SQL Assignment

1. **Create a table called employees with the following structure**
   * **emp_id (integer, should not be N<LL and should be a primary key)`**
   * **emp_name (text, should not be N<LL)`**
   * **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)a Write the SQL query to create the above table with all constraints.**

   Ans –

   ```
   create table employees  (
   emp_id int primary key,
   emp_name varchar(50) not null,
   age  int  CHECK (age >= 18),
   email varchar(300) UNIQUE NOT NULL ,
   salary DECIMAL(10, 2) DEFAULT 30000.00
   );
   ```

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

   Ans- Constraints are rules in a database that ensure data is accurate and reliable. They help maintain data integrity by preventing invalid data entry.

   Just like above , Primary Key: Ensures each record is unique and identifiable (e.g., emp_id must be unique for every employee).

   NOT NULL: Ensures a column can't have missing values (e.g., emp_name cannot be blank).

   UNIQUE: Ensures all values in a column are different (e.g., email must be unique). CHECK: Enforces specific rules (e.g., age >= 18).

   DEFAULT: Sets a default value if none is provided (e.g., salary defaults to 30,000).

3. **Why would you apply the NOT NuLL constraint to a column? Can a primary key contain N<LL values? Justify your answer.**

   Ans- The NOT NULL constraint makes sure that a column cannot have empty or missing values. You use it when you always need information in that column (like a name or age) and don't want it left blank. A primary key cannot have NULL values because it's used to uniquely identify each row in a table. If a primary key had NULL values, it wouldn't be able to uniquely identify the rows, which is its main job.

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**

Ans- To add or remove constraints on an existing table, you use the ALTER TABLE command.

For add-

```
ALTER TABLE employees ADD CONSTRAINT check_age CHECK (age >= 18);
```

For delete –

```
ALTER TABLE employees DROP CONSTRAINT check_age;
```

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.**

Ans- When you attempt to insert, update, or delete data that violates a constraint, the database will prevent the action and throw an error. This helps maintain the integrity of the data by ensuring that the rules (constraints) you set are followed.

Consequences of Violating Constraints:

- Insert Violations: Trying to insert a row that breaks a rule (e.g., entering NULL in a NOT NULL column) will fail.
- Update Violations: If you try to change data to something that violates a constraint (e.g., updating a value to a duplicate in a UNIQUE column), the update won't succeed.
- Delete Violations: If deleting a row causes other constraints (like foreign keys) to break, the delete will fail.

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

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**

Ans-

-
  ```
  ALTER TABLE products
   ADD CONSTRAINT product_id PRIMARY KEY ;
  ```

-
  ```
  Alter table products
   alter column set default 50.00;
  ```

7. **Write a query that shows all order_id, student_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).**

   Ans –

```
SELECT
   p.order_id,
   s.student_name,
   p.product_name
FROM
   products p
LEFT JOIN
   orders o
 ON p.order_id = o.order_id

LEFT JOIN
   students s ON o.student_id = s.student_id;
```

9 .   Write a query to find the total sales amount for each product using an INNER JOIN and the SUM() function

```
Select sum(s.sales), p.product_name from sales s

Join products p

On

s.product_id = p.product_id

group by p.product_name;
```

10 . 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.

Ans-

```
Select

o.order_id ,

 c.customer_name ,

 sum( od.quantity) as total_quantity from orders o

Join customers c

On o.customer_id = c. customer_id

Join order_details od
```

```
On o.order_id = od.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**
   Ans-

   For primary key command

```
SELECT
   tc.table_name,
   kcu.column_name,
   tc.constraint_name
FROM
   information_schema.table_constraints AS tc
JOIN
   information_schema.key_column_usage AS kcu
ON
   tc.constraint_name = kcu.constraint_name
WHERE
   tc.table_name = 'actor'
   AND tc.constraint_type = 'PRIMARY KEY'
   AND tc.table_schema = 'sakila';
```

For foreign key command

```
SELECT
    kcu.table_name,
    kcu.column_name,
    kcu.constraint_name,
    kcu.referenced_table_name,
    kcu.referenced_column_name
FROM
    information_schema.key_column_usage AS kcu
WHERE
    kcu.referenced_table_name = 'actor'
    AND kcu.table_schema = 'sakila';
```

2. List all details of actors

Ans – select * from actors;

3. -List all customer information from DB.

Ans-

```
 SELECT *
FROM customer;
```

4. List different countries.

Ans-

```
SELECT
    country
FROM
    country;
```

5. Display all active customers.

Ans-

