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

```
CREATE TABLE film (
    film_id INT PRIMARY KEY,
    title VARCHAR(255)
);

CREATE TABLE special_features (
    feature_id INT PRIMARY KEY,
    feature_name VARCHAR(255)
);

CREATE TABLE film_special_features (
    film_id INT,
    feature_id INT,
    FOREIGN KEY (film_id) REFERENCES film(film_id),
    FOREIGN KEY (feature_id) REFERENCES special_features(feature_id)
);
```

-- 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.

```
CREATE TABLE film_actor (
    actor_id INT,
    film_id INT,
    role VARCHAR(255),
    PRIMARY KEY (actor_id, film_id)
```

```
);
```

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

```
CREATE TABLE film (
    film_id INT PRIMARY KEY,
    title VARCHAR(255),
    language_id INT,
    rental_rate DECIMAL(4,2),
    length INT
);

CREATE TABLE original_language (
    film_id INT PRIMARY KEY,
    original_language_id INT,
    FOREIGN KEY (film_id) REFERENCES film(film_id),
    FOREIGN KEY (original_language_id) REFERENCES language(language_id)
);
```

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

```
CREATE TABLE rental (
    rental_id INT PRIMARY KEY,
    rental_date DATETIME,
    inventory_id INT,
    customer_id INT,
```

```
    return_date DATETIME,
    staff_id INT
);
```

-- 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 and tables.

```
WITH ActorFilmCount 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, actor_name
)

SELECT
    actor_name,
    film_count
FROM
    ActorFilmCount
ORDER BY
    actor_name;
```

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

```
WITH RecursiveCategory AS (
SELECT
  category_id,
  name AS category_name,
  parent_id
FROM
  category
WHERE
  parent_id IS NULL
UNION ALL
SELECT
  c.category_id,
  CONCAT(RC.category_name, ' > ', c.name) AS category_name,
  c.parent_id
FROM
  category c
JOIN RecursiveCategory RC ON c.parent_id = RC.category_id
)
SELECT
  category_id,
  category_name
FROM
  RecursiveCategory
ORDER BY
  category_id;
```

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

```
WITH FilmLanguageInfo 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
    FilmLanguageInfo
ORDER BY
    film_title;
```

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

```
WITH CustomerRevenue 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, customer_name
  )

  SELECT
    customer_id,
    customer_name,
    COALESCE(total_revenue, 0) AS total_revenue
  FROM
    CustomerRevenue
  ORDER BY
    customer_id;
```

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

```
  WITH RankedFilms AS (
  SELECT
    film_id,
    title,
    rental_duration,
    ROW_NUMBER() OVER (ORDER BY rental_duration) AS duration_rank
  FROM
    film
  )
```

```
SELECT
    film_id,
    title,
    rental_duration,
    duration_rank
FROM
    RankedFilms
ORDER BY
    rental_duration;
```

-- 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 CustomerRentals AS (
SELECT
    r.customer_id,
    COUNT(*) AS total_rentals
FROM
    rental r
GROUP BY
    r.customer_id
HAVING
    COUNT(*) > 2
)

SELECT
    c.customer_id,
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
```

```
    c.email,
    c.address_id,
    cr.total_rentals
FROM
    customer c
JOIN CustomerRentals cr ON c.customer_id = cr.customer_id
ORDER BY
    cr.total_rentals DESC;
```

-- 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 MonthlyRentals AS (
 SELECT
    DATE_FORMAT(rental_date, '%Y-%m') AS rental_month,
    COUNT(*) AS total_rentals
 FROM
    rental
 GROUP BY
    rental_month
 )

 SELECT
    rental_month,
    total_rentals
FROM
    MonthlyRentals
ORDER BY
```

```
    rental_month;
```

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

```
WITH CustomerPayments AS (
 SELECT
    customer_id,
    SUM(CASE WHEN payment_type = 'Cash' THEN amount ELSE 0 END) AS cash_payments,
    SUM(CASE WHEN payment_type = 'Credit Card' THEN amount ELSE 0 END) AS credit_card_payments,
    SUM(CASE WHEN payment_type = 'Debit Card' THEN amount ELSE 0 END) AS debit_card_payments,
    SUM(CASE WHEN payment_type = 'Check' THEN amount ELSE 0 END) AS check_payments
 FROM
    payment
 GROUP BY
    customer_id
)

SELECT
    cp.customer_id,
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
    cp.cash_payments,
    cp.credit_card_payments,
    cp.debit_card_payments,
    cp.check_payments
```

```
FROM
    CustomerPayments cp
JOIN
    customer c ON cp.customer_id = c.customer_id
ORDER BY
    customer_id;
```

-- 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 ActorPairs AS (
SELECT
    fa1.actor_id AS actor1_id,
    fa2.actor_id AS actor2_id,
    f.film_id,
    f.title AS film_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
    ap.actor1_id,
    CONCAT(a1.first_name, ' ', a1.last_name) AS actor1_name,
    ap.actor2_id,
    CONCAT(a2.first_name, ' ', a2.last_name) AS actor2_name,
    ap.film_id,
    ap.film_title
```

```
FROM
    ActorPairs ap
JOIN
    actor a1 ON ap.actor1_id = a1.actor_id
JOIN
    actor a2 ON ap.actor2_id = a2.actor_id
ORDER BY
    ap.film_id, ap.actor1_id, ap.actor2_id;
```

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

```
WITH RecursiveEmployeeHierarchy AS (
  SELECT
    staff_id,
    first_name,
    last_name,
    reports_to
  FROM
    staff
  WHERE
    staff_id = :manager_id  -- Specify the manager's staff_id here
  UNION ALL
  SELECT
    s.staff_id,
    s.first_name,
    s.last_name,
    s.reports_to
```

```
FROM
    staff s
JOIN RecursiveEmployeeHierarchy reh ON s.reports_to = reh.staff_id
)

SELECT
    staff_id,
    first_name,
    last_name,
    reports_to
FROM
    RecursiveEmployeeHierarchy
ORDER BY
    staff_id;
```