.99	136	22.99
14	ALICE FANTASIA	A Emotional Drama of a Shark And a Database...	2006	1	NULL	6	0.99	94	23.99
16	ALLEY EVOLUTION	A Fast-Paced Drama of a Robot And a Composer...	2006	1	NULL	6	2.99	180	23.99
- Action Output:** Displays the execution log:

#	Time	Action	Message	Duration / Fetch
5	10:21:29	SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0	0 row(s) affected	0.000 sec
6	10:21:29	CREATE SCHEMA mavenmovies	Error Code: 1007. Can't create database 'mavenmovies'; database exists	0.000 sec
7	10:21:49	SELECT * FROM customer WHERE active = 1 LIMIT 0, 1000	584 row(s) returned	0.000 sec / 0.000 sec
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

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

The screenshot shows the MySQL Workbench interface. In the top-left pane, the SQL editor window titled "Mavenmovies\*" contains the following SQL code:

```
4 •  SELECT COUNT(*) AS total_films
5   FROM film
6  WHERE replacement_cost > 15 AND replacement_cost < 20;
7
8 •  SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
9 •  SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
10 -- SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';
11
12 -- DROP SCHEMA IF EXISTS mavenmovies; -- commenting out for Maven Course to avoid concerning warning message
```

In the bottom-left pane, the "Result Grid" shows the output of the query:

total_films
214

The bottom-right pane displays the "Action Output" log:

#	Time	Action	Message	Duration / Fetch
6	10:21:29	CREATE SCHEMA mavenmovies	Error Code: 1007. Can't create database 'mavenmovies'; database exists	0.000 sec
7	10:21:49	SELECT * FROM customer WHERE active = 1 LIMIT 0, 1000	584 row(s) returned	0.000 sec / 0.000 sec
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

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

The screenshot shows the MySQL Workbench interface. In the top-left pane, the SQL editor window titled "Mavenmovies\*" contains the following SQL code:

```
4 •  SELECT COUNT(DISTINCT first_name) AS unique_first_names
5   FROM actor;
6
7 •  SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
8 •  SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
9 -- SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';
10
11 -- DROP SCHEMA IF EXISTS mavenmovies; -- commenting out for Maven Course to avoid concerning warning message
12 •  CREATE SCHEMA mavenmovies;
```

In the bottom-left pane, the "Result Grid" shows the output of the query:

unique_first_names
128

The bottom-right pane displays the "Action Output" log:

#	Time	Action	Message	Duration / Fetch
7	10:21:49	SELECT * FROM customer WHERE active = 1 LIMIT 0, 1000	584 row(s) returned	0.000 sec / 0.000 sec
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

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

The screenshot shows the MySQL Workbench interface with a query editor and results grid. The query is:

```
4 • SELECT *
  5   FROM customer
  6   LIMIT 10;
  7
  8
  9 • SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
10 • SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
11 -- SET @OLD_SQL_MODE=@SQL_MODE, SQL_MODE='TRADITIONAL';
12
```

The results grid displays 10 rows of customer data:

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

The output pane shows the execution log:

#	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_fims 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

## 11. Display the first 3 records from the customer table whose first name starts with 'b'.

The screenshot shows the MySQL Workbench interface with a query editor and results grid. The query is:

```
1 • USE mavenmovies;
2 • SELECT * FROM customer;
3
4 • SELECT *
  5   FROM customer
  6   WHERE first_name LIKE 'B%';
  7   LIMIT 3;
  8
  9 • SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
```

The results grid displays 3 rows of customer data:

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

The output pane shows the execution log:

#	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_fims 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'.

The screenshot shows the MySQL Workbench interface with a query editor and results grid. The query is:

```
4 • SELECT *  
5   FROM customer  
6  WHERE first_name LIKE 'A%'  
7  
8 • SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;  
9 • SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;  
10 -- SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';  
11  
12 -- DROP SCHEMA IF EXISTS mavenmovies; -- commenting out for Maven Course to avoid concerning warning message
```

The results grid shows customer data, and the action output shows the execution of the query.

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

The screenshot shows the MySQL Workbench interface with a query editor and results grid. The query is:

```
1 • SELECT * FROM customer  
2 WHERE first_name LIKE 'A%'  
3  
4  
5 • SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;  
6 • SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;  
7 -- SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';  
8
```

The results grid shows customer data, and the action output shows the execution of the query. It includes an error message for the CREATE TABLE statement due to the table already existing.

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

The screenshot shows the MySQL Workbench interface with the following details:

- SQL Editor:** Contains the following SQL code:

```
1 •  SELECT * FROM customer
2 WHERE first_name LIKE '%a';
3
4
5 •  SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
6 •  SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
7 -- SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';
8
```
- Result Grid:** Displays a table of customer data, filtered by first\_name like '%a'. The table includes columns: customer\_id, store\_id, first\_name, last\_name, email, address\_id, active, create\_date, and last\_update. The results show 29 rows.
- Action Output:** Shows the execution log with the following entries:

#	Time	Action	Message	Duration / Fetch
144	10:45:01	CREATE TABLE customers ( id INT AUTO_INCREMENT PRIMARY KEY, first_name VARCHAR(100), ... )	0 row(s) affected	0.015 sec
145	10:45:05	SELECT * FROM customers WHERE first_name LIKE '%a' LIMIT 0, 1000	0 row(s) returned	0.016 sec / 0.000 sec
146	10:45:35	SELECT COUNT(*) FROM customers LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec
147	10:48:17	SELECT * FROM customer WHERE first_name LIKE 'A%' LIMIT 0, 1000	44 row(s) returned	0.000 sec / 0.000 sec
148	10:49:27	SELECT * FROM customer WHERE first_name LIKE '%a' LIMIT 0, 1000	96 row(s) returned	0.016 sec / 0.000 sec

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

The screenshot shows the MySQL Workbench interface with the following details:

- SQL Editor:** Contains the following SQL code:

```
1 •  SELECT * FROM city
2 WHERE city LIKE 'a%a'
3 LIMIT 4;
4
5 •  SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
6 •  SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
7 -- SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';
8
```
- Result Grid:** Displays a table of city data, filtered by city like 'a%a'. The table includes columns: city\_id, city, country\_id, and last\_update. The results show 4 rows.
- Action Output:** Shows the execution log with the following entries:

#	Time	Action	Message	Duration / Fetch
145	10:45:05	SELECT * FROM customers WHERE first_name LIKE '%a' LIMIT 0, 1000	0 row(s) returned	0.016 sec / 0.000 sec
146	10:45:35	SELECT COUNT(*) FROM customers LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec
147	10:48:17	SELECT * FROM customer WHERE first_name LIKE 'A%' LIMIT 0, 1000	44 row(s) returned	0.000 sec / 0.000 sec
148	10:49:27	SELECT * FROM customer WHERE first_name LIKE '%a' LIMIT 0, 1000	96 row(s) returned	0.016 sec / 0.000 sec
149	10:50:30	SELECT * FROM city WHERE city LIKE 'a%a' LIMIT 4	4 row(s) returned	0.000 sec / 0.000 sec

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

The screenshot shows the MySQL Workbench interface with the following details:

- SQL Editor:** Contains the following SQL code:

```
1 •  SELECT * FROM customer
2 WHERE LOWER(first_name) LIKE '%ni%';
3
4 •  SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
5 •  SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
6 -- SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';
7
8 -- DROP SCHEMA IF EXISTS mavenmovies; -- commenting out for Maven Course to avoid concerning warning message
```
- Result Grid:** Displays a table of customer data. The columns are: customer\_id, store\_id, first\_name, last\_name, email, address\_id, active, create\_date, and last\_update. The data includes rows for Jennifer Davis, Virginia Green, Stephanie Mitchell, Janice Ward, Nicole Peterson, Denise Kelly, Bonnie Hughes, Annie Russell, Connie Wallace, Monica Hicks, and Juanita Mason.
- Action Output:** Shows the execution history of the session, listing each query with its time, action, message, and duration/fetch time.

