

2.4. Constraints

Data types are a way to limit the kind of data that can be stored in a table. For many applications, however, the constraint they provide is too coarse. For example, a column containing a product price should probably only accept positive values. But there is no data type that accepts only positive numbers. Another issue is that you might want to constrain column data with respect to other columns or rows. For example, in a table containing product information, there should only be one row for each product number.

To that end, SQL allows you to define constraints on columns and tables. Constraints give you as much control over the data in your tables as you wish. If a user attempts to store data in a column that would violate a constraint, an error is raised. This applies even if the value came from the default value definition.

2.4.1. Check Constraints

A check constraint is the most generic constraint type. It allows you to specify that the value in a certain column must satisfy an arbitrary expression. For instance, to require positive product prices, you could use:

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CHECK (price > 0)  
);
```

As you see, the constraint definition comes after the data type, just like default value definitions. Default values and constraints can be listed in any order. A check constraint consists of the key word `CHECK` followed by an expression in parentheses. The check constraint expression should involve the column thus constrained, otherwise the constraint would not make too much sense.

You can also give the constraint a separate name. This clarifies error messages and allows you to refer to the constraint when you need to change it. The syntax is:

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CONSTRAINT positive_price CHECK (price > 0)  
);
```

To specify a named constraint, use the key word `CONSTRAINT` followed by an identifier followed by the constraint definition.

A check constraint can also refer to several columns. Say you store a regular price and a discounted price, and you want to ensure that the discounted price is lower than the regular price.

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CHECK (price > 0),  
    discounted_price numeric CHECK (discounted_price > 0),  
    CHECK (price > discounted_price)  
);
```

The first two constraints should look familiar. The third one uses a new syntax. It is not attached to a particular column, instead it appears as a separate item in the comma-separated column list. Column definitions and these constraint definitions can be listed in mixed order.

We say that the first two are column constraints, whereas the third one is a table constraint because it is written separately from the column definitions. Column constraints can also be written as table constraints, while the reverse is not necessarily possible. The above example could also be written as

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric,  
    CHECK (price > 0),  
    discounted_price numeric,  
    CHECK (discounted_price > 0),  
    CHECK (price > discounted_price)  
);
```

or even

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CHECK (price > 0),  
    discounted_price numeric,  
    CHECK (discounted_price > 0 AND price > discounted_price)  
);
```

It's a matter of taste.

It should be noted that a check constraint is satisfied if the check expression evaluates to true or the null value. Since most expressions will evaluate to the null value if one operand is null they will not prevent null values in the constrained columns. To ensure that a column does not contain null values, the not-null constraint described in the next section should be used.

2.4.2. Not-Null Constraints

A not-null constraint simply specifies that a column must not assume the null value. A syntax example:

```
CREATE TABLE products (  
    product_no integer NOT NULL,  
    name text NOT NULL,  
    price numeric  
);
```

A not-null constraint is always written as a column constraint. A not-null constraint is functionally equivalent to creating a check constraint `CHECK (column_name IS NOT NULL)`, but in PostgreSQL creating an explicit not-null constraint is more efficient. The drawback is that you cannot give explicit names to not-null constraints created that way.

Of course, a column can have more than one constraint. Just write the constraints after one another:

```
CREATE TABLE products (  
    product_no integer NOT NULL,  
    name text NOT NULL,  
    price numeric NOT NULL CHECK (price > 0)  
);
```

The order doesn't matter. It does not necessarily affect in which order the constraints are checked.

The `NOT NULL` constraint has an inverse: the `NULL` constraint. This does not mean that the column must be null, which would surely be useless. Instead, this simply defines the default behavior that the column may be null. The `NULL` constraint is not defined in the SQL standard and should not be used in portable applications. (It was only added to PostgreSQL to be compatible with other database systems.) Some users, however, like it because it makes it easy to toggle the constraint in a script file. For example, you could start with

```
CREATE TABLE products (
    product_no integer NULL,
    name text NULL,
    price numeric NULL
);
```

and then insert the NOT key word where desired.

Tip: In most database designs the majority of columns should be marked not null.

2.4.3. Unique Constraints

Unique constraints ensure that the data contained in a column or a group of columns is unique with respect to all the rows in the table. The syntax is

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric
);
```

when written as a column constraint, and

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
);
```

when written as a table constraint.

If a unique constraint refers to a group of columns, the columns are listed separated by commas:

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

It is also possible to assign names to unique constraints:

```
CREATE TABLE products (
    product_no integer CONSTRAINT must_be_different UNIQUE,
    name text,
    price numeric
);
```

In general, a unique constraint is violated when there are (at least) two rows in the table where the values of each of the corresponding columns that are part of the constraint are equal. However, null values are not considered equal in this consideration. That means, in the presence of a multicolumn unique constraint it is possible to store an unlimited number of rows that contain a null value in at least one of the constrained columns. This behavior conforms to the SQL standard, but we have heard that other SQL databases may not follow this rule. So be careful when developing applications that are intended to be portable.

2.4.4. Primary Keys

Technically, a primary key constraint is simply a combination of a unique constraint and a not-null constraint. So, the following two table definitions accept the same data:

```
CREATE TABLE products (  
    product_no integer UNIQUE NOT NULL,  
    name text,  
    price numeric  
);  
  
CREATE TABLE products (  
    product_no integer PRIMARY KEY,  
    name text,  
    price numeric  
);
```

Primary keys can also constrain more than one column; the syntax is similar to unique constraints:

```
CREATE TABLE example (  
    a integer,  
    b integer,  
    c integer,  
    PRIMARY KEY (a, c)  
);
```

A primary key indicates that a column or group of columns can be used as a unique identifier for rows in the table. (This is a direct consequence of the definition of a primary key. Note that a unique constraint does not, in fact, provide a unique identifier because it does not exclude null values.) This is useful both for documentation purposes and for client applications. For example, a GUI application that allows modifying row values probably needs to know the primary key of a table to be able to identify rows uniquely.

A table can have at most one primary key (while it can have many unique and not-null constraints). Relational database theory dictates that every table must have a primary key. This rule is not enforced by PostgreSQL, but it is usually best to follow it.

2.4.5. Foreign Keys

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values in some other column. We say this maintains the *referential integrity* between two related tables.

Say you have the product table that we have used several times already:

```
CREATE TABLE products (  
    product_no integer PRIMARY KEY,  
    name text,  
    price numeric  
);
```

Let's also assume you have a table storing orders of those products. We want to ensure that the orders table only contains orders of products that actually exist. So we define a foreign key constraint in the orders table that references the products table:

```
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    product_no integer REFERENCES products (product_no),  
    quantity integer  
);
```

Now it is impossible to create orders with `product_no` entries that do not appear in the products table.

We say that in this situation the orders table is the *referencing* table and the products table is the *referenced* table. Similarly, there are referencing and referenced columns.

You can also shorten the above command to

```
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    product_no integer REFERENCES products,  
    quantity integer  
);
```

because in absence of a column list the primary key of the referenced table is used as referenced column.

A foreign key can also constrain and reference a group of columns. As usual, it then needs to be written in table constraint form. Here is a contrived syntax example:

```
CREATE TABLE t1 (  
    a integer PRIMARY KEY,  
    b integer,  
    c integer,  
    FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)  
);
```

Of course, the number and type of the constrained columns needs to match the number and type of the referenced columns.

A table can contain more than one foreign key constraint. This is used to implement many-to-many relationships between tables. Say you have tables about products and orders, but now you want to allow one order to contain possibly many products (which the structure above did not allow). You could use this table structure:

```
CREATE TABLE products (  
    product_no integer PRIMARY KEY,  
    name text,  
    price numeric  
);  
  
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    shipping_address text,  
    ...  
);  
  
CREATE TABLE order_items (  
    product_no integer REFERENCES products,  
    order_id integer REFERENCES orders,  
    quantity integer,  
    PRIMARY KEY (product_no, order_id)  
);
```

Note also that the primary key overlaps with the foreign keys in the last table.

We know that the foreign keys disallow creation of orders that don't relate to any products. But what if a product is removed after an order is created that references it? SQL allows you to specify that as well. Intuitively, we have a few options:

- Disallow deleting a referenced product

- Delete the orders as well
- Something else?

To illustrate this, let's implement the following policy on the many-to-many relationship example above: When someone wants to remove a product that is still referenced by an order (via `order_items`), we disallow it. If someone removes an order, the order items are removed as well.

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products ON DELETE RESTRICT,
    order_id integer REFERENCES orders ON DELETE CASCADE,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

Restricting and cascading deletes are the two most common options. `RESTRICT` can also be written as `NO ACTION` and it's also the default if you don't specify anything. There are two other options for what should happen with the foreign key columns when a primary key is deleted: `SET NULL` and `SET DEFAULT`. Note that these do not excuse you from observing any constraints. For example, if an action specifies `SET DEFAULT` but the default value would not satisfy the foreign key, the deletion of the primary key will fail.

Analogous to `ON DELETE` there is also `ON UPDATE` which is invoked when a primary key is changed (updated). The possible actions are the same.

More information about updating and deleting data is in Chapter 3.

Finally, we should mention that a foreign key must reference columns that are either a primary key or form a unique constraint. If the foreign key references a unique constraint, there are some additional possibilities regarding how null values are matched. These are explained in the `CREATE TABLE` entry in the *PostgreSQL 7.3 Reference Manual*.

Prev
Default Values

Home
Up

Next
Inheritance