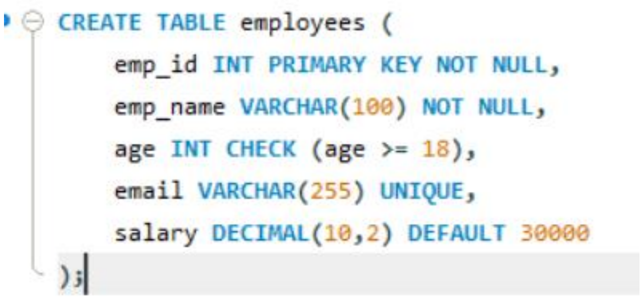
1. Create a table called employees with the following structure?
    - emp_id (integer, should not be NULL and should be a primary key)
    - emp_name (text, should not be NULL)
    - age (integer, should have a check constraint to ensure the age is at least 18)
    - email (text, should be unique for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

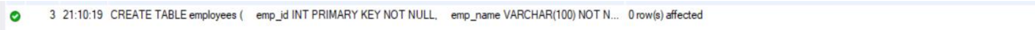
    Write the SQL query to create the above table with all constraints.

Sollution :



```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY NOT NULL,
    emp_name VARCHAR(100) NOT NULL,
    age INT CHECK (age >= 18),
    email VARCHAR(255) UNIQUE,
    salary DECIMAL(10,2) DEFAULT 30000
);
```

Output



```
✓   3  21:10:19  CREATE TABLE employees (    emp_id INT PRIMARY KEY NOT NULL,    emp_name VARCHAR(100) NOT N...  0 row(s) affected
```

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

In SQL, **constraints** are rules applied to table columns to ensure that the data stored in a database is **valid, consistent, and reliable**.
They help **maintain data integrity** by preventing invalid or inconsistent entries.

- **Prevent invalid data** from being entered.
- **Avoid duplicates** where they shouldn't exist.
- **Ensure relationships** between tables remain correct.

| Constraint | Purpose | Example |
|---|---|---|
| **PRIMARY KEY** | Uniquely identifies each row in a table; cannot be NULL | `emp_id INT PRIMARY KEY` |
| **FOREIGN KEY** | Ensures a value exists in another table (maintains referential integrity) | `FOREIGN KEY (dept_id) REFERENCES departments(dept_id)` |

|  |  |  |
|---|---|---|
| **UNIQUE** | Ensures all values in a column are different | `email VARCHAR(255) UNIQUE` |
| **NOT NULL** | Ensures the column cannot have NULL values | `emp_name VARCHAR(50) NOT NULL` |
| **CHECK** | Ensures values meet a condition | `age INT CHECK (age >= 18)` |
| **DEFAULT** | Assigns a default value if none is provided | `salary DECIMAL(10,2) DEFAULT 30000` |

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 is used when **a column must always have a value** — meaning it's a **mandatory field**.

- Ensure essential data is always present
- Avoid incomplete records

No, a primary key cannot contain NULL values.

**Justification :-**

**Primary Key = Uniquely Identifies a Row**

- Every row must have a unique identifier.
- If it's NULL, SQL cannot determine which row it is referring to.

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 SQL, you use the ALTER TABLE statement to modify constraints on an existing table.

```
        CREATE TABLE employees (
            emp_id INT,
            emp_name VARCHAR(50),
            age INT,
            email VARCHAR(100),
            salary DECIMAL(10,2)
);
```

**Adding Constraints**

- **Add PRIMARY KEY :-**

ALTER TABLE employees
ADD CONSTRAINT pk_emp PRIMARY KEY (emp_id);

- Add UNIQUE

ALTER TABLE employees
ADD CONSTRAINT unique_email UNIQUE (email);

**Removing Constraints**

- **Drop PRIMARY KEY**

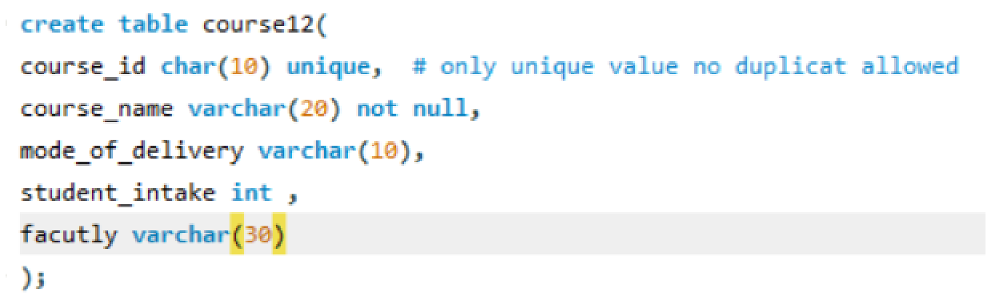
ALTER TABLE employees
DROP PRIMARY KEY;

