```
#Sql basics

-- 1. Create a table called employees with the following structure?
-- : emp_id (integer, should not be NULL and should be a primary key)Q
-- : emp_name (text, should not be NULL)Q
-- : age (integer, should have a check constraint to ensure the age is at least 18)Q
-- : email (text, should be unique for each employee)Q
-- : salary (decimal, with a default value of 30,000).

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

CREATE TABLE employees (
emp_id INTEGER PRIMARY KEY NOT NULL,
emp_name TEXT NOT NULL,
age INTEGER CHECK(age >= 18),
email VARCHAR(255) UNIQUE,
salary DECIMAL DEFAULT 30000.00
);


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

-- Purpose of constraints

-- - Data accuracy
-- - Data consistency
-- - Data security

-- Common constraints:

-- 1. Primary Key (PK)
-- 2. Foreign Key (FK)
-- 3. Unique (UQ)
-- 4. Not Null (NN)
-- 5. Check (CK)
-- 6. Default (DF)

-- Example:

-- In employess example table

-- - PK: EmployeeID
-- - FK: DepartmentID
-- - UQ: Name
-- - NN: Age
-- - CK: Age > 18
-- - DF: DepartmentID = 1

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

-- NOT NULL constraint:

-- - Prevents NULL values in a column
-- - Makes the column value mandatory

-- Primary Key (PK) and NULL values:

-- - PK cannot have NULL values
-- - PK automatically implies NOT NULL constraint

#4.Explain the steps and SQL commands used to add or remove constraints on an existing table. Provide an example for b
#To add a constraint to an existing table, use the ALTER TABLE command followed by the ADD CONSTRAINT clause.

#Example:
CREATE TABLE employees (
emp_id INT,
emp_name VARCHAR(50)
```

```
);

-- Primary key constraint add karne ke liye
ALTER TABLE employees
ADD CONSTRAINT pk_emp_id PRIMARY KEY (emp_id);

-- Constraint drop karne ke liye
ALTER TABLE employees
DROP CONSTRAINT pk_emp_id;

#5. Explain the consequences of attempting to insert, update, or delete data in a way that violates constraints. Provi

#Suppose we have a table with a primary key constraint:


CREATE TABLE employees (
emp_id INT PRIMARY KEY,
emp_name VARCHAR(50)
);

INSERT INTO employees (emp_id, emp_name) VALUES (1, 'John Doe');
INSERT INTO employees (emp_id, emp_name) VALUES (1, 'Jane Doe');


#Error Message:


#Error Code: 1062. Duplicate entry '1' for key 'PRIMARY'

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

-- CREATE TABLE products (

--     product_id INT,

--     product_name VARCHAR(50),

--     price DECIMAL(10, 2));


#To add constraints to the products table:


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

ALTER TABLE products
ADD CONSTRAINT pk_product_id PRIMARY KEY (product_id);

ALTER TABLE products
ALTER COLUMN price SET DEFAULT 50.00;


-- 7. You have two tables:
-- Write a query to fetch the student_name and class_name for each student using an INNER JOIN.
CREATE TABLE Students (
student_id INT,
student_name VARCHAR(255),
class_id INT
);

CREATE TABLE Classes (
class_id INT,
class_name VARCHAR(255)
```

```
);

INSERT INTO Students (student_id, student_name, class_id)
VALUES
(1, 'Alice', 101),
(2, 'Bob', 102),
(3, 'Charlie', 101);

INSERT INTO Classes (class_id, class_name)
VALUES
(101, 'Math'),
(102, 'Science'),
(103, 'History');

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

#8. Consider the following three tables:
-- 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)5
CREATE TABLE Orders (
order_id INT,
order_date DATE,
customer_id INT
);

CREATE TABLE Customers (
customer_id INT,
customer_name VARCHAR(255)
);

CREATE TABLE Products (
product_id INT,
product_name VARCHAR(255)
);

-- Add order_id column to Products table
ALTER TABLE Products
ADD COLUMN order_id INT;

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

INSERT INTO Customers (customer_id, customer_name)
VALUES (101, 'Alice'), (102, 'Bob');

INSERT INTO Products (product_id, product_name, order_id)
VALUES (1, 'Laptop', 1), (2, 'Phone', NULL);

-- Query to retrieve all order_id, customer_name, and product_name
SELECT O.order_id, C.customer_name, P.product_name
FROM Products P
LEFT JOIN Orders O ON P.order_id = O.order_id
LEFT JOIN Customers C ON O.customer_id = C.customer_id;

