CS232L – Database Management System

LAB#04
POSTGRESQL CONSTRAINTS

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS232L – Database Management system Lab

Lab 4 – PostgreSQL Constraints

Objective

The objective of this session is to

1. SQL Constraint Introduction
2. The NOT NULL Constraint
3. The UNIQUE Key Constraint
4. The PRIMARY KEY Constraint
5. The FOREIGN KEY Constraint
6. The CHECK Constraint
7. Adding a Constraint
8. Dropping a Constraint
9. Disabling a Constraint
10. Enabling Constraints
11. Viewing Constraints
12. Practice Problem

Instructions

- Open the handout/lab manual in front of a computer in the lab during the session.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering problems, indicate the commands you entered and the output displayed.

PostgreSQL Constraint Introduction

The Postgres Server uses constraints to prevent invalid data entry into tables.

You can use constraints to do the following:

- Enforce rules at the table level whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
- Prevent the deletion of a table if there are dependencies from other tables.

Constraint	**Description**
NOT NULL	Specifies that this column my not contain a null value
UNIQUE	Specifies a column or combination of columns whose values must be unique for all rows in the table
PRIMARY KEY	Uniquely identifies each row of the table
FOREIGN KEY	Establishes and enforces a foreign key relationship between the column and a column of the referenced table
CHECK	Specifies a condition that must be true

Constraints Guidelines

All constraints are stored in the data dictionary. Constraints are easy to reference if you give them a meaningful name. Constraint names must follow the standard object naming rules. Constraints can be defined at the time of table creation or after the table has been created. You can view the constraints defined for a specific table by looking at the USER_CONSTRAINTS data dictionary table.

Syntax:

```
CREATE TABLES table
(column datatype
[column_constraint],
column datatype [column_constraint],
…..
[table_constraint] [……]);
```

- column is the name of the column
- datatype is the column’s datatype and length
- column_constraint is an integrity constraint as part of the column definition
- table_constraint is an integrity constraint as part of the table definition

Constraints are usually created at the same time as the table. Constraints can be added to a table after its creation and also temporarily disabled. Constraints can be defined at one of two levels.

Constraint Level	Description
Column	References a single column and is defined within a specification for the owning column: can define any type of integrity constraint
Table	References one or more columns and is defined separately from the definitions of the columns in the table: can define any constraints except NOT NULL

PostgreSQL Primary Key

A primary key is a column or a group of columns used to identify a row uniquely in a table.

You define primary keys through primary key constraints. Technically, a primary key constraint is the combination of a not-null constraint and a UNIQUE constraint.

A table can have one and only one primary key. It is a good practice to add a primary key to every table.

Define primary key when creating the table

Normally, we add the primary key to a table when we define the table's structure using CREATE TABLE statement.

```
CREATE TABLE TABLE (
	column_1 data_type PRIMARY KEY,
	column_2 data_type,
	…
);
```

The following statement creates a purchase order (PO) header table with the name po_headers.

```
CREATE TABLE po_headers (
	po_no INTEGER PRIMARY KEY,
	vendor_no INTEGER,
	description TEXT,
	shipping_address TEXT
);
```

The po_no is the primary key of the po_headers table, which uniquely identifies purchase order in the po_headers table.

In case the primary key consists of two or more columns, you define the primary key constraint as follows:

```
CREATE TABLE TABLE (
        column_1 data_type,
        column_2 data_type,
        …
          PRIMARY KEY (column_1, column_2)
);
```

For example, the following statement creates the purchase order line items table whose primary key is a combination of purchase order number (po_no) and line item number (item_no).

```
CREATE TABLE po_items (
        po_no INTEGER,
        item_no INTEGER,
        product_no INTEGER,
        qty INTEGER,
        net_price NUMERIC,
        PRIMARY KEY (po_no, item_no)
);
```

If you don't specify explicitly the name for primary key constraint, PostgreSQL will assign a default name to the primary key constraint. By default, PostgreSQL uses table-name_pkey as the default name for the primary key constraint. In this example, PostgreSQL creates the primary key constraint with the name po_items_pkey for the po_items table.

