SQL Basics Assignment Questions

1. SQL query to create the employees table:

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

Explanation of the constraints:

- PRIMARY KEY on emp_id → ensures each employee has a unique identifier and cannot be NULL.
- NOT NULL on emp_id and emp_name → these fields must always have a value.
- CHECK (age >= 18) → ensures only employees aged 18 or older can be added.
- UNIQUE on email → no two employees can share the same email.
- DEFAULT 30000 on salary → if no salary is provided, it defaults to 30,000.

2. Purpose of constraints and how they help maintain data integrity:

Constraints ensure that the data in the database remains accurate, reliable, and consistent.

They act as rules that the database enforces whenever data is inserted, updated, or deleted.

Examples of common constraints:

- **PRIMARY KEY:** Guarantees each row in a table is uniquely identifiable.
- **FOREIGN KEY:** Ensures data in one table corresponds to valid data in another, maintaining referential integrity.
- **NOT NULL:** Makes sure important fields are always filled.
- **UNIQUE:** Prevents duplicate values in a column.
- **CHECK:** Enforces domain-specific rules, like ensuring age > 0.
- **DEFAULT:** Automatically assigns a value if none is provided.

These constraints help avoid issues like duplicate records, invalid data, or orphaned rows.

3. Why apply NOT NULL? Can a primary key contain NULL values?

- **Why NOT NULL:**
  To ensure that a column always has a meaningful value. For example, you wouldn't want an employee record without a name.
- **Can a primary key contain NULL values?**
  No.
  A primary key must be both:
    - **Unique:** No two rows can have the same primary key value.
    - **NOT NULL:** Every row must have a value for the primary key.
      Allowing NULL would defeat the purpose of uniquely identifying each row.

4. Adding or removing constraints on an existing table:

Adding a constraint:

Suppose you want to add a NOT NULL constraint to the salary column:

```
ALTER TABLE employees
ALTER COLUMN salary SET NOT NULL;
```

Or adding a CHECK constraint:

```
ALTER TABLE employees
ADD CONSTRAINT chk_salary_positive CHECK (salary > 0);
```

Removing a constraint:

To remove the CHECK constraint chk_salary_positive:

sql

CopyEdit

```
ALTER TABLE employees
DROP CONSTRAINT chk_salary_positive;
```

To remove NOT NULL from the salary column:

```
ALTER TABLE employees
ALTER COLUMN salary DROP NOT NULL;
```

Note: The actual syntax might differ slightly depending on your database (e.g., MySQL, PostgreSQL, SQL Server).

5. Consequences of violating constraints (with example):

When you attempt to insert, update, or delete data that violates constraints, the database will reject the operation and return an error.

Examples:

- Inserting an employee younger than 18:

```
INSERT INTO employees (emp_id, emp_name, age, email)
VALUES (1, 'John Doe', 16, 'john@example.com');
```

❗ Error message (example):

```
ERROR: new row for relation "employees" violates check constraint "employees_age_check"
DETAIL: Failing row contains (1, John Doe, 16, john@example.com, 30000).
```

- Trying to insert two employees with the same email:

vbnet

CopyEdit

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

Constraints protect data consistency by rejecting any operations that would break the integrity rules.

6.we created the table:

```
CREATE TABLE products (
   product_id INT,
   product_name VARCHAR(50),
   price DECIMAL(10, 2)
);
```

Now you want to:

Make product_id the primary key

Set a default value of 50.00 for price

SQL commands to modify the existing table:

a) Add primary key constraint to product_id:

```
ALTER TABLE products
ADD CONSTRAINT pk_product_id PRIMARY KEY (product_id);
```

b) Set default value for price:

```
ALTER TABLE products
ALTER COLUMN price SET DEFAULT 50.00;
```

7.I have two tables:

1)Students

2)Classes

need to write a query to fetch each student's name and their class name using an **INNER JOIN**.