- **Drop UNIQUE**

ALTER TABLE employees
DROP INDEX unique_email;

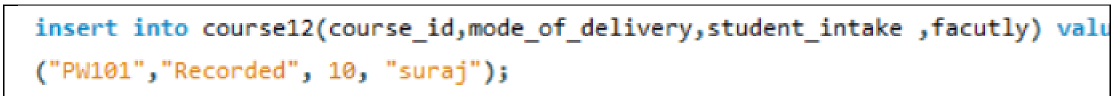
5. Explain the consequences of attempting to insert, update, or delete data in a way that violates constraints. Provide an example of an error message that might occur when violating a constraint.

**INSERT** → If you insert a value that violates constraints, the record won't be saved.
**UPDATE** → If you try to change an existing value to something that breaks a constraint, the update fails.
**DELETE** → If deleting a row violates a constraint (e.g., foreign key dependency), deletion is blocked.



```
create table course12(
course_id char(10) unique,  # only unique value no duplicat allowed
course_name varchar(20) not null,
mode_of_delivery varchar(10),
student_intake int ,
facutly varchar(30)
);
```

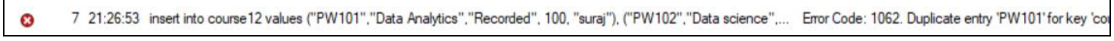
Insert



```
insert into course12(course_id,mode_of_delivery,student_intake ,facutly) valu
("PW101","Recorded", 10, "suraj");
```

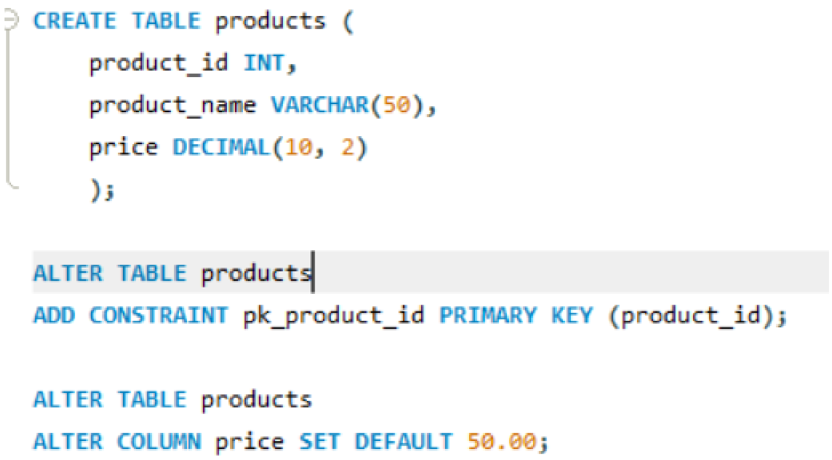
Already course_id PW101 is present in the table

Error



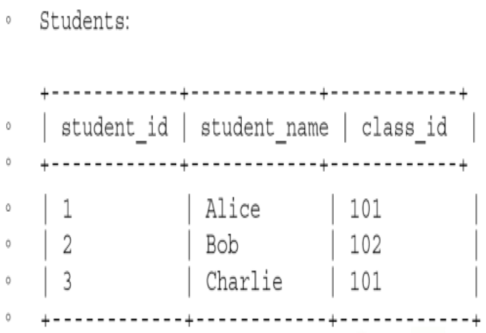
```
7 21:26:53 insert into course12 values ("PW101","Data Analytics","Recorded", 100, "suraj"), ("PW102","Data science",... Error Code: 1062. Duplicate entry 'PW101' for key 'co
```

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
   Q : The price should have a default value of 50.00



