Assignment : SQL

1. Create a table called employees with the following structure

emp_id (integer, should not be NULL and should be a primary key)

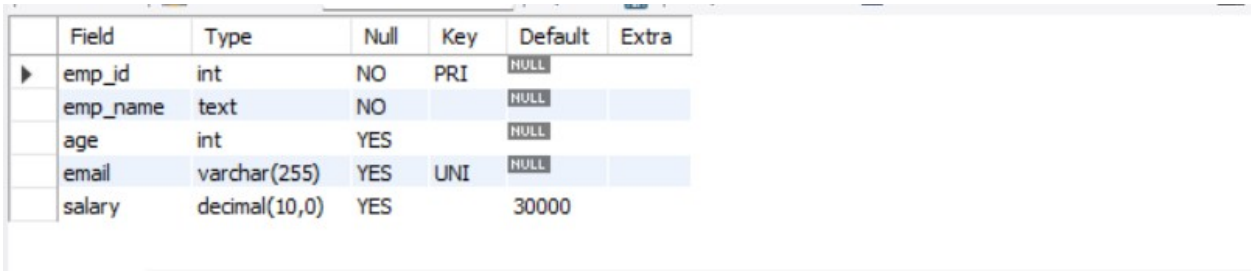
emp_name (text, should not be NULL)

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.



Field	Type	Null	Key	Default	Extra
emp_id	int	NO	PRI	NULL	
emp_name	text	NO		NULL	
age	int	YES		NULL	
email	varchar(255)	YES	UNI	NULL	
salary	decimal(10,0)	YES		30000	

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

constrains define conditions or restrictions on columns or groups of columns in a table, ensuring that the data meets certain criteria. Constraints help maintain data accuracy, consistency, and reliability within a database.

common types of constraints:

1.PRIMARY KEY

```
example:
DROP DATABASE IF EXISTS db;
CREATE DATABASE db;
USE db;
CREATE TABLE student(roll_no INT PRIMARY KEY,name VARCHAR(10),age
INT);
DESCRIBE student;
```

2.FOREIGN KEY

```
example:
DROP DATABASE IF EXISTS db;
CREATE DATABASE db;
USE db;
CREATE TABLE student(roll_no INT PRIMARY KEY,name VARCHAR(10),age
INT);
INSERT INTO student VALUES(1,"abc",16);
INSERT INTO student VALUES(2,"bca",16);
INSERT INTO student VALUES(3,"cab",16);
CREATE TABLE student1(roll_no INT PRIMARY KEY ,marks INT,FOREIGN
KEY(roll_no) REFERENCES student(roll_no));
INSERT INTO student1 VALUES(1,97);
INSERT INTO student1 VALUES(2,98);
INSERT INTO student1 VALUES(3,89);
DESCRIBE student1;
```

3.NOT NULL

```
example:
DROP DATABASE IF EXISTS db;
CREATE DATABASE db;
USE db;
CREATE TABLE student(roll_no INT PRIMARY KEY,name VARCHAR(10) NOT
NULL,age INT);
DESCRIBE student;
```

4.UNIQUE

```
example:
DROP DATABASE IF EXISTS db;
CREATE DATABASE db;
USE db;
CREATE TABLE student(roll_no INT PRIMARY KEY,name VARCHAR(10)
UNIQUE ,age INT);
DESCRIBE student;
```

5.CHECK

```
example:
DROP DATABASE IF EXISTS db;
CREATE DATABASE db;
USE db;
CREATE TABLE student(roll_no INT PRIMARY KEY,name VARCHAR(10) ,age INT
CHECK(age>18));
DESCRIBE student;
```

6.DEFAUL

```
example:
DROP DATABASE IF EXISTS db;
CREATE DATABASE db;
USE db;
CREATE TABLE student(roll_no INT PRIMARY KEY,name VARCHAR(10) DEFAULT
"no",age INT );
INSERT INTO student (roll_no,age)VALUES(2,18);
DESCRIBE student;
```

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

The NOT NULL constraint ensures that a column cannot contain NULL values. It requires every row to have a value for that column

primary key cannot contain NULL values because it should contain unique values to identify each row uniquely

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.