```
SELECT
    customer_id,
    first_name,
    last_name,
    email,
    active,
    create_date,
    last_update
```

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

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

Ans-

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

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

Ans-

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

8. **8 - List the total number of films whose replacement cost is greater than $15 and less than $20.**

Ans –

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

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

Ans-

```
SELECT
   COUNT(DISTINCT (first_name))
```

```
FROM
   actor;
```

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

Ans-

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

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

Ans-

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

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

Ans-

```
SELECT
   title, rating
FROM
   film
WHERE
   rating = 'g';
```

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

Ans-

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

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

Ans-

```
SELECT
```

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

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

Ans-

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

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

Ans-

```
SELECT
    first_name
FROM
    Customer
WHERE
    first_name LIKE '%NI%'
LIMIT 4;
```

17. **Find all customers whose first name have "r" in the second position .**

Ans-

```
SELECT
    first_name
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.**

Ans-

```
SELECT
    first_name
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".**

Ans –

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

20. **Get the films with pg and pg-13 rating using IN operator.**

Ans-

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

21. **Get the films with length between 50 to 100 using between operator.**

Ans-

```
SELECT
   title,
   length
FROM
   film
WHERE
   length BETWEEN 50 AND 100;
```

22. **- Get the top 50 actors using limit operator.**

Ans-

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

23. **Get the distinct film ids from inventory table.**

Ans-

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

Functions

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

   Ans-

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

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

   Ans-

   ```
   SELECT
      AVG(DATEDIFF(return_date, rental_date)) AS average_rental_duration_days
   FROM
      rental
   WHERE
      return_date IS NOT NULL;
   ```

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

   Ans-

   ```
   SELECT
      upper(first_name) as Cap_First_Name, upper(last_name) as Cap_Last_Name
   FROM
      customer;
   ```

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

   Ans-

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

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**

   Ans-

```
SELECT
   COUNT(rental_id) as total_rent, customer_id
FROM
   rental
GROUP BY customer_id;
```

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

   Ans-

```
SELECT
   s.store_id,
   SUM(p.amount) AS total_amount
FROM
   payment p
JOIN
   customer c ON p.customer_id = c.customer_id
JOIN
   store s ON c.store_id = s.store_id
GROUP BY
   s.store_id;
```

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.**

   Ans-

```
SELECT
   c.name AS category_name,
   COUNT(r.rental_id) AS total_rentals
FROM
   rental AS r
JOIN
   inventory AS i ON r.inventory_id = i.inventory_id
JOIN
   film AS f ON i.film_id = f.film_id
JOIN
```

```
   film_category AS fc ON f.film_id = fc.film_id
JOIN
   category AS c ON fc.category_id = c.category_id
GROUP BY
   c.name;
```

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

Ans-

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

JOINS

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.

Ans-

```
SELECT
   f.title,
   c.first_name,
   c.last_name
FROM
   film AS f
JOIN
   inventory AS i ON f.film_id = i.film_id
JOIN
```

```
    rental AS r ON i.inventory_id = r.inventory_id
JOIN
    customer AS c ON r.customer_id = c.customer_id;
```

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.**

Ans-

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

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..**

Ans –

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

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**

Ans-

```
SELECT
    c.first_name,
    c.last_name,
    f.title AS movie_title,
    ci.city AS city_name
```

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

Advanced Joins and GROUP BY:

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.

Ans –

```
SELECT

   f.title,

   COUNT(r.rental_id) AS rental_count

FROM

   film AS f
```

```
JOIN
   inventory AS i ON f.film_id = i.film_id
JOIN
   rental AS r ON i.inventory_id = r.inventory_id
GROUP BY
   f.title
ORDER BY
   rental_count DESC
LIMIT 5;
```

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.

Ans-

```
SELECT
   c.first_name,
   c.last_name
FROM
   customer AS c
JOIN
   rental AS r ON c.customer_id = r.customer_id
JOIN
   inventory AS 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.**

   Ans-

```
WITH most_rent AS (
   SELECT
      c.first_name,
      c.last_name,
      SUM(p.amount) AS total_amount,
      RANK() OVER (ORDER BY SUM(p.amount) DESC) AS total_rank
   FROM
      customer AS c
   JOIN
      payment AS p ON c.customer_id = p.customer_id
   GROUP BY
      c.customer_id, c.first_name, c.last_name
)

SELECT
   first_name,
   last_name,
   total_amount,
   total_rank
FROM
   most_rent;
```