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

The screenshot shows the MySQL Workbench interface with the following details:

- SQL Editor:** Contains the following SQL code:

```
1 •  SELECT * FROM customer
2 WHERE first_name LIKE '_r%';
3
4 •  SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
5 •  SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
6 -- SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';
7
8 -- DROP SCHEMA IF EXISTS mavenmovies; -- commenting out for Maven Course to avoid concerning warning message
```
- Result Grid:** Displays a table of customer data. The columns are: customer\_id, store\_id, first\_name, last\_name, email, address\_id, active, create\_date, and last\_update. The data includes rows for Brenda Wright, Frances Parker, Irene Price, Crystal Ford, Tracy Cole, Grace Ellis, Erin Dunn, Erica Matthews, Brittany Riley, Kristen Chavez, and Kristin Johnston.
- Action Output:** Shows the execution history of the session, listing each query with its time, action, message, and duration/fetch time.

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

The screenshot shows the MySQL Workbench interface with the following details:

- SQL Editor:** Contains the following SQL code:
 

```

1 •  SELECT * FROM customer
2 WHERE first_name LIKE 'a%' AND LENGTH(first_name) >= 5;
3
4 •  SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
5 •  SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
6 -- SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';
7
8 -- DROP SCHEMA IF EXISTS mavenmovies; -- commenting out for Maven Course to avoid concerning warning message
      
```
- Result Grid:** Displays the results of the query, showing 19 rows of customer data. The columns include customer\_id, store\_id, first\_name, last\_name, email, address\_id, active, create\_date, and last\_update. Some rows are highlighted in blue.
- Action Output:** Shows the history of actions taken during the session, including the execution of the query and other database operations.

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

The screenshot shows the MySQL Workbench interface with the following details:

- SQL Editor:** Contains the following SQL code:
 

```

1 •  SELECT * FROM customer
2 WHERE first_name LIKE 'a%o';
3
4 •  SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
5 •  SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
6 -- SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';
7
8 -- DROP SCHEMA IF EXISTS mavenmovies; -- commenting out for Maven Course to avoid concerning warning message
      
```
- Result Grid:** Displays the results of the query, showing 4 rows of customer data. The columns include customer\_id, store\_id, first\_name, last\_name, email, address\_id, active, create\_date, and last\_update. Some rows are highlighted in blue.
- Action Output:** Shows the history of actions taken during the session, including the execution of the query and other database operations.

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

The screenshot shows the MySQL Workbench interface with the following details:

- SQL Editor:** Contains the following SQL code:

```
1 •  SELECT * FROM film
2 WHERE rating IN ('PG', 'PG-13');

3
4 •  SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
5 •  SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
6 -- SET @OLD_SQL_MODE=@SQL_MODE, SQL_MODE='TRADITIONAL';
7
8 -- DROP SCHEMA IF EXISTS mavenmovies; -- commenting out for Maven Course to avoid concerning warning message
```
- Result Grid:** Displays a table of film records. The results show 33 rows of data, including titles like 'ACADEMY DINOSAUR', 'AGENT TRUMAN', and 'AIRPLANE SIERRA'.
- Action Output:** Shows the execution log with 155 entries, detailing each query execution and its duration.

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

The screenshot shows the MySQL Workbench interface with the following details:

- SQL Editor:** Contains the following SQL code:

```
1 •  SELECT * FROM film
2 WHERE length BETWEEN 50 AND 100;

3
4 •  SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
5 •  SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
6 -- SET @OLD_SQL_MODE=@SQL_MODE, SQL_MODE='TRADITIONAL';
7
8 -- DROP SCHEMA IF EXISTS mavenmovies; -- commenting out for Maven Course to avoid concerning warning message
```
- Result Grid:** Displays a table of film records. The results show 22 rows of data, including titles like 'ADAPTATION HOLES', 'AIRPORT POLLOCK', and 'ALADDIN CALENDAR'.
- Action Output:** Shows the execution log with 156 entries, detailing each query execution and its duration.

## 22. Get the top 50 actors using limit operator.

The screenshot shows the MySQL Workbench interface with a query editor titled "Mavenmovies". The query is:

```
1 • SELECT
2     a.actor_id,
3     CONCAT(a.first_name, ' ', a.last_name) AS actor_name,
4     COUNT(fa.film_id) AS film_count
5   FROM actor a
6  JOIN film_actor fa ON a.actor_id = fa.actor_id
7  GROUP BY a.actor_id, a.first_name, a.last_name
8 ORDER BY film_count DESC
9 LIMIT 50;
```

The results grid shows the top 50 actors with their names and the number of films they have appeared in:

actor_id	actor_name	film_count
107	GINA DEGENERES	42
102	WALTER TORN	41
198	MARY KETTEL	40
181	MATTHEW CARREY	39
23	SANDRA KILMER	37
81	SCARLETT DAMON	36
158	VIVIAN BASINGER	35
60	HENRY BERRY	35
144	ANGELA WITHERSPOON	35
37	VAL BOLGER	35
13	UMA WOOD	35

The output pane shows the following log entries:

#	Time	Action	Message	Duration / Fetch
4	12:05:03	SELECT * FROM actors LIMIT 50	Error Code: 1046. No database selected. Select the default DB to be used by double-clicking its name in the SC...	0.000 sec
5	12:05:30	USE mavenmovies	0 row(s) affected	0.000 sec
6	12:05:37	CREATE SCHEMA mavenmovies	Error Code: 1007. Can't create database 'mavenmovies'; database exists	0.000 sec
7	12:05:41	SELECT * FROM actors LIMIT 50	Error Code: 1146. Table 'mavenmovies.actors' doesn't exist	0.000 sec
8	12:07:07	SELECT a.actor_id, CONCAT(a.first_name, ' ', a.last_name) AS actor_name, COUNT(fa.film_id) AS fil...	50 row(s) returned	0.046 sec / 0.000 sec
9	12:07:57	SELECT DISTINCT film_id FROM inventory LIMIT 0, 1000	958 row(s) returned	0.016 sec / 0.000 sec

## 23. Get the distinct film ids from inventory table.

The screenshot shows the MySQL Workbench interface with a query editor titled "Mavenmovies". The query is:

```
1 • SELECT DISTINCT film_id
2   FROM inventory;
3
4
5 • SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
6 • SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
7 -- SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';
8
9 -- DROP SCHEMA IF EXISTS mavenmovies; -- commenting out for Maven Course to avoid concerning warning message
```

The results grid shows the distinct film IDs:

film_id
1
2
3
4
5
6
7
8
9
10
11

The output pane shows the following log entries:

#	Time	Action	Message	Duration / Fetch
5	12:05:30	USE mavenmovies	0 row(s) affected	0.000 sec
6	12:05:37	CREATE SCHEMA mavenmovies	Error Code: 1007. Can't create database 'mavenmovies'; database exists	0.000 sec
7	12:05:41	SELECT * FROM actors LIMIT 50	Error Code: 1146. Table 'mavenmovies.actors' doesn't exist	0.000 sec
8	12:07:07	SELECT a.actor_id, CONCAT(a.first_name, ' ', a.last_name) AS actor_name, COUNT(fa.film_id) AS fil...	50 row(s) returned	0.046 sec / 0.000 sec
9	12:07:57	SELECT DISTINCT film_id FROM inventory LIMIT 0, 1000	958 row(s) returned	0.016 sec / 0.000 sec

