7

Server-Side Programming

In previous chapters, we learned how to execute SQL queries. We started by writing simple queries, then moved on to writing more complex queries; we learned how to use aggregates in the traditional way, and in *Chapter 5*, *Advanced Statements*, we talked about window functions, which are another way to write aggregates. In this chapter, we will add server-side programming to this list of skills. Server-side programming can be useful in many cases as it moves the programming logic from the client side to the database side. For example, we could use it to take a function that has been written many times at different points of the application program and move it inside the server so that it is written only once, meaning that in case of modification, we only have to modify one function. In this chapter, we will also look at how PostgreSQL can manage different server-side programming languages, and we will see that server-side programming can be very useful if you need to process a large amount of data that has been extracted from tables. We will address the fact that all the functions we will write can be called in any SQL statement. We will also see that in some cases, for certain types of functions, it is also possible to create indices on the functions.

Another feature of server-side programming is the chance to define customized data. In this chapter, we will look at some examples of this.

In simple terms, this chapter will discuss the following:

- Exploring data types
- Exploring functions and languages
- The NoSQL data type

Technical requirements

Before starting, remember to start the Docker container named `chapter_07`, as shown below:

```
$ bash run-pg-docker.sh chapter_07
postgres@learn_postgresql:~$ psql -U forum forumdb
```

Exploring data types

As users, we have already had the opportunity to experience the power and versatility of server-side functions – for example, in *Chapter 5*, *Advanced Statements*, we used a query similar to the following:

```
forumdb=> select * from categories where upper(title) like 'A%';
 pk | title |         description
----+-------+-----------------------------
  4 | A.I   | Machine Learning discussions
(1 row)
```

In this piece of code, the `upper` function is a server-side function; this function turns all the characters of a string into uppercase. In this chapter, we will acquire the knowledge to be able to write functions such as the `upper` function that we called in the preceding query.

In this section, we'll talk about data types. We will briefly mention the standard types managed by PostgreSQL and how to create new ones.

The concept of extensibility

What is extensibility? Extensibility is PostgreSQL's ability to extend its functionality and its data types. Extensibility is an extremely useful PostgreSQL feature because it enables us to have data types, functions, and functional indexes that are not present in the base system. In this chapter, we will cover extension at the data type level, as well as the addition of new functions.

Standard data types

In previous chapters, even if not explicitly obvious, we already used standard data types. This was when we learned how to use **Data Definition Language** (**DDL**) commands. However, we will now be looking more deeply into this topic. The following is a short list of the most used data types:

- Boolean type
- Numeric types
- Character types

- Date/time
- NoSQL data types: `hstore`, `xml`, `json`, and `jsonb`

For each data type, we will show an example operation followed by a brief explanation. For further information on the standard data types supported by PostgreSQL, please refer to the official documentation at `https://www.postgresql.org/docs/current/extend-type-system.html`.

Boolean data type

First, we will introduce the Boolean data type. PostgreSQL supports Boolean data types. The Boolean type (identified by `BOOLEAN` or `BOOL`), like all data types supported by PostgreSQL, can assume the `NULL` value. Therefore, a Boolean data type can take the `NULL`, `FALSE`, and `TRUE` values. The data type input function for the Boolean type accepts the following representations for the `TRUE` state:

State	true	yes	on	1

For the `false` state, we have the following:

State	false	no	off	0

Let's look at some examples, starting with the `users` table:

1. Let's first display the contents of the `users` table:

```
forumdb=> select * from users;
 pk |    username    | gecos |         email
----+----------------+-------+-----------------------
  1 | luca_ferrari   |       | luca@pgtraining.com
  2 | enrico_pirozzi |       | enrico@pgtraiing.com
  3 | newuser        |       | newuser@pgtraining.com
(3 rows)
```

2. Now let's add a Boolean data type to the `users` table:

```
forumdb=> alter table users add user_on_line boolean;
ALTER TABLE
```

