Complete SQL Assignment Solutions (Mavenmovies Dataset)

Yeh poora solution aapke assignment ke structure ke hisaab se arrange kiya gaya hai. Sabhi practical queries Mavenmovies database ke schema par आधारित hain.

Section 1: SQL Basics

Is section ke liye dataset ki zaroorat nahi hai. Yeh general SQL concepts par आधारित hai.

1. Create a table called `employees`

Query:

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

2. Purpose of constraints

Answer: Constraints database mein data ki quality, accuracy, aur reliability banaye rakhne ke liye rules hote hain. Yeh galat data ko table mein enter hone se rokte hain, jisse **Data Integrity** (data ki shuddhta) bani rehti hai.

- **`NOT NULL`**: Column ko khaali (NULL) nahi chhoda ja sakta.
- **`UNIQUE`**: Column ki har value alag honi chahiye.
- **`PRIMARY KEY`**: Table ke har record ko uniquely identify karta hai. Yeh `NOT NULL` aur `UNIQUE` ka combination hota hai.
- **`FOREIGN KEY`**: Do tables ke beech mein link banata hai.
- **`CHECK`**: Sunishchit karta hai ki column ki value ek di gayi condition ko poora kare.
- **`DEFAULT`**: Agar koi value na di jaaye, to ek default value set karta hai.

3. `NOT NULL` constraint and Primary Keys

Answer: `NOT NULL` constraint isliye lagaya jaata hai taaki kisi zaroori column (jaise user ka naam, order date) mein hamesha data ho. Yeh adhoore records ko rokta hai.

Nahi, **ek primary key mein `NULL` value nahi ho sakti. Kaaran:** Primary key ka mool uddeshya table ke har record ko ek anokhi pehchaan dena hai. `NULL` ka matlab 'koi value nahi' hota hai, isliye yeh kisi record ko anokhi pehchaan nahi de sakta. Isliye, primary key hamesha `NOT NULL` aur `UNIQUE` hoti hai.

4. Adding or removing constraints

Answer: `ALTER TABLE` command ka istemaal karke existing table par constraints jode ya hataye jaate hain.

Constraint Jodna (Adding a Constraint):

```
-- 'email' column par UNIQUE constraint jodna
ALTER TABLE employees
ADD CONSTRAINT uq_email UNIQUE (email);
```

-

Constraint Hatana (Removing a Constraint):

```
-- Abhi joda gaya 'uq_email' constraint hatana
ALTER TABLE employees
DROP CONSTRAINT uq_email;
```

-

5. Consequences of violating constraints

Answer: Jab koi `INSERT`, `UPDATE`, ya `DELETE` operation kisi constraint ko todne ki koshish karta hai, to database us operation ko **reject** kar deta hai aur ek **error message** deta hai. Isse table ka data jaisa tha waisa hi rehta hai.

Error Message ka Udaharan (UNIQUE constraint todne par):

```
ERROR: duplicate key value violates unique constraint "uq_email"
DETAIL: Key (email)=(existing.email@example.com) already exists.
```

-

6. Add constraints to an existing `products` table

Answer:

```
-- Step 1: product_id ko PRIMARY KEY banana
ALTER TABLE products
ADD PRIMARY KEY (product_id);

-- Step 2: price column par DEFAULT value set karna
ALTER TABLE products
ALTER COLUMN price SET DEFAULT 50.00;
```

7. Fetch student and class names using `INNER JOIN`

Answer:

```
SELECT
    s.student_name,
    c.class_name
FROM
    Students s
INNER JOIN
    Classes c ON s.class_id = c.class_id;
```

8. List all orders and products using `INNER JOIN` and `LEFT JOIN`

Answer:

```
SELECT
    p.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. Find the total sales amount for each product

Answer:

```
SELECT
    p.product_name,
    SUM(s.amount) AS total_sales_amount
FROM
    Sales s
INNER JOIN
    Products p ON s.product_id = p.product_id
GROUP BY
    p.product_name;
```

10. Display order details with customer names

Answer:

```
SELECT
   o.order_id,
   c.customer_name,
   od.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;
```

Section 2: SQL Commands (Mavenmovies)

1. Identify primary keys and foreign keys

- **Primary Keys (PK):** Har table mein ek unique identifier, jaise `actor.actor_id`, `film.film_id`, `customer.customer_id`.
- **Foreign Keys (FK):** Ek table se doosri table ka link, jaise `film.language_id` (jo `language` table ke `language_id` se judta hai) ya `rental.customer_id` (jo `customer` table ke `customer_id` se judta hai).

2. List all details of actors

```
SELECT * FROM actor;
```

3. List all customer information

```
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 ID 1

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

7. Films with rental duration greater than 5

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

8. Total films with replacement cost between $15 and $20

```
SELECT COUNT(*) AS total_films
FROM film
WHERE replacement_cost > 15 AND replacement_cost < 20;
```

9. Count of unique first names of actors

```
SELECT COUNT(DISTINCT first_name) AS unique_first_names FROM actor;
```

10. First 10 records from the customer table

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

11. First 3 customers whose first name starts with 'b'

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

12. First 5 movies rated 'G'

```
SELECT title FROM film WHERE rating = 'G' LIMIT 5;
```

13. Customers whose first name starts with "a"

```
SELECT first_name, last_name FROM customer WHERE first_name LIKE 'a%';
```

14. Customers whose first name ends with "a"

```
SELECT first_name, last_name FROM customer WHERE first_name LIKE '%a';
```

15. First 4 cities which start and end with 'a'

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

16. Customers with "NI" in their first name

```
SELECT first_name, last_name FROM customer WHERE first_name LIKE '%NI%';
```

17. Customers with "r" in the second position of their first name

```
SELECT first_name, last_name FROM customer WHERE first_name LIKE '_r%';
```

18. Customers with first name starting with "a" and at least 5 characters long

```
SELECT first_name, last_name FROM customer WHERE first_name LIKE 'a____%';
```

19. Customers with first name starting with "a" and ending with "o"

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

20. Films with 'PG' and 'PG-13' rating

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

21. Films with length between 50 and 100

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

22. Top 50 actors

```
SELECT * FROM actor ORDER BY actor_id LIMIT 50;
```

23. Distinct film IDs from inventory

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

Section 3: Functions (Mavenmovies)

1. Total number of rentals made

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

2. Average rental duration

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

3. Customer names in uppercase

```
SELECT UPPER(first_name) AS first_name_upper, UPPER(last_name) AS
last_name_upper FROM customer;
```

4. Extract month from rental date

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

5. Count of rentals for each customer

```
SELECT customer_id, COUNT(rental_id) AS rental_count
FROM rental
GROUP BY customer_id
ORDER BY rental_count DESC;
```

6. Total revenue by each store

```
SELECT i.store_id, SUM(p.amount) AS total_revenue
FROM payment p
JOIN rental r ON p.rental_id = r.rental_id
JOIN inventory i ON r.inventory_id = i.inventory_id
GROUP BY i.store_id;
```

7. Total rentals for each movie category

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

8. Average rental rate for each language

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

Section 4: Joins (Mavenmovies)

9. Movie title and customer name who rented it

```
SELECT f.title, c.first_name, c.last_name
FROM rental r
JOIN customer c ON r.customer_id = c.customer_id
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN film f ON i.film_id = f.film_id
LIMIT 100; -- Bahut saare results aayenge, isliye limit laga di
```

10. Actors in the film "ACADEMY DINOSAUR"

```sql
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 = 'ACADEMY DINOSAUR';
```

11. Customer names and their total spending

```sql
SELECT 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
ORDER BY total_spent DESC;
```

12. Movie titles rented by customers in 'London'

```sql
SELECT c.first_name, c.last_name, f.title
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 ct ON a.city_id = ct.city_id
WHERE ct.city = 'London';
```

13. Top 5 rented movies

```sql
SELECT f.title, COUNT(r.rental_id) AS rental_count
FROM rental r
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN film f ON i.film_id = f.film_id
GROUP BY f.title
ORDER BY rental_count DESC
LIMIT 5;
```

14. Customers who rented from both store 1 and store 2

```sql
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
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;
```

Section 5: Window Functions (Mavenmovies)

1. Rank customers based on total spending

```
SELECT
   customer_id,
   SUM(amount) AS total_spent,
   RANK() OVER (ORDER BY SUM(amount) DESC) AS customer_rank
FROM payment
GROUP BY customer_id;
```