## ❖ Functions

### Basic Aggregate Functions:

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

The screenshot shows the MySQL Workbench interface with a query editor and a results grid. The query is:1 • SELECT COUNT(\*) AS total\_rentals  
2 FROM rental;  
3  
4  
5 • SET @OLD\_UNIQUE\_CHECKS=@@UNIQUE\_CHECKS, UNIQUE\_CHECKS=0;  
6 • SET @OLD\_FOREIGN\_KEY\_CHECKS=@@FOREIGN\_KEY\_CHECKS, FOREIGN\_KEY\_CHECKS=0;  
7 -- SET @OLD\_SQL\_MODE=@@SQL\_MODE, SQL\_MODE='TRADITIONAL';  
8  
9 -- DROP SCHEMA IF EXISTS mavenmovies; -- commenting out for Maven Course to avoid concerning warning message

The results grid shows:

total_rentals
16044

The execution history (Result 3) shows the following steps:

#	Time	Action	Message	Duration / Fetch
6	12:05:37	CREATE SCHEMA mavenmovies	Error Code: 1007. Can't create database 'mavenmovies'; database exists	0.000 sec
7	12:05:41	SELECT * FROM actors LIMIT 50	Error Code: 1146. Table 'mavenmovies.actors' doesn't exist	0.000 sec
8	12:07:07	SELECT a.actor_id, CONCAT(a.first_name, ' ', a.last_name) AS actor_name, COUNT(f.film_id) AS film_count FROM actors a JOIN film_actor fa ON a.actor_id = fa.actor_id JOIN film f ON fa.film_id = f.film_id GROUP BY a.actor_id ORDER BY film_count DESC LIMIT 50	50 row(s) returned	0.046 sec / 0.000 sec
9	12:07:57	SELECT DISTINCT film_id FROM inventory LIMIT 0, 1000	958 row(s) returned	0.016 sec / 0.000 sec
10	12:10:34	SELECT COUNT(*) AS total_rentals FROM rental LIMIT 0, 1000	1 row(s) returned	0.032 sec / 0.000 sec

**Question 2: Find the average rental duration (in days) of movies rented from the Sakila database.**

**Hint: Utilize the AVG() function.**

The screenshot shows the MySQL Workbench interface with a query editor and a results grid. The query is:1 • SELECT AVG(DATEDIFF(return\_date, rental\_date)) AS avg\_rental\_duration\_days  
2 FROM rental;  
3  
4  
5 • SET @OLD\_UNIQUE\_CHECKS=@@UNIQUE\_CHECKS, UNIQUE\_CHECKS=0;  
6 • SET @OLD\_FOREIGN\_KEY\_CHECKS=@@FOREIGN\_KEY\_CHECKS, FOREIGN\_KEY\_CHECKS=0;  
7 -- SET @OLD\_SQL\_MODE=@@SQL\_MODE, SQL\_MODE='TRADITIONAL';  
8  
9 -- DROP SCHEMA IF EXISTS mavenmovies; -- commenting out for Maven Course to avoid concerning warning message

The results grid shows:

avg_rental_duration_days
5.025

The execution history (Result 4) shows the following steps:

#	Time	Action	Message	Duration / Fetch
7	12:05:41	SELECT * FROM actors LIMIT 50	Error Code: 1146. Table 'mavenmovies.actors' doesn't exist	0.000 sec
8	12:07:07	SELECT a.actor_id, CONCAT(a.first_name, ' ', a.last_name) AS actor_name, COUNT(f.film_id) AS film_count FROM actors a JOIN film_actor fa ON a.actor_id = fa.actor_id JOIN film f ON fa.film_id = f.film_id GROUP BY a.actor_id ORDER BY film_count DESC LIMIT 50	50 row(s) returned	0.046 sec / 0.000 sec
9	12:07:57	SELECT DISTINCT film_id FROM inventory LIMIT 0, 1000	958 row(s) returned	0.016 sec / 0.000 sec
10	12:10:34	SELECT COUNT(*) AS total_rentals FROM rental LIMIT 0, 1000	1 row(s) returned	0.032 sec / 0.000 sec
11	12:11:13	SELECT AVG(DATEDIFF(return_date, rental_date)) AS avg_rental_duration_days FROM rental LIMIT 0, 1000	1 row(s) returned	0.047 sec / 0.000 sec

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

**Hint:** Use the **UPPER ()** function.

The screenshot shows the MySQL Workbench interface with a query editor and results grid. The query uses the **UPPER()** function to convert first and last names to uppercase. The results grid displays 5 rows of data with columns **first\_name\_upper** and **last\_name\_upper**. The output pane shows the execution log with 12 entries, all completed successfully.

```
1 • SELECT
2     UPPER(first_name) AS first_name_upper,
3     UPPER(last_name) AS last_name_upper
4   FROM customer;
5
6 • SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
7 • SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
8 -- SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';
9
```

first_name_upper	last_name_upper
MARY	SMITH
PATRICIA	JOHNSON
LINDA	WILLIAMS
BARBARA	JONES
ELIZABETH	BROWN
JENNIFER	DAVIS
MARIA	MILLER
SUSAN	WILSON
MARGARET	MOORE
DOROTHY	TAYLOR
LISA	ANDERSON

Result 5 x

Action Output

#	Time	Action	Message	Duration / Fetch
8	12:07:07	SELECT a.actor_id, CONCAT(a.first_name, ' ', a.last_name) AS actor_name, COUNT(film_id) AS fil...	50 row(s) returned	0.046 sec / 0.000 sec
9	12:07:57	SELECT DISTINCT film_id FROM inventory LIMIT 0, 1000	958 row(s) returned	0.016 sec / 0.000 sec
10	12:10:34	SELECT COUNT(*) AS total_rentals FROM rental LIMIT 0, 1000	1 row(s) returned	0.032 sec / 0.000 sec
11	12:11:13	SELECT AVG(DATEDIFF(return_date, rental_date)) AS avg_rental_duration_days FROM rental LIMIT 0, 1000	1 row(s) returned	0.047 sec / 0.000 sec
12	12:12:07	SELECT UPPER(first_name) AS first_name_upper, UPPER(last_name) AS last_name_upper FROM cust...	599 row(s) returned	0.000 sec / 0.000 sec

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

**Hint:** Employ the **MONTH()** function.

The screenshot shows the MySQL Workbench interface with a query editor and results grid. The query uses the **MONTH()** function to extract the month from the rental date. The results grid displays 11 rows of data with columns **rental\_id** and **rental\_month**. The output pane shows the execution log with 13 entries, all completed successfully.

```
1 • SELECT
2     Execute the statement under the keyboard cursor
3     MONTH(rental_date) AS rental_month
4   FROM rental;
5
6 • SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
7 • SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
8 -- SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';
9
```

rental_id	rental_month
1	5
2	5
3	5
4	5
5	5
6	5
7	5
8	5
9	5
10	5
11	5

Result 6 x

Action Output

#	Time	Action	Message	Duration / Fetch
9	12:07:57	SELECT DISTINCT film_id FROM inventory LIMIT 0, 1000	958 row(s) returned	0.016 sec / 0.000 sec
10	12:10:34	SELECT COUNT(*) AS total_rentals FROM rental LIMIT 0, 1000	1 row(s) returned	0.032 sec / 0.000 sec
11	12:11:13	SELECT AVG(DATEDIFF(return_date, rental_date)) AS avg_rental_duration_days FROM rental LIMIT 0, 1000	1 row(s) returned	0.047 sec / 0.000 sec
12	12:12:07	SELECT UPPER(first_name) AS first_name_upper, UPPER(last_name) AS last_name_upper FROM cust...	599 row(s) returned	0.000 sec / 0.000 sec
13	12:14:01	SELECT rental_id, MONTH(rental_date) AS rental_month FROM rental LIMIT 0, 1000	1000 row(s) returned	0.000 sec / 0.000 sec

