**NAME :- NILESH SINHA ,** **Batch :- Data Analytics Course**

**Email.id :- nileshrick1@gamil.com**

# -----: SQL Basics Assignment Questions :----

**Q2. Explain the purpose of constraints and how they help maintain data integrity in a database. Provide examples of common types of constraints.**

**=>** Integrity Constraints are the protocols that a table's data columns must follow. These are used to restrict the types of information that can be entered into a table. This means that the data in the database is accurate and reliable. You may apply integrity Constraints at the column or table level. The table-level Integrity constraints apply to the entire table, while the column level constraints are only applied to one column.

Here are some common types of constraints:

- **Primary key**

Uniquely identifies each record in a table, and can be made up of one or more columns.

- **Foreign key**

Ensures that a record is not loaded into a table if the foreign key values are not found in the primary key values of the parent table.

- **UNIQUE**

Prevents duplicate values from being entered into a column. This is useful for fields like email addresses or user IDs.

- **NOT NULL**

Prevents blank or null values from being entered into a column.

- **Check**

Specifies a range of rules for the contents of a table. A search condition must be satisfied for all rows in the table

**Q3. Why would you apply the NOT NULL constraint to a column? Can a primary key contain NULL values? Justify your answer.**

**=>** Why would you apply the NOT NULL constraint to a column?

= In SQL, the NOT NULL constraint in a column means that the column cannot store NULL values.

Can a primary key contain NULL values?

= The primary key cannot have NULL and duplicate values. It is used to add integrity to the table. In the case of a primary key, both Duplicate and NULL values are not valid.

**Q4. Explain the steps and SQL commands used to add or remove constraints on an existing table. Provide an example for both adding and removing a constraint.**

**=>** The **ALTER TABLE statement in SQL** is used to add, remove, or modify columns in an existing table. The ALTER TABLE statement is also used to add and remove various constraints on existing tables.

**ADD – To add a new column to the table:**

```
ALTER TABLE table_name
ADD column_name datatype;
```

**DROP – To delete an existing column from the table:**

ALTER TABLE table_name
DROP COLUMN column_name;

**Q5. Explain the consequences of attempting to insert, update, or delete data in a way that violates constraints. Provide an example of an error message that might occur when violating a constraint**.

**=>** There are mainly three operations that have the ability to change the state of relations, these modifications are given below:

1. **Insert –**
   To insert new tuples in a relation in the database.
2. **Delete –**
   To delete some of the existing relation on the database.
3. **Update (Modify) –**
   To make changes in the value of some existing tuples.

The ORA-00001 error message is indicative of this pattern of thinking. The ORA-00001 message is triggered when a unique constraint has been violated. Essentially the user causes the error when trying to execute an INSERT or UPDATE statement that has generated a duplicate value in a restricted field.

# SQL COMMANDS :------

**Q1. Identify the primary keys and foreign keys in maven movies db. Discuss the differences.**

**=> Primary vs. Foreign Keys – Key Differences**

| Aspect | Primary Key | Foreign Key |
|---|---|---|
| **Definition** | Uniquely identifies a record within its own table. | Establishes a relationship between two tables. |
| **Uniqueness** | Must have unique values for each row. | Can have duplicate values across rows. |
| **Null Values** | Cannot contain NULL values. | Can contain NULL values, depending on the relationship. |
| **Purpose** | Ensures that each record in the table is unique. | Ensures data integrity by linking records across tables. |
| **Example (in Movies DB)** | MovieID in Movies, ActorID in Actors | MovieID in Movie_Actors, ActorID in Movie_Actors, DirectorID in Movie_Directors |
| **Enforcement** | Ensures each record can be uniquely identified. | Ensures that a value in one table matches a value in another (or is NULL). |

# Normalisation & CTE :-----

**Q1. . First Normal Form (1NF):**

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

**=>** To bring this structure into **1NF**, we need to eliminate repeating groups and ensure each field contains only a single value.

- **Option 1**: Instead of storing multiple actors' names or IDs in a single column, create a new table to represent the relationship between films and actors. For example, we could create a **film_actor** junction table with the following columns:
    - film_id (foreign key referencing film.film_id)
    - actor_id (foreign key referencing actor.actor_id)

The new structure would look like this:

- film table: Only includes unique details for each film (e.g., film_id, title, etc.).
- actor table: Contains details about each actor (e.g., actor_id, name).
- film_actor table: Contains combinations of film_id and actor_id, representing the many-to-many relationship between films and actors.

## Q2. Second Normal Form (2NF):

### a. 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.

**=> Normalization to 2NF:**

To bring the table into **2NF**, we must remove the partial dependency. The steps to do so are as follows:

1. **Create a new table for customer information** to remove the dependency between last_name and first_name from the rental table.
    - We will create a **customer** table containing customer_id, last_name, and first_name.
2. **Modify the rental table**:
    - The rental table should now only store rental-specific information, which will include rental_id, rental_date, inventory_id, customer_id (as a foreign key), return_date, and staff_id.

**Explanation of Normalization:**

- After the normalization, the rental table is in **2NF** because:
    - All non-key attributes (such as rental_date, inventory_id, return_date, staff_id) are now fully dependent on the primary key (rental_id).
    - The attributes last_name and first_name, which were partially dependent on customer_id, have been moved to a new **customer** table, ensuring that there are no partial dependencies remaining.

## Q3. Third Normal Form (3NF):

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

**=> Steps to Normalize the Table to 3NF:**

To bring the **staff** table into **3NF**, we need to eliminate these transitive dependencies by creating separate tables for the entities that are transitively dependent. Here's how we can proceed:

1. **Create a address table**: Since **address_id** references the **address** table, we need to ensure that all address-related data (e.g., street, city, country) is moved to the **address** table to eliminate the transitive dependency between staff and address-related attributes.

- New address table:

| address_id | address | city | country_id |
|---|---|---|---|
| 5 | '123 Elm St.' | 'New York' | 1 |
| 6 | '456 Oak Ave.' | 'Los Angeles' | 2 |

2. **Create a store table**: If **store_id** references a **store** table, we should move store-specific details like **store_name** and **store_address** to the **store** table, and only store the foreign key (store_id) in the **staff** table.
    - New store table:

| store_id | store_name | store_address |
|---|---|---|
| 1 | 'Main Store' | '123 Broadway' |
| 2 | 'Sub Store' | '456 Market St' |

3. **Modify the staff table**: After creating the address and store tables, the **staff** table should only contain references to those tables via foreign keys and staff-specific information (e.g., first_name, last_name, email, etc.).
    - New staff table:

| staff_id | first_name | last_name | address_id | email | store_id | active | username | password |
|---|---|---|---|---|---|---|---|---|
| 1 | 'Mike' | 'Hughes' | 5 | 'mike@example.com' | 1 | 1 | 'mikeh' | 'pass1' |
| 2 | 'Sarah' | 'Williams' | 6 | 'sarah@example.com' | 1 | 1 | 'sarahw' | 'pass2' |

Now, **staff** no longer has any transitive dependencies. All address-related information is stored in the address table, and store-specific information is stored in the store table.