3. Let's update some values:

```
forumdb=> update users set user_on_line = true where pk=1;
UPDATE 1
```

4. Now, if we want to search for all the records that have the `user_on_line` field set to `true`, we have to perform the following:

```
forumdb=> \x
Expanded display is on.
forumdb=> select * from users where user_on_line = true;
-[ RECORD 1 ]+--------------------
pk           | 1
username     | luca_ferrari
gecos        |
email        | luca@pgtraining.com
user_on_line | t
```

5. If we want the search for all the records that have the `user_on_line` field set to `NULL`, as we saw in *Chapter 4*, *Basic Statements*, we have to perform the following:

```
forumdb=> select * from users where user_on_line is NULL;
-[ RECORD 1 ]+-----------------------
pk           | 2
username     | enrico_pirozzi
gecos        |
email        | enrico@pgtraiing.com
user_on_line |
-[ RECORD 2 ]+-----------------------
pk           | 3
username     | newuser
gecos        |
email        | newuser@pgtraining.com
user_on_line |
```

Thus, we have explored the Boolean data type.

Numeric data type

PostgreSQL supports several types of numeric data types; the most used ones are as follows:

- `integer` or `int4` (4-byte integer number).
- `bigint` or `int8` (8-byte integer number).
- `real` (4-byte variable precision, inexact with 6-decimal-digit precision).

- `double precision` (8-byte variable precision, inexact with 15-decimal-digit precision).
- `numeric` (`precision`, `scale`), where the precision of a numeric is the total count of significant digits in the whole number, and the scale of a numeric is the count of decimal digits in the fractional part. For example, 5.827 has a precision of 4 and a scale of 3.

Now, we will look at some brief examples of each type in the upcoming sections.

Integer types

As we can see here, if we cast a number to an integer type such as `integer` or `bigint`, PostgreSQL will make a truncated value of the input number:

```
forumdb=> \x
Expanded display is off.
forumdb=> select 1.123456789::integer as my_field;
 my_field
----------
        1
(1 row)

forumdb=> select 1.123456789::int4 as my_field;
 my_field
----------
        1
(1 row)
forumdb=> select 1.123456789::bigint as my_field;
 my_field
----------
        1
(1 row)

forumdb=> select 1.123456789::int8 as my_field;
 my_field
----------
        1
(1 row)
```

Numbers with a fixed precision data type

In the following example, we'll see the same query that we have seen previously, but this time, we'll make a cast to `real` and to `double precision`:

```
forumdb=> select 1.123456789::real as my_field;
 my_field
----------
 1.1234568
(1 row)

forumdb=> select 1.123456789::double precision as my_field;
  my_field
-------------
 1.123456789
(1 row)
```

As can be seen here, in the first query, the result was cut to the sixth digit; this happened because the real type has at least 6-decimal-digit precision.

Now suppose we want to perform the sum of the value `0.1` 10 times. The correct result would be the number 1. Instead, if we execute:

```
forumdb=> select sum(0.1::real) from generate_series(1,10);
    sum
----------
 1.0000001
(1 row)
```

We get the value `1.0000001`. This happens due to the intrinsic rounding error in the `real` data type, so it is not recommended to use the `real` data type in fields representing money. The correct way to make this sum is using the `numeric` data type.

Numbers with an arbitrary precision data type

In this last section about `numeric` data types, we'll make the same query that we saw earlier, but we'll make a cast to arbitrary precision:

```
forumdb=> select 1.123456789::numeric(10,1) as my_field;
 my_field
---------
```

```
      1.1
(1 row)

forumdb=> select 1.123456789::numeric(10,5) as my_field;
 my_field
----------
  1.12346
(1 row)

forumdb=> select 1.123456789::numeric(10,9) as my_field;
  my_field
-------------
 1.123456789
(1 row)
```

As we can see from the examples shown here, we decide how many digits the scale should be.

But what about if we perform something like the following?

```
forumdb=> select 1.123456789::numeric(10,11) as my_field;
ERROR:  numeric field overflow
DETAIL:  A field with precision 10, scale 11 must round to an absolute
value less than 10^-1.
```

The result is an error. This is because the data type was defined as a `numeric` type with a precision value equal to `10`, so we can't have a scale parameter equal to or greater than the precision value.

