SQL assignment DA SEPT 24 batch

Ritik Gupta
Gupritik.786@gmail.com

1. Create a table called employees with the following structure?
   : emp_id (integer, should not be NULL and should be a primary key) Q
   : emp_name (text, should not be NULL) Q
   : age (integer, should have a check constraint to ensure the age is at least 18) Q
   : email (text, should be unique for each employee) Q
   : salary (decimal, with a default value of 30,000). Write the SQL query to create the above table with all constraints.

```
create table employees
(emp_id int not null,
emp_name text not null,
age int check(age >=18),
email int unique ,
salary dec default(30000));
```

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

   Constraints in a database are rules enforced on the data in tables to maintain accuracy, consistency, and integrity. They ensure that only valid and meaningful data is stored, reducing the risk of data corruption or errors.

   Purpose of Constraints

1. **Data Integrity**: Constraints ensure the accuracy and reliability of data.
   - Example: A constraint can ensure a person's age is always a non-negative number.
2. **Consistency**: They enforce uniformity in data entry and relationships between tables.

   - Example: Ensuring that a product ID entered in a sales table exists in the products table.
3. **Error Prevention**: They prevent invalid data from being entered into the database.
   - Example: Rejecting entries where a required field is missing.
4. **Business Rules Enforcement**: Constraints can enforce specific rules relevant to an organization.
   - Example: A salary must be greater than a minimum wage.

Common Types of Constraints

1. **Primary Key Constraint**
   - **Purpose**: Ensures each row in a table has a unique identifier.
   - **Example**: A StudentID column in a Students table must have unique values and cannot be NULL.
   - **SQL Syntax**:

```
CREATE TABLE Students (
   StudentID INT PRIMARY KEY,
   Name VARCHAR(50)
);
```

2. **Foreign Key Constraint**
   - **Purpose**: Maintains referential integrity by ensuring that a value in one table corresponds to a value in another.
   - **Example**: A StudentID in a Enrollments table must exist in the Students table.
   - **SQL Syntax**:

```
CREATE TABLE Enrollments (
   EnrollmentID INT PRIMARY KEY,
   StudentID INT,
   FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
);
```

3. **Unique Constraint**
   - **Purpose**: Ensures that all values in a column are unique, allowing NULL values.
   - **Example**: An email column in a Users table must have unique values.
   - **SQL Syntax**:

```
CREATE TABLE Users (
   UserID INT PRIMARY KEY,
   Email VARCHAR(100) UNIQUE
);
```

4. **Not Null Constraint**
   - **Purpose**: Ensures that a column cannot have NULL values.
   - **Example**: A Name column in a Employees table must always have a value.
   - **SQL Syntax**:

```
CREATE TABLE Employees (
   EmployeeID INT PRIMARY KEY,
   Name VARCHAR(50) NOT NULL
);
```

5. **Check Constraint**
   - **Purpose**: Ensures that all values in a column satisfy a specific condition.
   - **Example**: A Salary column in a Jobs table must be greater than 0.
   - **SQL Syntax**:

```
CREATE TABLE Jobs (
   JobID INT PRIMARY KEY,
   Salary DECIMAL(10, 2),
   CHECK (Salary > 0)
);
```

6. **Default Constraint**
   - **Purpose**: Assigns a default value to a column when no value is provided.
   - **Example**: A Status column in an Orders table defaults to 'Pending'.
   - **SQL Syntax**:

```
CREATE TABLE Orders (
   OrderID INT PRIMARY KEY,
   Status VARCHAR(20) DEFAULT 'Pending');
```

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

Why Apply the NOT NULL Constraint to a Column?
The **NOT NULL** constraint is applied to ensure that a column always contains a value and cannot be left empty. This is crucial when the presence of data in that column is essential for the database to function correctly or to maintain data integrity.
Reasons to Use NOT NULL:

1. **Data Completeness**: Ensures critical information is always recorded.
   - Example: A Name field in a Users table must not be NULL to identify users.
2. **Avoid Errors in Queries**: Prevents issues when performing operations or calculations that assume a value is present.
   - Example: Calculating total = quantity * price would fail if price is NULL.
3. **Consistency in Data Relationships**: Supports referential integrity, particularly in foreign key relationships.
   - Example: A foreign key referencing a parent table cannot be NULL unless explicitly designed for optional relationships.

