PostgreSQL CREATE TABLE syntax

A relational database consists of multiple related tables. A table consists of rows and columns. Tables allow you to store structured data like customers, products, employees, etc.

To create a new table, you use the CREATE TABLE statement. The following illustrates the basic syntax of the CREATE TABLE statement:

```
CREATE TABLE [IF NOT EXISTS] table_name (
  column1 datatype(length) column_contraint,
  column2 datatype(length) column_contraint,
  column3 datatype(length) column_contraint,
  table_constraints
                                );
```

In this syntax:

- First, specify the name of the table after the CREATE TABLE keywords.
- Second, creating a table that already exists will result in a error. The IF NOT EXISTS option allows you to create the new table only if it does not exist. When you use the IF NOT EXISTS option and the table already exists, PostgreSQL issues a notice instead of the error and skips creating the new table.
- Third, specify a comma-separated list of table columns. Each column consists of the column name, the kind of data that column stores, the length of data, and the column constraint. The column constraints specify rules that data stored in the column must follow. For example, the not-null constraint enforces the values in the column cannot be NULL. The column constraints include not null, unique, primary key, check, foreign key constraints.
- Finally, specify the table constraints including primary key, foreign key, and check constraints.

Note that some table constraints can be defined as column constraints like primary key, foreign key, check, unique constraints.

Constraints

PostgreSQL includes the following column constraints:

- NOT NULL – ensures that values in a column cannot be NULL.
- UNIQUE – ensures the values in a column unique across the rows within the same table.
- PRIMARY KEY – a primary key column uniquely identify rows in a table. A table can have one and only one primary key. The primary key constraint allows you to define the primary key of a table.
- CHECK – a CHECK constraint ensures the data must satisfy a boolean expression.
- FOREIGN KEY – ensures values in a column or a group of columns from a table exists in a column or group of columns in another table. Unlike the primary key, a table can have many foreign keys.

An introduction to PostgreSQL column and table constraints

Introduction

Constraints are *additional* requirements for acceptable values in addition to those provided by data types. They allow you to define narrower conditions for your data than those found in the general purpose data types.
These are often reflections on the specific characteristics of a field based on additional context provided by your applications. For example, an age field might use the int data type to store whole numbers. However, certain ranges of acceptable integers do not make sense as valid ages. For instance, negative integers would not be reasonable in this scenario. We can express this logical requirement in PostgreSQL using constraints.

Where constraints are defined: column vs table constraints

Postgres allows you to create constraints associated with a specific column or with a table in general.
Almost all constraints can be used in both forms without modification:

Constraint	Column	Table
CHECK	Yes	Yes
NOT NULL	Yes	No*
UNIQUE	Yes	Yes
PRIMARY KEY	Yes	Yes
FOREIGN KEY	Yes	Yes

*: NOT NULL cannot be used as a table constraint. However, you can approximate the results by using IS NOT NULL as the statement within a CHECK table constraint.

Let's look at how column and table constraints differ.

Column constraints

Column constraints are constraints attached to a single column. They are used to determine whether a proposed value for a column is valid or not. Column constraints are evaluated after the input is validated against basic type requirements (like making sure a value is a whole number for int columns).

Column constraints are great for expressing requirements that are limited to a single field. They attach the constraint condition directly to the column involved. For instance, we could model the age restriction in a person table by adding a constraint after the column name and data type:

```
CREATE TABLE person (
    . . .
    age int CHECK (age >= 0),
    . . .
);
```

This snippet defines a person table with one of the columns being an int called age.
The age must be greater than or equal to zero. Column constraints are easy to understand because they are added as additional requirements onto the column they affect.

Table constraints

The other type of constraint is called a table constraint. Table constraints can express any restrictions that a column constraint can, but can additionally express restrictions that involve more than one column. Instead of being attached to a specific column, table constraints are defined as a separate component of the table and can reference any of the table's columns.

The column constraint we saw earlier could be expressed as a table constraint like this:

```
CREATE TABLE person (
    . . .
    age int,
    . . .
    CHECK (age >= 0)
);
```

The same basic syntax is used, but the constraint is listed separately. To take advantage of the ability for table constraints to introduce compound restrictions, we can use the logical AND operator to join multiple conditions from different columns.

For example, in a banking database, a table called qualified_borrowers might need to check whether individuals have an existing account and the ability to offer collateral in order to qualify for a loan. It might make sense to include both of these in the same check:

```
CREATE TABLE qualified_borrowers (
. . .
account_number int,
acceptable_collateral boolean,
. . .
CHECK (account_number IS NOT NULL AND acceptable_collateral = 't')
);
```

Here, we use the CHECK constraint again to check that the account_number is not null and that the loan officer has marked the client as having acceptable collateral by checking the acceptable_collateral column. A table constraint is necessary since multiple columns are being checked.

