1. Create a table called employees with constraints

Question:
Create a table employees with:

- emp_id INT, NOT NULL, PRIMARY KEY
- emp_name TEXT, NOT NULL
- age INT, with CHECK age ≥ 18
- email TEXT, UNIQUE
- salary DECIMAL, DEFAULT 30000

Answer:

sql

CopyEdit

```
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 DEFAULT 30000
);
```

---

2. Explain the purpose of constraints

Answer:

Constraints ensure **data integrity** by placing rules on table columns. Common types include:

- **PRIMARY KEY**: Ensures uniqueness and not null.
- **FOREIGN KEY**: Enforces referential integrity between tables.
- **UNIQUE**: Ensures no duplicate values.
- **NOT NULL**: Prevents null entries.
- **CHECK**: Validates conditions on data (e.g., age ≥ 18).
- **DEFAULT**: Sets a default value if none is provided.

They help prevent bad data entry, like negative ages or duplicate IDs.

---

3. Why use NOT NULL? Can a primary key be NULL?

Answer:

- NOT NULL ensures that a column **must have a value**, preventing incomplete data.
- A **primary key CANNOT contain NULL**, because it must uniquely identify each row — and NULL is unknown, so it breaks uniqueness.

---

4. Add/Remove Constraints on Existing Table

Answer:

Add Constraint Example (Adding a CHECK to age column):

sql

CopyEdit

```
ALTER TABLE employees

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

Remove Constraint Example (Dropping that CHECK):

sql

CopyEdit

```
ALTER TABLE employees

DROP CONSTRAINT chk_age;
```

(Note: Constraint names may be auto-generated if not explicitly named.)

---

5. Consequences of Violating Constraints

Answer:

If a constraint is violated, the database throws an error and blocks the operation.

Example:
Inserting a row with duplicate email:

sql

CopyEdit

```
INSERT INTO employees (emp_id, emp_name, age, email)

VALUES (1, 'John Doe', 25, 'john@example.com');
```

If 'john@example.com' already exists, you'll get:

sql

CopyEdit

```
ERROR: duplicate key value violates unique constraint "employees_email_key"
```

---

6. Add constraints to existing products table

Given:

sql

CopyEdit

```
CREATE TABLE products (

   product_id INT,

   product_name VARCHAR(50),

   price DECIMAL(10, 2)

);
```

Update: Make product_id a primary key, price default to 50.00

Answer:

sql

CopyEdit

```
ALTER TABLE products

ADD CONSTRAINT pk_product PRIMARY KEY (product_id);


ALTER TABLE products

ALTER COLUMN price SET DEFAULT 50.00;
```

✅ Question 7

Question:
You have two tables:

- students(student_id, student_name, class_id)
- classes(class_id, class_name)

Write a query to fetch the student_name and class_name using an **INNER JOIN**.

Answer:

sql

CopyEdit

```
SELECT s.student_name, c.class_name

FROM students s
```

```
INNER JOIN classes c ON s.class_id = c.class_id;
```

✅ Question 8

Question:
You have three tables:

- orders(order_id, customer_id, product_id)
- customers(customer_id, customer_name)
- products(product_id, product_name)

Show all order_id, customer_name, and product_name, ensuring **all products** are listed, even if **not ordered**.
➡️ Use **LEFT JOIN** and **INNER JOIN**.

Answer:

sql

CopyEdit

```
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;
```

✅ Question 9

Question:
Given the same tables (orders, products) plus quantity and price assumed:
Find the **total sales amount** per product using **INNER JOIN** and SUM().

➡️ Total Sales = quantity * price

Assumed columns:

- orders(order_id, product_id, quantity)
- products(product_id, product_name, price)

Answer:

sql

CopyEdit

```
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;
```

✅ Question 10

Question:
You have three tables:

- orders(order_id, customer_id)
- order_items(order_id, product_id, quantity)
- customers(customer_id, customer_name)

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

Answer:

sql

CopyEdit

