SQL Assignment - Complete Answers

SQL BASICS

Q1: Create a table called employees with specified constraints

```
-- Create employees table with constraints
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(10,2) DEFAULT 30000
);
```

Q2: Purpose of constraints and examples

```
-- Purpose: Constraints enforce rules at the database level to ensure data integrity and consistency.
-- Examples:
-- PRIMARY KEY: uniquely identifies a row (no NULLs).
-- FOREIGN KEY: enforces referential integrity between tables.
-- NOT NULL: column must have a value.
-- UNIQUE: no duplicate values allowed.
-- CHECK: enforces a boolean condition on values.
-- DEFAULT: provides default value when none supplied.
```

Q3: Why apply NOT NULL; can PK be NULL?

```
-- NOT NULL ensures that a column always contains a value; useful when a value is required logically.
-- A PRIMARY KEY cannot contain NULL values because it must uniquely identify each row; NULL means unknown and
```

Q4: Steps and SQL commands to add/remove constraints (example)

```
-- Add a CHECK constraint (name may vary by RDBMS):
ALTER TABLE employees ADD CONSTRAINT chk_age CHECK (age >= 18);

-- Add UNIQUE constraint on email:
ALTER TABLE employees ADD CONSTRAINT uq_email UNIQUE (email);

-- Drop constraint (name depends on how it was created):
ALTER TABLE employees DROP CONSTRAINT chk_age;

-- For primary key addition (if product table exists):
ALTER TABLE products ADD CONSTRAINT pk_products PRIMARY KEY (product_id);

-- Remove primary key (syntax varies):
ALTER TABLE products DROP CONSTRAINT pk_products;
```

Q5: Consequences of violating constraints (example error)

```
-- Consequence: Operation fails; transaction may be rolled back.
-- Example: inserting NULL into NOT NULL column:
-- ERROR (Postgres): ERROR:  null value in column "emp_name" violates not-null constraint
-- Example: violating UNIQUE on email:
-- ERROR: duplicate key value violates unique constraint "uq_email".
```

Q6: Alter products table to add primary key and default price

```
-- Given products table without constraints:
-- Add primary key
ALTER TABLE products ADD CONSTRAINT pk_products PRIMARY KEY (product_id);

-- Set default price (Postgres syntax)
ALTER TABLE products ALTER COLUMN price SET DEFAULT 50.00;

-- Or if ALTER COLUMN SET DEFAULT not supported, use:
ALTER TABLE products ALTER COLUMN price SET DEFAULT 50.00;
```

Q7: INNER JOIN to fetch student_name and class_name

```
SELECT s.student_name, c.class_name
FROM students s
INNER JOIN classes c ON s.class_id = c.class_id;
```

Q8: Show all products and their orders (ensure all products listed even if no orders)

```
-- Use LEFT JOIN from products to orders to keep all products
SELECT p.product_id, p.product_name, o.order_id, o.order_date, co.customer_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;
```

Q9: Total sales amount for each product using INNER JOIN and SUM()

```
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
ORDER BY total_sales DESC;
```

Q10: Display order_id, customer_name, and quantity using INNER JOIN between three tables

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

SQL COMMANDS (Maven Movies / Sakila examples)

1. Identify primary keys and foreign keys in Maven Movies DB (examples and explanation)

```
-- Example (Sakila):
-- Primary Keys:
-- actor.actor_id, film.film_id, customer.customer_id, payment.payment_id
-- Foreign Keys:
-- film_actor.actor_id REFERENCES actor(actor_id)
-- film_actor.film_id REFERENCES film(film_id)
-- inventory.film_id REFERENCES film(film_id)
-- rental.inventory_id REFERENCES inventory(inventory_id)
-- Explanation: PK uniquely identify rows; FK link tables to enforce referential integrity.
```

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 with ID 1

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

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

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

8. Count of films whose replacement cost is >15 and <20

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

9. Count of unique first names of actors

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

10. Display the first 10 records from the customer table

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

11. First 3 records from customer whose first name starts with 'b'

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

12. Names of the first 5 movies rated 'G'

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

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

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

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

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

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

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

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

```
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 LOWER(first_name) LIKE '_r%';
```

18. Customers whose first name starts with 'a' and are at least 5 characters long

```
SELECT * FROM customer WHERE LOWER(first_name) LIKE 'a%' AND LENGTH(first_name) >= 5;
```

19. Customers whose first name starts with 'a' and ends with 'o'

```
SELECT * FROM customer WHERE LOWER(first_name) LIKE 'a%o';
```

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

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

21. Films with length between 50 and 100

```
SELECT * FROM film WHERE length BETWEEN 50 AND 100;
```

22. Get the top 50 actors using LIMIT

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

23. Get distinct film ids from inventory

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

FUNCTIONS (Aggregate, String, Date, GROUP BY)