Creating names for constraints

When you create constraints using the syntax above, PostgreSQL automatically chooses a reasonable, but vague, name. In the case of the qualified_borrowers table above, Postgres would name the constraint qualified_borrowers_check:

```
INSERT INTO qualified_borrowers VALUES (123, false);
```

```
ERROR:  new row for relation "qualified_borrowers" violates check constraint "qualified_borrowers_check"
DETAIL:  Failing row contains (123, f).
```

This name gives you information about the table and type of constraint when a constraint is violated. In cases where multiple constraints are present on a table, however, more descriptive names are helpful to help troubleshooting.

You can optionally specify the name for your constraints by preceding the constraint definition with the CONSTRAINT keyword followed by the name.

For example, if you wanted to name the constraint in the qualified_borrowers table loan_worthiness, you could instead define the table like this:

```
CREATE TABLE qualified_borrowers (
. . .
account_number int,
acceptable_collateral boolean,
. . .
```

```
    CONSTRAINT loan_worthiness CHECK (account_number IS NOT NULL AND acceptable_collateral = 't')
);
```

Now, when we violate a constraint, we get our more descriptive label:

```
INSERT INTO qualified_borrowers VALUES (123, false);
```

```
ERROR:  new row for relation "qualified_borrowers" violates check constraint "loan_worthiness"
DETAIL:  Failing row contains (123, f).
```

You can name column constraints in the same way:

```
CREATE TABLE teenagers (
    . . .
    age int CONSTRAINT is_teenager CHECK (age >= 13 AND age <= 19),
    . . .
);
```

PostgreSQL's list of available constraints

Now that we've covered some of the basics of how constraints work, we can take a deeper look at what constraints are available and how they may be used.

Check constraints

Check constraints are a general purpose constraint that allows you to specify an expression involving column or table values that evaluates to a boolean.

You've already seen a few examples of check constraints earlier. Check constraints begin with the keyword CHECK and then provide an expression enclosed in parentheses. For column constraints, this is placed after the data type declaration. For table constraints, these can be placed anywhere after the columns that they interact with are defined.

For example, we can create a film_nominations table that contains films that have been nominated and are eligible for a feature length award for 2019:

```
CREATE TABLE film_nominations (
    title text,
    director text,
    release_date date CHECK ('01-01-2019' <= release_date AND release_date <= '12-31-2019'),
    length int,
    votes int,
    CHECK (votes >= 10 AND length >= 40)
);
```

We have one column check restraint that checks that the release_date is within 2019. Afterwards, we have a table check constraint ensuring that the film has received enough votes to be nominated and that the length qualifies it for the "feature length" category.

When evaluating check constraints, acceptable values return *true*. If the new record's values satisfy all type requirements and constraints, the record will be added to the table:

```
INSERT INTO film_nominations VALUES (
    'A great film',
    'Talented director',
    '07-16-2019',
    117,
    45
);
```

```
INSERT 0 1
```

Values that yield *false* produce an error indicating that the constraint was not satisfied:

```
INSERT INTO film_nominations VALUES (
    'A poor film',
    'Misguided director',
    '10-24-2019',
    128,
    1
);
```

```
ERROR:  new row for relation "film_nominations" violates check constraint "film_nominations_check"
DETAIL:  Failing row contains (A poor film, Misguided director, 2019-07-16, 128, 1).
```

In this case, the film has satisfied every condition except for the number of votes required. PostgreSQL rejects the submission since it does not pass the final table check constraint.

Not null constraints

The NOT NULL constraint is much more focused. It guarantees that values within a column are not null. While this is a simple constraint, it is used very frequently.

To mark a column as requiring a non-null value, add NOT NULL after the type declaration:

```
CREATE TABLE national_capitals (
    country text NOT NULL,
    capital text NOT NULL,
);
```

In the above example, we have a simple two column table mapping countries to their national capitals. Since both of these are required fields that would not make sense to leave blank, we add the NOT NULL constraint.

Inserting a null value now results in an error:

```
INSERT INTO national_capitals VALUES (
    NULL,
    'London',
);
```

```
ERROR:  null value in column "country" violates not-null constraint
DETAIL:  Failing row contains (null, London).
```

The NOT NULL constraint functions only as a column constraint (it cannot be used as a table constraint). However, you can easily work around this by using IS NOT NULL within a table CHECK constraint.