Can a Primary Key Contain NULL Values?
No, a primary key cannot contain NULL values.

Justification:

1. **Uniqueness Requirement**: The primary key uniquely identifies each row in a table. A NULL value represents "unknown" or "missing" data, which cannot serve as a unique identifier.
   - If NULL were allowed, multiple rows could have NULL values, violating the uniqueness rule.
2. **Logical Integrity**: The primary key is often used to reference rows in other tables via foreign keys. If a primary key contained NULL, the referenced rows would become ambiguous or invalid.
3. **SQL Standard**: According to SQL standards, the primary key constraint implicitly enforces both **NOT NULL** and **UNIQUE** constraints.

Example:

The following would result in an error:

```
CREATE TABLE Students (
   StudentID INT PRIMARY KEY,
   Name VARCHAR(50)
);

-- Attempt to insert a NULL value into the primary key column
INSERT INTO Students (StudentID, Name) VALUES (NULL, 'John Doe'); -- Error: Primary
key cannot be NULL
```

In conclusion:

- Apply the **NOT NULL** constraint to ensure essential data is always provided.
- A **primary key** cannot contain NULL values because it must uniquely and unambiguously identify each record in the table.

4. 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.

Steps and SQL Commands for Adding or Removing Constraints on an Existing Table

Adding Constraints

1. **Identify the Table and Column**: Determine the table and column(s) where you want to add the constraint.
2. **Use the ALTER TABLE Command**: Use the ALTER TABLE statement with the appropriate syntax to add the constraint.
3. **Specify the Constraint Type**: Define the type of constraint (NOT NULL, UNIQUE, CHECK, etc.) and the rules it enforces.

Example: Adding a Constraint

Suppose we have a table Employees without any constraints on the Email column.

```
CREATE TABLE Employees (
 EmployeeID INT PRIMARY KEY,
 Name VARCHAR(50),
 Email VARCHAR(100));
```

To add a **UNIQUE** constraint on the Email column:

```
ALTER TABLE Employees
ADD CONSTRAINT UniqueEmail UNIQUE (Email);
```

To add a **CHECK** constraint to ensure EmployeeID is greater than 0:

```
ALTER TABLE Employees
ADD CONSTRAINT CheckEmployeeID CHECK (EmployeeID > 0);
```

Removing Constraints

1. **Identify the Constraint Name**: Determine the name of the constraint you want to remove. You can find constraint names by querying the database metadata (e.g., INFORMATION_SCHEMA in SQL Server).
2. **Use the ALTER TABLE Command**: Use the ALTER TABLE statement with the DROP CONSTRAINT clause.

Example: Removing a Constraint

To remove the **UNIQUE** constraint on the Email column:

```
ALTER TABLE Employees
DROP CONSTRAINT UniqueEmail;
```

To remove the **CHECK** constraint on EmployeeID:

```
ALTER TABLE Employees
DROP CONSTRAINT CheckEmployeeID;
```

5. 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.

Consequences of Violating Constraints

When you attempt to insert, update, or delete data in a way that violates constraints in a database, the database management system (DBMS) enforces the rules defined by the constraints and prevents the operation. This ensures data integrity but results in an error.

Common Scenarios and Their Consequences

1. **Violation of a NOT NULL Constraint**

- **Attempt**: Inserting or updating a row with a NULL value in a column defined as NOT NULL.
- **Consequence**: The operation fails because the column requires a value.
- **Example**: If the Name column in an Employees table has a NOT NULL constraint:

```
INSERT INTO Employees (EmployeeID, Name) VALUES (1, NULL);
```
Error Message (MySQL):

```
ERROR 1048 (23000): Column 'Name' cannot be null
```

2. **Violation of a UNIQUE Constraint**
   - **Attempt**: Inserting or updating a row with a duplicate value in a column defined as UNIQUE.
   - **Consequence**: The operation fails because duplicate values are not allowed.
   - **Example**: If the Email column in an Users table has a UNIQUE constraint:

```
INSERT INTO Users (UserID, Email) VALUES (1, 'example@example.com');
INSERT INTO Users (UserID, Email) VALUES (2, 'example@example.com');
```
Error Message (PostgreSQL):