In case you want to specify the name of the primary key constraint, you use CONSTRAINT clause as follows:

```
CONSTRAINT constraint_name PRIMARY KEY(column_1,
column_2,...);
```

Define primary key when changing the existing table structure

It is rare to define a primary key for existing table. In case you have to do it, you can use the ALTER TABLE statement to add a primary key constraint.

```
ALTER TABLE table_name ADD PRIMARY KEY (column_1, column_2);
```

The following statement creates a table named products without defining any primary key.

```
CREATE TABLE products (
        product_no INTEGER,
        description TEXT,
        product_cost NUMERIC
);
```

Suppose you want to add a primary key constraint to the products table, you can execute the following statement:

```
ALTER TABLE products
ADD PRIMARY KEY (product_no);
```

How to add an auto-incremented primary key to an existing table

Suppose, we have a vendors table that does not have any primary key.

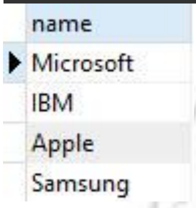
```
CREATE TABLE vendors (name VARCHAR(255));
```

And we add few rows to the vendors table using INSERT statement:

```
INSERT INTO vendors (NAME)
VALUES
       ('Microsoft'),
       ('IBM'),
       ('Apple'),
       ('Samsung');
```

To verify the insert operation, we query data from the vendors table using the following SELECT statement:

```
SELECT
      *
FROM
      vendors;
```



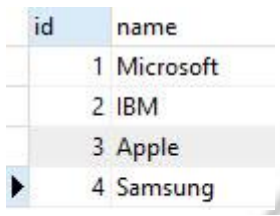
name
Microsoft
IBM
Apple
Samsung

Now, if we want to add a primary key named id into the vendors table and the id field is auto-incremented by one, we use the following statement:

```
ALTER TABLE vendors ADD COLUMN ID SERIAL PRIMARY KEY;
```

Let's check the vendors table again.

```
SELECT
      id,name
FROM
      vendors;
```



id	name
1	Microsoft
2	IBM
3	Apple
4	Samsung

Remove primary key

To remove an existing primary key constraint, you also use the ALTER TABLE statement with the following syntax:

```
ALTER TABLE table_name DROP CONSTRAINT primary_key_constraint;
```

For example, to remove the primary key constraint of the products table, you use the following statement:

```
ALTER TABLE products
DROP CONSTRAINT products_pkey;
```

In this tutorial, you have learned how to add and remove primary key constraints using CREATE TABLE and ALTER TABLE statements.

PostgreSQL Foreign Key

Introduction to PostgreSQL Foreign Key Constraint

A foreign key is a column or a group of columns in a table that reference the primary key of another table.

The table that contains the foreign key is called the referencing table or child table. And the table referenced by the foreign key is called the referenced table or parent table.

A table can have multiple foreign keys depending on its relationships with other tables.

In PostgreSQL, you define a foreign key using the foreign key constraint. The foreign key constraint helps maintain the referential integrity of data between the child and parent tables.

A foreign key constraint indicates that values in a column or a group of columns in the child table equal the values in a column or a group of columns of the parent table.

PostgreSQL foreign key constraint syntax

The following illustrates a foreign key constraint syntax:

```
[CONSTRAINT fk_name]
   FOREIGN KEY(fk_columns)
   REFERENCES parent_table(parent_key_columns)
   [ON DELETE delete_action]
   [ON UPDATE update_action]
```

In this syntax:

- First, specify the name for the foreign key constraint after the CONSTRAINT keyword. The CONSTRAINT clause is optional. If you omit it, PostgreSQL will assign an auto-generated name.
- Second, specify one or more foreign key columns in parentheses after the FOREIGN KEY keywords.
- Third, specify the parent table and parent key columns referenced by the foreign key columns in the REFERENCES clause.
- Finally, specify the delete and update actions in the ON DELETE and ON UPDATE clauses.

The delete and update actions determine the behaviors when the primary key in the parent table is deleted and updated. Since the primary key is rarely updated, the ON UPDATE action is not often used in practice. We'll focus on the ON DELETE action.