**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.**

```

Mavenmovies* x
1 • SELECT
   COUNT(*) AS rental_count
FROM rental
GROUP BY customer_id;
2 • SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
3 • SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;

```

The screenshot shows the MySQL Workbench interface with a query editor window titled "Mavenmovies\*". The query is:

```

1 • SELECT
   COUNT(*) AS rental_count
FROM rental
GROUP BY customer_id;
2 • SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
3 • SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;

```

The results grid displays the following data:

customer_id	rental_count
1	32
2	27
3	26
4	22
5	38
6	28
7	33
8	24
9	23
10	25
11	24

The output pane shows the execution log:

#	Time	Action	Message	Duration / Fetch
10	12:10:34	SELECT COUNT(*) AS total_rentals FROM rental LIMIT 0, 1000	1 row(s) returned	0.032 sec / 0.000 sec
11	12:11:13	SELECT AVG(DATEDIFF(return_date, rental_date)) AS avg_rental_duration_days FROM rental LIMIT 0, 1000	1 row(s) returned	0.047 sec / 0.000 sec
12	12:12:07	SELECT UPPER(first_name) AS first_name_upper, UPPER(last_name) AS last_name_upper FROM customer	599 row(s) returned	0.000 sec / 0.000 sec
13	12:14:01	SELECT rental_id, MONTH(rental_date) AS rental_month FROM rental LIMIT 0, 1000	1000 row(s) returned	0.000 sec / 0.000 sec
14	12:15:20	SELECT customer_id, COUNT(*) AS rental_count FROM rental GROUP BY customer_id LIMIT 0, 1000	599 row(s) returned	0.031 sec / 0.000 sec

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

**Hint: Combine SUM() and GROUP BY.**

```

Mavenmovies* x
1 • SELECT
   s.store_id,
   SUM(p.amount) AS total_revenue
FROM payment p
JOIN staff s ON p.staff_id = s.staff_id
GROUP BY s.store_id;
2 • SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;

```

The screenshot shows the MySQL Workbench interface with a query editor window titled "Mavenmovies\*". The query is:

```

1 • SELECT
   s.store_id,
   SUM(p.amount) AS total_revenue
FROM payment p
JOIN staff s ON p.staff_id = s.staff_id
GROUP BY s.store_id;
2 • SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;

```

The results grid displays the following data:

store_id	total_revenue
1	33489.47
2	33927.04

The output pane shows the execution log:

#	Time	Action	Message	Duration / Fetch
11	12:11:13	SELECT AVG(DATEDIFF(return_date, rental_date)) AS avg_rental_duration_days FROM rental LIMIT 0, 1000	1 row(s) returned	0.047 sec / 0.000 sec
12	12:12:07	SELECT UPPER(first_name) AS first_name_upper, UPPER(last_name) AS last_name_upper FROM customer	599 row(s) returned	0.000 sec / 0.000 sec
13	12:14:01	SELECT rental_id, MONTH(rental_date) AS rental_month FROM rental LIMIT 0, 1000	1000 row(s) returned	0.000 sec / 0.000 sec
14	12:15:20	SELECT customer_id, COUNT(*) AS rental_count FROM rental GROUP BY customer_id LIMIT 0, 1000	599 row(s) returned	0.031 sec / 0.000 sec
15	12:16:19	SELECT s.store_id, SUM(p.amount) AS total_revenue FROM payment p JOIN staff s ON p.staff_id = s.staff_id GROUP BY s.store_id	2 row(s) returned	0.047 sec / 0.000 sec

## 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.

The screenshot shows the MySQL Workbench interface. The SQL editor contains the following query:

```
1 • SELECT
2     c.name AS category_name,
3     COUNT(r.rental_id) AS total_rentals
4 FROM
5     category c
6 JOIN
7     film_category fc ON c.category_id = fc.category_id
8 JOIN
9     film f ON fc.film_id = f.film_id
```

The results grid displays the following data:

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

The output pane shows the following log entries:

#	Time	Action	Message	Duration / Fetch
16	12:17:51	SELECT c.name AS category_name, COUNT(r.rental_id) AS total_rentals FROM film_category fc JOIN ...	Error Code: 1054. Unknown column 'f.film_id' in 'on clause'	0.000 sec
17	12:27:47	SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0	0 row(s) affected	0.000 sec
18	12:27:47	SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0	0 row(s) affected	0.000 sec
19	12:27:47	CREATE SCHEMA mavenmovies	Error Code: 1007. Can't create database 'mavenmovies'; database exists	0.000 sec
20	12:46:51	SELECT c.name AS category_name, COUNT(r.rental_id) AS total_rentals FROM category c JOIN f...	16 row(s) returned	0.015 sec / 0.000 sec

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

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

The screenshot shows the MySQL Workbench interface. The SQL editor contains the following query:

```
1 • SELECT
2     l.name AS language,
3     ROUND(AVG(f.rental_rate), 2) AS average_rental_rate
4 FROM
5     film f
6 JOIN
7     language l ON f.language_id = l.language_id
8 GROUP BY
9     l.name
```

The results grid displays the following data:

language	average_rental_rate
English	2.98

The output pane shows the following log entries:

#	Time	Action	Message	Duration / Fetch
17	12:27:47	SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0	0 row(s) affected	0.000 sec
18	12:27:47	SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0	0 row(s) affected	0.000 sec
19	12:27:47	CREATE SCHEMA mavenmovies	Error Code: 1007. Can't create database 'mavenmovies'; database exists	0.000 sec
20	12:46:51	SELECT c.name AS category_name, COUNT(r.rental_id) AS total_rentals FROM category c JOIN f...	16 row(s) returned	0.015 sec / 0.000 sec
21	12:48:17	SELECT l.name AS language, ROUND(AVG(f.rental_rate), 2) AS average_rental_rate FROM film f JOIN language l...	1 row(s) returned	0.016 sec / 0.000 sec

**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**

The screenshot shows the MySQL Workbench interface with a query editor and results pane. The query is:

```

Mavenmovies* 
1 • SELECT
2     f.title AS movie_title,
3     c.first_name,
4     c.last_name
5   FROM
6     film f
7   JOIN
8     inventory i ON f.film_id = i.film_id
9   JOIN

```

The results grid shows the following data:

movie_title	first_name	last_name
ACADEMY DINOSAUR	BEATRICE	ARNOLD
ACADEMY DINOSAUR	CARL	ARTIS
ACADEMY DINOSAUR	ROBERT	BAUGHMAN
ACADEMY DINOSAUR	HENRY	BILLINGSLEY
ACADEMY DINOSAUR	NORMAN	CURRIER
ACADEMY DINOSAUR	FREDDIE	DUGGAN
ACADEMY DINOSAUR	JOE	FRANCISCO
ACADEMY DINOSAUR	GABRIEL	HARDER
ACADEMY DINOSAUR	MATTIE	HOFFMAN
ACADEMY DINOSAUR	WILLIE	MARIKHAM
ACADEMY DINOSAUR	DEBRA	NELSON

The output pane shows the following log entries:

#	Time	Action	Message	Duration / Fetch
18	12:27:47	SET @OLD_FOREIGN_KEY_CHECKS=@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0	0 row(s) affected	0.000 sec
19	12:27:47	CREATE SCHEMA mavenmovies	Error Code: 1007. Can't create database 'mavenmovies'; database exists	0.000 sec
20	12:46:51	SELECT c.name AS category_name, COUNT(i.rental_id) AS total_rentals FROM category c JOIN film f ON c.category_id = f.category_id GROUP BY c.name ORDER BY total_rentals DESC LIMIT 10;	16 row(s) returned	0.015 sec / 0.000 sec
21	12:48:17	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 ORDER BY average_rental_rate DESC LIMIT 1;	1 row(s) returned	0.016 sec / 0.000 sec
22	12:49:23	SELECT f.title AS movie_title, c.first_name, c.last_name FROM film f JOIN inventory i ON f.film_id = i.film_id JOIN customer c ON i.customer_id = c.customer_id WHERE f.title = 'Gone with the Wind' ORDER BY c.last_name, c.first_name;	1000 row(s) returned	0.109 sec / 0.000 sec

**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.**

The screenshot shows the MySQL Workbench interface with a query editor and results pane. The query is:

```

Mavenmovies* 
5   film f
6   JOIN
7     film_actor fa ON f.film_id = fa.film_id
8   JOIN
9     actor a ON fa.actor_id = a.actor_id
10  WHERE
11    f.title = 'Gone with the Wind'
12  ORDER BY
13    a.last_name, a.first_name;

```

The results grid shows the following data:

first_name	last_name
ANNETTE	BALFOUR
ANNETTE	DEAN
ANNETTE	LEIGH
ANNETTE	REED
ANNETTE	ROBBINS
ANNETTE	WHITE

The output pane shows the following log entries:

#	Time	Action	Message	Duration / Fetch
18	12:27:47	SET @OLD_FOREIGN_KEY_CHECKS=@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0	0 row(s) affected	0.000 sec
19	12:27:47	CREATE SCHEMA mavenmovies	Error Code: 1007. Can't create database 'mavenmovies'; database exists	0.000 sec
20	12:46:51	SELECT c.name AS category_name, COUNT(i.rental_id) AS total_rentals FROM category c JOIN film f ON c.category_id = f.category_id GROUP BY c.name ORDER BY total_rentals DESC LIMIT 10;	16 row(s) returned	0.015 sec / 0.000 sec
21	12:48:17	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 ORDER BY average_rental_rate DESC LIMIT 1;	1 row(s) returned	0.016 sec / 0.000 sec
22	12:49:23	SELECT f.title AS movie_title, c.first_name, c.last_name FROM film f JOIN inventory i ON f.film_id = i.film_id JOIN customer c ON i.customer_id = c.customer_id WHERE f.title = 'Gone with the Wind' ORDER BY c.last_name, c.first_name;	1000 row(s) returned	0.109 sec / 0.000 sec

**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.**

```

Mavenmovies* x
1 • SELECT
2   c.first_name,
3   c.last_name,
4   ROUND(SUM(p.amount), 2) AS total_spent
5   FROM
6     customer c
7   JOIN
8     payment p ON c.customer_id = p.customer_id
9   GROUP BY

```

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

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

Action Output

#	Time	Action	Message	Duration / Fetch
21	12:48:17	SELECT lname AS language, ROUND(AVG(f.rental_rate), 2) AS average_rental_rate FROM film f JOIN inventory i ON f.film_id = i.film_id JOIN rental r ON i.inventory_id = r.inventory_id JOIN customer c ON r.customer_id = c.customer_id GROUP BY lname	1 row(s) returned	0.016 sec / 0.000 sec
22	12:49:23	SELECT title AS movie_title, c.first_name, c.last_name FROM film f JOIN inventory i ON f.film_id = i.film_id JOIN rental r ON i.inventory_id = r.inventory_id JOIN customer c ON r.customer_id = c.customer_id WHERE c.city_id = 1	1000 row(s) returned	0.109 sec / 0.000 sec
23	12:50:22	SELECT a.first_name, a.last_name FROM film f JOIN film_actor fa ON f.film_id = fa.film_id JOIN actor a ON fa.actor_id = a.actor_id WHERE f.title = 'ACADEMY DINOSAUR'	0 row(s) returned	0.016 sec / 0.000 sec
24	12:50:25	SELECT a.first_name, a.last_name FROM film f JOIN film_actor fa ON f.film_id = fa.film_id JOIN actor a ON fa.actor_id = a.actor_id WHERE f.title = 'ACADEMY DINOSAUR' AND a.first_name = 'MATTHEW'	0 row(s) returned	0.000 sec / 0.000 sec
25	12:51:04	SELECT c.first_name, c.last_name, ROUND(SUM(p.amount), 2) AS total_spent FROM payment p JOIN rental r ON p.rental_id = r.rental_id JOIN customer c ON r.customer_id = c.customer_id WHERE c.city_id = 1 GROUP BY c.first_name, c.last_name	599 row(s) returned	0.031 sec / 0.000 sec

**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.**

```

Mavenmovies* x
1 • SELECT
2   c.first_name,
3   c.last_name,
4   f.title AS movie_title
5   FROM
6     customer c
7   JOIN
8     address a ON c.address_id = a.address_id

```

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

first_name	last_name	movie_title
MATTIE	HOFFMAN	ACADEMY DINOSAUR
MATTIE	HOFFMAN	CHICKEN HELFIGHTERS
MATTIE	HOFFMAN	COLD-BLOODED 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

Action Output

#	Time	Action	Message	Duration / Fetch
22	12:49:23	SELECT title AS movie_title, c.first_name, c.last_name FROM film f JOIN inventory i ON f.film_id = i.film_id JOIN rental r ON i.inventory_id = r.inventory_id JOIN customer c ON r.customer_id = c.customer_id WHERE c.city_id = 1	1000 row(s) returned	0.109 sec / 0.000 sec
23	12:50:22	SELECT a.first_name, a.last_name FROM film f JOIN film_actor fa ON f.film_id = fa.film_id JOIN actor a ON fa.actor_id = a.actor_id WHERE f.title = 'ACADEMY DINOSAUR'	0 row(s) returned	0.016 sec / 0.000 sec
24	12:50:25	SELECT a.first_name, a.last_name FROM film f JOIN film_actor fa ON f.film_id = fa.film_id JOIN actor a ON fa.actor_id = a.actor_id WHERE f.title = 'ACADEMY DINOSAUR' AND a.first_name = 'MATTHEW'	0 row(s) returned	0.000 sec / 0.000 sec
25	12:51:04	SELECT c.first_name, c.last_name, ROUND(SUM(p.amount), 2) AS total_spent FROM payment p JOIN rental r ON p.rental_id = r.rental_id JOIN customer c ON r.customer_id = c.customer_id WHERE c.city_id = 1 GROUP BY c.first_name, c.last_name	599 row(s) returned	0.031 sec / 0.000 sec
26	12:51:44	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 WHERE a.city_id = 1	47 row(s) returned	0.016 sec / 0.000 sec

**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.**

```

Mavenmovies* x
Limit to 1000 rows
7   inventory i ON f.film_id = i.film_id
8   JOIN
9     rental r ON i.inventory_id = r.inventory_id
10  GROUP BY
11    f.film_id, f.title
12  ORDER BY
13    rental_count DESC
14  LIMIT 5;
15

```

movie_title	rental_count
BUCKET BROTHERHOOD	34
ROCKETEER MOTHER	33
JUGGLER HARDLY	32
FORWARD TEMPLE	32
RIDGEMONT SUBMARINE	32