Functions Q1: Retrieve total number of rentals

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

Functions Q2: Average rental duration (from film table)

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

Functions Q3: Display first and last names of customers in uppercase

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

Functions Q4: Extract month from rental_date alongside rental_id

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

Functions Q5: Count of rentals per customer

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

Functions Q6: Total revenue generated by each store

```
SELECT store_id, SUM(amount) AS total_revenue FROM payment GROUP BY store_id;
```

Functions Q7: Total number of rentals for each movie category

```
SELECT c.name AS category, COUNT(*) AS total_rentals
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;
```

Functions Q8: Average rental rate of movies in each language

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

JOINS

Q9: Display title of the movie, customer's first and last name who rented it

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

Q10: Retrieve actors who appeared in 'Gone with the Wind'

```
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';
```

Q11: Customer names with total amount spent on rentals

```
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
ORDER BY total_spent DESC;
```

Q12: Titles of movies rented by customers in a particular city (e.g., London)

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

ADVANCED JOINS & GROUP BY

Q13: Top 5 rented movies with count

```
SELECT f.title, COUNT(*) AS rental_count
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.title
ORDER BY rental_count DESC
LIMIT 5;
```

Q14: Customers who rented from both stores (store_id 1 and 2)

```
SELECT r.customer_id, c.first_name, c.last_name
FROM rental r
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN customer c ON r.customer_id = c.customer_id
GROUP BY r.customer_id, c.first_name, c.last_name
HAVING COUNT(DISTINCT i.store_id) = 2;
```

WINDOW FUNCTIONS

1. Rank the customers based on total amount spent on rentals

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

2. Calculate cumulative revenue generated by each film over time

```
SELECT f.film_id, r.rental_date,
       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. Average rental duration for each film, considering similar lengths

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

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

```
WITH film_rental_counts AS (
    SELECT fc.category_id, f.film_id, f.title, COUNT(*) AS rental_count
    FROM film f
    JOIN film_category fc ON f.film_id = fc.film_id
    JOIN inventory i ON f.film_id = i.film_id
    JOIN rental r ON i.inventory_id = r.inventory_id
    GROUP BY fc.category_id, f.film_id, f.title
)
SELECT category_id, film_id, title, rental_count
FROM (
    SELECT *, ROW_NUMBER() OVER (PARTITION BY category_id ORDER BY rental_count DESC) AS rn
    FROM film_rental_counts
) sub WHERE rn <= 3;
```

5. Monthly revenue trend for the entire rental store over time

```
SELECT DATE_TRUNC('month', p.payment_date) AS month, SUM(p.amount) AS revenue
FROM payment p
GROUP BY DATE_TRUNC('month', p.payment_date)
ORDER BY month;
```

6. Identify customers whose total spending falls in top 20%

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

7. Running total of rentals per category ordered by rental count

```
SELECT category_id, rental_count,
       SUM(rental_count) OVER (ORDER BY rental_count DESC ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
FROM (
    SELECT fc.category_id, COUNT(*) AS rental_count
    FROM film_category fc
    JOIN inventory i ON fc.film_id = i.film_id
    JOIN rental r ON i.inventory_id = r.inventory_id
    GROUP BY fc.category_id
) s;
```

8. Find films rented less than average rental count for their category

```
WITH category_avg AS (
    SELECT fc.category_id, AVG(cnt) AS avg_rentals
    FROM (
        SELECT fc.category_id, f.film_id, COUNT(*) AS cnt
        FROM film f
        JOIN film_category fc ON f.film_id = fc.film_id
        JOIN inventory i ON f.film_id = i.film_id
        JOIN rental r ON i.inventory_id = r.inventory_id
        GROUP BY fc.category_id, f.film_id
    ) t
    GROUP BY fc.category_id
)
SELECT f.film_id, f.title, fc.category_id, COUNT(*) AS film_rentals, ca.avg_rentals
FROM film f
JOIN film_category fc ON f.film_id = fc.film_id
JOIN inventory i ON f.film_id = i.film_id
JOIN rental r ON i.inventory_id = r.inventory_id
JOIN category_avg ca ON fc.category_id = ca.category_id
GROUP BY f.film_id, f.title, fc.category_id, ca.avg_rentals
HAVING COUNT(*) < ca.avg_rentals;
```

9. Find films rented less than the average rental count for their categories (alternative)

```
-- Similar to previous; included to address question variations.
```

10. Identify top 5 months with highest revenue and show revenue

```
SELECT DATE_TRUNC('month', p.payment_date) AS month, SUM(p.amount) AS revenue
FROM payment p
GROUP BY DATE_TRUNC('month', p.payment_date)
ORDER BY revenue DESC
LIMIT 5;
```

11. Difference in rental counts between each customer's total rentals and the average across all customers