PostgreSQL supports the following actions:

- SET NULL
- SET DEFAULT
- RESTRICT
- NO ACTION
- CASCADE

PostgreSQL foreign key constraint examples

The following statements create the customers and contacts tables:

```
DROP TABLE IF EXISTS customers;
DROP TABLE IF EXISTS contacts;

CREATE TABLE customers(
   customer_id INT GENERATED ALWAYS AS IDENTITY,
   customer_name VARCHAR(255) NOT NULL,
   PRIMARY KEY(customer_id)
);

CREATE TABLE contacts(
   contact_id INT GENERATED ALWAYS AS IDENTITY,
   customer_id INT,
   contact_name VARCHAR(255) NOT NULL,
   phone VARCHAR(15),
   email VARCHAR(100),
   PRIMARY KEY(contact_id),
   CONSTRAINT fk_customer
      FOREIGN KEY(customer_id)
        REFERENCES customers(customer_id)
);
```

In this example, the customers table is the parent table and the contacts table is the child table.

Each customer has zero or many contacts and each contact belongs to zero or one customer.

The customer_id column in the contacts table is the foreign key column that references the primary key column with the same name in the customers table.

The following foreign key constraint fk_customer in the contacts table defines the customer_id as the foreign key:

```
CONSTRAINT fk_customer
   FOREIGN KEY(customer_id)
      REFERENCES customers(customer_id)
```

Because the foreign key constraint does not have the ON DELETE and ON UPDATE action, they default to NO ACTION.

NO ACTION

The following inserts data into the customers and contacts tables:

```
INSERT INTO customers(customer_name)
VALUES('BlueBird Inc'),
      ('Dolphin LLC');

INSERT INTO contacts(customer_id, contact_name, phone, email)
VALUES(1,'John Doe','(408)-111-1234','john.doe@bluebird.dev'),
      (1,'Jane Doe','(408)-111-1235','jane.doe@bluebird.dev'),
      (2,'David Wright','(408)-222-
1234','david.wright@dolphin.dev');
```

The following statement deletes the customer id 1 from the customers table:

```
DELETE FROM customers
WHERE customer_id = 1;
```

Because of the ON DELETE NO ACTION, PostgreSQL issues a constraint violation because the referencing rows of the customer id 1 still exist in the contacts table:

```
ERROR:  update or delete on table "customers" violates foreign
key constraint "fk_customer" on table "contacts"
DETAIL:  Key (customer_id)=(1) is still referenced from table
"contacts".
SQL state: 23503
```

CASCADE

The ON DELETE CASCADE automatically deletes all the referencing rows in the child table when the referenced rows in the parent table are deleted. In practice, the ON DELETE CASCADE is the most commonly used option.

The following statements recreate the sample tables. However, the delete action of the fk_customer changes to CASCADE:

```
DROP TABLE IF EXISTS contacts;
DROP TABLE IF EXISTS customers;

CREATE TABLE customers(
   customer_id INT GENERATED ALWAYS AS IDENTITY,
   customer_name VARCHAR(255) NOT NULL,
   PRIMARY KEY(customer_id)
);

CREATE TABLE contacts(
   contact_id INT GENERATED ALWAYS AS IDENTITY,
   customer_id INT,
   contact_name VARCHAR(255) NOT NULL,
   phone VARCHAR(15),
   email VARCHAR(100),
   PRIMARY KEY(contact_id),
   CONSTRAINT fk_customer
      FOREIGN KEY(customer_id)
        REFERENCES customers(customer_id)
        ON DELETE CASCADE
);

INSERT INTO customers(customer_name)
VALUES('BlueBird Inc'),
      ('Dolphin LLC');

INSERT INTO contacts(customer_id, contact_name, phone, email)
VALUES(1,'John Doe','(408)-111-1234','john.doe@bluebird.dev'),
      (1,'Jane Doe','(408)-111-1235','jane.doe@bluebird.dev'),
      (2,'David Wright','(408)-222-
1234','david.wright@dolphin.dev');
```

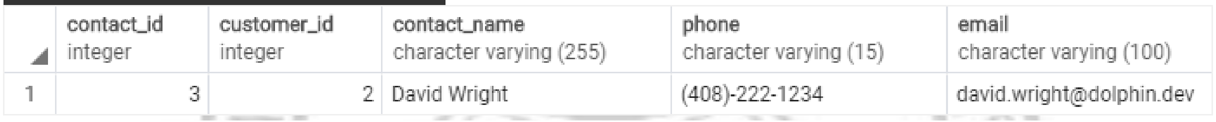
The following statement deletes the customer id 1:

```
DELETE FROM customers
```

```
WHERE customer_id = 1;
```

Because of the ON DELETE CASCADE action, all the referencing rows in the contacts table are automatically deleted:

```
SELECT * FROM contacts;
```



	contact_id integer	customer_id integer	contact_name character varying (255)	phone character varying (15)	email character varying (100)
1	3	2	David Wright	(408)-222-1234	david.wright@dolphin.dev

SET DEFAULT

The ON DELETE SET DEFAULT sets the default value to the foreign key column of the referencing rows in the child table when the referenced rows from the parent table are deleted.