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

   Ans-

```
SELECT
   f.title,
   r.rental_date,
   SUM(p.amount) OVER (PARTITION BY f.film_id ORDER BY r.rental_date) AS
cumulative_revenue
FROM
   film AS f
```

```
JOIN
  inventory AS i ON f.film_id = i.film_id
JOIN
  rental AS r ON i.inventory_id = r.inventory_id
JOIN
  payment AS 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.**
Ans-

```
SELECT
  f.film_id,
  f.title,
  f.length AS film_length,
  DATEDIFF(r.return_date, r.rental_date) AS rental_duration,
  AVG(DATEDIFF(r.return_date, r.rental_date))
    OVER (PARTITION BY f.length) AS avg_rental_duration_for_length
FROM
  film f
JOIN
  inventory i ON f.film_id = i.film_id
JOIN
  rental r ON i.inventory_id = r.inventory_id
ORDER BY
  f.length, f.film_id;
```

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

```
WITH RankedFilms AS (
  SELECT
    c.name AS category_name,
    f.title,
    COUNT(r.rental_id) AS rental_count,
    ROW_NUMBER() OVER (PARTITION BY c.name ORDER BY COUNT(r.rental_id) DESC) AS
category_rank
  FROM
    film AS f
  JOIN
    film_category AS fc ON f.film_id = fc.film_id
  JOIN
    category AS c ON fc.category_id = c.category_id
  JOIN
```

```
        inventory AS i ON f.film_id = i.film_id
    JOIN
        rental AS r ON i.inventory_id = r.inventory_id
    GROUP BY
        c.name, f.title
)

SELECT
    category_name,
    title,
    rental_count,
    category_rank
FROM
    RankedFilms
WHERE
    category_rank <= 3
ORDER BY
    category_name, category_rank;
```

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

Ans-

```
WITH CustomerRentals AS (
    SELECT
        customer_id,
        COUNT(rental_id) AS total_rentals
    FROM
        rental
    GROUP BY
        customer_id
),
AverageRentals AS (
    SELECT
        AVG(total_rentals) AS avg_rentals
    FROM
        CustomerRentals
)

SELECT
    cr.customer_id,
    cr.total_rentals,
    ar.avg_rentals,
    cr.total_rentals - ar.avg_rentals AS rental_difference
```

```
FROM
    CustomerRentals AS cr
CROSS JOIN
    AverageRentals AS ar
ORDER BY
    cr.customer_id;
```

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

Ans-

```
SELECT
    DATE_FORMAT(p.payment_date, '%Y-%m') AS month_year,
    SUM(p.amount) AS total_monthly_revenue
FROM
    payment AS p
JOIN
    rental AS r ON p.rental_id = r.rental_id
GROUP BY
    month_year
ORDER BY
    month_year;
```

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

Ans-

```
WITH CustomerSpending AS (
    SELECT
        customer_id,
        SUM(p.amount) AS total_spending
    FROM
        payment AS p
    GROUP BY
        customer_id
),
RankedCustomerSpending AS (
    SELECT
        customer_id,
        total_spending,
        NTILE(5) OVER (ORDER BY total_spending DESC) AS spending_percentile
    FROM
        CustomerSpending
)

SELECT
    customer_id,
   total_spending
FROM
   RankedCustomerSpending
WHERE
   spending_percentile = 1
ORDER BY
   total_spending DESC;
```

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

ANS-

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

SELECT
   category_name,
   rental_count,
   @running_total := @running_total + rental_count AS running_total_rentals
FROM
   CategoryRentals, (SELECT @running_total := 0) AS init
ORDER BY
   rental_count DESC;
```

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

Ans-

```
WITH FilmRentals AS (
   SELECT
      f.film_id,
      f.title,
        c.category_id,
        c.name AS category_name,
        COUNT(r.rental_id) AS rental_count
    FROM
        film AS f
    JOIN
        film_category AS fc ON f.film_id = fc.film_id
    JOIN
        category AS c ON fc.category_id = c.category_id
    JOIN
        inventory AS i ON f.film_id = i.film_id
    JOIN
        rental AS r ON i.inventory_id = r.inventory_id
    GROUP BY
        f.film_id, f.title, c.category_id, c.name
),
AverageRentals AS (
    SELECT
        category_id,
        AVG(rental_count) AS avg_rental_count
    FROM
        FilmRentals
    GROUP BY
        category_id
)

SELECT
    fr.film_id,
    fr.title,
    fr.rental_count,
    ar.avg_rental_count
FROM
    FilmRentals AS fr
JOIN
    AverageRentals AS ar ON fr.category_id = ar.category_id
WHERE
    fr.rental_count < ar.avg_rental_count
ORDER BY
    fr.category_id, fr.rental_count;
```