SQL query:

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

8.

i have three tables:

- Orders (order_id, order_date, customer_id)
- Customers (customer_id, customer_name)
- Products (product_id, product_name, order_id)

You need to show all:

- order_id
- customer_name

- product_name
  and ensure that all products are listed even if not associated with an order (using INNER JOIN and LEFT JOIN).

SQL query:

```
SELECT
   orders.order_id,
   customers.customer_name,
   products.product_name
FROM
   products
LEFT JOIN
   orders ON products.order_id = orders.order_id
LEFT JOIN
   customers ON orders.customer_id = customers.customer_id;
```

Explanation:

- We start from products so that all products are listed.
- LEFT JOIN with orders keeps products that might not have an order.
- LEFT JOIN with customers ensures customer names are shown where there *is* an order, otherwise NULL.

9.

You have two tables:

- Sales (sale_id, product_id, amount)
- Products (product_id, product_name)

You need to find the total sales amount for each product using INNER JOIN and the SUM() function.

SQL query:

```
SELECT
   products.product_name,
   SUM(sales.amount) AS total_sales
FROM
   products
```

```
INNER JOIN
   sales ON products.product_id = sales.product_id
GROUP BY
   products.product_name;
```

Explanation:

- INNER JOIN combines only matching products that have sales.
- SUM() calculates total sales amount per product.
- GROUP BY groups the results by product name.

10.i have three tables:

- Orders (order_id, order_date, customer_id)
- Customers (customer_id, customer_name)
- Order_Details (order_id, product_id, quantity)

need a query to display:

- order_id
- customer_name
- quantity of products ordered by each customer
  using an **INNER JOIN** between all three tables.

SQL query:

```
SELECT
   orders.order_id,
   customers.customer_name,
   order_details.quantity
FROM
   orders
INNER JOIN
   customers ON orders.customer_id = customers.customer_id
INNER JOIN
   order_details ON orders.order_id = order_details.order_id;
```

Explanation:

- Join orders with customers to get customer names.
- Join orders with order_details to get product quantities.

- The result shows each order along with the customer name and quantity of each product in that order.

SQL Commands

1 - Identify the primary keys and foreign keys in sakila DB. Discuss the differences

Example from sakila:

- **Primary keys:**
  - actor_id in actor table
  - customer_id in customer table
  - film_id in film table
- **Foreign keys:**
  - film_id in inventory references film
  - customer_id in payment references customer
  - address_id in customer references address

Differences:

- **Primary key** uniquely identifies each row in its own table. Cannot be NULL.
- **Foreign key** creates a link between two tables, referencing the primary key of another table to maintain referential integrity.

✅ **2 - List all details of actors**

```
SELECT * FROM actor;
```

✅ **3 - List all customer information from DB**

```
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 the films whose rental duration is greater than 5**

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

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

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

✅ **9 - Display the count of unique first names of actors**

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

✅ **10 - Display the first 10 records from the customer table**

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

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

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

✅ **12 - Display the names of the first 5 movies which are rated as ‘G’**

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

✅ **13 - Find all customers whose first name starts with "a"**

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

✅ **14 - Find all customers whose first name ends with "a"**

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

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

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

✅ **16 - Find all customers whose first name has "NI" in any position**

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

✅ **17 - Find all customers whose first name has "r" in the second position**

```
SELECT * 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**

```
SELECT * 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"**

```
SELECT * FROM customer
WHERE first_name LIKE 'A%o';
```

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

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

✅ **21 - Get the films with length between 50 to 100 using BETWEEN operator**

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

✅ **22 - Get the top 50 actors using LIMIT operator**

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

✅ **23 - Get the distinct film ids from inventory table**

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

Basic Aggregate Functions

Question 1: Retrieve the total number of rentals made

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

Question 2: Find the average rental duration (in days) of movies rented

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

(rental_duration is stored in the film table, which shows for how many days a film is rented by default)

String Functions

Question 3: Display the first name and last name of customers in uppercase

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

Question 4: Extract the month from the rental date and display it with rental ID

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

GROUP BY

Question 5: Retrieve the count of rentals for each customer

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

Question 6: Find the total revenue generated by each store

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

Question 7: Total number of rentals for each movie category