```
SELECT o.order_id, c.customer_name, oi.quantity

FROM orders o

INNER JOIN order_items oi ON o.order_id = oi.order_id

INNER JOIN customers c ON o.customer_id = c.customer_id;
```

("1. Identify the primary keys and foreign keys in Maven Movies DB. Discuss the differences",

"*Answer:\n\n- **Primary Keys* uniquely identify a record in a table (e.g., actor_id in actor, film_id in film).\n- *Foreign Keys* establish relationships between tables (e.g., customer_id in rental refers to customer.customer_id).\n\n*Differences:*\n- A primary key ensures uniqueness.\n- A foreign key is used to enforce referential integrity."),

("2. List all details of actors", "sql\nSELECT * FROM actor;\n"),

("3. List all customer information from DB.", "sql\nSELECT * FROM customer;\n"),

("4. List different countries.", "sql\nSELECT DISTINCT country FROM country;\n"),

("5. Display all active customers.", "sql\nSELECT * FROM customer WHERE active = 1;\n"),

("6. List of all rental IDs for customer with ID 1.", "sql\nSELECT rental_id FROM rental WHERE customer_id = 1;\n"),

("7. Display all the films whose rental duration is greater than 5.", "sql\nSELECT * FROM film WHERE rental_duration > 5;\n"),

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

"sql\nSELECT COUNT(*) FROM film WHERE replacement_cost > 15 AND replacement_cost < 20;\n"),

("9. Display the count of unique first names of actors.", "sql\nSELECT COUNT(DISTINCT first_name) FROM actor;\n"),

("10. Display the first 10 records from the customer table.", "sql\nSELECT * FROM customer LIMIT 10;\n"),

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

"sql\nSELECT * FROM customer WHERE first_name LIKE 'b%' LIMIT 3;\n"),

("12. Display the names of the first 5 movies which are rated as ‘G’.",

"sql\nSELECT title FROM film WHERE rating = 'G' LIMIT 5;\n"),

("13. Find all customers whose first name starts with \"a\".", "sql\nSELECT * FROM customer WHERE first_name LIKE 'a%';\n"),

("14. Find all customers whose first name ends with \"a\".", "sql\nSELECT * FROM customer WHERE first_name LIKE '%a';\n"),

("15. Display the list of first 4 cities which start and end with ‘a’.",

"sql\nSELECT city FROM city WHERE city LIKE 'a%a' LIMIT 4;\n"),

("16. Find all customers whose first name has \"NI\" in any position.",

"sql\nSELECT * FROM customer WHERE first_name ILIKE '%ni%';\n"),

("17. Find all customers whose first name has \"r\" in the second position.",

"sql\nSELECT * FROM customer WHERE first_name LIKE '_r%';\n"),

```
 ("18. Find all customers whose first name starts with \"a\" and are at least 5 characters in length.",
  "sql\nSELECT * FROM customer WHERE first_name LIKE 'a%' AND LENGTH(first_name) >= 5;\n"),
 ("19. Find all customers whose first name starts with \"a\" and ends with \"o\".",
  "sql\nSELECT * FROM customer WHERE first_name LIKE 'a%o';\n"),
 ("20. Get the films with PG and PG-13 rating using IN operator.",
  "sql\nSELECT * FROM film WHERE rating IN ('PG', 'PG-13');\n"),
 ("21. Get the films with length between 50 to 100 using BETWEEN operator.",
  "sql\nSELECT * FROM film WHERE length BETWEEN 50 AND 100;\n"),
 ("22. Get the top 50 actors using LIMIT operator.", "sql\nSELECT * FROM actor LIMIT 50;\n"),
 ("23. Get the distinct film ids from inventory table.", "sql\nSELECT DISTINCT film_id FROM inventory;\n")
```

SQL Questions and Answers

Functions

1. Total number of rentals

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

2. Average rental duration (in days)

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

3. Display first and last names of customers in uppercase

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

4. Extract month from rental date with rental ID

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

5. Count of rentals for each customer

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

6. Total revenue by each store

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

7. Total number of rentals per movie category

```
SELECT fc.category_id, c.name AS category_name, COUNT(r.rental_id) AS
rental_count
FROM film_category fc
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
JOIN category c ON fc.category_id = c.category_id
GROUP BY fc.category_id, c.name;
```

8. Average rental rate by 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

9. Movie title, customer’s first and last name who rented it

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

10. Actors 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';
```

11. Customer names and total amount they’ve spent

```
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;
```

12. 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 ct ON a.city_id = ct.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 ct.city = 'London';
```

13. Top 5 rented movies with rental counts

```
SELECT f.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.title
ORDER BY rental_count DESC
LIMIT 5;
```

14. Customers who rented from both store 1 and store 2

```
SELECT customer_id
FROM rental r
JOIN inventory i ON r.inventory_id = i.inventory_id
GROUP BY customer_id
HAVING COUNT(DISTINCT i.store_id) = 2;
```

Window Functions

1. Rank customers by total rental spending

```
SELECT customer_id,
       SUM(amount) AS total_spent,
       RANK() OVER (ORDER BY SUM(amount) DESC) AS rank
FROM payment
GROUP BY customer_id;
```

2. Cumulative revenue by film over time

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

3. Avg rental duration by similar film lengths

```
SELECT film_id, title, length,
       AVG(rental_duration) OVER (PARTITION BY length) AS
```

```
avg_rental_duration
FROM film;
```

4. Top 3 films in each category by rental counts

```
WITH film_rentals AS (
    SELECT c.name AS category, f.title, COUNT(r.rental_id) AS rental_count
    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
)
SELECT *,
       RANK() OVER (PARTITION BY category ORDER BY rental_count DESC) AS
rank
FROM film_rentals
WHERE rank <= 3;
```

5. Difference between customer's rentals and average

```
WITH customer_rentals AS (
    SELECT customer_id, COUNT(*) AS total_rentals
    FROM rental
    GROUP BY customer_id
),
avg_rentals AS (
    SELECT AVG(total_rentals) AS avg_rental_count FROM customer_rentals
)
SELECT cr.customer_id, cr.total_rentals,
       (cr.total_rentals - ar.avg_rental_count) AS rental_difference
FROM customer_rentals cr, avg_rentals ar;
```

6. Monthly revenue trend