Result 16 x

Output

Action Output

#	Time	Action	Message	Duration / Fetch
23	12:50:22	SELECT	a.first_name, a.last_name FROM film f JOIN film_actor fa ON f.film_id = fa.film_id JOIN ... 0 row(s) returned	0.016 sec / 0.000 sec
24	12:50:25	SELECT	a.first_name, a.last_name FROM film f JOIN film_actor fa ON f.film_id = fa.film_id JOIN ... 0 row(s) returned	0.000 sec / 0.000 sec
25	12:51:04	SELECT	c.first_name, c.last_name, ROUND(SUM(p.amount), 2) AS total_spent FROM customer c ... 599 row(s) returned	0.031 sec / 0.000 sec
26	12:51:44	SELECT	c.first_name, c.last_name, f.title AS movie_title FROM customer c JOIN address a ON ... 47 row(s) returned	0.016 sec / 0.000 sec
27	12:52:43	SELECT	f.title AS movie_title, COUNT(r.rental_id) AS rental_count FROM film f JOIN inventory i O... 5 row(s) returned	0.031 sec / 0.000 sec

**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.**

```

Mavenmovies* x
Limit to 1000 rows
7   JOIN
8     rental r ON c.customer_id = r.customer_id
9   JOIN
10    inventory i ON r.inventory_id = i.inventory_id
11  GROUP BY
12    c.customer_id, c.first_name, c.last_name
13  HAVING
14    COUNT(DISTINCT i.store_id) = 2;
15

```

customer_id	first_name	last_name
1	MARY	SMITH
2	PATRICIA	JOHNSON
3	LINDA	WILLIAMS
4	BARBARA	JONES
5	ELIZABETH	BROWN
6	JENNIFER	DAVIS
7	MARIA	MILLER
8	SUSAN	WILSON
9	MARGARET	MOORE
10	DOROTHY	TAYLOR
11	LISA	ANDERSON

Result 18 x

Output

Action Output

#	Time	Action	Message	Duration / Fetch
25	12:51:04	SELECT	c.first_name, c.last_name, ROUND(SUM(p.amount), 2) AS total_spent FROM customer c ... 599 row(s) returned	0.031 sec / 0.000 sec
26	12:51:44	SELECT	c.first_name, c.last_name, f.title AS movie_title FROM customer c JOIN address a ON ... 47 row(s) returned	0.016 sec / 0.000 sec
27	12:52:43	SELECT	f.title AS movie_title, COUNT(r.rental_id) AS rental_count FROM film f JOIN inventory i O... 5 row(s) returned	0.031 sec / 0.000 sec
28	12:54:02	SELECT	c.customer_id, c.first_name, c.last_name FROM customer c JOIN rental r ON c.custom... 599 row(s) returned	0.031 sec / 0.016 sec
29	12:54:51	SELECT	c.customer_id, c.first_name, c.last_name FROM customer c JOIN rental r ON c.custom... 599 row(s) returned	0.016 sec / 0.016 sec

## ❖ Windows Function:

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

The screenshot shows a SQL query in the Query Editor window titled 'Mavenmovies'. The code uses joins to connect three tables: rental, customer, and inventory. It groups by customer\_id and includes a having clause to filter for customers who have rented exactly 2 movies. The results are displayed in a grid, showing columns for customer\_id, first\_name, and last\_name. The output pane at the bottom shows the execution log with various SELECT statements and their execution times.

```

21   JOIN
22     rental r ON c.customer_id = r.customer_id
23   JOIN
24     inventory i ON r.inventory_id = i.inventory_id
25   GROUP BY
26     c.customer_id, c.first_name, c.last_name
27   HAVING
28     COUNT(DISTINCT i.store_id) = 2;
29

```

customer_id	first_name	last_name
1	MARY	SMITH
2	PATRICIA	JOHNSON
3	LINDA	WILLIAMS
4	BARBARA	JONES
5	ELIZABETH	BROWN
6	JENNIFER	DAVIS
7	MARIA	MILLER
8	SUSAN	WILSON
9	MARGARET	MOORE
10	DOROTHY	TAYLOR
11	LISA	ANDERSON

Action Output

#	Time	Action	Message	Duration / Fetch
27	12:52:43	SELECT f.title AS movie_title, COUNT(r.rental_id) AS rental_count FROM film f JOIN inventory i ON f.film_id = i.film_id JOIN rental r ON i.inventory_id = r.inventory_id WHERE r.rental_date > '2005-05-27' GROUP BY f.title ORDER BY rental_count DESC	5 row(s) returned	0.031 sec / 0.000 sec
28	12:54:02	SELECT c.customer_id, c.first_name, c.last_name FROM customer c JOIN rental r ON c.customer_id = r.customer_id WHERE r.rental_date > '2005-05-27' GROUP BY c.customer_id, c.first_name, c.last_name ORDER BY c.customer_id	599 row(s) returned	0.031 sec / 0.016 sec
29	12:54:51	SELECT c.customer_id, c.first_name, c.last_name FROM customer c JOIN rental r ON c.customer_id = r.customer_id WHERE r.rental_date > '2005-05-27' GROUP BY c.customer_id, c.first_name, c.last_name ORDER BY c.customer_id	599 row(s) returned	0.016 sec / 0.016 sec
30	12:57:26	SELECT c.customer_id, c.first_name, c.last_name, ROUND(SUM(p.amount), 2) AS total_spent, COUNT(r.rental_id) AS rental_count FROM customer c JOIN rental r ON c.customer_id JOIN payment p ON r.rental_id = p.rental_id WHERE r.rental_date > '2005-05-27' GROUP BY c.customer_id, c.first_name, c.last_name ORDER BY total_spent DESC	599 row(s) returned	0.047 sec / 0.000 sec
31	12:57:26	SELECT c.customer_id, c.first_name, c.last_name FROM customer c JOIN rental r ON c.customer_id WHERE r.rental_date > '2005-05-27' GROUP BY c.customer_id, c.first_name, c.last_name ORDER BY c.customer_id	599 row(s) returned	0.016 sec / 0.016 sec

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

The screenshot shows a SQL query in the Query Editor window titled 'Mavenmovies'. The code uses a windowed sum function to calculate the cumulative revenue for each film over time. The results are displayed in a grid, showing columns for film\_title, payment\_date, amount, and cumulative\_revenue. The output pane at the bottom shows the execution log with various SELECT statements and their execution times.

```

1  •  SELECT
2    f.title AS film_title,
3    p.payment_date,
4    p.amount,
5    SUM(p.amount) OVER (
6      PARTITION BY f.film_id
7      ORDER BY p.payment_date
8    ) AS cumulative_revenue
9  FROM

```

film_title	payment_date	amount	cumulative_revenue
ACADEMY DINOSAUR	2005-05-27 07:03:28	0.99	0.99
ACADEMY DINOSAUR	2005-05-30 20:21:07	1.99	2.98
ACADEMY DINOSAUR	2005-06-15 02:57:51	0.99	3.97
ACADEMY DINOSAUR	2005-06-17 20:24:00	0.99	4.96
ACADEMY DINOSAUR	2005-06-21 00:30:26	1.99	6.95
ACADEMY DINOSAUR	2005-07-07 10:41:31	0.99	7.94
ACADEMY DINOSAUR	2005-07-07 20:59:00	0.99	8.93
ACADEMY DINOSAUR	2005-07-09 19:03:15	0.99	9.92
ACADEMY DINOSAUR	2005-07-10 13:07:31	0.99	10.91
ACADEMY DINOSAUR	2005-07-27 07:51:11	0.99	11.90
ACADEMY DINOSAUR	2005-07-29 09:41:38	1.99	13.89