-- #9.Given the following tables:
-- Write a query to find the total sales amount for each product using an INNER JOIN and the SUM() function
CREATE TABLE Sales (
sale_id INT,
product_id INT,
amount INT
);

CREATE TABLE Products (
product_id INT,
product_name VARCHAR(20)
);

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

INSERT INTO Products (product_id, product_name) VALUES
(101, 'Laptop'),
(102, 'Phone');
SELECT P.product_name, SUM(S.amount) AS total_sales
FROM Sales S
JOIN Products P ON S.product_id = P.product_id
GROUP BY P.product_name;
#10. You are given three tables
-- 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.
CREATE TABLE Orders (
order_id INT,
order_date DATE,
customer_id INT
);

INSERT INTO Orders (order_id, order_date, customer_id)
VALUES
(1, '2024-01-02', 1),
(2, '2024-01-05', 2);

CREATE TABLE Customers (
customer_id INT,
customer_name VARCHAR(255)
);

INSERT INTO Customers (customer_id, customer_name)
VALUES
(1, 'Alice'),
(2, 'Bob');

CREATE TABLE Order_Details (
order_id INT,
product_id INT,
quantity INT
);

INSERT INTO Order_Details (order_id, product_id, quantity)
VALUES
(1, 101, 2),
(1, 102, 1),
(2, 101, 3);
SELECT O.order_id, C.customer_name, SUM(OD.quantity) AS total_quantity
FROM Orders O
INNER JOIN Customers C ON O.customer_id = C.customer_id
INNER JOIN Order_Details OD ON O.order_id = OD.order_id
GROUP BY O.order_id, C.customer_name;


##SQL Commands
use sakila;
-- 1-Identify the primary keys and foreign keys in maven movies db. Discuss the differences
-- Primary keys:

-- - actor_id (actors table)
-- - customer_id (customers table)
-- - film_id (films table)
```

```sql
-- - inventory_id (inventory table)
-- - order_id (orders table)
-- - payment_id (payments table)
-- - rental_id (rentals table)
-- - staff_id (staff table)
-- - store_id (stores table)

-- Foreign keys:

-- - customer_id (orders table) references customers table
-- - inventory_id (rentals table) references inventory table
-- - film_id (inventory table) references films table
-- - staff_id (payments table) references staff table
-- - rental_id (payments table) references rentals table

-- Primary keys uniquely identify each record in a table, while foreign keys link related data between tables.
-- 2- List all details of actors
select * from actor;
-- 3 -List all customer information from DB.
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 of all rental IDs for customer with ID 1.
SELECT rental_id FROM rental WHERE customer_id = 1;
-- 7 - Display all the films whose rental duration is greater than 5 .
SELECT * FROM film WHERE rental_duration > 5;
-- 8 - List the total number of films whose replacement cost is greater than $15 and less than $20.
SELECT COUNT(*) FROM film WHERE replacement_cost BETWEEN 15 AND 20;
-- 9 - Display the count of unique first names of actors.
SELECT COUNT(DISTINCT first_name) 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 whose first name starts with b.
SELECT * FROM customer WHERE first_name LIKE 'b%' LIMIT 3;
-- 12 -Display the names of the first 5 movies which are 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 first 4 cities which start and end with a .
SELECT city FROM city WHERE city LIKE 'a%a' LIMIT 4;
-- 16- Find all customers whose first name have "NI" in any position.
SELECT * FROM customer WHERE first_name LIKE '%NI%';
-- 17- Find all customers whose first name have "r" in the second position .
SELECT * FROM customer WHERE first_name LIKE '_r%';
-- 18 - 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_____%';
-- 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 the films with pg and pg-13 rating using IN operator.
SELECT * FROM film WHERE rating IN ('PG', 'PG-13');
-- 21 - Get the films with length between 50 to 100 using between operator.
SELECT * FROM film WHERE length BETWEEN 50 AND 100;
-- 22 - Get the top 50 actors using limit operator.
SELECT * FROM actor LIMIT 50;
-- 23 - Get the distinct film ids from inventory table
SELECT DISTINCT film_id FROM inventory;

##Functions
-- Question 1: Retrieve the total number of rentals made in the Sakila database.

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