2. Calculate cumulative revenue generated by each film over time

```
SELECT
   p.payment_date,
   f.title,
   p.amount,
   SUM(p.amount) OVER (PARTITION BY f.film_id ORDER BY p.payment_date) AS
cumulative_revenue
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;
```

3. Top 3 films in each category based on rental counts

```
WITH CategoryRentalCounts AS (
   SELECT
      f.title,
      c.name AS category_name,
      COUNT(r.rental_id) AS rental_count,
      ROW_NUMBER() OVER (PARTITION BY c.name ORDER BY COUNT(r.rental_id)
DESC) as rn
   FROM rental r
   JOIN inventory i ON r.inventory_id = i.inventory_id
   JOIN film f ON i.film_id = f.film_id
   JOIN film_category fc ON f.film_id = fc.film_id
   JOIN category c ON fc.category_id = c.category_id
   GROUP BY f.title, c.name
)
SELECT title, category_name, rental_count
FROM CategoryRentalCounts
WHERE rn <= 3;
```

4. Find the monthly revenue trend

```
SELECT
   DATE_FORMAT(payment_date, '%Y-%m') AS payment_month,
   SUM(amount) AS monthly_revenue,
   SUM(SUM(amount)) OVER (ORDER BY DATE_FORMAT(payment_date, '%Y-%m')) AS
cumulative_revenue
FROM payment
GROUP BY payment_month
ORDER BY payment_month;
```

5. Top 5 months with the highest revenue

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

6. Monthly revenue trend for store

```
SELECT

    DATE_FORMAT(p.payment_date, '%Y-%m') AS month,

    SUM(p.amount) AS monthly_revenue,

    SUM(SUM(p.amount)) OVER (ORDER BY DATE_FORMAT(p.payment_date,
'%Y-%m')) AS cumulative_revenue

FROM payment p

GROUP BY DATE_FORMAT(p.payment_date, '%Y-%m')

ORDER BY month;
```

---

7. Customers in top 20% spending

```
SELECT *

FROM (
    SELECT
        c.customer_id,
        c.first_name,
        c.last_name,
        SUM(p.amount) AS total_spent,
        NTILE(5) OVER (ORDER BY SUM(p.amount) DESC) AS spending_percentile
    FROM customer c
    JOIN payment p ON c.customer_id = p.customer_id
    GROUP BY c.customer_id, c.first_name, c.last_name
) t
WHERE spending_percentile = 1;   -- Top 20%
```

---

8. Running total of rentals per category

```
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)) AS running_total
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.name;
```

---

9. Films rented less than avg rentals in category

```
SELECT *
FROM (
    SELECT
        f.title,
        c.name AS category_name,
        COUNT(r.rental_id) AS rental_count,
        AVG(COUNT(r.rental_id)) OVER (PARTITION BY c.category_id) AS avg_category_rentals
    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.category_id, c.name
) t
WHERE rental_count < avg_category_rentals;
```

---

10. Top 5 months with highest revenue

```
SELECT *
FROM (
    SELECT
        DATE_FORMAT(p.payment_date, '%Y-%m') AS month,
        SUM(p.amount) AS monthly_revenue,
        RANK() OVER (ORDER BY SUM(p.amount) DESC) AS rank_by_revenue
    FROM payment p
    GROUP BY DATE_FORMAT(p.payment_date, '%Y-%m')
) t
WHERE rank_by_revenue <= 5;
```

Section 6: Normalisation & CTE (Mavenmovies)

1. First Normal Form (1NF)

a. Identify a table that violates 1NF and how to normalize it

Sakila itself is already in 1NF. But a *hypothetical* denormalized table that **would violate 1NF**:

```
customer_contact_unnorm
-----------------------
customer_id (PK)
name
phones          -- e.g. '98765-43210, 91234-56789'  (multiple values in single column)
emails          -- e.g. 'a@x.com;b@y.com'
address_line
```

Why it violates 1NF

- A column (`phones` or `emails`) contains multiple values in a single field (non-atomic values). 1NF requires atomic (indivisible) column values.