Add a foreign key constraint to an existing table

To add a foreign key constraint to the existing table, you use the following form of the ALTER TABLE statement:

```
ALTER TABLE child_table
ADD CONSTRAINT constraint_name
FOREIGN KEY (fk_columns)
REFERENCES parent_table (parent_key_columns);
```

When you add a foreign key constraint with ON DELETE CASCADE option to an existing table, you need to follow these steps:

First, drop existing foreign key constraints:

```
ALTER TABLE child_table
DROP CONSTRAINT constraint_fkey;
```

First, add a new foreign key constraint with ON DELETE CASCADE action:

```
ALTER TABLE child_table
ADD CONSTRAINT constraint_fk
FOREIGN KEY (fk_columns)
REFERENCES parent_table(parent_key_columns)
ON DELETE CASCADE;
```

In this tutorial, you have learned about PostgreSQL foreign keys and how to use the foreign key constraint to create foreign keys for a table.

PostgreSQL CHECK Constraint

A CHECK constraint is a kind of constraint that allows you to specify if values in a column must meet a specific requirement.

The CHECK constraint uses a Boolean expression to evaluate the values before they are inserted or updated to the column.

If the values pass the check, PostgreSQL will insert or update these values to the column. Otherwise, PostgreSQL will reject the changes and issue a constraint violation error.

Define PostgreSQL CHECK constraint for new tables

Typically, you use the CHECK constraint at the time of creating the table using the CREATE TABLE statement.

The following statement defines an employees table.

```
DROP TABLE IF EXISTS employees;
CREATE TABLE employees (
	id SERIAL PRIMARY KEY,
	first_name VARCHAR (50),
	last_name VARCHAR (50),
	birth_date DATE CHECK (birth_date > '1900-01-01'),
	joined_date DATE CHECK (joined_date > birth_date),
	salary numeric CHECK(salary > 0)
);
```

The employees table has three CHECK constraints:

- First, the birth date (birth_date) of the employee must be greater than 01/01/1900. If you try to insert a birth date before 01/01/1900, you will receive an error message.
- Second, the joined date (joined_date) must be greater than the birth date (birth_date). This check will prevent from updating invalid dates in terms of their semantic meanings.
- Third, the salary must be greater than zero, which is obvious.

Let’s try to insert a new row into the employees table:

```
INSERT INTO employees (first_name, last_name, birth_date,
joined_date, salary)
VALUES ('John', 'Doe', '1972-01-01', '2015-07-01', - 100000);
```

The statement attempted to insert a negative salary into the salary column. However, PostgreSQL returned the following error message:

```
[Err] ERROR:  new row for relation "employees" violates check
constraint "employees_salary_check"
DETAIL:  Failing row contains (1, John, Doe, 1972-01-01, 2015-
07-01, -100000).
```

The insert failed because of the CHECK constraint on the salary column that accepts only positive values.

By default, PostgreSQL gives the CHECK constraint a name using the following pattern:

```
{table}_{column}_check
```

For example, the constraint on the salary column has the following constraint name:

```
employees_salary_check
```

However, if you want to assign aCHECK constraint a specific name, you can specify it after the CONSTRAINT expression as follows:

```
column_name data_type CONSTRAINT constraint_name CHECK(...)
```

See the following example:

```
...
salary numeric CONSTRAINT positive_salary CHECK(salary > 0)
...
```

Define PostgreSQL CHECK constraints for existing tables

To add CHECK constraints to existing tables, you use the ALTER TABLE statement. Suppose, you have an existing table in the database named `prices_list`

```
CREATE TABLE prices_list (
	id serial PRIMARY KEY,
	product_id INT NOT NULL,
	price NUMERIC NOT NULL,
	discount NUMERIC NOT NULL,
	valid_from DATE NOT NULL,
	valid_to DATE NOT NULL
);
```

Now, you can use ALTER TABLE statement to add the CHECK constraints to the prices_list table. The price and discount must be greater than zero and the discount is less than the price. Notice that we use a Boolean expression that contains the AND operators.

```
ALTER TABLE prices_list
ADD CONSTRAINT price_discount_check
CHECK (
        price > 0
        AND discount >= 0
        AND price > discount
);
```

The valid to date (valid_to) must be greater than or equal to valid from date (valid_from).

```
ALTER TABLE prices_list
ADD CONSTRAINT valid_range_check
CHECK (valid_to >= valid_from);
```

The CHECK constraints are very useful to place additional logic to restrict values that the columns can accept at the database layer. By using the CHECK constraint, you can make sure that data is updated to the database correctly.