```sql
-- Question 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;


-- Question 3: 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;

-- Question 4: 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;

-- Question 5: 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;

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

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

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

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

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

SELECT 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.name;

#Joins
-- Question 9: 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;

-- Question 10: Retrieve the names of all actors who have 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';


-- Question 11: 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_amount
FROM customer c
JOIN rental r ON c.customer_id = r.customer_id
JOIN payment p ON r.rental_id = p.rental_id
GROUP BY c.first_name, c.last_name;


-- Question 12: List the titles of movies rented by each customer in a particular city (e.g., 'London').

SELECT f.title,
c.first_name,
c.last_name,
ci.city
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
JOIN address a ON c.address_id = a.address_id
JOIN city ci ON a.city_id = ci.city_id
WHERE ci.city = 'London'
GROUP BY f.title, c.first_name, c.last_name, ci.city;

##Advanced Joins and Group By


-- Question 13: Display the top 5 rented movies along with the number of times they've been rented.

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;


-- Question 14: Determine the customers who have rented movies from both stores (store ID 1 and store ID 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
JOIN store s ON i.store_id = s.store_id
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING COUNT(DISTINCT s.store_id) = 2;


##Windows Function:

-- Question 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_amount
FROM customer c
```

```sql
JOIN rental r ON c.customer_id = r.customer_id
JOIN payment p ON r.rental_id = p.rental_id
GROUP BY c.customer_id, c.first_name, c.last_name
ORDER BY total_amount DESC;

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

SELECT f.film_id,
f.title,
SUM(p.amount) 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
GROUP BY f.film_id, f.title
ORDER BY cumulative_revenue DESC;

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

SELECT f.film_id,
f.title,
AVG(TIMESTAMPDIFF(DAY, r.rental_date, r.return_date)) AS average_rental_duration
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 average_rental_duration DESC;



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

SELECT c.name AS category_name,
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 c.name, f.title
ORDER BY c.name, rental_count DESC;

-- Question 5: Calculate the difference in rental counts between each customer's total rentals and the average rentals

SELECT c.customer_id,
c.first_name,
c.last_name,
COUNT(r.rental_id) AS total_rentals,
(COUNT(r.rental_id) - (SELECT AVG(total_rentals) FROM (SELECT customer_id, COUNT(rental_id) AS total_rentals FROM rent
FROM customer c
JOIN rental r ON c.customer_id = r.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name;



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

SELECT
EXTRACT(YEAR FROM p.payment_date) AS year,
EXTRACT(MONTH FROM p.payment_date) AS month,
SUM(p.amount) AS revenue
FROM
payment p
GROUP BY
EXTRACT(YEAR FROM p.payment_date),
EXTRACT(MONTH FROM p.payment_date)
```

```
ORDER BY
year, month;


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

SELECT c.customer_id,
c.first_name,
c.last_name,
SUM(p.amount) AS total_spending
FROM customer c
JOIN rental r ON c.customer_id = r.customer_id
JOIN payment p ON r.rental_id = p.rental_id
GROUP BY c.customer_id, c.first_name, c.last_name
ORDER BY total_spending DESC
LIMIT 59;


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

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

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

SELECT
f.film_id,
f.title,
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, f.title, c.name
HAVING
COUNT(r.rental_id) < (SELECT AVG(rental_count) FROM (SELECT COUNT(r.rental_id) AS rental_count FROM rental r GROUP BY


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

SELECT EXTRACT(YEAR FROM p.payment_date) AS year,
EXTRACT(MONTH FROM p.payment_date) AS month,
SUM(p.amount) AS revenue
FROM payment p
GROUP BY EXTRACT(YEAR FROM p.payment_date), EXTRACT(MONTH FROM p.payment_date)
ORDER BY revenue DESC
LIMIT 5;


##Normalisation & CTE
```

```
-- 1. First Normal Form (1NF)

-- a. Identify a table in the Sakila database that violates 1NF. Explain how you would normalize it to achieve 1NF.

-- As mentioned before, the Sakila database is generally well-designed. However, to illustrate 1NF, let's create a hyp
-- Normalization to 1NF:
-- Create a new table called film_actor_list.
-- This table would have columns: film_id (foreign key referencing film), actor_name.
-- For each film, insert a separate row for each actor.
-- Example:
-- Instead of film_id: 1, actors: "Penelope Guiness, Nick Wahlberg, Ed Chase",
-- You would have:
-- film_id: 1, actor_name: "Penelope Guiness"
-- film_id: 1, actor_name: "Nick Wahlberg"
-- film_id: 1, actor_name: "Ed Chase"
-- By doing this, each cell contains a single, atomic value, satisfying 1NF.