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

    Ans-

    ```
    SELECT
    ```

```
  DATE_FORMAT(payment_date, '%Y-%m') AS month_year,
  SUM(amount) AS total_revenue
FROM
  payment
GROUP BY
  month_year
ORDER BY
  total_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.

Ans- The customer table can violate First Normal Form (1NF) if it contains non-atomic values, such as multiple phone numbers stored in a single cell.

Example of a Violation:

customer_id	first_name	last_name	address	phone_numbers
1	John	Doe	123 Elm St	123-456-7890, 987-654-3210

Normalization Steps:

1. **Create Separate Tables:**
   - **Customer Table:** Store unique customer records with atomic values.
   - **Customer Phone Table:** Create a new table to store phone numbers linked to the customer.

Normalized Structure:

Customer Table:

customer_id	first_name	last_name	address
1	John	Doe	123 Elm St
2	Jane	Smith	456 Oak St

customer_id	**first_name**	**last_name**	**address**
3	Alice	Johnson	789 Pine St

Customer Phone Table:

customer_id	**phone_number**
1	123-456-7890
1	987-654-3210
2	555-123-4567
3	222-333-4444

Outcome:

- Each cell now contains atomic values, ensuring adherence to 1NF.
- The relationship between customers and their phone numbers is maintained through foreign keys, facilitating easier data management and querying.

4o mini

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.

Ans-

Table: film_actor

Determining if the Table is in 2NF:

1. Identify the Primary Key: The composite primary key for the film_actor table is (film_id, actor_id).
2. Check for Partial Dependencies: A table is in Second Normal Form (2NF) if it is in First Normal Form (1NF) and has no partial dependencies. This means that all non-key attributes must depend on the entire primary key, not just part of it.

Example Structure of film_actor:

film_id	actor_id	actor_name
1	1	Actor A

film_id	actor_id	actor_name
1	2	Actor B
2	1	Actor A

Analysis:

- In this table, actor_name only depends on actor_id, not on the full primary key (film_id, actor_id). Thus, it creates a partial dependency, violating 2NF.

Steps to Normalize to 2NF:

1. Separate the Data into Related Tables:
   - Film_Actor Table: Keep film_id and actor_id.
   - Actor Table: Create a new table for actor details, including actor_id and actor_name.

Normalized Structure:

Film_Actor Table:

film_id	actor_id
1	1
1	2
2	1

Actor Table:

actor_id	actor_name
1	Actor A
2	Actor B

Outcome:

- The film_actor table is now in 2NF, as there are no partial dependencies. All non-key attributes in both tables depend on their respective primary keys.

By normalizing to 2NF, data redundancy is reduced, and the integrity of relationships between films and actors is improved.

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.

Ans-

Chosen Table: customer

Identifying 3NF Violations:

- A table is in Third Normal Form (3NF) if it is in Second Normal Form (2NF) and has no transitive dependencies.
- In the customer table, transitive dependencies are present:
  - city depends on city_id.
  - country depends on country_id.

Example Structure of customer:

customer_id first_name last_name address_id address city_id city country_id country

Transitive Dependencies:

- city → country_id
- country → country_id

Steps to Normalize to 3NF:

1. Separate into Related Tables:
   - Customer Table: Keep customer_id, first_name, last_name, and address_id.
   - Address Table: Include address_id, address, and city_id.
   - City Table: Include city_id, city, and country_id.
   - Country Table: Include country_id and country.

Normalized Structure:

Customer Table:

customer_id first_name last_name address_id

Address Table:

address_id address city_id

City Table:

city_id city country_id

Country Table:

country_id country

Outcome

By normalizing to 3NF:

- All transitive dependencies are eliminated, ensuring that non-key attributes depend only on the primary key.
- This structure improves data integrity and reduces redundancy across the database.

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.

Ans-

1. Unnormalized Form (UNF):

- The original film table contains multiple actor names in a single cell, violating First Normal Form (1NF).

Example:

film_id	title	description	release_year	language	actors
1	"Film A"	"Action film about heroes."	2021	"English"	"Actor 1, Actor 2"
2	"Film B"	"Romantic film."	2020	"English"	"Actor 3"

---

2. Convert to First Normal Form (1NF):

- Split actor names into separate rows to ensure atomic values.

Normalized Structure (1NF):

film_id	title	description	release_year	language	actor_name
1	"Film A"	"Action film about heroes."	2021	"English"	"Actor 1"
1	"Film A"	"Action film about heroes."	2021	"English"	"Actor 2"
2	"Film B"	"Romantic film."	2020	"English"	"Actor 3"

3. Convert to Second Normal Form (2NF):

- Identify and eliminate partial dependencies by creating separate tables.

Normalized Structure (2NF):

Film Table:

film_id	title	description	release_year	language
1	"Film A"	"Action film about heroes."	2021	"English"
2	"Film B"	"Romantic film."	2020	"English"

Film_Actor Table:

film_id	actor_name
1	"Actor 1"
1	"Actor 2"
2	"Actor 3"

Outcome

The film table is now in 2NF, with all attributes depending only on the primary key. The film_actor table maintains the relationship between films and actors, enhancing data integrity and reducing redundancy.

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.

Ans-