In MySQL, We can add or remove constraints on an existing table using the ALTER TABLE statement.

1. Adding Constraints

We use the ALTER TABLE statement followed by the specific constraint type to add. Common constraints include PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK, NOT NULL and DEFAULT

1. Adding a Primary Key Constraint

Syntax:

ALTER PRIMARY table_name ADD PRIMARY KEY (column_name);

Example:

ALTER TABLE employees ADD PRIMARY KEY (emp_id);

2. Adding a Foreign Key Constraint

Syntax:

ALTER TABLE table_name ADD CONSTRAINT constraint_name FOREIGN KEY (column_name) REFERENCES other_table(other_column);

Example:

ALTER TABLE employees ADD CONSTRAINT fk_dept FOREIGN KEY (dept_id) REFERENCES departments(dept_id);

3. Adding a NOT NULL Constraint

Syntax:

ALTER TABLE table_name MODIFY COLUMN column_name column_type NOT NULL;

Example:

ALTER TABLE employees MODIFY COLUMN emp_name VARCHAR(50) NOT NULL;

4. Adding a Unique Constraint

Syntax:

ALTER TABLE table_name ADD CONSTRAINT constraint_name UNIQUE (column_name);

Example:

ALTER TABLE employees ADD CONSTRAINT unique_emp_name UNIQUE (emp_name);

5. Adding a Check Constraint (MySQL 8.0.16+)

Syntax:

ALTER TABLE table_name ADD CONSTRAINT constraint_name CHECK (condition);

Example:

ALTER TABLE employees ADD CONSTRAINT chk_salary CHECK (salary > 0);

6. Adding a Default Constraint

MySQL doesn't directly use `DEFAULT` as a named constraint, but you can set a default value by modifying the column with `DEFAULT`.

Syntax:

ALTER TABLE table_name MODIFY COLUMN column_name column_type DEFAULT default_value;

Example:

ALTER TABLE employees MODIFY COLUMN dept_id INT DEFAULT 1;

2. Removing Constraints

We use different ALTER TABLE commands depending on the constraint type.

Certainly! Here are the commands for each constraint type in the specified format:

1. Removing a Primary Key Constraint

Syntax:

ALTER TABLE table_name DROP PRIMARY KEY;

Example:

ALTER TABLE employees DROP PRIMARY KEY;

2. Removing a Foreign Key Constraint

Syntax:

ALTER TABLE table_name DROP FOREIGN KEY constraint_name;

Example:

ALTER TABLE employees DROP FOREIGN KEY fk_dept;

3. Removing a NOT NULL Constraint

Syntax:

ALTER TABLE table_name MODIFY COLUMN column_name column_type;

Example:

ALTER TABLE employees MODIFY COLUMN emp_name VARCHAR(50);

4. Removing a Unique Constraint

Syntax:

ALTER TABLE table_name DROP INDEX index_name;

Example:

ALTER TABLE employees DROP INDEX unique_emp_name;

5. Removing a Check Constraint

Syntax:

ALTER TABLE table_name DROP CHECK constraint_name;

Example:

ALTER TABLE employees DROP CHECK chk_salary;

6. Removing a Default Constraint

Syntax:

ALTER TABLE table_name ALTER COLUMN column_name DROP DEFAULT;

Example:

ALTER TABLE employees ALTER COLUMN dept_id DROP DEFAULT;

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.

1. **Primary Key Constraint Violation**

The **Primary Key** constraint enforces that each value in the primary key column(s) must be unique and not null.

- **Consequence**: Attempting to insert or update a duplicate primary key value will result in an error. Deleting a record with primary keys is allowed as long as there are no related foreign keys referencing it (otherwise, see Foreign Key Violation below).
- **Example Error Message**:
  - `Error Code: 1062. Duplicate entry '1' for key 'PRIMARY'`

Example Scenario:

- Assume emp_id 1 already exists in the employees table INSERT INTO employees (emp_id, emp_name, dept_id) VALUES (1, 'John Doe', 101);

2. **Foreign Key Constraint Violation**

A **Foreign Key** constraint ensures that a value in a column corresponds to an existing value in a referenced column of another table. It enforces referential integrity.