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

3. **Violation of a PRIMARY KEY Constraint**
   - **Attempt**: Inserting or updating a row with a duplicate or NULL value in a primary key column.
   - **Consequence**: The operation fails because a primary key must be unique and cannot be NULL.
   - **Example**:

```
INSERT INTO Employees (EmployeeID, Name) VALUES (1, 'Alice');
INSERT INTO Employees (EmployeeID, Name) VALUES (1, 'Bob');
```
Error Message (SQL Server):

```
Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'PK_Employees'. Cannot insert duplicate key in
object 'dbo.Employees'.
```

4. **Violation of a FOREIGN KEY Constraint**
   - **Attempt**: Inserting or updating a row with a value in a foreign key column that does not exist in the referenced table, or deleting a referenced row in the parent table.
   - **Consequence**: The operation fails to preserve referential integrity.
   - **Example**:

INSERT INTO Orders (OrderID, CustomerID) VALUES (1, 999); -- Assuming 999 does not exist in the Customers table

Error Message (MySQL):

ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails (`Orders`, CONSTRAINT `fk_customer` FOREIGN KEY (`CustomerID`) REFERENCES `Customers` (`CustomerID`))

5. **Violation of a CHECK Constraint**
   - **Attempt**: Inserting or updating a row with a value that does not satisfy the condition defined in a CHECK constraint.
   - **Consequence**: The operation fails because the value does not meet the defined rule.
   - **Example**: If a Salary column has a CHECK constraint ensuring it must be greater than 0:

   INSERT INTO Jobs (JobID, Salary) VALUES (1, -500);

   Error Message (Oracle):
   ORA-02290: check constraint (SCHEMA.CHECK_SALARY) violated

6. You created a products table without constraints as follows:
   CREATE TABLE products (
     product_id INT,
   product_name VARCHAR(50),
   price DECIMAL(10, 2));
   Now, you realise that?
    : The product_id should be a primary key Q
   : The price should have a default value of 50.00

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

   ALTER TABLE products
   ALTER COLUMN price SET DEFAULT 50.00;

7. You have two tables:
   SELECT
      students.student_name,
      classes.class_name
   FROM
      students

```
INNER JOIN
   classes
ON
   students.class_id = classes.class_id;
```

8. Consider the following three tables:

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

9. Given the following tables:

```
SELECT
   products.product_id,
   products.product_name,
   SUM(sales.amount) AS sales_amt
FROM
   sales
      INNER JOIN
   products ON sales.product_id = products.product_id
GROUP BY products.product_id , products.product_name;
```

10. You are given three tables:

```
SELECT
   orders.customer_id,
   customers.customer_name,
   SUM(order_details.quantity)
```

```
FROM
    orders
        INNER JOIN
    order_details ON orders.order_id = order_details.order_id
        INNER JOIN
    customers ON customers.customer_id = orders.customer_id
GROUP BY orders.customer_id , customers.customer_name;
```

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.

In the Sakila database, a table that could potentially violate **First Normal Form (1NF)** is the address table. Specifically, consider a hypothetical case where the address table might have a column named phone_numbers that stores multiple phone numbers as a comma-separated list.

This violates 1NF because:

- **1NF Rule**: All columns must contain atomic (indivisible) values. Storing multiple values in a single column (e.g., a list of phone numbers) violates this rule.

To normalize it ;

```
CREATE TABLE phone (

  phone_id INT AUTO_INCREMENT PRIMARY KEY,

  address_id INT NOT NULL,

  phone_number VARCHAR(15) NOT NULL,

  FOREIGN KEY (address_id) REFERENCES address(address_id));
```

2. 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.

Understanding 2NF Requirements

A table is in **2NF** if:

1. It is already in **First Normal Form (1NF)** (i.e., it has atomic values and no repeating groups).
2. Every non-prime attribute (non-key column) is fully functionally dependent on the entire primary key and not just part of it.