Normalization to 1NF

1. Create separate rows for each phone/email (or better: separate tables).
2. New design:

```
customer (customer_id PK, name, address_id, ...)
customer_phone (phone_id PK, customer_id FK, phone_number,
phone_type)
customer_email (email_id PK, customer_id FK, email_address,
email_type)
```

Now each column is atomic and each phone/email is a separate row.

---

2. Second Normal Form (2NF)

a. Choose a table and how to test for 2NF; if violates, how to normalize

2NF applies to tables with a **composite primary key**. It states: *no partial dependency* of a non-key column on part of the composite key.

Example (hypothetical violation):
Suppose we have a denormalized table:

```
film_store_info
-------------------------
film_id (PK part)
store_id (PK part)
film_title
store_address
stock_count
```

Composite PK = (film_id, store_id).

Check for 2NF violation

- `film_title` depends only on `film_id` (part of composite key) — partial dependency.
- `store_address` depends only on `store_id` — partial dependency.

Thus table is **not in 2NF**.

Normalize to 2NF

1. Split into tables where attributes depend on the whole key:
    - `film_store` (film_id, store_id, stock_count) — this keeps attributes that truly depend on both film & store.
    - `film` (film_id PK, title, etc.)
    - `store` (store_id PK, address, etc.)

After splitting, non-key attributes no longer depend on just part of the composite key.

---

3. Third Normal Form (3NF)

a. Identify a table that violates 3NF, describe transitive dependencies and normalize

Again, Sakila is mostly 3NF already. Example of a hypothetical violation:

```
payment_denorm
------------------------------
payment_id PK
customer_id FK
customer_name
customer_city
amount
payment_date
```

Transitive dependency

- `customer_name` and `customer_city` depend on `customer_id`, while `customer_city` might determine `country_id` (or `country_name`) — i.e., non-key attributes depend on other non-key attributes via `customer_id`. This is transitive because `payment -> customer_id -> customer_city -> country`.

Normalize to 3NF

1. Remove customer details from `payment`.

    - `payment` (payment_id, customer_id, amount, payment_date, staff_id, rental_id)

2. Keep full customer details in `customer` table and address/city/country in separate normalized tables:

    - `customer` (customer_id, address_id, ...)
    - `address` (address_id, address, city_id, ...)
    - `city` (city_id, city, country_id)
    - `country` (country_id, country)

This removes transitive dependencies; `payment` attributes depend only on the PK.

---

4. Normalization Process (example: unnormalized → 1NF → 2NF)

a. Walkthrough with a concrete table

Start with a fully unnormalized hypothetical table `order_unnorm` (similar pattern):

```
order_unnorm
------------
order_id
customer_name
customer_phone1,customer_phone2   -- repeated columns
order_date
item1, item2, item3               -- repeating group
qty1, qty2, qty3
price1, price2, price3
```

Step 1: To 1NF

- Remove repeating columns and repeating groups. Create rows for each item:

```
orders (order_id PK, customer_id FK, order_date)
```

```
order_items (order_item_id PK, order_id FK, product_id, quantity,
unit_price)
customer_phone (phone_id, customer_id, phone_number)
```

Step 2: To 2NF

- If `orders` used a composite key (say order_id + product_id) — ensure attributes that depend only on `order_id` are moved to `orders` and not kept in the composite table.
- Example: If `order_items` had `customer_name` — move `customer_name` to `customer` table.

Step 3: To 3NF (and beyond)

- Eliminate transitive dependencies: move address → city → country into separate tables.

---

5. CTE Basics

a. Distinct list of actor names and number of films they acted in (actor + film_actor)

```
WITH actor_film_count 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 actor_id, CONCAT(first_name, ' ', last_name) AS actor_name,
film_count
FROM actor_film_count
ORDER BY film_count DESC;
```

---

6. CTE with Joins

a. Combine film & language to display film title, language name, rental_rate

```
WITH film_lang AS (
    SELECT
        f.film_id,
        f.title,
        l.name AS language_name,
        f.rental_rate
    FROM film f
    LEFT JOIN language l ON f.language_id = l.language_id
)
SELECT film_id, title, language_name, rental_rate
FROM film_lang
ORDER BY title;
```