For example, this offers equivalent guarantees using a table constraint:

```
CREATE TABLE national_capitals (
    country text,
    capital text,
    CHECK (country IS NOT NULL AND capital IS NOT NULL)
);
```

Unique constraints

The UNIQUE constraint tells PostgreSQL that each value within a column must not be repeated. This is useful in many different scenarios where having the same value in multiple records should be impossible.

For example, columns that deals with IDs of any kind should, by definition, have unique values. A social security number, a student or customer ID, or a product UPC (barcode number) would be useless if they were not able to differentiate between specific people or items.

A UNIQUE constraint can be specified at the column level:

```
CREATE TABLE supplies (
    supply_id integer UNIQUE,
    name text,
    inventory integer
);
```

They can also be specified as table constraints:

```
CREATE TABLE supplies (
    supply_id integer,
    name text,
    inventory integer,
    UNIQUE (supply_id)
);
```

One of the advantages of using UNIQUE table constraints is that it allows you to perform uniqueness checks on a combination of columns. This works by specifying two or more columns that PostgreSQL should evaluate together. The values in individual columns may repeat but the combination of values specified must be unique.

As an example, let's look back at the national_capitals table we used before:

```
CREATE TABLE national_capitals (
    country text NOT NULL,
    capital text NOT NULL,
);
```

If we wanted to make sure that we don't add multiple records for the same pair, we could add UNIQUE constraints to the columns here:

```
CREATE TABLE national_capitals (
    country text NOT NULL UNIQUE,
    capital text NOT NULL UNIQUE,
);
```

This would ensure that both the countries and capitals are only present once in each table. However, some countries have multiple capitals. This would mean we may have multiple entries with the same country value. These wouldn't work with the current design:

```
INSERT INTO national_capitals VALUES (
    'Bolivia',
    'Sucre'
);
INSERT INTO national_capitals VALUES (
    'Bolivia',
    'La Paz'
);
```

```
INSERT 0 1
ERROR:  duplicate key value violates unique constraint "national_capitals_country_key"
DETAIL:  Key (country)=(Bolivia) already exists.
```

If we still want to make sure we don't end up with duplicate entries while allowing for repeated values in individual columns, a unique check on the combination of country and capital would suffice:

```
CREATE TABLE national_capitals (
    country text,
    captial text,
    UNIQUE (country, capital)
);
```

Now, we can add both of Bolivia's capitals to the table without an error:

```
INSERT INTO national_capitals VALUES (
    'Bolivia',
    'Sucre'
);
INSERT INTO national_capitals VALUES (
    'Bolivia',
    'La Paz'
);
```

```
INSERT 0 1
INSERT 0 1
```

However, attempting to add the same combination twice is still caught by the constraint:

```
INSERT INTO national_capitals VALUES (
    'Bolivia',
    'Sucre'
);
INSERT INTO national_capitals VALUES (
    'Bolivia',
    'Sucre'
);
```

```
INSERT 0 1
ERROR:  duplicate key value violates unique constraint "national_capitals_country_capital_key"
DETAIL:  Key (country, capital)=(Bolivia, Sucre) already exists.
```

Primary key constraints

The PRIMARY KEY constraint serves a special purpose. It indicates that the column can be used to uniquely identify a record within the table. This means that it must be reliably unique and that every record must have a value in that column.

Primary keys are recommended for every table not required, and every table may only have one primary key. Primary keys are mainly used to identify, retrieve, modify, or delete individual records within a table. They allow users and administrators to target the operation using an identifier that is guaranteed by PostgreSQL to match exactly one record.

Let's use the supplies table we saw before as an example:

```
CREATE TABLE supplies (
    supply_id integer UNIQUE,
    name text,
    inventory integer
);
```

Here we've identified that the supply_id should be unique. If we wanted to use this column as our primary key (guaranteeing uniqueness and a non-null value), we could simply change the UNIQUE constraint to PRIMARY KEY:

```
CREATE TABLE supplies (
    supply_id integer PRIMARY KEY,
    name text,
    inventory integer
);
```

This way, if we needed to update the inventory amounts for a specific supply, we could target it using the primary key:

```
INSERT INTO supplies VALUES (
    38,
    'nails',
    5
);
UPDATE supplies set inventory = 10 WHERE supply_id = 38;
```

```
INSERT 0 1
UPDATE 1
```

While many tables use a single column as the primary key, it is also possible to create a primary key using a set of columns, as a table constraint.