Action Output

#	Time	Action	Message	Duration / Fetch
29	12:54:51	SELECT c.customer_id, c.first_name, c.last_name FROM customer c JOIN rental r ON c.customer_id = r.customer_id WHERE r.rental_date > '2005-05-27' GROUP BY c.customer_id, c.first_name, c.last_name ORDER BY c.customer_id	599 row(s) returned	0.016 sec / 0.016 sec
30	12:57:26	SELECT c.customer_id, c.first_name, c.last_name, ROUND(SUM(p.amount), 2) AS total_spent, COUNT(r.rental_id) AS rental_count FROM customer c JOIN rental r ON c.customer_id JOIN payment p ON r.rental_id = p.rental_id WHERE r.rental_date > '2005-05-27' GROUP BY c.customer_id, c.first_name, c.last_name ORDER BY total_spent DESC	599 row(s) returned	0.047 sec / 0.000 sec
31	12:57:26	SELECT c.customer_id, c.first_name, c.last_name FROM customer c JOIN rental r ON c.customer_id WHERE r.rental_date > '2005-05-27' GROUP BY c.customer_id, c.first_name, c.last_name ORDER BY c.customer_id	599 row(s) returned	0.016 sec / 0.016 sec
32	12:58:12	SELECT f.title AS film_title, p.payment_date, p.amount, SUM(p.amount) OVER (PARTITION BY f.film_id ORDER BY p.payment_date) AS cumulative_revenue FROM film f JOIN payment p ON f.film_id = p.film_id WHERE p.payment_date > '2005-05-27' ORDER BY f.title	16044 row(s) returned	0.188 sec / 0.015 sec
33	12:59:22	SELECT f.title AS film_title, p.payment_date, p.amount, SUM(p.amount) OVER (PARTITION BY f.film_id ORDER BY p.payment_date) AS cumulative_revenue FROM film f JOIN payment p ON f.film_id = p.film_id WHERE p.payment_date > '2005-05-27' ORDER BY f.title	16044 row(s) returned	0.172 sec / 0.015 sec

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

Mavenmovies\*

```

1 • SELECT
2     f.length AS film_length,
3     f.title AS film_title,
4     ROUND(AVG(DATEDIFF(r.return_date, r.rental_date)), 2) AS avg_rental_duration
5   FROM
6     film f
7   JOIN
8     inventory i ON f.film_id = i.film_id
9   JOIN

```

Result Grid | Filter Rows: Export: Wrap Cell Content: Result Grid Form Editor Field Types Read Only Context Help Snippets

film_length	film_title	avg_rental_duration
46	RIDGEVENTH 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

Result 23 x

Output:

#	Time	Action	Message	Duration / Fetch
30	12:57:26	SELECT c.customer_id, c.first_name, c.last_name, ROUND(SUM(p.amount), 2) AS total_spent, ...	599 row(s) returned	0.047 sec / 0.000 sec
31	12:57:26	SELECT c.customer_id, c.first_name, c.last_name FROM customer c JOIN rental r ON c.customer_id = r.customer_id	599 row(s) returned	0.016 sec / 0.016 sec
32	12:58:12	SELECT f.title AS film_title, p.payment_date, p.amount, SUM(p.amount) OVER () PARTITION BY f.category_id	16044 row(s) returned	0.188 sec / 0.015 sec
33	12:59:22	SELECT f.title AS film_title, p.payment_date, p.amount, SUM(p.amount) OVER () PARTITION BY f.category_id	16044 row(s) returned	0.172 sec / 0.015 sec
34	13:00:50	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	958 row(s) returned	0.047 sec / 0.000 sec

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

Mavenmovies\*

```

1 • SELECT *
2   FROM (
3     SELECT
4       c.name AS category,
5       f.title AS film_title,
6       COUNT(r.rental_id) AS rental_count,
7       RANK() OVER (
8         PARTITION BY c.category_id
9         ORDER BY COUNT(r.rental_id) DESC

```

Result Grid | Filter Rows: Export: Wrap Cell Content: Result Grid Form Editor Field Types Read Only Context Help Snippets

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	IDOLIS SNATCHERS	30	2
Children	SWEETHEARTS SUSPECTS	29	3

Result 24 x

Output:

#	Time	Action	Message	Duration / Fetch
32	12:58:12	SELECT f.title AS film_title, p.payment_date, p.amount, SUM(p.amount) OVER () PARTITION BY f.category_id	16044 row(s) returned	0.188 sec / 0.015 sec
33	12:59:22	SELECT f.title AS film_title, p.payment_date, p.amount, SUM(p.amount) OVER () PARTITION BY f.category_id	16044 row(s) returned	0.172 sec / 0.015 sec
34	13:00:50	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	958 row(s) returned	0.047 sec / 0.000 sec
35	13:01:52	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)	Error Code: 4015. Window function is allowed only in SELECT list and ORDER BY clause	0.000 sec
36	13:02:23	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) AS rank_within_category ) t	56 row(s) returned	0.031 sec / 0.000 sec

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

The screenshot shows the MySQL Workbench interface with a query editor and results grid. The query is:

```

1 • SELECT
2     c.customer_id,
3     c.first_name,
4     c.last_name,
5     COUNT(r.rental_id) AS total_rentals,
6     ROUND(AVG(COUNT(r.rental_id)) OVER (), 2) AS avg_rentals_across_all,
7     COUNT(r.rental_id) - ROUND(AVG(COUNT(r.rental_id)) OVER (), 2) AS difference_from_avg
8
9 FROM
10    customer c
  
```

The results grid displays 25 rows of data:

customer_id	first_name	last_name	total_rentals	avg_rentals_across_all	difference_from_avg
148	ELEANOR	HUNT	46	26.78	19.22
526	KARA	SEAL	45	26.78	18.22
236	MARICIA	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

The output pane shows the execution log:

#	Time	Action	Message	Duration / Fetch
33	12:59:22	SELECT f.title AS film_title, p.payment_date, p.amount, SUM(p.amount) OVER (PARTITION BY f.title) AS monthly_revenue	16044 row(s) returned	0.172 sec / 0.015 sec
34	13:00:50	SELECT f.length AS film_length, f.title AS film_title, ROUND(AVG(DATEDIFF(r.return_date, r.rental_date)), 2) AS avg_rentals_across_all	958 row(s) returned	0.047 sec / 0.000 sec
35	13:01:52	SELECT c.name AS category, f.title AS film_title, COUNT(r.rental_id) AS rental_count, RANK() OVER (PARTITION BY f.title ORDER BY COUNT(r.rental_id) DESC) AS rank	Error Code: 4015. Window function is allowed only in SELECT list and ORDER BY clause	0.000 sec
36	13:02:23	SELECT * FROM ( SELECT c.name AS category, f.title AS film_title, COUNT(r.rental_id) AS rental_count, RANK() OVER (PARTITION BY f.title ORDER BY COUNT(r.rental_id) DESC) AS rank ) t	56 row(s) returned	0.031 sec / 0.000 sec
37	13:03:39	SELECT c.customer_id, c.first_name, c.last_name, COUNT(r.rental_id) AS total_rentals, ROUND((SUM(p.amount) / COUNT(r.rental_id)), 2) AS avg_rentals_across_all	599 row(s) returned	0.015 sec / 0.000 sec
38	13:07:59	SELECT DATE_FORMAT(payment_date, '%Y-%m') AS month, ROUND(SUM(amount), 2) AS monthly_revenue	5 row(s) returned	0.031 sec / 0.000 sec

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

The screenshot shows the MySQL Workbench interface with a query editor and results grid. The query is:

```

1 • SELECT
2     DATE_FORMAT(payment_date, '%Y-%m') AS month,
3     ROUND(SUM(amount), 2) AS monthly_revenue
4
5 FROM
6     payment
7 GROUP BY
8     DATE_FORMAT(payment_date, '%Y-%m')
9 ORDER BY
10    month;
  