```
WITH actor_film_counts 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
    actor_name,
    film_count
FROM
    actor_film_counts
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.

Ans-

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

```
SELECT

   film_title,

   language_name,

   rental_rate

FROM

   film_language_info

ORDER BY

   film_title;
```

6/7. 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.
Ans-

```
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;
```

7 CTE with Window Functions: a. Utilize a CTE with a window function to rank films based on their rental duration from the film table.

Ans-

```
WITH ranked_films AS (
    SELECT
        f.title AS film_title,
        f.rental_duration,
        RANK() OVER (ORDER BY f.rental_duration DESC) AS rental_rank
    FROM
        film f
)

SELECT
    film_title,
    rental_duration,
    rental_rank
FROM
    ranked_films
ORDER BY
    rental_rank;
```

8. 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.

Ans-

```
WITH active_customers AS (
    SELECT
        c.customer_id,
        COUNT(r.rental_id) AS rental_count
    FROM
        customer c
    JOIN
        payment p ON c.customer_id = p.customer_id
    JOIN
```

```
        rental r ON p.rental_id = r.rental_id
    GROUP BY
        c.customer_id
    HAVING
        COUNT(r.rental_id) > 2
)

SELECT
    c.customer_id,
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
    c.email,
    c.address_id
FROM
    customer c
JOIN
    active_customers ac ON c.customer_id = ac.customer_id
ORDER BY
    customer_name;
```

9 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.

Ans-

```
WITH monthly_rentals AS (
    SELECT
        DATE_FORMAT(r.rental_date, '%Y-%m') AS rental_month,
        COUNT(r.rental_id) AS total_rentals
    FROM
        rental r
    GROUP BY
        rental_month
)

SELECT
    rental_month,
    total_rentals
FROM
    monthly_rentals
ORDER BY
    rental_month;
```

10. 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.

Ans-

```
WITH actor_pairs AS (
   SELECT
      fa1.actor_id AS actor1_id,
      fa2.actor_id AS actor2_id,
      f.title AS film_title
   FROM
      film_actor fa1
   JOIN
      film_actor fa2 ON fa1.film_id = fa2.film_id
   JOIN
      film f ON fa1.film_id = f.film_id
   WHERE
      fa1.actor_id < fa2.actor_id  -- This condition avoids duplicate pairs
)

SELECT
   ap.actor1_id,
   ap.actor2_id,
   ap.film_title
FROM
   actor_pairs ap
ORDER BY
   ap.film_title, ap.actor1_id, ap.actor2_id;
```

11. **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.**
    Ans-

```
WITH RECURSIVE employee_hierarchy AS (

   SELECT
      s.staff_id,
      CONCAT(s.first_name, ' ', s.last_name) AS employee_name,
      s.<correct_column_name>
   FROM
      staff s
   WHERE
      s.<correct_column_name> = 1
   UNION ALL


   SELECT
      s.staff_id,
      CONCAT(s.first_name, ' ', s.last_name) AS employee_name,
        s.<correct_column_name>
    FROM
        staff s
    JOIN
        employee_hierarchy eh ON s.<correct_column_name> = eh.staff_id
)

SELECT
    staff_id,
    employee_name,
    <correct_column_name>
    FROM
    employee_hierarchy
ORDER BY
    staff_id;
```