Composite Key Assumption

If the table had a composite primary key such as (inventory_id, rental_date), this table would need to be checked for 2NF compliance.

Violations of 2NF

If a non-prime attribute (e.g., staff_id) depends only on a part of the composite primary key (inventory_id) and not the entire composite key (inventory_id, rental_date), it violates 2NF.

Example of Partial Dependency:

- staff_id might be associated with inventory_id because staff manage specific inventories.
- This means staff_id is not dependent on the entire primary key (inventory_id, rental_date), violating 2NF.

Steps to Normalize to 2NF

1. **Identify the Partial Dependency**:
    - Determine which non-prime attributes depend only on part of the composite key. In this case, staff_id depends on inventory_id.
2. **Decompose the Table**:
    - Create a separate table to store the relationship between inventory_id and staff_id.
3. **Revised Tables**:
    - **rental Table**:

rental_id	inventory_id	customer_id	rental_date	return_date
1	100	1	2023-01-01 10:00:00	2023-01-03 15:00:00
2	101	1	2023-01-02 12:00:00	2023-01-04 18:00:00

- **inventory_staff Table**:

inventory_id	staff_id
100	2
101	3

SQL to Normalize

1. **Create the inventory_staff Table**:

```
CREATE TABLE inventory_staff (
   inventory_id INT NOT NULL,
   staff_id INT NOT NULL,
   PRIMARY KEY (inventory_id),
   FOREIGN KEY (inventory_id) REFERENCES inventory(inventory_id),
   FOREIGN KEY (staff_id) REFERENCES staff(staff_id));
```

3. 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.

Third Normal Form (3NF)

A table is in **Third Normal Form (3NF)** if:

1. It is in **Second Normal Form (2NF)**.
2. It has no **transitive dependencies**, meaning that non-prime attributes (non-key attributes) are not dependent on other non-prime attributes.

Identifying Transitive Dependencies

In this table:

- **Primary Key**: film_id
- **Non-prime Attributes**: title, description, release_year, language_id, language_name, rental_duration, rental_rate

There is a **transitive dependency**:

- language_name depends on language_id, not directly on the primary key film_id.
- language_id determines language_name, which causes a transitive dependency: film_id -> language_id -> language_name.

This violates **3NF** because language_name is not directly dependent on film_id but depends on language_id (a non-prime attribute).

Steps to Normalize to 3NF

To normalize the film table to **3NF**, we need to remove the transitive dependency by creating a new table to store the language_id and language_name.

1. **Create a New language Table**:
    - The language table will store language_id and language_name, ensuring that language_name is only dependent on language_id, not on film_id.

```
CREATE TABLE language (
  language_id INT PRIMARY KEY,
  language_name VARCHAR(255) NOT NULL
);
```

2. **Modify the film Table**:
    - Remove language_name from the film table since it is now stored in the language table. The film table will still store language_id as a foreign key.

```
CREATE TABLE film (
  film_id INT PRIMARY KEY,
  title VARCHAR(255) NOT NULL,
  description TEXT,
  release_year YEAR,
  language_id INT,
  rental_duration INT,
  rental_rate DECIMAL(5,2),
  FOREIGN KEY (language_id) REFERENCES language(language_id)
);
```

3. **Populate the language Table**:
    - Insert distinct language_id and language_name combinations from the film table into the language table.

```
INSERT INTO language (language_id, language_name)
SELECT DISTINCT language_id, language_name
FROM film;
```

4. **Update the film Table**:
    - Remove the language_name column from the film table.

ALTER TABLE film DROP COLUMN language_name;

Normalized Tables in 3NF

After normalization, the two tables are:

1. **language Table**:

plaintext

Copy code

```
| language_id | language_name |
|---------|----------|
| 1         | English       |
| 2         | Italian       |
```

2. **film Table**:

plaintext

Copy code

```
| film_id | title           | description   | release_year | language_id | rental_duration | rental_rate |
|------|------------|----------------|----------|---------|------------|---------|
| 1      | The Matrix      | Sci-fi thriller | 1999        | 1          | 7              | 4.99       |
| 2      | The Godfather    | Crime drama      | 1972        | 2          | 5              | 3.99       |
```

Outcome

- The **transitive dependency** (film_id -> language_id -> language_name) has been removed.
- The film table now only contains attributes that are directly dependent on the primary key (film_id), and the language table handles the relationship between language_id and language_name.
- Both tables are now in **Third Normal Form (3NF)**.

