Assignment - Normalisation and CTE queries

1. First Normal Form (1NF)

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

Table: The `film_actor` table could be considered as a candidate for violating 1NF if it stores multiple actor IDs in a single field.

Normalization Steps:

- Ensure that each field in the table contains only atomic (indivisible) values.
- If a film has multiple actors, each actor's ID should have its own row in the table. Thus, if the `film_actor` table had a column storing a list of actor IDs, we would need to split those into separate rows.

---

2. Second Normal Form (2NF)

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.

Table: The `film` table.

Determining 2NF:

- A table is in 2NF if it is in 1NF and all non-key attributes are fully functionally dependent on the entire primary key.
- In the `film` table, if it has a composite primary key (e.g., `film_id`, `category_id`), check if any non-key attributes (like `title`, `release_year`) depend on only part of the key.

Normalization Steps:

- If any non-key attributes depend only on part of the primary key, split them into a separate table, ensuring that all attributes depend on the entire primary key.

---

3. Third Normal Form (3NF)

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

Table: The `customer` table.

Transitive Dependencies:

- If the `customer` table contains `customer_id`, `first_name`, `last_name`, `address_id`, and `address`, the address depends on `address_id`, which is not the primary key, creating a transitive dependency.

Normalization Steps:

- Remove the transitive dependency by creating a separate `address` table.
- Keep `address_id` in the `customer` table and link it to the new `address` table to eliminate transitive dependencies.

---

4. Normalization Process

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

Table: The `payment` table (as an example).

Step 1: Unnormalized Form (UNF)

- Assume it includes non-atomic fields, such as `payment_date` and `amounts`.

Step 2: Convert to 1NF

- Ensure all fields are atomic. If `amounts` is a list, split it into separate rows for each payment record.

Step 3: Convert to 2NF

- Identify the composite key (e.g., `payment_id`, `customer_id`). Ensure all non-key attributes (like `amount`, `payment_date`) are fully functionally dependent on the entire key.
- If `payment_date` depends only on `payment_id`, split into a new table.

---

5. CTE Basics

Write a query using a CTE to retrieve the distinct list of actor names and the number of films they have acted in.

```
WITH actor_film_count AS (
   SELECT 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
)
SELECT DISTINCT first_name, last_name, film_count
FROM actor_film_count;
```

---

6. Recursive CTE

Use a recursive CTE to generate a hierarchical list of categories and their subcategories from the category table in Sakila.

```
WITH RECURSIVE category_hierarchy AS (
   SELECT category_id, name, parent_id
   FROM category
   WHERE parent_id IS NULL  -- Start from top-level categories
   UNION ALL
   SELECT c.category_id, c.name, c.parent_id
   FROM category c
   JOIN category_hierarchy ch ON c.parent_id = ch.category_id
)
SELECT * FROM category_hierarchy;
```

---

7. CTE with Joins

Create a CTE that combines information from the film and language tables to display the film title, language name, and rental rate.

```
WITH film_language 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 * FROM film_language;
```

---

8. CTE for Aggregation

Write a query using a CTE to find the total revenue generated by each customer (sum of payments) from the payment and customer tables.

```
WITH customer_revenue AS (
   SELECT c.customer_id, c.first_name, c.last_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 * FROM customer_revenue;
```

---

9. CTE with Window Functions

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

```
WITH film_ranking AS (
   SELECT f.title, f.length,
        RANK() OVER (ORDER BY f.length DESC) AS rental_duration_rank
   FROM film f
)
SELECT * FROM film_ranking;
```

---

10. CTE and Filtering

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
   FROM rental
   GROUP BY customer_id
   HAVING COUNT(rental_id) > 2
)
SELECT c.customer_id, c.first_name, c.last_name
FROM customer c
JOIN frequent_customers fc ON c.customer_id = fc.customer_id;
```

---

11. CTE for Date Calculations

Write a query using a CTE to find the total number of rentals made each month, considering the rental table.

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

```
)
SELECT * FROM monthly_rentals;
```

---

12. CTE for Pivot Operations

Use a CTE to pivot the data from the payment table to display the total payments made by each customer in separate columns for different payment methods.

```
WITH payment_summary AS (
   SELECT c.customer_id,
        SUM(CASE WHEN p.payment_type = 'Credit Card' THEN p.amount ELSE 0 END) AS credit_card,
        SUM(CASE WHEN p.payment_type = 'Cash' THEN p.amount ELSE 0 END) AS cash,
        SUM(CASE WHEN p.payment_type = 'Debit Card' THEN p.amount ELSE 0 END) AS debit_card
   FROM customer c
   JOIN payment p ON c.customer_id = p.customer_id
   GROUP BY c.customer_id
)
SELECT * FROM payment_summary;
```

---

13. CTE and Self-Join

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, fa2.actor_id AS actor2, f.title
   FROM film_actor fa1
   JOIN film_actor fa2 ON fa1.film_id = fa2.film_id AND fa1.actor_id < fa2.actor_id
   JOIN film f ON fa1.film_id = f.film_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,
     title
FROM actor_pairs ap
JOIN actor a1 ON ap.actor1 = a1.actor_id
JOIN actor a2 ON ap.actor2 = a2.actor_id;
```

---

14. CTE for Recursive Search

Implement a recursive CTE to find all employees in the staff table who report to a specific manager, considering the staff_id column.

```
WITH RECURSIVE employee_hierarchy AS (
   SELECT staff_id, first_name, last_name, manager_id
   FROM staff
   WHERE manager_id = :specific_manager_id  -- Replace with actual manager ID

   UNION ALL

   SELECT s.staff_id, s.first_name, s.last_name, s.manager_id
   FROM staff s
   JOIN employee_hierarchy eh ON s.manager_id = eh.staff_id
)
SELECT * FROM employee_hierarchy;
```