- **Consequence**:
  - **Insert or Update**: Inserting or updating a foreign key column with a value that does not exist in the referenced table will cause an error.
  - **Delete**: Attempting to delete a row from the referenced table (parent table) while it's still referenced by a foreign key in another table (child table) will also cause an error.
- **Example Error Message**:
  - `Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails`

Example Scenario:

-- Attempting to insert a dept_id that does not exist in the departments table INSERT INTO employees (emp_id, emp_name, dept_id) VALUES (2, 'Jane Smith', 999);

3. **NOT NULL Constraint Violation**

The **NOT NULL** constraint prevents columns from having NULL values.

- **Consequence**: Attempting to insert or update a row with a NULL value in a column defined with NOT NULL will cause an error.
- **Example Error Message**:
  - `Error Code: 1048. Column 'emp_name' cannot be null`

Example Scenario:

-- Attempting to insert a NULL value in a NOT NULL column INSERT INTO employees (emp_id, emp_name, dept_id) VALUES (3, NULL, 101);

4. **Unique Constraint Violation**

A **Unique** constraint enforces that all values in the constrained column or group of columns must be unique.

- **Consequence**: Attempting to insert or update a row with a value that already exists in a unique-constrained column will cause an error.
- **Example Error Message**:
  - `Error Code: 1062. Duplicate entry 'John Doe' for key 'unique_emp_name'`

Example Scenario:

-- Assume 'John Doe' already exists in the emp_name column with a unique constraint INSERT INTO employees (emp_id, emp_name, dept_id) VALUES (4, 'John Doe', 102);

5. **Check Constraint Violation** (MySQL 8.0.16+)

The **Check** constraint enforces a specific condition that must be met for the data to be valid.

- **Consequence**: Attempting to insert or update a row where the check condition is not met will result in an error.
- **Example Error Message**:
  - `Error Code: 3819. Check constraint 'chk_salary' is violated.`

Example Scenario:

-- Assume there is a check constraint on salary requiring it to be > 0 INSERT INTO employees (emp_id, emp_name, dept_id, salary) VALUES (5, 'Michael Scott', 101, -500);

6. **Default Constraint Violation**

A **Default** constraint sets a default value for a column if no value is provided.

- **Consequence**: If an insert does not specify a value for a column with a default, the default value will be used. No direct error is triggered for violating a default, but inserting `NULL` into a NOT NULL column with a default value may trigger a NOT NULL constraint error instead.

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**

To Add primary key constraint to product_id ALTER TABLE products ADD PRIMARY KEY (product_id);

To Set a default value of 50.00 for the price column ALTER TABLE products ALTER COLUMN price SET DEFAULT 50.00;

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.

SELECT Students.student_name, Classes.class_name FROM Students INNER JOIN Classes ON Students.class_id = Classes.class_id;

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).

SELECT Orders.order_id, Customers.customer_name, Products.product_name FROM Products LEFT JOIN Orders ON Products.order_id = Orders.order_id INNER JOIN Customers ON Orders.customer_id = Customers.customer_id;

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.

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

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

SELECT Orders.order_id, Customers.customer_name, Order_Details.quantity FROM Orders INNER JOIN Customers ON Orders.customer_id = Customers.customer_id INNER JOIN Order_Details ON Orders.order_id = Order_Details.order_id;

SQL Commands

1. **Identify the primary keys and foreign keys in the Maven Movies database. Discuss the differences:**

   Primary Keys:

   - `actor` table: `actor_id`
   - `address` table: `address_id`
   - `customer` table: `customer_id`

- Other tables contain similar primary keys, typically the table name followed by `_id`.