4. Normalization Process: a. Take a specific table in Sakila and guide through the process of normalizing it from the initial unnormalized form up to at least 2NF.

Let's walk through the **normalization process** for a specific table in the Sakila database. We will choose the **rental** table for this example and guide you through the steps to normalize it from **unnormalized form** to **Second Normal Form (2NF)**.

Starting with the Unnormalized Form (UNF)

In the unnormalized form, data is typically stored with repeating groups and non-atomic values. For example, let's assume the rental table contains both rental information and customer details in a single row:

rental_id	**inventory_id**	**customer_id**	**customer_name**	**customer_address**	**rental_date**	**return_date**	**staff_id**
1	100	1	John Doe	123 Elm St.	2023-01-01 10:00	2023-01-03 15:00	2
2	101	2	Jane Smith	456 Oak Ave.	2023-01-02 12:00	2023-01-04 18:00	3

In this unnormalized form, we have:

1. **Repeating groups**: Customer details (like customer_name and customer_address) are repeated for each rental.
2. **Non-atomic values**: Customer details should be stored in separate fields to ensure atomicity.

Step 1: Convert to First Normal Form (1NF)

To convert this table into **1NF**, we need to:

- Ensure that all columns contain atomic values (no repeating groups or multi-valued attributes).
- Each record should represent a unique entity, so we'll remove repeating customer data for each rental by separating customer information into another table.

1NF Design

We'll split the customer information into a separate customer table and keep the rental details in the rental table.

1NF Tables:

1. **customer Table**:

customer_id	customer_name	customer_address
1	John Doe	123 Elm St.
2	Jane Smith	456 Oak Ave.

2. **rental Table**:

rental_id	inventory_id	customer_id	rental_date	return_date	staff_id
1	100	1	2023-01-01 10:00:00	2023-01-03 15:00:00	2
2	101	2	2023-01-02 12:00:00	2023-01-04 18:00:00	3

Now, the **rental** table has only atomic values, and the repeating customer data has been moved to the **customer** table. This satisfies **1NF**.

Step 2: Convert to Second Normal Form (2NF)

To convert the tables into **2NF**, we need to:

- Ensure that the table is in **1NF**.
- Remove any **partial dependencies** (where non-prime attributes depend on only part of a composite primary key).

In the **rental** table, the primary key is rental_id, which uniquely identifies each rental. However, in the unnormalized design, customer_id (a non-prime attribute) is dependent on rental_id, and customer-related information like customer_name and customer_address are repeated for every rental.

The **rental** table is in **1NF**, but it violates **2NF** due to the partial dependency:

- customer_name and customer_address depend on customer_id (not the entire primary key, rental_id).

Solution to Achieve 2NF:

We need to separate customer-specific information that depends solely on customer_id into a new table. The rental table should only store rental-related information, referencing customer_id as a foreign key.

2NF Design:

1. **customer Table**:

customer_id	customer_name	customer_address
1	John Doe	123 Elm St.
2	Jane Smith	456 Oak Ave.

2. **rental Table**:

rental_id	inventory_id	customer_id	rental_date	return_date	staff_id

1	100	1	2023-01-01 10:00:00	2023-01-03 15:00:00	2
2	101	2	2023-01-02 12:00:00	2023-01-04 18:00:00	3

Here, the rental table now only stores rental-related data, with customer_id being a foreign key. All customer details are stored in the customer table. The partial dependency has been removed, and the table is now in **2NF**.

Conclusion

- The original unnormalized rental table contained repeating groups and non-atomic values.
- After applying the steps to **1NF**, we separated customer data into the customer table, removing the repeating customer details.
- We then applied **2NF** by removing the partial dependency of customer data on rental_id and putting customer information in its own table.

The table is now normalized to **2NF**, ensuring all non-prime attributes are fully dependent on the primary key.