Database

Q1. Create a table called employees with the following structure

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY NOT NULL,
    emp_name TEXT NOT NULL,
    age INT CHECK (age >= 18),
    email TEXT UNIQUE,
    salary DECIMAL(10,2) DEFAULT 30000
);
```

Q2. Explain the purpose of constraints and how they help maintain data integrity in a database.

Ans: Constraints enforce rules on data in a table. They prevent invalid data and maintain consistency.

- **NOT NULL** → ensures no missing values.
- **UNIQUE** → ensures values are not duplicated.
- **PRIMARY KEY** → uniquely identifies rows.
- **CHECK** → ensures values meet conditions.
- **FOREIGN KEY** → maintains referential integrity.

Q3. Why would you apply the NOT NULL constraint to a column? Can a primary key contain NULL values?

Ans:

- **NOT NULL** → ensures important fields (like name, ID) always have data.
- **Primary key** cannot have NULL because it must uniquely identify every row.

Q4. Explain the steps and SQL commands used to add or remove constraints on an existing table.

Ans:

- **Add constraint:**

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

- **Remove constraint:**

```
ALTER TABLE employees
DROP CONSTRAINT chk_salary;
```

Q5. Explain the consequences of attempting to insert, update, or delete data in a way that violates constraints.
Ans: Operation fails, error is raised, data integrity stays protected.
Example:

```
INSERT INTO employees (emp_id, emp_name, age, email)
VALUES (1, 'John', 15, 'john@mail.com');
```

Error → **"ERROR: new row for relation violates check constraint 'employees_age_check'"**

Q6. Modify products table to add constraints

```
ALTER TABLE products
ADD CONSTRAINT pk_product PRIMARY KEY (product_id);

ALTER TABLE products
ALTER COLUMN price SET DEFAULT 50.00;
```

Q7. Fetch student_name and class_name for each student (INNER JOIN)

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

Q8. Show all order_id, customer_name, and product_name ensuring all products are listed

```
SELECT o.order_id, c.customer_name, p.product_name
FROM products p
LEFT JOIN orders o ON p.product_id = o.product_id
LEFT JOIN customers c ON o.customer_id = c.customer_id;
```

Q9. Find total sales amount for each product

```
SELECT p.product_name, SUM(o.quantity * p.price) AS total_sales
FROM orders o
INNER JOIN products p
ON o.product_id = p.product_id
GROUP BY p.product_name;
```

Q10. Display order_id, customer_name, and quantity of products ordered

```
SELECT o.order_id, c.customer_name, o.quantity
FROM orders o
INNER JOIN customers c ON o.customer_id = c.customer_id
INNER JOIN products p ON o.product_id = p.product_id;
```

Functions

Q1. Retrieve the total number of rentals made in the Sakila database.

```
SELECT COUNT(*) FROM rental;
```

Q2. Find the average rental duration (in days) of movies rented.

```
SELECT AVG(rental_duration) FROM film;
```

String Functions

Q3. Display the first name and last name of customers in uppercase.

```
SELECT UPPER(first_name), UPPER(last_name) FROM customer;
```

Q4. Extract the month from the rental date and display it with rental ID.

```
SELECT rental_id, MONTH(rental_date) FROM rental;
```

Group By

Q5. Retrieve the count of rentals for each customer.

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

Q6. Find the total revenue generated by each store.

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

Q7. Determine the total number of rentals for each category of movies.

```
SELECT c.name AS category, COUNT(*) AS rental_count
FROM rental r
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN film f ON i.film_id = f.film_id
JOIN film_category fc ON f.film_id = fc.film_id
JOIN category c ON fc.category_id = c.category_id
GROUP BY c.name;
```

Q8. Find the average rental rate of movies in each language.

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

Joins

Q9. Display the title of the movie, customer's first name, and last name who rented it.

```
SELECT f.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;
```

Q10. Retrieve the names of all actors who appeared in the film "Gone with the Wind."

```
SELECT a.first_name, a.last_name
FROM actor a
JOIN film_actor fa ON a.actor_id = fa.actor_id
JOIN film f ON fa.film_id = f.film_id
WHERE f.title = 'Gone with the Wind';
```

Q11. Retrieve the customer names along with the total amount 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.first_name, c.last_name;
```

Q12. List the titles of movies rented by each customer in London.

```
SELECT c.first_name, c.last_name, f.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.first_name, c.last_name, f.title;
```

Advanced Joins and Group By

Q13. Display the top 5 rented movies with number of times rented.

```
SELECT f.title, COUNT(*) AS times_rented
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.title
ORDER BY times_rented DESC
LIMIT 5;
```

Q14. Determine the customers who have rented movies from both stores (1 and 2).

```
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
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING COUNT(DISTINCT i.store_id) = 2;
```

Windows Functions

Q1. Rank the customers based on total amount spent.

```
SELECT c.customer_id, c.first_name, c.last_name,
       RANK() OVER (ORDER BY SUM(p.amount) DESC) AS rank_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;
```