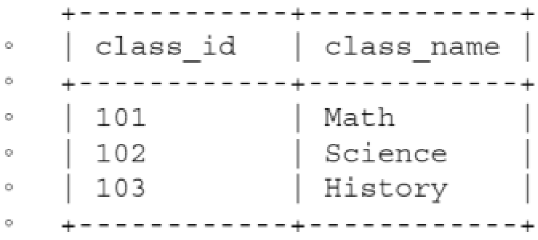
```
CREATE TABLE products (
    product_id INT,
    product_name VARCHAR(50),
    price DECIMAL(10, 2)
    );

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

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

7.



- Students:

| student_id | student_name | class_id |
|---|---|---|
| 1 | Alice | 101 |
| 2 | Bob | 102 |
| 3 | Charlie | 101 |

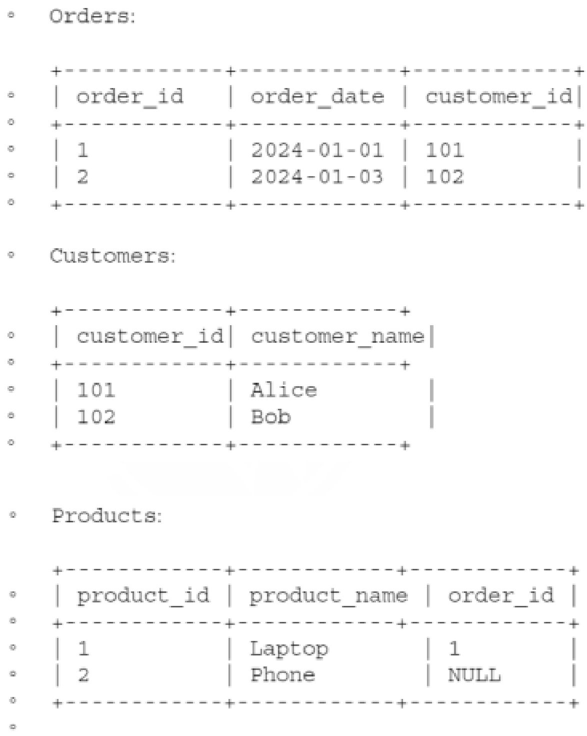


| class_id | class_name |
|---|---|
| 101 | Math |
| 102 | Science |
| 103 | History |

Write a query to fetch the student_name and class_name for each student using an INNER JOIN.

```
SELECT 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:



◦ 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 |

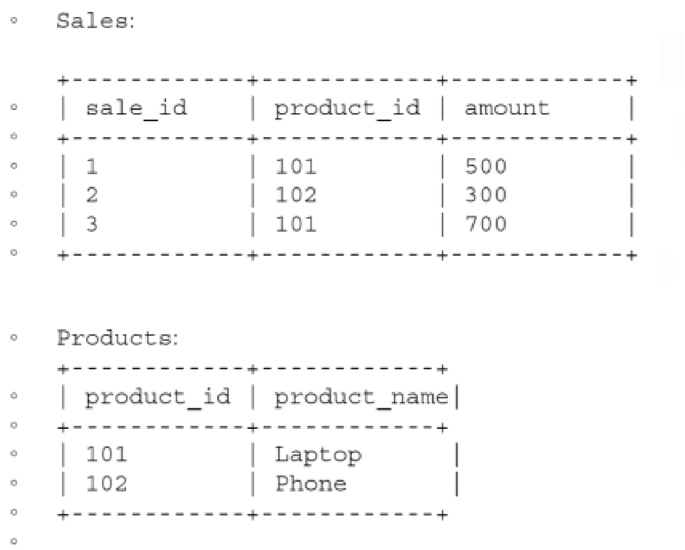
◦ Products:

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

9.