Foreign Keys:

- `address` table: `city_id` references `city(city_id)`
- `customer` table: `address_id` references `address(address_id)` and `store_id` references `store(store_id)`

Differences:

- A **primary key** uniquely identifies each record in a table and cannot be null.
- A **foreign key** is a reference to a primary key in another table, establishing a relationship between the two tables and ensuring referential integrity.

2. **List all details of actors:**

   SELECT * FROM actor;

3. **List all customer information:**

   SELECT * FROM customer;

4. **List different countries:**

   SELECT DISTINCT country FROM country;

5. **Display all active customers:**

   SELECT * FROM customer WHERE active = 1;

6. **List all rental IDs for customer with ID 1:**

   SELECT rental_id FROM rental WHERE customer_id = 1;

7. **Display all films with a rental duration greater than 5:**

   SELECT * FROM film WHERE rental_duration > 5;

8. **List the total number of films with a replacement cost between $15 and $20:**

   SELECT COUNT(*) AS total_films FROM film WHERE replacement_cost > 15 AND replacement_cost < 20;

9. **Display the count of unique actor first names:**

   SELECT COUNT(DISTINCT first_name) AS unique_first_names FROM actor;

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

    SELECT * FROM customer LIMIT 10;

11. **Display the first 3 records from the customer table where the first name starts with 'B':**

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

12. **Display the names of the first 5 movies rated as ‘G’:**

    SELECT title FROM film WHERE rating = 'G' LIMIT 5;

13. **Find all customers whose first name starts with "A":**

    SELECT * FROM customer WHERE first_name LIKE 'A%';

14. **Find all customers whose first name ends with "A":**

    SELECT * FROM customer WHERE first_name LIKE '%A';

15. **Display the list of the first 4 cities starting and ending with 'A':**

    SELECT DISTINCT city FROM city WHERE city LIKE 'A%' AND city LIKE '%A' LIMIT 4;

16. **Find all customers whose first name contains "NI":**

    SELECT * FROM customer WHERE first_name LIKE '%NI%';

17. **Find all customers whose first name has "R" as the second character:**

    SELECT * FROM customer WHERE first_name LIKE '_R%';

18. **Find all customers whose first name starts with "A" and is at least 5 characters:**

    SELECT * FROM customer WHERE first_name LIKE 'A%' AND LENGTH(first_name) >= 5;

19. **Find all customers whose first name starts with "A" and ends with "O":**

    SELECT * FROM customer WHERE first_name LIKE 'A%O';

20. **Get films rated PG and PG-13 using the `IN` operator:**

    SELECT * FROM film WHERE rating IN ('PG', 'PG-13');

21. **Get films with length between 50 and 100 using the `BETWEEN` operator:**

    SELECT * FROM film WHERE length BETWEEN 50 AND 100;

22. **Get the top 50 actors using the `LIMIT` operator:**

    SELECT * FROM actor LIMIT 50;

23. **Get the distinct film IDs from the inventory table:**

    SELECT DISTINCT film_id FROM inventory;

Functions:

Basic Aggregate Functions:

Basic Aggregate Functions:

1. **Retrieve the total number of rentals made in the Sakila database:**

   SELECT COUNT(*) AS total_rentals FROM rental;

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

   SELECT AVG(rental_duration) AS average_rental_duration FROM film;

String Functions:

1. **Display the first name and last name of customers in uppercase:**

   SELECT UPPER(first_name) AS first_name_upper, UPPER(last_name) AS last_name_upper FROM customer;

2. **Extract the month from the rental date and display it alongside the rental ID:**

   SELECT rental_id, MONTH(rental_date) AS rental_month FROM rental;

GROUP BY:

1. **Retrieve the count of rentals for each customer (display customer ID and the count of rentals):**

   SELECT customer_id, COUNT(*) AS rental_count FROM rental GROUP BY customer_id;

2. **Find the total revenue generated by each store:**

   SELECT store_id, SUM(amount) AS total_revenue FROM payment GROUP BY store_id;