Similarly, the next example will also produce an error:

```
forumdb=> select 1.123456789::numeric(10,10) as my_field;
ERROR:  numeric field overflow
DETAIL:  A field with precision 10, scale 10 must round to an absolute
value less than 1.
```

In the preceding example, the query generates an error because the scale was `10`, meaning we should have 10 digits, but we have 11 digits in total:

Digits	1	2	3	4	5	6	7	8	9	10	11
	1	.	1	2	3	4	5	6	7	8	9

However, if in our number we don't have the first digit, the query will work:

```
forumdb=> select 0.123456789::numeric(10,10) as my_field;
   my_field
--------------
 0.1234567890
(1 row)
```

Now let's go back to the example of the previous paragraph, which provided an incorrect sum, and let's repeat it using the numeric type:

```
forumdb=> select sum(0.1::numeric(2,2)) from generate_series(1,10);
 sum
------
 1.00
(1 row)
```

As we can see, now the value of the sum is correct; so, the correct way to represent money is using a `numeric` data type.

Thus, we have learned all about the various numeric data types.

Character data type

The most used character data types in PostgreSQL are the following:

- `character(n)/char(n)` (fixed-length, blank-padded)
- `character varying(n)/varchar(n)` (variable length with a limit)
- `varchar/text` (variable unlimited length)

Now, we will look at some examples to see how PostgreSQL manages these kinds of data types.

Chars with fixed-length data types

We will check out how they work using the following example:

1. Let's start by creating a new test table:

```
forumdb=> create table new_tags (
pk integer not null primary key,
tag char(10)
);
CREATE TABLE
```

In the previous code, we created a new table named `new_tags` with a `char(10)` field name tag.

2. Now, let's add some records and see how PostgreSQL behaves:

```
forumdb=> insert into new_tags values (1,'first tag');
INSERT 0 1
forumdb=> insert into new_tags values (2,'tag');
INSERT 0 1
```

In order to continue with our analysis, we must introduce two new functions:

- `length(p)`: This counts the number of characters, where `p` is an input parameter and a string
- `octet_length(p)`: This counts the number of bytes, where `p` is an input parameter and a string

3. Let's execute the following query:

```
forumdb=> \x
Expanded display is on.
forumdb=> select pk,tag,length(tag),octet_length(tag),char_
length(tag) from new_tags;
-[ RECORD 1 ]+-----------
pk           | 1
tag          | first tag
length       | 9
octet_length | 10
char_length  | 9
-[ RECORD 2 ]+-----------
pk           | 2
tag          | tag
length       | 3
octet_length | 10
char_length  | 3
```

As we can see, the overall length of the space occupied internally by the field is always `10`; this is true even if the number of characters entered is different. This happens because we have defined the field as `char(10)`, with a fixed length of `10`, so even if we insert a string with a shorter length, the difference between 10 and the number of real characters of the string will be filled with blank characters.

Chars with variable length with a limit data types

In this section, we are going to repeat the same example that we used in the previous section, but this time, we'll use the `varchar(10)` data type for the `tag` field:

1. Let's recreate the `new_tags` table:

```
forumdb=> drop table if exists new_tags;
DROP TABLE


forumdb=> create table new_tags (
pk integer not null primary key,
tag varchar(10)
);
CREATE TABLE
```

2. Then, let's insert some data:

```
forumdb=> insert into new_tags values (1,'first tag');
INSERT 0 1

forumdb=> insert into new_tags values (2,'tag');
INSERT 0 1
```

3. Now, if we repeat the same query as before, we obtain the following:

```
forumdb=> \x
Expanded display is off.
forumdb=> select pk,tag,length(tag),octet_length(tag) from new_tags
;
 pk |    tag    | length | octet_length
----+-----------+--------+--------------
  1 | first tag |      9 |            9
  2 | tag       |      3 |            3
(2 rows)
```

As we can see, this time, the real internal size and the number of characters in the string are the same.

4. Now, let's try to insert a string longer than 10 characters and see what happens:

```
forumdb=> insert into new_tags values (3,'this sentence has more
than 10 characters');
ERROR:  value too long for type character varying(10)
```

PostgreSQL answers correctly with an error because the input string exceeds the dimension of the field.

