SQL FUNDAMENTALS: CONCEPTS, QUERIES, AND BEST PRACTICES

CONSTRAINTS IN SQL: DEFINITION AND IMPORTANCE

In SQL, **constraints** are rules applied to table columns to enforce data integrity and ensure accuracy, consistency, and reliability within a relational database. They prevent invalid data entry and maintain meaningful relationships between tables.

COMMON TYPES OF CONSTRAINTS

- **PRIMARY KEY:** Uniquely identifies each record in a table and cannot contain NULL values.

  ```
  emp_id INT PRIMARY KEY NOT NULL
  ```

  Primary keys must be unique and always have a value, hence cannot be NULL.

- **FOREIGN KEY:** Enforces referential integrity by linking to a primary key in another table.

  ```
  department_id INT,
  FOREIGN KEY (department_id) REFERENCES
  departments(dept_id)
  ```

- **UNIQUE:** Ensures all values in the column are distinct.

  ```
  email VARCHAR(255) UNIQUE
  ```

- **NOT NULL:** Prevents NULL values, enforcing mandatory data entry.

```
emp_name VARCHAR(100) NOT NULL
```

- **CHECK:** Defines a condition that values must satisfy.

```
age INT CHECK (age >= 18)
```

Importance of NOT NULL: It guarantees that essential fields always have valid data, which is critical for data quality and reliable query results.

EXAMPLE SCHEMA ILLUSTRATING PRIMARY AND FOREIGN KEYS

Consider two tables: `employees` and `departments`.

```
CREATE TABLE departments (
  dept_id INT PRIMARY KEY,
  dept_name VARCHAR(50) NOT NULL
);

CREATE TABLE employees (
  emp_id INT PRIMARY KEY,
  emp_name VARCHAR(100) NOT NULL,
  department_id INT,
  FOREIGN KEY (department_id) REFERENCES
departments(dept_id)
);
```

This structure ensures each employee is linked to an existing department, preserving relational integrity.

CREATING SQL TABLES WITH CONSTRAINTS: A PRACTICAL EXAMPLE

Below is the SQL `CREATE TABLE` command for creating an `employees` table with various constraints enforcing data integrity:

```
CREATE TABLE employees (
  emp_id INTEGER PRIMARY KEY NOT NULL,
```

```
  emp_name TEXT NOT NULL,
  age INTEGER CHECK (age >= 18),
  email TEXT UNIQUE,
  salary DECIMAL DEFAULT 30000
);
```

EXPLANATION OF CONSTRAINTS

- **emp_id INTEGER PRIMARY KEY NOT NULL:** This defines `emp_id` as the primary key, uniquely identifying each employee and disallowing `NULL` values.
- **emp_name TEXT NOT NULL:** Ensures every employee record has a name; `NOT NULL` prevents omission.
- **age INTEGER CHECK (age >= 18):** The `CHECK` constraint restricts age entries to 18 or older, enforcing a business rule.
- **email TEXT UNIQUE:** Guarantees that email addresses are unique across employees.
- **salary DECIMAL DEFAULT 30000:** If no salary is provided at insertion, it defaults to 30,000.

ALTERING CONSTRAINTS ON EXISTING TABLES

You can modify constraints using `ALTER TABLE`. For example:

```
-- Add a CHECK constraint to ensure salary >= 10000
ALTER TABLE employees
ADD CONSTRAINT chk_salary CHECK (salary >= 10000);

-- Drop the UNIQUE constraint on email (assuming
constraint named 'email_unique')
ALTER TABLE employees
DROP CONSTRAINT email_unique;
```

HANDLING CONSTRAINT VIOLATIONS

Inserting or updating data that violates constraints will result in errors. For example, attempting to insert an employee aged 16:

```
INSERT INTO employees (emp_id, emp_name, age, email,
salary)
VALUES (1, 'John Doe', 16, 'john@example.com', 35000);
-- Error: CHECK constraint failed: age must be at least
18
```

Similarly, trying to insert a duplicate `emp_id` will cause:

```
-- Error: UNIQUE constraint failed: emp_id
```

BASIC CRUD OPERATIONS IN SQL

The fundamental operations to manage data in SQL databases are often summarized as CRUD: Create, Read, Update, and Delete. These operations enable interaction with tables such as `employees` or `products`.