```
SELECT DATE_TRUNC('month', payment_date) AS month,
       SUM(amount) AS monthly_revenue
FROM payment
GROUP BY month
ORDER BY month;
```

7. Customers in top 20% by spending

```
WITH customer_total AS (
    SELECT customer_id, SUM(amount) AS total_spent
    FROM payment
    GROUP BY customer_id
),
ranked_customers AS (
    SELECT *,
           NTILE(5) OVER (ORDER BY total_spent DESC) AS percentile
    FROM customer_total
)
SELECT customer_id, total_spent
FROM ranked_customers
WHERE percentile = 1;
```

8. Running total of rentals per category

```
WITH category_rentals AS (
    SELECT c.name AS category, COUNT(r.rental_id) AS rental_count
    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
)
SELECT category, rental_count,
       SUM(rental_count) OVER (ORDER BY rental_count DESC) AS running_total
FROM category_rentals;
```

9. Films rented less than average in their category

```
WITH rentals_per_film AS (
    SELECT f.film_id, c.name AS category, 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.film_id, c.name
),
category_avg AS (
    SELECT category, AVG(rental_count) AS avg_count
    FROM rentals_per_film
    GROUP BY category
)
SELECT rpf.*
FROM rentals_per_film rpf
JOIN category_avg ca ON rpf.category = ca.category
WHERE rpf.rental_count < ca.avg_count;
```

10. Top 5 months with highest revenue

```
SELECT TO_CHAR(payment_date, 'YYYY-MM') AS month,
       SUM(amount) AS revenue
FROM payment
GROUP BY month
ORDER BY revenue DESC
LIMIT 5;
```

Normalization & CTEs

1. What is normalization in SQL? Why is it important?

```
**Answer:**

**Normalization** is the process of organizing data in a database to reduce
**redundancy** and improve **data integrity**.

**Benefits:**
- Eliminates duplicate data
```

- Ensures data consistency
- Makes updates, inserts, and deletes more efficient

2. Different Normal Forms (1NF to 5NF)

Answer:

Normal Form	Rule
1NF	Atomic values, no repeating groups
2NF	1NF + No partial dependency
3NF	2NF + No transitive dependency
BCNF	Every determinant is a candidate key
4NF	No multivalued dependencies
5NF	Eliminates join dependency

3. Create CTE that returns top 5 customers by revenue

```
WITH customer_spending AS (
    SELECT customer_id, SUM(amount) AS total_spent
    FROM payment
    GROUP BY customer_id
)
SELECT * FROM customer_spending
ORDER BY total_spent DESC
LIMIT 5;
```

4. Use CTE to find customers who rented more than average

```
WITH rental_counts AS (
    SELECT customer_id, COUNT(*) AS rental_count
    FROM rental
    GROUP BY customer_id
),
avg_rentals AS (
    SELECT AVG(rental_count) AS avg_rentals FROM rental_counts
)
SELECT rc.*
FROM rental_counts rc, avg_rentals ar
WHERE rc.rental_count > ar.avg_rentals;
```

5. Use multiple CTEs to calculate revenue per customer and percent of total

```
WITH customer_revenue AS (
    SELECT customer_id, SUM(amount) AS total_spent
    FROM payment
    GROUP BY customer_id
),
total_revenue AS (
    SELECT SUM(total_spent) AS grand_total FROM customer_revenue
)
SELECT cr.customer_id, cr.total_spent,
       ROUND((cr.total_spent / tr.grand_total) * 100, 2) AS
percent_of_total
FROM customer_revenue cr, total_revenue tr;
```

6. Recursive CTE example to calculate factorial

```
WITH RECURSIVE factorial(n, fact) AS (
    SELECT 1, 1
    UNION ALL
    SELECT n + 1, (n + 1) * fact
    FROM factorial
    WHERE n < 5
)
SELECT * FROM factorial;
```

7. Recursive CTE to return numbers from 1 to 100

```
WITH RECURSIVE numbers AS (
    SELECT 1 AS num
    UNION ALL
    SELECT num + 1
    FROM numbers
    WHERE num < 100
)
SELECT * FROM numbers;
```

8. Return category, film, and total rentals using CTE

```
WITH film_rentals AS (
    SELECT c.name AS category, f.title, COUNT(r.rental_id) AS rental_count
    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
)
SELECT * FROM film_rentals;
```

9. Use CTE to return customers who rented "Family" films more than 3 times

```
WITH family_rentals AS (
    SELECT c.customer_id, COUNT(*) AS rental_count
    FROM customer c
    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
    JOIN film_category fc ON f.film_id = fc.film_id
    JOIN category cat ON fc.category_id = cat.category_id
    WHERE cat.name = 'Family'
    GROUP BY c.customer_id
)
SELECT customer_id
FROM family_rentals
WHERE rental_count > 3;
```

10. Find top 3 actors by number of films using CTE

```
WITH actor_film_count 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 *
FROM actor_film_count
ORDER BY film_count DESC
LIMIT 3;
```