3. **Determine the total number of rentals for each category of movies:**

   SELECT fc.category_id, c.name AS category_name, COUNT(r.rental_id) AS total_rentals FROM rental r JOIN inventory i ON r.inventory_id = i.inventory_id JOIN film_category fc ON i.film_id = fc.film_id JOIN category c ON fc.category_id = c.category_id GROUP BY fc.category_id, c.name;

4. **Find the average rental rate of movies in each language:**

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

Joins

1. **Display the title of the movie, and the customer's first and last name who rented it:**

   SELECT f.title AS movie_title, c.first_name AS customer_first_name, c.last_name AS customer_last_name FROM rental r JOIN inventory i ON r.inventory_id = i.inventory_id JOIN film f ON i.film_id = f.film_id JOIN customer c ON r.customer_id = c.customer_id;

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

   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 = 'Gone with the Wind';

3. **Retrieve the customer names along with the total amount they've spent on rentals:**

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

4. **List the titles of movies rented by each customer in a particular city (e.g., 'London'):**

   SELECT c.first_name AS customer_first_name, c.last_name AS customer_last_name, f.title AS movie_title FROM customer c JOIN address a ON c.address_id = a.address_id JOIN city ci ON a.city_id = ci.city_id JOIN rental r ON c.customer_id = r.customer_id JOIN inventory i ON r.inventory_id = i.inventory_id JOIN film f ON i.film_id = f.film_id WHERE ci.city = 'London' GROUP BY c.customer_id, f.title;

Advanced Joins and GROUP BY:

1. **Display the top 5 rented movies along with the number of times they've been rented:** ~~~sql 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 GROUP BY f.film_id, f.title ORDER BY rental_count DESC LIMIT 5;

2. **Determine the customers who have rented movies from both stores (store ID 1 and store ID 2):** ```sql SELECT c.customer_id, c.first_name, c.last_name FROM customer c JOIN rental r ON c.customer_id = r.customer_id JOIN inventory i ON r.inventory_id = i.inventory_id WHERE i.store_id IN (1, 2) GROUP BY c.customer_id, c.first_name, c.last_name HAVING COUNT(DISTINCT i.store_id) = 2;

Windows Function:

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

```
SELECT c.customer_id, c.first_name, c.last_name, SUM(p.amount) AS
total_spent,
       RANK() OVER (ORDER BY SUM(p.amount) DESC) AS spending_rank
FROM customer c
JOIN payment p ON c.customer_id = p.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name;
```

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

```
SELECT f.title, r.rental_date, SUM(p.amount) OVER (PARTITION BY
f.film_id ORDER BY r.rental_date) AS cumulative_revenue
FROM rental r
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN film f ON i.film_id = f.film_id
JOIN payment p ON r.rental_id = p.rental_id
ORDER BY f.title, r.rental_date;
```

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

```
SELECT f.title, f.length, AVG(f.rental_duration) OVER (PARTITION
BY f.length) AS avg_rental_duration
FROM film f;
```

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

```
SELECT c.name AS category, f.title, COUNT(r.rental_id) AS
rental_count,
       RANK() OVER (PARTITION BY c.category_id ORDER BY
COUNT(r.rental_id) DESC) AS category_rank
FROM category c
JOIN film_category fc ON c.category_id = fc.category_id
JOIN film f ON fc.film_id = f.film_id
JOIN inventory i ON f.film_id = i.film_id
JOIN rental r ON i.inventory_id = r.inventory_id
GROUP BY c.category_id, c.name, f.title
HAVING category_rank <= 3;
```

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

```
SELECT c.customer_id, c.first_name, c.last_name,
COUNT(r.rental_id) AS total_rentals,
```

```
       COUNT(r.rental_id) - AVG(COUNT(r.rental_id)) OVER () AS
rental_diff
FROM customer c
JOIN rental r ON c.customer_id = r.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name;
```