In this tutorial, you have learned how to use PostgreSQL CHECK constraint to check the values of columns based on a Boolean expression.

PostgreSQL UNIQUE Constraint

Sometimes, you want to ensure that values stored in a column or a group of columns are unique across the whole table such as email addresses or usernames.

PostgreSQL provides you with the UNIQUE constraint that maintains the uniqueness of the data correctly.

When a UNIQUE constraint is in place, every time you insert a new row, it checks if the value is already in the table. It rejects the change and issues an error if the value already exists. The same process is carried out for updating existing data.

When you add a UNIQUE constraint to a column or a group of columns, PostgreSQL will automatically create a unique index on the column or the group of columns.

PostgreSQL UNIQUE constraint example

The following statement creates a new table named person with a UNIQUE constraint for the email column.

```
CREATE TABLE person (
        id SERIAL PRIMARY KEY,
        first_name VARCHAR (50),
        last_name VARCHAR (50),
        email VARCHAR (50) UNIQUE
);
```

Note that the UNIQUE constraint above can be rewritten as a table constraint as shown in the following query:

```
CREATE TABLE person (
        id SERIAL  PRIMARY KEY,
        first_name VARCHAR (50),
        last_name  VARCHAR (50),
        email      VARCHAR (50),
          UNIQUE(email)
);
```

First, insert a new row into the person table using INSERT statement:

```
INSERT INTO person(first_name,last_name,email)
VALUES('john','doe','j.doe@postgresqltutorial.com');
```

Second, insert another row with duplicate email.

```
INSERT INTO person(first_name,last_name,email)
VALUES('jack','doe','j.doe@postgresqltutorial.com');
```

PostgreSQL issued an error message.

```
[Err] ERROR:  duplicate key value violates unique constraint
"person_email_key"
DETAIL:  Key (email)=(j.doe@postgresqltutorial.com) already
exists.
```

Creating a UNIQUE constraint on multiple columns

PostgreSQL allows you to create a UNIQUE constraint to a group of columns using the following syntax:

```
CREATE TABLE table (
    c1 data_type,
    c2 data_type,
    c3 data_type,
    UNIQUE (c2, c3)
);
```

The combination of values in column c2 and c3 will be unique across the whole table. The value of the column c2 or c3 needs not to be unique.

Dropping a unique constraint

```
ALTER TABLE tbl_name
DROP CONSTRAINT constraint_name UNIQUE (col_name);
```

PostgreSQL Not-Null Constraint

Introduction to NULL

In database theory, NULL represents unknown or information missing. NULL is not the same as an empty string or the number zero.

Suppose that you need to insert an email address of a contact into a table. You can request his or her email address. However, if you don't know whether the contact has an email address or not, you can insert NULL into the email address column. In this case, NULL indicates that the email address is not known at the time of recording.

NULL is very special. It does not equal anything, even itself. The expression NULL = NULL returns NULL because it makes sense that two unknown values should not be equal.

To check if a value is NULL or not, you use the IS NULL boolean operator. For example, the following expression returns true if the value in the email address is NULL.

```
email_address IS NULL
```

The IS NOT NULL operator negates the result of the IS NULL operator.

PostgreSQL NOT NULL constraint

To control whether a column can accept NULL, you use the NOT NULL constraint:

```
CREATE TABLE table_name(
   ...
   column_name data_type NOT NULL,
   ...
);
```

If a column has a NOT NULL constraint, any attempt to insert or update NULL in the column will result in an error.

Declaring NOT NULL columns

The following CREATE TABLE statement creates a new table name invoices with the not-null constraints.

```
CREATE TABLE invoices(
  id SERIAL PRIMARY KEY,
  product_id INT NOT NULL,
  qty numeric NOT NULL CHECK(qty > 0),
  net_price numeric CHECK(net_price > 0)
);
```

This example uses the NOT NULL keywords that follow the data type of the product_id and qty columns to declare NOT NULL constraints.