```

The results grid displays 6 rows of data:

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

The output pane shows the execution log:

#	Time	Action	Message	Duration / Fetch
34	13:00:50	SELECT f.length AS film_length, f.title AS film_title, ROUND(AVG(DATEDIFF(r.return_date, r.rental_date)), 2) AS avg_rentals_across_all	958 row(s) returned	0.047 sec / 0.000 sec
35	13:01:52	SELECT c.name AS category, f.title AS film_title, COUNT(r.rental_id) AS rental_count, RANK() OVER (PARTITION BY f.title ORDER BY COUNT(r.rental_id) DESC) AS rank	Error Code: 4015. Window function is allowed only in SELECT list and ORDER BY clause	0.000 sec
36	13:02:23	SELECT * FROM ( SELECT c.name AS category, f.title AS film_title, COUNT(r.rental_id) AS rental_count, RANK() OVER (PARTITION BY f.title ORDER BY COUNT(r.rental_id) DESC) AS rank ) t	56 row(s) returned	0.031 sec / 0.000 sec
37	13:03:39	SELECT c.customer_id, c.first_name, c.last_name, COUNT(r.rental_id) AS total_rentals, ROUND((SUM(p.amount) / COUNT(r.rental_id)), 2) AS avg_rentals_across_all	599 row(s) returned	0.015 sec / 0.000 sec
38	13:07:59	SELECT DATE_FORMAT(payment_date, '%Y-%m') AS month, ROUND(SUM(amount), 2) AS monthly_revenue	5 row(s) returned	0.031 sec / 0.000 sec

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

The screenshot shows the MySQL Workbench interface with the following details:

- SQL Editor:** Contains the following SQL query:
 

```

1 • SELECT *
2   FROM (
3     SELECT
4       c.customer_id,
5       c.first_name,
6       c.last_name,
7       ROUND(SUM(p.amount), 2) AS total_spent,
8       PERCENT_RANK() OVER (ORDER BY SUM(p.amount)) AS spending_percentile
9   FROM
      
```
- Result Grid:** Displays the results of the query, showing 27 rows of customer information, including their ID, first name, last name, total spent, and spending percentile. The results are ordered by spending percentile.
- Action Output:** Shows the execution log with 5 entries, detailing the time, action, message, and duration for each query step.
- SQLAdditions:** A panel on the right containing a note about automatic context help being disabled.

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

The screenshot shows the MySQL Workbench interface with the following details:

- SQL Editor:** Contains the following SQL query:
 

```

1 • SELECT
2   Execute the statement under the keyboard cursor
3   COUNT(r.rental_id) AS rental_count,
4   SUM(COUNT(r.rental_id)) OVER (
5     ORDER BY COUNT(r.rental_id) DESC
6   ) AS running_total
7   FROM
8     category c
9   JOIN
      
```
- Result Grid:** Displays the results of the query, showing 28 categories with their rental count and running total.
- Action Output:** Shows the execution log with 10 entries, detailing the time, action, message, and duration for each query step.
- SQLAdditions:** A panel on the right containing a note about automatic context help being disabled.

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

The screenshot shows the MySQL Workbench interface. The SQL editor contains the following query:

```

1 • SELECT *
2   FROM (
3     SELECT
4       c.name AS category,
5       f.title AS film_title,
6       COUNT(r.rental_id) AS film_rental_count,
7       ROUND(AVG(COUNT(r.rental_id)) OVER (PARTITION BY c.category_id), 2) AS avg_category_rentals
8     FROM
9       category c

```

The results grid displays the following data:

category	film_title	film_rental_count	avg_category_rentals
Action	GOSFORD DONNIE	8	18.23
Action	PARK CITIZEN	8	18.23
Action	CASUALTIES ENCINO	9	18.23
Action	MONTEZUMA COMMAND	9	18.23
Action	ANTITRUST TOMATOES	10	18.23
Action	GRAIL FRANKENSTEIN	10	18.23
Action	DRAGON SQUAD	11	18.23
Action	MAGNOLIA FORRESTER	11	18.23
Action	LORD ARIZONA	11	18.23
Action	DARKO DORADO	11	18.23
Action	SPEAKEASY DATE	12	18.23

The output pane shows the execution log:

- 38 13:07:59 SELECT \* FROM (SELECT c.customer\_id, c.first\_name, c.last\_name, ROUND(SUM(amount), 2) AS monthly\_revenue FROM payment GROUP BY DATE\_FORMAT(payment\_date, '%Y-%m') ORDER BY monthly\_revenue DESC LIMIT 1) AS result
- 39 13:08:49 SELECT \* FROM (SELECT c.name AS category, COUNT(r.rental\_id) AS rental\_count, SUM(COUNT(r.rental\_id)) OVER (PARTITION BY c.category\_id) AS avg\_category\_rentals FROM category c)
- 40 13:09:52 SELECT c.name AS category, COUNT(r.rental\_id) AS rental\_count, SUM(COUNT(r.rental\_id)) OVER (PARTITION BY c.category\_id) AS avg\_category\_rentals FROM category c
- 41 13:10:47 SELECT c.name AS category, f.title AS film\_title, COUNT(r.rental\_id) AS film\_rental\_count, ROUND(AVG(COUNT(r.rental\_id)) OVER (PARTITION BY c.category\_id), 2) AS avg\_category\_rentals FROM category c
- 42 13:11:19 SELECT \* FROM (SELECT c.name AS category, f.title AS film\_title, COUNT(r.rental\_id) AS film\_rental\_count, ROUND(AVG(COUNT(r.rental\_id)) OVER (PARTITION BY c.category\_id), 2) AS avg\_category\_rentals FROM category c)

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

The screenshot shows the MySQL Workbench interface. The SQL editor contains the following query:

```

1 • SELECT
2   DATE_FORMAT(payment_date, '%Y-%m') AS month,
3   ROUND(SUM(amount), 2) AS total_revenue
4   FROM
5   payment
6   GROUP BY
7   DATE_FORMAT(payment_date, '%Y-%m')
8   ORDER BY
9   total_revenue DESC

```

The results grid displays the following data:

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

The output pane shows the execution log:

- 39 13:08:49 SELECT \* FROM (SELECT c.customer\_id, c.first\_name, c.last\_name, ROUND(SUM(amount), 2) AS monthly\_revenue FROM payment GROUP BY DATE\_FORMAT(payment\_date, '%Y-%m') ORDER BY monthly\_revenue DESC LIMIT 1) AS result
- 40 13:09:52 SELECT c.name AS category, COUNT(r.rental\_id) AS rental\_count, SUM(COUNT(r.rental\_id)) OVER (PARTITION BY c.category\_id) AS avg\_category\_rentals FROM category c
- 41 13:10:47 SELECT c.name AS category, f.title AS film\_title, COUNT(r.rental\_id) AS film\_rental\_count, ROUND(AVG(COUNT(r.rental\_id)) OVER (PARTITION BY c.category\_id), 2) AS avg\_category\_rentals FROM category c
- 42 13:11:19 SELECT \* FROM (SELECT c.name AS category, f.title AS film\_title, COUNT(r.rental\_id) AS film\_rental\_count, ROUND(AVG(COUNT(r.rental\_id)) OVER (PARTITION BY c.category\_id), 2) AS avg\_category\_rentals FROM category c)
- 43 13:12:21 SELECT DATE\_FORMAT(payment\_date, '%Y-%m') AS month, ROUND(SUM(amount), 2) AS total\_revenue FROM payment GROUP BY DATE\_FORMAT(payment\_date, '%Y-%m')

## ❖ Normalisation & CTE:

### 1. First Normal Form (1NF):

- 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**