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

```
SELECT DATE_FORMAT(p.payment_date, '%Y-%m') AS month,
SUM(p.amount) AS monthly_revenue,
       SUM(SUM(p.amount)) OVER (ORDER BY
DATE_FORMAT(p.payment_date, '%Y-%m')) AS cumulative_revenue
FROM payment p
GROUP BY DATE_FORMAT(p.payment_date, '%Y-%m');
```

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

```
SELECT customer_id, first_name, last_name, total_spent
FROM (
    SELECT c.customer_id, c.first_name, c.last_name,
SUM(p.amount) AS total_spent,
           NTILE(5) OVER (ORDER BY SUM(p.amount) DESC) AS
spending_percentile
    FROM customer c
    JOIN payment p ON c.customer_id = p.customer_id
    GROUP BY c.customer_id, c.first_name, c.last_name
) AS spending_data
WHERE spending_percentile = 1;
```

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

```
SELECT c.name AS category, f.title, COUNT(r.rental_id) AS
rental_count,
       SUM(COUNT(r.rental_id)) OVER (PARTITION BY c.category_id
ORDER BY COUNT(r.rental_id) DESC) AS running_total
FROM category c
JOIN film_category fc ON c.category_id = fc.category_id
JOIN film f ON fc.film_id = f.film_id
JOIN inventory i ON f.film_id = i.film_id
JOIN rental r ON i.inventory_id = r.inventory_id
GROUP BY c.category_id, c.name, f.title;
```

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

```sql
SELECT f.title, c.name AS category, COUNT(r.rental_id) AS
rental_count,
       AVG(COUNT(r.rental_id)) OVER (PARTITION BY c.category_id)
AS avg_category_rentals
FROM category c
JOIN film_category fc ON c.category_id = fc.category_id
JOIN film f ON fc.film_id = f.film_id
JOIN inventory i ON f.film_id = i.film_id
JOIN rental r ON i.inventory_id = r.inventory_id
GROUP BY f.film_id, f.title, c.category_id, c.name
HAVING COUNT(r.rental_id) < avg_category_rentals;
```

10. **Identify the top 5 months with the highest revenue and display the revenue generated in each month:** `sql SELECT month, monthly_revenue FROM ( SELECT DATE_FORMAT(p.payment_date, '%Y-%m') AS month, SUM(p.amount) AS monthly_revenue, RANK() OVER (ORDER BY SUM(p.amount) DESC) AS revenue_rank FROM payment p GROUP BY DATE_FORMAT(p.payment_date, '%Y-%m') ) AS monthly_revenue_data WHERE revenue_rank <= 5;`

Normalisation & CTE

1. **First Normal Form (1NF):**
   - **Violation**: A table violates 1NF if it has repeating groups or multiple values in a single column.
   - **Example**: Suppose the `customer` table had a column for `phone_numbers` that stored multiple phone numbers in a single cell, separated by commas.
   - **Normalization to 1NF**: To achieve 1NF, separate the `phone_numbers` column into individual rows or create a new table `customer_phone` with columns `customer_id` and `phone_number`, where each phone number occupies its own row.
2. **Second Normal Form (2NF):**
   - **Check for 2NF**: A table is in 2NF if it's in 1NF and all non-key attributes are fully dependent on the primary key. Partial dependencies (where non-key attributes depend on only part of a composite key) indicate a violation.
   - **Example**: Suppose the `rental` table had a composite primary key of `customer_id` and `inventory_id`, with `store_id` dependent only on `inventory_id`.
   - **Normalization to 2NF**: Separate `store_id` into a new table (`inventory_store`) with columns `inventory_id` and `store_id`, ensuring all non-key attributes depend entirely on the composite key.
3. **Third Normal Form (3NF):**
   - **Violation**: A table violates 3NF if it has transitive dependencies (a non-key attribute depends on another non-key attribute).

- **Example**: Suppose the `payment` table includes `customer_id`, `customer_name`, and `amount`. Here, `customer_name` is transitively dependent on `customer_id` rather than directly on the primary key of `payment`.
- **Normalization to 3NF**: Remove `customer_name` from `payment` and create a `customer` table where `customer_id` is the primary key, linking it back to `payment`.