The national_capitals table is a good candidate to demonstrate this. If we wanted to create a primary key using the existing columns, we could replace the UNIQUE table constraint with PRIMARY KEY:

```
CREATE TABLE national_capitals (
    country text,
    captial text,
    PRIMARY KEY (country, capital)
);
```

Foreign keys constraints

Foreign keys are columns within one table that reference column values within another table. This is desirable and often necessary in a variety of scenarios where tables contain related data. This ability for the database to easily connect and reference data stored in separate tables is one of the primary features of relational databases.

For example, you may have a orders table to track individual orders and a customers table to track contact info and information about your customers. It makes sense to put this information separately since customers may have many orders. However, it also makes sense to be able to easily link the records in these two tables to allow more complex operations.

Let's start by trying to model the customers table:

```
CREATE TABLE customers (
    customer_id serial PRIMARY KEY,
    first_name text,
    last_name text,
    phone_number bigint,
);
```

This table is pretty simple. It includes columns to store the parent's first name, last name, and phone number. It also specifies an ID column that uses the PRIMARY KEY constraint.
The serial data type is used to automatically generate the next ID in the sequence if an ID is not specified.

For the orders table, we want to be able to specify information about individual orders. One essential piece of data is what customer placed the order. We can use a foreign key to link the order to the customer without duplicating information. We do this with
the REFERENCES constraint, which defines a foreign key relationship to a column in another table:

```
CREATE TABLE orders (
    order_id serial PRIMARY KEY,
    order_date date,
    customer integer REFERENCES customers
);
```

Here, we are indicating that the customer column in the orders table has a foreign key relationship with the customers table. Since we do not specify a specific column within the customers table, PostgreSQL assumes that we want to link to the primary key in the customers table: customer_id.

If we try to insert a value into the orders table that doesn't reference a valid customer, PostgreSQL will reject it:

```
INSERT INTO orders VALUES (
    100,
    '11-19-2019',
    300
);
```

```
ERROR:  insert or update on table "orders" violates foreign key constraint "orders_customer_fkey"
DETAIL:  Key (customer)=(300) is not present in table "customers".
```

If we add the customer first, our order will then be accepted by the system:

```
INSERT INTO customers VALUES (
    300,
    'Jill',
    'Smith',
    '5551235677'
);
INSERT INTO orders VALUES (
    100,
    '11-19-2019',
    300
);
```

```
INSERT 0 1
INSERT 0 1
```

While the primary key is a great candidate for foreign keys because it guarantees to match only one record, you can also use other columns as long as they're unique. To do so you just have to specify the column in parentheses after the table name in the REFERENCES definition:

```
CREATE TABLE example (
    . . .
    column type REFERENCES other_table (column)
);
```

You can also use sets of columns that are guaranteed unique. To do so, you need to use a table constraint that begins with FOREIGN KEY and refers to columns you've defined earlier in the table description:

```
CREATE TABLE example (
    . . .
    FOREIGN KEY (column1, column2) REFERENCES other_table (column1, column2)
);
```

Deciding what to do with foreign keys when deleting or updating

One consideration you'll need to think about when defining foreign key constraints is what to do when a referenced table is deleted or updated.

As an example, let's look at the customers and orders tables again. We need to specify how we want the system to respond when we delete a customer from the customers table when the customer has an associated order in the orders table.

We can choose between the following options:

RESTRICT: Choosing to restrict deletions means that PostgreSQL will refuse to delete the customer record if it's referenced by a record in the orders table. To delete a customer, you will first have to remove any associated records from the orders table. Only then will you be able to remove the value from the customer table.

CASCADE: Selecting the cascade option means that when we delete the customer record, the records that reference it in the orders table are *also* deleted. This is useful in many cases but must be used with care to avoid deleting data by mistake.

NO ACTION: The no action option tells PostgreSQL to simply remove the customer and not do anything with the associated orders records. If the constraint is checked later, it will still cause an error, but this won't happen during the initial deletion. This is the default action if no other is specified.

SET NULL: This option tells PostgreSQL to set the referencing columns to null when the referenced records are removed. So if we delete a customer from the customers table, the customer column in the orders table will be set to NULL.

Set DEFAULT: If this option is chosen, PostgreSQL will change the referencing column to the default value if the referenced record is deleted. So if the customer column in the orders table had a default value and we remove a customer from the customers table, the record in the orders value would be assigned the default value.

These actions can be specified when defining a foreign key constraint by adding ON DELETE followed by the action. So if we want to remove associated orders from our system when a customer is deleted, we could specify that like this:

```
CREATE TABLE orders (
    order_id serial PRIMARY KEY,
    order_date date,
    customer integer REFERENCES customers ON DELETE CASCADE
);
```

These type of actions can also be applied when *updating* a referenced column instead of deleting one by using ON UPDATE instead of ON DELETE.