Chars with a variable length without a limit data types

In this section, we will again use the same example as before, but this time, we'll use a `text` data type for the `tag` field.

Let's recreate the `new_tags` table and re-insert the same data that we inserted previously:

```
forumdb=>  drop table if exists new_tags;
DROP TABLE

forumdb=> create table new_tags (
pk integer not null primary key,
tag text
);
CREATE TABLE

forumdb=> insert into new_tags values (1,'first tag'), (2,'tag'),(3,'this
sentence has more than 10 characters');
INSERT 0 3
```

This time, PostgreSQL correctly inserts all three records. This is because the `text` data type is a `char` data type with unlimited length, as we can see in the following query:

```
forumdb=> select pk,substring(tag from 0 for 20),length(tag),octet_
length(tag) from new_tags ;
 pk |      substring      | length | octet_length
----+---------------------+--------+--------------
  1 | first tag           |      9 |            9
  2 | tag                 |      3 |            3
  3 | this sentence has m |     41 |           41
(3 rows)
```

In the preceding example, we can see that the `text` data type behaves exactly like the `varchar(n)` data type we saw earlier. The only difference between `text` and `varchar(n)` is that the `text` type has no size limit. It is important to note that in the preceding query, we used the `substring` function. The `substring` function takes a piece of the string starting from the `from` parameter for `n` characters; for example, if we write `substring(tag from 0 for 20)`, it means that we want the first 20 characters of the `tag` string as output.

With this, we have covered all the `char` data types.

Date/timestamp data types

In this section, we will talk about how to store dates and times in PostgreSQL. PostgreSQL supports both dates and times and the combination of date and time (timestamp). PostgreSQL manages hours both with time zone settings and without time zone settings, as described in the official documentation (`https://www.postgresql.org/docs/current/datatype-datetime.html`).

> PostgreSQL supports the full set of SQL date and time types. Dates are counted according to the Gregorian calendar.

Date data types

Managing dates often becomes a puzzle for developers. This is because dates are represented differently depending on the country for which we have to store the data – for example, the American way is month/day/year, whereas the European format is day/month/year. PostgreSQL helps us by providing the necessary tools to best solve this problem, as seen here:

1. The first thing we have to do is to see how PostgreSQL internally stores dates. To do this, we have to perform the following query:

```
forumdb=> \x
Expanded display is on.
forumdb=> select * from pg_settings where name ='DateStyle';
-[ RECORD 1 ]---+-------------------------------------------------
--
name            | DateStyle
setting         | ISO, MDY
[..]
sourcefile      |
```

```
sourceline      |
pending_restart | f
```

First of all, let's take a look at the pg_settings view. Using the pg_settings view, we can view the parameters set in the postgresql.conf configuration file. In the preceding result, we can see that the configuration for displaying the date is MDY (month/day/year). If we want to change this parameter globally, we have to edit the postgresql.conf file.

2. On a Debian or Debian-based server, we can edit the file as follows:

```
root@pgdev:/# vim /etc/postgresql/16/main/postgresql.conf
```

3. Then, we have to modify the following section:

```
#Locale and Formatting

datestyle = 'iso, mdy'
```

4. After changing this parameter, in the query on pg_settings, the context parameter is 'user'; we just need to do a reload of the server. In this case, a restart is not necessary:

```
root@pgdev:/# service postgresql reload
[ ok ] Reloading postgresql configuration (via systemctl):
postgresql.service.
```

For further information about the pg_settings view, we suggest visiting https://www.postgresql.org/docs/current/view-pg-settings.html.

5. We have learned what the internal parameters for date display are, so now, let's look at how to insert, update, and display dates. If we know the value of the date-style parameter, the PostgreSQL way of converting a string into a date is as follows:

```
forumdb=> \x
Expanded display is off.
forumdb=> select '12-31-2020'::date;
    date
------------
 2020-12-31
(1 row)
```

This way is simple but not particularly user-friendly. The best way to manage dates is by using some functions that PostgreSQL provides for us.