```
SELECT c.name AS category_name, COUNT(*) 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
JOIN film_category fc ON f.film_id = fc.film_id
JOIN category c ON fc.category_id = c.category_id
GROUP BY c.name;
```

Question 8: Average rental rate of movies in each language

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

Joins

Question 9: Display the title of the movie, customer's first name, and last name who rented it

```
SELECT f.title, c.first_name, c.last_name
FROM rental r
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN film f ON i.film_id = f.film_id
JOIN customer c ON r.customer_id = c.customer_id;
```

Question 10: Retrieve the names of all actors who appeared in the film "Gone with the Wind"

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

Question 11: Retrieve customer names along with the total amount they've spent on rentals

```
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, c.first_name, c.last_name;
```

Question 12: List titles of movies rented by each customer in a particular city (e.g., 'London')

```
SELECT 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.first_name, c.last_name, f.title;
```

✅ **Advanced Joins and GROUP BY**

Question 13: Display the top 5 rented movies along with number of times they've been rented

```
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;
```

Question 14: Determine customers who have rented movies from both stores (store ID 1 and store ID 2)

```
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;
```

Windows Function:

1. Rank the customers based on the total amount they've spent on rentals

```
SELECT customer_id, first_name, last_name,
    SUM(amount) AS total_spent,
    RANK() OVER (ORDER BY SUM(amount) DESC) AS customer_rank
FROM customer c
JOIN payment p ON c.customer_id = p.customer_id
GROUP BY customer_id, first_name, last_name;
```

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

```
SELECT f.film_id, f.title, p.payment_date,
    SUM(p.amount) OVER (PARTITION BY f.film_id ORDER BY p.payment_date) AS
cumulative_revenue
FROM film f
JOIN inventory i ON f.film_id = i.film_id
JOIN rental r ON i.inventory_id = r.inventory_id
JOIN payment p ON r.rental_id = p.rental_id;
```

3. Determine the average rental duration for each film, considering films with similar lengths

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

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

```
SELECT category_id, title, rental_count,
    RANK() OVER (PARTITION BY category_id ORDER BY rental_count DESC) AS rank_in_category
FROM (
  SELECT c.category_id, f.title, COUNT(r.rental_id) AS rental_count
  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 c.category_id, f.title
) AS sub
WHERE rank_in_category <= 3;
```

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

```
SELECT customer_id, first_name, last_name,
    customer_rental_count,
    customer_rental_count - avg_rental_count AS difference_from_avg
FROM (
  SELECT c.customer_id, c.first_name, c.last_name,
      COUNT(r.rental_id) AS customer_rental_count,
      AVG(COUNT(r.rental_id)) OVER () AS avg_rental_count
  FROM customer c
  JOIN rental r ON c.customer_id = r.customer_id
  GROUP BY c.customer_id, c.first_name, c.last_name
) AS sub;
```

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

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

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

```
WITH customer_spending AS (
   SELECT customer_id, SUM(amount) AS total_spent
   FROM payment
   GROUP BY customer_id
),
percentile AS (
   SELECT PERCENTILE_CONT(0.8) WITHIN GROUP (ORDER BY total_spent) AS top_20_threshold
   FROM customer_spending
)
SELECT cs.customer_id, cs.total_spent
FROM customer_spending cs, percentile p
WHERE cs.total_spent >= p.top_20_threshold;
```

(If your SQL doesn't support PERCENTILE_CONT, use rank-based filtering instead.)

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

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

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

```
WITH film_counts AS (
   SELECT f.film_id, f.title, c.category_id, COUNT(r.rental_id) AS rental_count
   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
)
SELECT film_id, title, rental_count, avg_category_rental
FROM (
    SELECT *,
        AVG(rental_count) OVER (PARTITION BY category_id) AS avg_category_rental
    FROM film_counts
) AS sub
WHERE rental_count < avg_category_rental;
```

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

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

Normalization & 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.

The film table has a special_features column, which contains a comma-separated list (like 'Trailers,Deleted Scenes,Behind the Scenes').

This violates 1NF because it has **multiple values in a single column**.

To normalize:

- Create a new table: film_special_feature