```
WITH cust_counts AS (
    SELECT customer_id, COUNT(*) AS cust_rentals
    FROM rental
    GROUP BY customer_id
), avg_all AS (
    SELECT AVG(cust_rentals) AS avg_rentals FROM cust_counts
)
SELECT c.customer_id, c.cust_rentals, (c.cust_rentals - a.avg_rentals) AS diff_from_avg
FROM cust_counts c, avg_all a;
```

NORMALIZATION & CTE

1. First Normal Form (1NF): identify violation and normalize

```
-- Violation example: a table 'customer_phones' with 'phones' column storing comma-separated numbers.
-- To normalize (1NF): Create rows per phone number:
CREATE TABLE customer_phone (
    customer_id INT,
    phone VARCHAR(20),
    PRIMARY KEY(customer_id, phone),
    FOREIGN KEY(customer_id) REFERENCES customer(customer_id)
);
```

2. Second Normal Form (2NF): how to determine and normalize

```
-- 2NF requires 1NF and that every non-key attribute is fully dependent on the whole primary key.
-- Example violation: table order_items(order_id, product_id, product_name, quantity)
-- product_name depends on product_id, not the composite PK(order_id, product_id).
-- Normalize by moving product_name to products table and keep product_id in order_items.
```

3. Third Normal Form (3NF): identify transitive dependency and normalize

```
-- 3NF requires 2NF and no transitive dependencies.
-- Example violation: table employees(emp_id, dept_id, dept_name)
-- dept_name depends on dept_id, not emp_id (transitive). Normalize by creating department table:
CREATE TABLE department (dept_id INT PRIMARY KEY, dept_name TEXT);
-- Then keep dept_id in employees table as FK.
```

4. Normalization process example (unnormalized → 2NF)

```
-- Start with an unnormalized sales table with repeated product information per order.
-- 1. Convert repeated columns into separate rows (1NF).
-- 2. Identify composite keys and move partially dependent attributes to new tables (2NF).
-- e.g., split into orders(order_id, customer_id, order_date), order_items(order_id, product_id, quantity).
```

5. CTE Basics: distinct actor names and film counts

```
WITH actor_films AS (
    SELECT a.actor_id, a.first_name, a.last_name, COUNT(fa.film_id) AS film_count
    FROM actor a
    LEFT JOIN film_actor fa ON a.actor_id = fa.actor_id
    GROUP BY a.actor_id, a.first_name, a.last_name
)
SELECT * FROM actor_films ORDER BY film_count DESC;
```

6. CTE with Joins: 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
    JOIN language l ON f.language_id = l.language_id
)
SELECT * FROM film_lang;
```

7. CTE for Aggregation: total revenue per customer

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

8. CTE with Window Functions: rank films by rental_duration

```
WITH film_rank AS (
    SELECT film_id, title, rental_duration,
           RANK() OVER (ORDER BY rental_duration DESC) AS rnk
    FROM film
)
SELECT * FROM film_rank WHERE rnk <= 20;
```

9. CTE and Filtering: customers with more than two rentals and join to get 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;
```

10. CTE for Date Calculations: rentals per month

```
WITH monthly_rentals AS (
    SELECT DATE_TRUNC('month', rental_date) AS month, COUNT(*) AS rentals_count
    FROM rental
    GROUP BY DATE_TRUNC('month', rental_date)
)
SELECT * FROM monthly_rentals ORDER BY month;
```

11. CTE and Self-Join: pairs of actors who appeared together in same film

```
WITH actor_pairs AS (
    SELECT fa1.film_id, fa1.actor_id AS actor1, fa2.actor_id AS actor2
    FROM film_actor fa1
    JOIN film_actor fa2 ON fa1.film_id = fa2.film_id AND fa1.actor_id < fa2.actor_id
)
SELECT ap.film_id, f.title, a1.first_name || ' ' || a1.last_name AS actor1,
       a2.first_name || ' ' || a2.last_name AS actor2
FROM actor_pairs ap
JOIN film f ON ap.film_id = f.film_id
JOIN actor a1 ON ap.actor1 = a1.actor_id
JOIN actor a2 ON ap.actor2 = a2.actor_id
ORDER BY ap.film_id;
```

12. Recursive CTE: find all employees reporting to a specific manager (reports_to)

```
-- Assuming staff or employee table with employee_id and reports_to columns
WITH RECURSIVE reports_cte AS (
    SELECT employee_id, first_name, last_name, reports_to
    FROM staff -- or employee
    WHERE employee_id = 2 -- starting manager id
    UNION ALL
    SELECT s.employee_id, s.first_name, s.last_name, s.reports_to
    FROM staff s
    JOIN reports_cte r ON s.reports_to = r.employee_id
)
SELECT * FROM reports_cte;
```