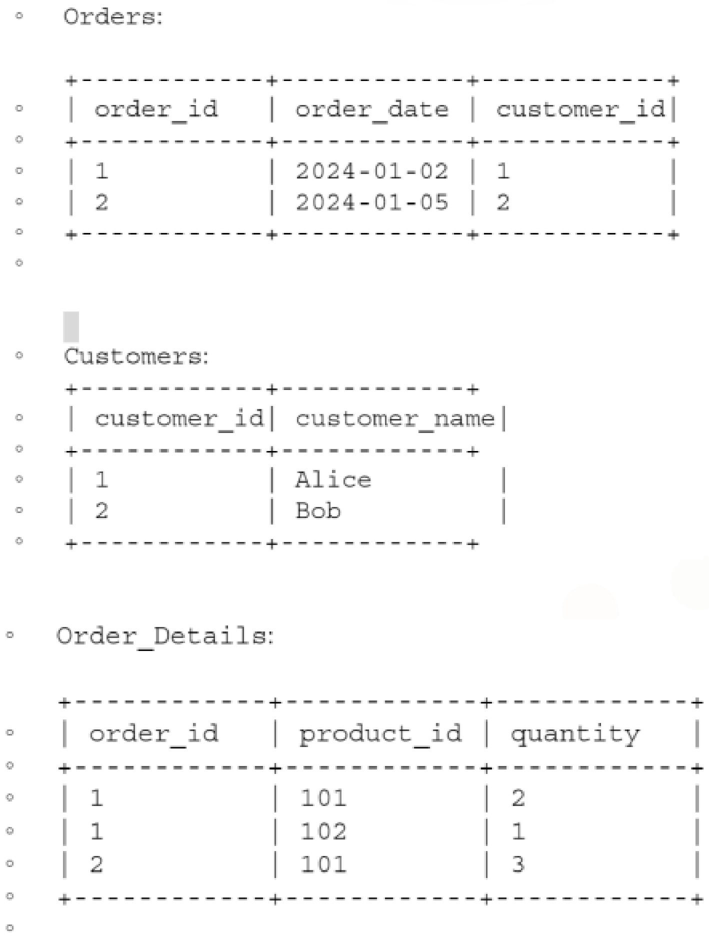


```
◦   Sales:

    +------------+------------+------------+
◦   | 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 p.product_id, p.product_name,
             SUM(oi.quantity * oi.unit_price) AS total_sales
        FROM products p
        INNER JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY p.product_id, p.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         |
◦   +-----------+-----------+-----------+
◦
```

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.

```
SELECT o.order_id, c.customer_name, SUM(oi.quantity) AS total_quantity
FROM orders o
INNER JOIN customers c ON o.customer_id = c.customer_id
INNER JOIN order_items oi ON o.order_id = oi.order_id
GROUP BY o.order_id, c.customer_name;
```

SQL Commands

1-Identify the primary keys and foreign keys in maven movies db. Discuss the differences

Primary keys: e.g., actor.actor_id, film.film_id, customer.customer_id, rental.rental_id, inventory.inventory_id, payment.payment_id

Foreign keys: e.g., film_actor.actor_id (FK -> actor), film_actor.film_id (FK -> film), inventory.film_id (FK -> film), rental.inventory_id (FK -> inventory), payment.customer_id (FK -> customer), address.city_id (FK -> city).
**Differences:** Primary key uniquely identifies a row; foreign key references a key in another table and enforces referential integrity.

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;
-- or
SELECT country_id, 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(*) AS cnt
FROM film
```

```
WHERE replacement_cost > 15 AND replacement_cost < 20;
```

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

```
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 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%' AND city LIKE '%a'
LIMIT 4;
```

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

```
SELECT * FROM customer WHERE first_name LIKE '%NI%';
-- or to be safe:
SELECT * FROM customer WHERE UPPER(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%' AND CHAR_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 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

Basic Aggregate Functions:

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

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

Question 2: Find the average rental duration (in days) of movies rented from the Sakila database. Hint: Utilize the AVG() function.

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

String Functions:

Question 3: Display the first name and last name of customers in uppercase. Hint: Use the UPPER () function.

```
SELECT UPPER(first_name) AS first_name_up, UPPER(last_name) AS
last_name_up
FROM customer;
```

Question 4: Extract the month from the rental date and display it alongside the rental ID. Hint: Employ the MONTH() function.

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

GROUP BY:

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.

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

Question 6: Find the total revenue generated by each store. Hint: Combine SUM() and GROUP BY.

```
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. Hint: JOIN film_category, film, and rental tables, then use cOUNT () and GROUP BY.

```
SELECT c.name AS category_name, COUNT(r.rental_id) AS rentals_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 c.name;
```

Question 8: Find the average rental rate of movies in each language. Hint: JOIN film and language tables, then use AVG () and GROUP BY.

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

Joins

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.

```
SELECT f.title, cu.first_name, cu.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 cu ON r.customer_id = cu.customer_id;
```

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 tables.