CREATE (INSERT)

To add new records, use the `INSERT INTO` statement. For example, adding a new employee:

```
INSERT INTO employees (emp_id, emp_name, age, email,
salary)
VALUES (101, 'Alice Smith', 28, 'alice@example.com',
32000);
```

READ (SELECT)

Retrieving data is done using `SELECT`. Filtering can be applied with `WHERE`:

```
SELECT emp_name, email FROM employees WHERE age >= 30;
```

UPDATE

To modify existing records, use `UPDATE` with a condition to target specific rows:

```
UPDATE employees SET salary = 35000 WHERE emp_id = 101;
```

DELETE

Records can be removed using `DELETE`. Constraints like foreign keys may prevent deletions if dependent data exists:

```
DELETE FROM employees WHERE emp_id = 101;
```

Attempting to delete an employee referenced by another table with a foreign key constraint will result in an error, protecting data integrity.

USING SQL JOINS: INNER, LEFT, RIGHT, AND FULL

SQL JOINs are used to combine rows from two or more tables based on related columns, allowing for flexible data retrieval. The most common types of JOINs include **INNER JOIN**, **LEFT JOIN**, **RIGHT JOIN**, and **FULL OUTER JOIN**. Each serves a specific purpose in how matched and unmatched rows are included in the result set.

EXAMPLE TABLES

Consider two tables:

```
Students
+------------+--------------+
| student_id | student_name |
+------------+--------------+
| 1          | Alice        |
| 2          | Bob          |
| 3          | Charlie      |
+------------+--------------+

Classes
+----------+------------+
| class_id | class_name |
+----------+------------+
| 1        | Math       |
```

```
| 2          | History    |
| 4          | Science    |
+----------+------------+
```

INNER JOIN

Returns only rows with matching keys in both tables.

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

Result will include only rows where `student_id` equals `class_id` (if this example represents such a relationship).

LEFT JOIN

Returns all rows from the left table and matching rows from the right table. Non-matching right-side rows return NULL.

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

RIGHT JOIN

Returns all rows from the right table and matching rows from the left table. Non-matching left-side rows return NULL.

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

FULL OUTER JOIN

Returns all rows where there is a match in one of the tables. Rows without a match in either table show NULL for the missing side.

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

USE CASE SUMMARY

- **INNER JOIN:** Retrieve only related data where matching keys exist in both tables.
- **LEFT JOIN:** Get all records from the left table, with matching data from the right when available.
- **RIGHT JOIN:** Get all records from the right table, with matching data from the left when available.
- **FULL OUTER JOIN:** Combine all rows from both tables, showing matches and unmatched records on either side.

VISUAL REPRESENTATION

Consider two overlapping circles representing the tables:

- **INNER JOIN** shows only the intersection.
- **LEFT JOIN** shows the entire left circle plus intersection.
- **RIGHT JOIN** shows the entire right circle plus intersection.
- **FULL OUTER JOIN** shows both circles entirely, including overlaps and exclusives.

AGGREGATION FUNCTIONS IN SQL

Aggregation functions in SQL are used to summarize or compute values over a set of rows. The most common aggregate functions include:

- **COUNT()**: Returns the number of rows.
- **AVG()**: Calculates the average (mean) of numeric values.
- **SUM()**: Computes the total sum of numeric values.
- **MIN()**: Finds the minimum value.
- **MAX()**: Finds the maximum value.

These functions are often used with the `GROUP BY` clause to aggregate data by categories. For example:

```
SELECT p.product_name, SUM(s.quantity) AS total_sales
FROM products p
INNER JOIN sales s ON p.product_id = s.product_id
GROUP BY p.product_name;
```

This query calculates total sales per product by summing the quantities sold.