```
CREATE TABLE film_special_feature (
   film_id SMALLINT,
   special_feature VARCHAR(50),
   FOREIGN KEY (film_id) REFERENCES film(film_id)
);
```

- Each special feature will be stored as a separate row instead of as a comma-separated string.

2. Second Normal Form (2NF)

a. Choose a table and describe how you'd check for 2NF & normalize if needed.

Example: Suppose we design a table like:

rental_details (rental_id, film_id, film_title)

- Composite primary key: (rental_id, film_id)
- Non-prime attribute film_title depends only on film_id (part of key).

Violates 2NF: film_title depends on part of the key, not the whole key.

To normalize:

- Remove film_title into separate film table, keep only:

rental_details (rental_id, film_id)

- film table keeps film_id, film_title.

3. Third Normal Form (3NF)

a. Identify a table violating 3NF & fix it.

Example: hypothetical table:

customer_details (customer_id, address_id, city, country)

- customer_id → address_id
- address_id → city
- city → country

Transitive dependency: customer_id → city → country.

Normalize:

- Keep only:

customer_details (customer_id, address_id)

- Separate address table: address_id, city_id
- Separate city table: city_id, country_id
- Separate country table: country_id, country

4. Normalization Process

a. Normalize a table from unnormalized to at least 2NF.

Unnormalized example:

order (order_id, customer_id, customer_name, product_id, product_name)

1. **1NF**: Remove repeating groups, each order-product pair in separate row.
2. **2NF**: Move partial dependencies:
    - **Create customer table: customer_id, customer_name**
    - **Create product table: product_id, product_name**
    - **Keep order table: order_id, customer_id**
    - **Create order_details table: order_id, product_id**

CTEs

5. CTE Basics

a. List actor names and number of films they've acted in

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

6. CTE with Joins

a. Film title, language name, rental rate

```
WITH film_language AS (
   SELECT f.film_id, f.title, f.rental_rate, l.name AS language_name
   FROM film f
   JOIN language l ON f.language_id = l.language_id
)
SELECT * FROM film_language;
```

7. CTE for Aggregation

a. Total revenue by customer

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

8. CTE with Window Functions

a. Rank films by rental duration

```
WITH film_rank AS (
   SELECT film_id, title, rental_duration,
        RANK() OVER (ORDER BY rental_duration DESC) AS duration_rank
   FROM film
)
SELECT * FROM film_rank;
```

9. CTE and Filtering

a. Customers with >2 rentals

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

```
    HAVING COUNT(*) > 2
)
SELECT c.*
FROM customer c
JOIN frequent_customers fc ON c.customer_id = fc.customer_id;
```

10. CTE for Date Calculations

a. Number of rentals per month

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

11. CTE and Self-Join

a. Actor pairs 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
    WHERE fa1.actor_id < fa2.actor_id
)
SELECT ap.film_id, a1.first_name AS actor1_first, a1.last_name AS actor1_last,
      a2.first_name AS actor2_first, a2.last_name AS actor2_last
FROM actor_pairs ap
JOIN actor a1 ON ap.actor1 = a1.actor_id
JOIN actor a2 ON ap.actor2 = a2.actor_id;
```

12. CTE for Recursive Search

a. Find employees reporting to specific manager (e.g., manager_id=1)

```
WITH RECURSIVE subordinates AS (
    SELECT staff_id, first_name, last_name, reports_to
    FROM staff
    WHERE staff_id = 1 -- starting manager
```

```
    UNION ALL
    SELECT s.staff_id, s.first_name, s.last_name, s.reports_to
    FROM staff s
    JOIN subordinates sub ON s.reports_to = sub.staff_id
)
SELECT * FROM subordinates;
```