```
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. Hint: JOIN customer, payment, and rental tables, then use SUM() and GROUP BY.

```
SELECT c.customer_id, 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;
```

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.

```
SELECT cu.customer_id, cu.first_name, cu.last_name, f.title
FROM customer cu
JOIN address a ON cu.address_id = a.address_id
JOIN city ci ON a.city_id = ci.city_id
JOIN rental r ON cu.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 cu.customer_id, cu.first_name, cu.last_name, f.title;
```

Advanced Joins and GROUP BY:

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.

```
SELECT f.title, COUNT(r.rental_id) 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.film_id, f.title
ORDER BY times_rented DESC
LIMIT 5;
```

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.

```
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 SUM(CASE WHEN i.store_id = 1 THEN 1 ELSE 0 END) > 0
  AND SUM(CASE WHEN i.store_id = 2 THEN 1 ELSE 0 END) > 0;
```

Windows Function:

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

```
SELECT customer_id, total_spent,
      RANK() OVER (ORDER BY total_spent DESC) AS rank_pos
```

```
FROM (
  SELECT customer_id, SUM(amount) AS total_spent
  FROM payment
  GROUP BY customer_id
) t;
```

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

```
SELECT film_id, payment_date,
       SUM(amount) OVER (PARTITION BY film_id ORDER BY payment_date
                         ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW) AS cumulative_revenue
FROM (
  SELECT f.film_id, p.payment_date, p.amount
  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
) t;
```

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

```
SELECT film_id, length, AVG(rental_duration) OVER (PARTITION BY length)
AS avg_rental_duration_by_length
FROM film;
```

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

```
SELECT category_id, film_id, title, rentals_count
FROM (
  SELECT c.category_id, f.film_id, f.title,
         COUNT(r.rental_id) AS rentals_count,
         ROW_NUMBER() OVER (PARTITION BY c.category_id ORDER BY
COUNT(r.rental_id) DESC) AS rn
  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, f.film_id, f.title
) t
WHERE rn <= 3;
```

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

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

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

```
SELECT customer_id, total_spent
FROM (
  SELECT customer_id, SUM(amount) AS total_spent,
         NTILE(5) OVER (ORDER BY SUM(amount) DESC) AS quintile
  FROM payment
  GROUP BY customer_id
) t
WHERE quintile = 1; -- top 20%
```

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

```
SELECT category_id, rentals_count,
       SUM(rentals_count) OVER (ORDER BY rentals_count DESC) AS
running_total
FROM (
  SELECT c.category_id, COUNT(r.rental_id) AS rentals_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.category_id
) t;
```

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

```
WITH cat_counts AS (
  SELECT fc.category_id, f.film_id, f.title, COUNT(r.rental_id) AS film_rentals
  FROM film_category fc
  JOIN film f ON fc.film_id = f.film_id
  LEFT JOIN inventory i ON f.film_id = i.film_id
  LEFT JOIN rental r ON i.inventory_id = r.inventory_id
  GROUP BY fc.category_id, f.film_id, f.title
),
cat_avg AS (
  SELECT category_id, AVG(film_rentals) AS avg_rentals
  FROM cat_counts
  GROUP BY category_id
)
SELECT cc.category_id, cc.film_id, cc.title, cc.film_rentals, ca.avg_rentals
FROM cat_counts cc
JOIN cat_avg ca ON cc.category_id = ca.category_id
WHERE cc.film_rentals < ca.avg_rentals;
```

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

```
SELECT DATE_FORMAT(payment_date, '%Y-%m') AS month, SUM(amount)
AS revenue
FROM payment
GROUP BY month
ORDER BY revenue DESC
LIMIT 5;
```

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

```
WITH cust_counts AS (
  SELECT customer_id, COUNT(*) AS total_rentals
  FROM rental
  GROUP BY customer_id
), avg_all AS (
  SELECT AVG(total_rentals) AS avg_rentals FROM cust_counts
)
SELECT c.customer_id, c.total_rentals, a.avg_rentals, c.total_rentals -
a.avg_rentals AS diff
FROM cust_counts c CROSS JOIN avg_all a;
```

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.