Another example computes the average salary within each department:

```
SELECT department_id, AVG(salary) AS avg_salary
FROM employees
GROUP BY department_id;
```

To filter aggregated results, `HAVING` is used:

```
SELECT department_id, COUNT(*) AS employee_count
FROM employees
GROUP BY department_id
HAVING COUNT(*) > 5;
```

This returns departments with more than 5 employees.

WINDOW FUNCTIONS FOR ADVANCED ANALYTICS

SQL **window functions** such as `RANK()` and `ROW_NUMBER()` enable advanced data analysis by performing calculations across a set of rows related to the current row without collapsing the result set. Unlike aggregate functions used with `GROUP BY`, window functions preserve each individual row while providing summary or ranking information.

HOW WINDOW FUNCTIONS WORK

Window functions operate over partitions defined by `PARTITION BY` and order rows within these partitions using `ORDER BY`. The general syntax is:

```
function_name() OVER (PARTITION BY column1 ORDER BY
column2)
```

This allows computations such as ranking, running totals, or assigning sequential row numbers to be done within groups.

EXAMPLE QUERIES

Ranking customers by total spending:

```
SELECT customer_id, total_spent,
       RANK() OVER (ORDER BY total_spent DESC) AS
spending_rank
FROM customers;
```

Calculating a running total of sales:

```
SELECT sale_date, amount,
       SUM(amount) OVER (ORDER BY sale_date) AS
running_total
FROM sales;
```

Assigning row numbers grouped by product category:

```
SELECT category, product_name,
       ROW_NUMBER() OVER (PARTITION BY category ORDER BY
product_name) AS row_num
FROM products;
```

ILLUSTRATION

For example, ranking customers by total rental spending in a rental database assigns equal ranks to tied values, while `ROW_NUMBER()` differentiates rows uniquely even if values are tied.

NORMALIZATION: CONCEPTS AND PRACTICAL APPLICATION

Database **normalization** is a systematic process to organize data in tables to reduce redundancy and improve data integrity. It involves structuring tables

and their relationships according to defined normal forms, primarily 1NF, 2NF, and 3NF.

FIRST NORMAL FORM (1NF)

A table is in 1NF if it meets the following conditions:

- All columns contain atomic (indivisible) values.
- There are no repeating groups or arrays.

Example Violation:

```
Orders
+---------+------------+------------------+
| orderID | customerID | productIDs       |
+---------+------------+------------------+
| 101     | 1          | 10, 20, 30       |
+---------+------------+------------------+
```

The `productIDs` column holds multiple values, violating 1NF.

SECOND NORMAL FORM (2NF)

To achieve 2NF, the table must first be in 1NF, and all non-key columns must depend on the whole primary key, eliminating partial dependencies (where a non-key attribute depends on only part of a composite key).

Example Violation:

```
OrderDetails
+---------+-----------+-------------+
| orderID | productID | productName |
+---------+-----------+-------------+
| 101     | 10        | Widget A    |
+---------+-----------+-------------+
```

If the primary key is (orderID, productID), the `productName` depends only on `productID`, causing redundancy.

THIRD NORMAL FORM (3NF)

3NF requires a table to be in 2NF and have no transitive dependencies, where non-key attributes depend on other non-key attributes.

Example Violation:

```
Employees
+--------+------------+------------+
| empID  | deptID     | deptName   |
+--------+------------+------------+
| 1      | 10         | Sales      |
+--------+------------+------------+
```

Here, `deptName` depends on `deptID`, not directly on `empID`. To normalize, separate department data into its own table.

NORMALIZATION WALKTHROUGH

Consider this unnormalized table storing student courses:

```
StudentsCourses
+----------+------------+------------------+
| studentID| studentName| courses          |
+----------+------------+------------------+
| 1        | Alice      | Math, History    |
+----------+------------+------------------+
```

Step 1 (1NF): Break multi-valued `courses` into separate rows:

```
studentID | studentName | course
----------+-------------+---------
1         | Alice       | Math
1         | Alice       | History
```

Step 2 (2NF): Remove partial dependencies; if primary key is (studentID, course), attributes fully depend on both or just on `studentID`.