---

7. CTE for Aggregation

a. Total revenue per customer (payments)

```
WITH customer_revenue AS (
    SELECT
        p.customer_id,
        SUM(p.amount) AS total_revenue
    FROM payment p
    GROUP BY p.customer_id
)
SELECT
    c.customer_id,
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
    cr.total_revenue
FROM customer_revenue cr
JOIN customer c ON cr.customer_id = c.customer_id
ORDER BY cr.total_revenue DESC;
```

---

8. CTE with Window Functions

a. Rank films based on their rental duration (film.length used as duration)

```
WITH film_duration AS (
    SELECT
        film_id,
        title,
        length
    FROM film
)
SELECT
    film_id,
    title,
    length,
    RANK() OVER (ORDER BY length DESC) AS length_rank
FROM film_duration
ORDER BY length_rank;
```

> Note: If you want to rank films by *actual average rental time* from `rental` (difference r.return_date - r.rental_date), replace `length` with that aggregated metric.

Example using average actual rental duration:

```
WITH avg_rental_duration AS (
    SELECT
        f.film_id,
        f.title,
        AVG(DATEDIFF(r.return_date, r.rental_date)) AS
avg_rental_days
    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
)
SELECT
    film_id, title, avg_rental_days,
    RANK() OVER (ORDER BY avg_rental_days DESC) AS
rank_by_rental_duration
FROM avg_rental_duration
ORDER BY rank_by_rental_duration;
```

9. CTE and Filtering

a. List customers who made more than two rentals, then join to retrieve 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,
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
    c.email,
    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. Total number of rentals made each month (using rental.rental_date)

MySQL:

```
WITH monthly_rentals AS (
    SELECT
        DATE_FORMAT(r.rental_date, '%Y-%m') AS year_month,
        COUNT(*) AS rentals_count
    FROM rental r
    GROUP BY DATE_FORMAT(r.rental_date, '%Y-%m')
)
SELECT year_month, rentals_count
FROM monthly_rentals
ORDER BY year_month;
```

Postgres:

```
WITH monthly_rentals AS (
    SELECT
        TO_CHAR(r.rental_date, 'YYYY-MM') AS year_month,
        COUNT(*) AS rentals_count
    FROM rental r
    GROUP BY TO_CHAR(r.rental_date, 'YYYY-MM')
)
SELECT year_month, rentals_count
FROM monthly_rentals
ORDER BY year_month;
```

---

11. CTE and Self-Join

a. Pairs of actors who have appeared together in the same film

```
WITH film_actors AS (
    SELECT film_id, actor_id
    FROM film_actor
)
SELECT
    fa1.film_id,
    fa1.actor_id AS actor1_id,
    CONCAT(a1.first_name, ' ', a1.last_name) AS actor1_name,
    fa2.actor_id AS actor2_id,
    CONCAT(a2.first_name, ' ', a2.last_name) AS actor2_name
FROM film_actors fa1
JOIN film_actors fa2 ON fa1.film_id = fa2.film_id AND fa1.actor_id <
fa2.actor_id
JOIN actor a1 ON fa1.actor_id = a1.actor_id
JOIN actor a2 ON fa2.actor_id = a2.actor_id
ORDER BY fa1.film_id, actor1_id, actor2_id;
```

This produces each unique unordered actor pair per film (actor1_id < actor2_id avoids duplicates and actor pairing with itself).

---

12. CTE for Recursive Search

a. Recursive CTE to find all employees (staff) who report to a specific manager (assuming `reports_to` column)

Assume `staff` table columns: `staff_id`, `first_name`, `last_name`, `reports_to` (manager's staff_id). Example for manager with `staff_id = 2`.

MySQL 8+ / Postgres compatible:

```
WITH RECURSIVE reports AS (
    -- Anchor member: direct reports of manager_id = 2
    SELECT
        s.staff_id,
        CONCAT(s.first_name, ' ', s.last_name) AS staff_name,
        s.reports_to,
        1 AS level
    FROM staff s
    WHERE s.reports_to = 2

    UNION ALL

    -- Recursive member: find staff who report to anyone already found
    SELECT
        s2.staff_id,
        CONCAT(s2.first_name, ' ', s2.last_name) AS staff_name,
        s2.reports_to,
        r.level + 1 AS level
    FROM staff s2
    JOIN reports r ON s2.reports_to = r.staff_id
)
SELECT staff_id, staff_name, reports_to, level
FROM reports
ORDER BY level, staff_id;
```