**a. Identify a table in Sakila that violates 1NF**

**In the Sakila database, suppose we had a denormalized table like this (not actually present, but possible in a bad design):**

| customer_id | name | phones |
|---|---|---|
| **1** | **John Smith** | **555-1234, 555-5678** |
| **2** | **Jane Doe** | **555-8765** |

**Violation of 1NF:**

- **The phones column stores multiple values in a single field (comma-separated list).**
- **1NF requires atomic values (no repeating groups or multi-valued attributes).**

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.

**a. Determine if a table is in 2NF**

- 2NF applies to tables with a **composite primary key**.
- All **non-key attributes** must depend on the **whole** composite key, not just part of it.

Example:
film_actor table in Sakila has composite PK (film_id, actor_id).
If this table also stored film_title, that would violate 2NF, because:

- film_title depends only on film_id (part of the key), not on (film_id, actor_id).

**Normalization steps:**

1. Remove film_title from film_actor.
2. Keep only the composite key and the direct relationship.
3. Store film_title in film table (which already exists in Sakila).

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.

**a. Identify a table that violates 3NF**

Rule for 3NF: **No transitive dependencies** — a non-key column should not depend on another non-key column.

Example (hypothetical violation in Sakila):
Suppose customer table stored:

| customer_id | address_id | city | country |
|---|---|---|---|

**Violation:**

- country depends on city, and city depends on address_id.

- Both city and country are **not directly dependent** on customer_id but on each other.

**Normalization steps:**

1. Remove city and country from customer.
2. Store them in the city and country tables (already in Sakila).
3. Link through address_id → city_id → country_id.

4. Normalization Process: a. Take a specific table in Sakila and guide through the process of normalizing it from the initial unnormalized form up to at least 2NF.

**Unnormalized Table (example):**

| rental_id | customer_name | film_title | category | amount |
|---|---|---|---|---|
| 1 | John Smith | Shrek | Comedy | 4.99 |
| 2 | Jane Doe | Titanic | Romance | 3.99 |

**Problems:**

- Repeating customer names and film titles.
- Multiple dependencies.

---

**Step 1 → 1NF**

Ensure atomic values, remove any multivalued cells. (Already atomic here.)

---

**Step 2 → 2NF**

Identify primary key — here it might be rental_id. But if we combined (customer_name, film_title) as a composite key, we'd have partial dependencies.

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_count AS (

```
    SELECT
        a.actor_id,
        CONCAT(a.first_name, ' ', a.last_name) AS actor_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 DISTINCT actor_name, film_count
FROM actor_film_count
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_lang AS (
                SELECT
                    f.film_id,
                    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 film_lang
 ORDER BY 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_id, 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. È\

```
        WITH film_rank AS (
           SELECT
              film_id,
              title,
              rental_duration,
              RANK() OVER (ORDER BY rental_duration DESC) AS
        duration_rank
           FROM film
        )
        SELECT *
        FROM film_rank
ORDER BY duration_rank, title;
```

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_customers AS (
           SELECT
              customer_id,
              COUNT(*) AS rental_count
           FROM rental
           GROUP BY customer_id
           HAVING COUNT(*) > 2
        )
        SELECT
           fc.customer_id,
           c.first_name,
           c.last_name,
           fc.rental_count
        FROM frequent_customers fc
        JOIN customer c
           ON fc.customer_id = c.customer_id
ORDER BY fc.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(*) 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 film_actors AS (
            SELECT film_id, actor_id
            FROM film_actor
        )
        SELECT
            fa1.actor_id AS actor1_id,
            fa2.actor_id AS actor2_id,
            fa1.film_id
        FROM film_actors fa1
        JOIN film_actors fa2
            ON fa1.film_id = fa2.film_id
           AND fa1.actor_id < fa2.actor_id
ORDER BY fa1.film_id, actor1_id, actor2_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 col

```

        WITH RECURSIVE employee_hierarchy AS (
            -- Base case: start with the manager
            SELECT staff_id, first_name, last_name, reports_to, 0 AS level
            FROM staff
            WHERE staff_id = 1  -- change this to the manager's ID

            UNION ALL

            -- Recursive case: find employees reporting to the ones found in
        the previous step
            SELECT s.staff_id, s.first_name, s.last_name, s.reports_to,
        eh.level + 1
            FROM staff s
            INNER JOIN employee_hierarchy eh
                ON s.reports_to = eh.staff_id
        )
        SELECT *
        FROM employee_hierarchy
```

```
ORDER BY level, staff_id;
```