Step 3 (3NF): Eliminate transitive dependencies by separating student details and course enrollment into two tables:

```
Students
+----------+-------------+
| studentID| studentName |
+----------+-------------+

Enrollments
+----------+---------+
| studentID| course  |
+----------+---------+
```

This approach ensures that updates, insertions, and deletions can be performed without introducing inconsistencies or redundancy.

COMMON TABLE EXPRESSIONS (CTES) IN SQL

Common Table Expressions (CTEs) are a powerful SQL feature that improve query readability, modularity, and manage complexity by defining temporary named result sets. CTEs are declared using the `WITH` clause at the start of a query, enabling better organization by breaking down complex operations into simpler building blocks.

BASIC SYNTAX

```
WITH cte_name AS (
      SELECT ...
)
SELECT * FROM cte_name;
```

BENEFITS OF CTES

- **Readability:** Separates complex logic into understandable parts.
- **Reusability:** CTEs can be referenced multiple times within the main query.
- **Complexity management:** Helps write recursive and hierarchical queries.

EXAMPLE 1: SIMPLE CTE FOR ACTOR FILM COUNTS

```
WITH ActorFilmCount 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 first_name, last_name, film_count
FROM ActorFilmCount
ORDER BY film_count DESC;
```

EXAMPLE 2: JOINING FILM AND LANGUAGE TABLES USING CTE

```
WITH FilmLanguage AS (
  SELECT f.film_id, f.title, l.name AS language_name,
f.rental_rate
  FROM film f
  INNER JOIN language l ON f.language_id = l.language_id
)
SELECT title, language_name, rental_rate
FROM FilmLanguage
WHERE rental_rate > 2.00;
```

EXAMPLE 3: CTE WITH WINDOW FUNCTION TO RANK FILMS BY RENTAL DURATION

```
WITH RankedFilms AS (
  SELECT title, rental_duration,
         RANK() OVER (ORDER BY rental_duration DESC) AS
rental_rank
  FROM film
)
SELECT title, rental_duration, rental_rank
FROM RankedFilms
WHERE rental_rank <= 5;
```

EXAMPLE 4: RECURSIVE CTE TO FIND EMPLOYEES REPORTING TO A MANAGER

Assuming an `employees` table with `emp_id` and `reports_to` columns:

```
WITH RECURSIVE EmployeeHierarchy AS (
  SELECT emp_id, emp_name, reports_to
  FROM employees
  WHERE emp_id = 100 -- starting manager ID

  UNION ALL

  SELECT e.emp_id, e.emp_name, e.reports_to
  FROM employees e
  INNER JOIN EmployeeHierarchy eh ON e.reports_to =
eh.emp_id
)
SELECT emp_id, emp_name, reports_to
FROM EmployeeHierarchy;
```

USING CTES FOR FILTERING AND AGGREGATION

```
WITH FrequentRenters AS (
  SELECT customer_id, COUNT(rental_id) AS rental_count
  FROM rental
  GROUP BY customer_id
  HAVING COUNT(rental_id) > 2
)
SELECT c.customer_id, c.first_name, c.last_name,
fr.rental_count
FROM FrequentRenters fr
JOIN customer c ON fr.customer_id = c.customer_id;
```

SUMMARY AND BEST PRACTICES FOR SQL FUNDAMENTALS

This report has covered essential SQL concepts including constraints, CRUD operations, various JOIN types, aggregation functions, window functions,

normalization, and common table expressions (CTEs). Properly defining constraints such as **PRIMARY KEY** and **FOREIGN KEY** is crucial for maintaining data integrity and ensuring consistent relationships.

Best practices include writing clear, efficient SQL queries by:

- Using explicit JOINs to optimize data retrieval and avoid Cartesian products.
- Leveraging aggregation and window functions for insightful analytics without losing row-level detail.
- Applying normalization rules (1NF, 2NF, 3NF) to reduce redundancy and improve database structure.
- Utilizing CTEs to modularize complex queries, enhancing readability and maintainability.

Students are encouraged to test queries incrementally and thoughtfully design schemas with indexing and constraints to improve performance and reliability.