Q2. Calculate cumulative revenue generated by each film over time.

```
SELECT f.title, p.payment_date,
       SUM(p.amount) OVER (PARTITION BY f.title ORDER BY
p.payment_date) AS cumulative_revenue
```

```
FROM film f
JOIN inventory i ON f.film_id = i.film_id
JOIN rental r ON i.inventory_id = r.inventory_id
JOIN payment p ON r.rental_id = p.rental_id;
```

Q3. Determine average rental duration for each film, grouped by similar lengths.

```
SELECT length, AVG(rental_duration)
FROM film
GROUP BY length;
```

Q4. Identify the top 3 films in each category based on rental counts.

```
SELECT c.name AS category, f.title,
       RANK() OVER (PARTITION BY c.name ORDER BY COUNT(r.rental_id)
DESC) AS rank_rented
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.name, f.title
HAVING RANK() <= 3;
```

Q5. Calculate the difference in rental counts between each customer's rentals and average rentals.

```
SELECT c.customer_id, COUNT(r.rental_id) AS total_rentals,
       COUNT(r.rental_id) - AVG(COUNT(r.rental_id)) OVER() AS
diff_from_avg
FROM customer c
JOIN rental r ON c.customer_id = r.customer_id
GROUP BY c.customer_id;
```

Q6. Find monthly revenue trend.

```
SELECT MONTH(payment_date) AS month, SUM(amount) AS revenue
FROM payment
```

```
GROUP BY MONTH(payment_date)
ORDER BY month;
```

Q7. Identify customers whose total spending falls in top 20%.

```
SELECT customer_id, SUM(amount) AS total_spent
FROM payment
GROUP BY customer_id
QUALIFY NTILE(5) OVER (ORDER BY SUM(amount) DESC) = 1;
```

Q8. Running total of rentals per category ordered by rental count.

```
SELECT c.name, COUNT(r.rental_id) AS rental_count,
       SUM(COUNT(r.rental_id)) OVER (ORDER BY COUNT(r.rental_id)) 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.name;
```

Q9. Find films rented less than average rental count in their category.

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

Q10. Identify top 5 months with highest revenue.

```
SELECT MONTH(payment_date) AS month, SUM(amount) AS revenue
FROM payment
GROUP BY MONTH(payment_date)
```

```
ORDER BY revenue DESC
LIMIT 5;
```

SQL Commands

Q1. Identify the primary keys and foreign keys in Maven Movies DB. Discuss the differences.

- Primary Key (PK): uniquely identifies a record (e.g., actor_id in actor).
- Foreign Key (FK): references PK in another table (e.g., film_id in inventory).
- Difference: PK = uniqueness, FK = referential integrity.

Q2. List all details of actors.

```
SELECT * FROM actor;
```

Q3. List all customer information from DB.

```
SELECT * FROM customer;
```

Q4. List different countries.

```
SELECT DISTINCT country FROM country;
```

Q5. Display all active customers.

```
SELECT * FROM customer WHERE active = 1;
```

Q6. List of all rental IDs for customer with ID 1.

```
SELECT rental_id FROM rental WHERE customer_id = 1;
```

Q7. Display all the films whose rental duration is greater than 5.

```
SELECT * FROM film WHERE rental_duration > 5;
```

Q8. List the total number of films whose replacement cost is greater than $15 and less than $20.

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

Q9. Display the count of unique first names of actors.

```
SELECT COUNT(DISTINCT first_name) FROM actor;
```

Q10. Display the first 10 records from the customer table.

```
SELECT * FROM customer LIMIT 10;
```

Q11. Display the first 3 records from the customer table whose first name starts with ‘b’.

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

Q12. Display the names of the first 5 movies which are rated as ‘G’.

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

Q13. Find all customers whose first name starts with "a".

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

Q14. Find all customers whose first name ends with "a".

```
SELECT * FROM customer WHERE first_name LIKE '%a';
```

Q15. Display the list of first 4 cities which start and end with ‘a’.

```
SELECT city FROM city
WHERE city LIKE 'A%a'
LIMIT 4;
```

Q16. Find all customers whose first name has "NI" in any position.

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

Q17. Find all customers whose first name has "r" in the second position.

```
SELECT * FROM customer WHERE first_name LIKE '_r%';
```

Q18. Find all customers whose first name starts with "a" and are at least 5 characters in length.

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

Q19. Find all customers whose first name starts with "a" and ends with "o".

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

Q20. Get the films with PG and PG-13 rating using IN operator.

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

Q21. Get the films with length between 50 to 100 using BETWEEN operator.

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

Q22. Get the top 50 actors using LIMIT operator.

```
SELECT * FROM actor LIMIT 50;
```

Q23. Get the distinct film IDs from inventory table.

```
SELECT DISTINCT film_id FROM inventory;
```