Note that a column can have multiple constraints such as NOT NULL, check, unique, foreign key appeared next to each other. The order of the constraints is not important. PostgreSQL can check the constraint in the list in any order.

If you use NULL instead of NOT NULL, the column will accept both NULL and non-NULL values. If you don't explicitly specify NULL or NOT NULL, it will accept NULL by default.

Adding NOT NULL Constraint to existing columns

To add the NOT NULL constraint to a column of an existing table, you use the following form of the ALTER TABLE statement:

```
ALTER TABLE table_name
ALTER COLUMN column_name SET NOT NULL;
```

To add set multiple NOT NULL constraint to multiple columns, you use the following syntax:

```
ALTER TABLE table_name
ALTER COLUMN column_name_1 SET NOT NULL,
ALTER COLUMN column_name_2 SET NOT NULL,
...;
```

Let's take a look at the following example.

First, create a new table called production orders (production_orders):

```
CREATE TABLE production_orders (
	id SERIAL PRIMARY KEY,
	description VARCHAR (40) NOT NULL,
	material_id VARCHAR (16),
	qty NUMERIC,
	start_date DATE,
	finish_date DATE
);
```

Next, insert a new row into the production_orders table:

```
INSERT INTO production_orders (description)
VALUES('Make for Infosys inc.');
```

Then, to make sure that the qty field is not null, you can add the not-null constraint to the qty column. However, the column already contains data. If you try to add the not-null constraint, PostgreSQL will issue an error.

To add the NOT NULL constraint to a column that already contains NULL, you need to update NULL to non-NULL first, like this:

```
UPDATE production_orders
SET qty = 1;
```

The values in the qty column are updated to one. Now, you can add the NOT NULL constraint to the qty column:

```
ALTER TABLE production_orders
ALTER COLUMN qty
SET NOT NULL;
```

After that, you can update the not-null constraints for material_id, start_date, and finish_date columns:

```
UPDATE production_orders
SET material_id = 'ABC',
    start_date = '2015-09-01',
    finish_date = '2015-09-01';
```

Add not-null constraints to multiple columns:

```
ALTER TABLE production_orders
ALTER COLUMN material_id SET NOT NULL,
ALTER COLUMN start_date SET NOT NULL,
ALTER COLUMN finish_date SET NOT NULL;
```

Finally, attempt to update values in the qty column to NULL:

```
UPDATE production_orders
SET qty = NULL;
```

PostgreSQL issued an error message:

```
[Err] ERROR:  null value in column "qty" violates not-null
constraint
DETAIL:  Failing row contains (1, make for infosys inc., ABC,
null, 2015-09-01, 2015-09-01).
```

The special case of NOT NULL constraint

Besides the NOT NULL constraint, you can use a CHECK constraint to force a column to accept not NULL values. The NOT NULL constraint is equivalent to the following CHECK constraint:

```
CHECK(column IS NOT NULL)
```

This is useful because sometimes you may want either column a or b is not null, but not both.

For example, you may want either username or email column of the user tables is not null or empty. In this case, you can use the CHECK constraint as follows:

```
CREATE TABLE users (
 id serial PRIMARY KEY,
 username VARCHAR (50),
 password VARCHAR (50),
 email VARCHAR (50),
 CONSTRAINT username_email_notnull CHECK (
   NOT (
     ( username IS NULL  OR  username = '' )
     AND
     ( email IS NULL  OR  email = '' )
   )
 )
);
```

The following statement works.

```
INSERT INTO users (username, email)
VALUES
       ('user1', NULL),
       (NULL, 'email1@example.com'),
       ('user2', 'email2@example.com'),
       ('user3', '');
```

However, the following statement will not work because it violates the CHECK constraint:

```
INSERT INTO users (username, email)
VALUES
       (NULL, NULL),
       (NULL, ''),
       ('', NULL),
       ('', '');
```

```
[Err] ERROR:  new row for relation "users" violates check
constraint "username_email_notnull"
```

Drop NOT NULL Constraints

To remove the NOT NULL constraint, use the DROP NOT NULL clause along with ALTER TABLE ALTER COLUMN statement.

```
ALTER TABLE Table_name
ALTER COLUMN column_name DROP NOT NULL;
```