4. **Normalization Process Example**:
   - **Table**: Assume a table `order_details` in an unnormalized form:

```
order_id | product_id | quantity | customer_id | customer_phone_numbers
```

   - **1NF**: Separate `customer_phone_numbers` into a new `customer_phones` table:

```
customer_phones(customer_id, phone_number)
```

   - **2NF**: If `order_id` and `product_id` together form a composite primary key, but `customer_id` is only dependent on `order_id`, split `order_details` into `orders` and `order_products`.

```
orders(order_id, customer_id)
order_products(order_id, product_id, quantity)
```

Common Table Expressions (CTEs):

1. **CTE to Retrieve Distinct Actor Names and Film Counts**:

```
WITH ActorFilmCount AS (
    SELECT a.actor_id, a.first_name, a.last_name,
COUNT(fa.film_id) AS film_count
    FROM actor a
    JOIN film_actor fa ON a.actor_id = fa.actor_id
    GROUP BY a.actor_id, a.first_name, a.last_name
)
SELECT first_name, last_name, film_count
FROM ActorFilmCount;
```

2. **CTE with Joins (Film and Language Information)**:

```
WITH FilmLanguage AS (
    SELECT f.title, l.name AS language_name, f.rental_rate
    FROM film f
    JOIN language l ON f.language_id = l.language_id
)
SELECT *
FROM FilmLanguage;
```

3. **CTE for Total Revenue by Customer**:

```
WITH CustomerRevenue AS (
    SELECT c.customer_id, c.first_name, c.last_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 *
FROM CustomerRevenue;
```

4. **CTE with Window Function for Ranking Films by Rental Duration**:

```
WITH FilmRankings AS (
    SELECT title, rental_duration,
           RANK() OVER (ORDER BY rental_duration DESC) AS
rental_duration_rank
    FROM film
)
SELECT *
FROM FilmRankings;
```

5. **CTE for Customers with More Than Two Rentals**:

```
WITH FrequentCustomers AS (
    SELECT r.customer_id, COUNT(r.rental_id) AS rental_count
    FROM rental r
    GROUP BY r.customer_id
    HAVING COUNT(r.rental_id) > 2
)
SELECT c.*
FROM customer c
JOIN FrequentCustomers fc ON c.customer_id = fc.customer_id;
```

6. **CTE for Monthly Rental Counts**: `sql WITH MonthlyRentals AS ( SELECT DATE_FORMAT(r.rental_date, '%Y-%m') AS rental_month, COUNT(r.rental_id) AS monthly_rental_count FROM rental r GROUP BY rental_month ) SELECT * FROM MonthlyRentals;`

7. **CTE with Self-Join for Actor Pairs in the Same Film**: `sql WITH ActorPairs AS (SELECT fa1.actor_id AS actor1_id, fa2.actor_id AS actor2_id, fa1.film_id FROM film_actor fa1 JOIN film_actor fa2 ON fa1.film_id = fa2.film_id AND fa1.actor_id < fa2.actor_id) SELECT a1.first_name AS actor1_first_name, a1.last_name AS actor1_last_name, a2.first_name AS actor2_first_name, a2.last_name AS actor2_last_name, ap.film_id FROM ActorPairs ap JOIN`

```
actor a1 ON ap.actor1_id = a1.actor_id     JOIN actor a2 ON
ap.actor2_id = a2.actor_id;
```

8. **Recursive CTE to Find Employees Reporting to a Specific Manager**:

```
WITH RECURSIVE EmployeeHierarchy AS (
    SELECT staff_id, first_name, last_name, reports_to
    FROM staff
    WHERE reports_to = [manager_id]  -- Replace [manager_id] with
the specific manager's ID

    UNION ALL

    SELECT s.staff_id, s.first_name, s.last_name, s.reports_to
    FROM staff s
    JOIN EmployeeHierarchy eh ON s.reports_to = eh.staff_id
)
SELECT *
FROM EmployeeHierarchy;
```