-##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

-- Let's examine the film_category table.
-- 2NF Check:
-- Primary key: (film_id, category_id) (composite key).
-- Non-key attributes: There are no other attributes.
-- Since there are no non key attributes, there can be no partial dependancies. Therefore the table is in 2NF.
-- If we were to hypothetically add a non key attribute to the film_category table, called category_name, then that ta
-- To fix this, the category_name column would be removed from the film_category table, and the category table would b


-- 3. Third Normal Form (3NF)

-- a. Identify a table in Sakila that violates 3NF. Describe the transitive dependencies present and outline the steps

-- As discussed previously, the address table is a good example to illustrate 3NF.
-- Transitive Dependencies:
-- address has city_id.
-- city has country_id.
-- Therefore, address transitively depends on country through city.
-- Normalization to 3NF:
-- The address table is already normalized to 3NF. The Country ID is stored in the city table, and the city id is stor

##4. Normalization Process

-- a. Take a specific table in Sakila and guide through the process of normalizing it from the initial unnormalized fo

-- Let's take a hypothetical unnormalized table: film_inventory.
-- Unnormalized film_inventory:
-- film_id, title, store_id, store_address, inventory_id
-- 1NF:
-- Assume the table already satisfies 1NF (each cell has a single value).
-- 2NF:
-- Primary key: (film_id, inventory_id).
-- title depends only on film_id.
-- store_address depends only on store_id.
-- Normalization:
-- Create a film_details table: film_id (PK), title.
-- Create a store table: store_id (PK), store_address.
-- Modify film_inventory: film_id (FK), inventory_id (PK), store_id (FK).
-- Now, film_inventory, film_details, and store are in 2NF.. No


##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 i
-- SQL
```

```
WITH ActorFilmCounts 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
ActorFilmCounts
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 nam
-- SQL

WITH FilmLanguageDetails 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
title,
language_name,
rental_rate
FROM
FilmLanguageDetails;


--  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 custom
-- SQL

WITH CustomerRevenue AS (
SELECT
c.customer_id,
SUM(p.amount) AS total_revenue
FROM
customer c
JOIN
payment p ON c.customer_id = p.customer_id
GROUP BY
c.customer_id
)
SELECT
customer_id,
total_revenue
FROM
```

```
CustomerRevenue
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.
-- SQL

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


-- 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 tab
-- SQL

WITH HighRentalCustomers AS (
SELECT
customer_id
FROM
rental
GROUP BY
customer_id
HAVING
COUNT(*) > 2
)
SELECT
c.*
FROM
customer c
JOIN
HighRentalCustomers hrc ON c.customer_id = hrc.customer_id;


-- 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
--

WITH monthly_rentals AS (
SELECT
EXTRACT(YEAR FROM rental_date) AS rental_year,
EXTRACT(MONTH FROM rental_date) AS rental_month,
COUNT(*) AS total_rentals
FROM
rental
GROUP BY
EXTRACT(YEAR FROM rental_date),
EXTRACT(MONTH FROM rental_date)
)
SELECT
rental_year,
```

```sql
rental_month,
total_rentals
FROM
monthly_rentals
ORDER BY
rental_year,
rental_month;

##11.EE' 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.actor_id AS actor1_id,
fa1.actor_id AS actor1_name,
fa2.actor_id AS actor2_id,
fa2.actor_id AS actor2_name,
fa1.film_id
FROM
film_actor fa1
JOIN
film_actor fa2 ON fa1.film_id = fa2.film_id
WHERE
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
actor_pairs ap
JOIN
actor a1 ON ap.actor1_id = a1.actor_id
JOIN
actor a2 ON ap.actor2_id = a2.actor_id;

## 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 (
SELECT
staff_id,
first_name,
last_name,
0 AS level
FROM
staff
WHERE
staff_id = 2
UNION ALL
SELECT
s.staff_id,
s.first_name,
s.last_name,
level + 1
FROM
staff s
JOIN
employee_hierarchy m ON s.staff_id = m.staff_id + 1
)
SELECT
staff_id,
first_name,
last_name,
level
FROM
employee_hierarchy;
```