6. The first function that we'll talk about is the `to_date()` function. The `to_date()` function converts a given string into a date. The syntax of the `to_date()` function is as follows:

```
forumdb=> select to_date('31/12/2020','dd/mm/yyyy') ;
  to_date
------------
 2020-12-31
(1 row)
```

The `to_date()` function accepts two string parameters. The first parameter contains the value that we want to convert into a date. The second parameter is the pattern of the date. The `to_date()` function returns a date value.

7. Now, let's go back to the `posts` table and execute this query:

```
forumdb=> \x
Expanded display is on.
forumdb=> select pk,title,created_on from posts;
-[ RECORD 1 ]-----------------------------
pk         | 5
title      | Indexing PostgreSQL
created_on | 2023-01-23 15:21:55.747463+00
-[ RECORD 2 ]-----------------------------
pk         | 6
title      | Indexing Mysql
created_on | 2023-01-23 15:22:02.38953+00
-[ RECORD 3 ]-----------------------------
pk         | 7
title      | A view of  Data types in C++
created_on | 2023-01-23 15:26:21.367814+00
```

How is it possible that we have date/time combinations (timestamps) if nobody has ever entered these values into the table? It is possible because the `posts` table has been created as follows:

```
forumdb=> \d posts;
 Table "public.posts"
 Column          | Type                     |[...]| Default
----------------+-------------------------+[...]+--------
```

```
pk             | integer                  |      | [..]
title          | text                     |      |
[......]
created_on     | timestamp with time zone|      | CURRENT_TIMESTAMP
```

As we can see, the `created_on` field has `CURRENT_TIMESTAMP` as the default value, which means that if no value has been inserted, the current timestamp of the server will be inserted. Suppose now that we want to display the date in a different format – for example, in the European format, `created_on: 03-01-2020`.

8. To reach this goal, we have to use another built-in function, the `to_char` function:

```
forumdb=> select pk,title,to_char(created_on,'dd-mm-yyyy') as
created_on
from posts;
-[ RECORD 1 ]---------------------------
pk         | 5
title      | Indexing PostgreSQL
created_on | 23-01-2023
-[ RECORD 2 ]---------------------------
pk         | 6
title      | Indexing Mysql
created_on | 23-01-2023
-[ RECORD 3 ]---------------------------
pk         | 7
title      | A view of  Data types in C++
created_on | 23-01-2023
```

As shown here, the `to_char()` function is the inverse of the `to_date()` function.

Timestamp data types

PostgreSQL can manage dates and times with a time zone and without a time zone. We can store both date and time using the `timestamp` data type. In PostgreSQL, there is a data type called `timestamp with time zone` to display date and time with a time zone, and a data type called `timestamp without time zone` to store date and time without a time zone.

Let's now go through some examples. First of all, let's create a new table:

```
forumdb=> create table new_posts as select pk,title,created_on::timestamp
with time zone as created_on_t, created_on::timestamp without time zone as
```

```
create_on_nt from posts;
SELECT 3
```

We have just created a new table called `new_posts` with the following structure:

```
forumdb=# \d new_posts;
 Table "public.new_posts"
 Column       | Type                        | [...]
--------------+-----------------------------+----------
pk            | integer                     |
title         | text                        |
created_on_t  | timestamp with time zone    |
create_on_nt  | timestamp without time zone |
```

This table now has the same values for the `create_on_t (timestamp with time zone)` field and for the `created_on_nt (timestamp without time zone)` field, as we can see here:

```
forumdb=> select * from new_posts ;
-[ RECORD 1 ]+-----------------------------
pk           | 5
title        | Indexing PostgreSQL
created_on_t | 2023-01-23 15:21:55.747463+00
create_on_nt | 2023-01-23 15:21:55.747463
-[ RECORD 2 ]+-----------------------------
pk           | 6
title        | Indexing Mysql
created_on_t | 2023-01-23 15:22:02.38953+00
create_on_nt | 2023-01-23 15:22:02.38953
-[ RECORD 3 ]+-----------------------------
pk           | 7
title        | A view of  Data types in C++
created_on_t | 2023-01-23 15:26:21.367814+00
create_on_nt | 2023-01-23 15:26:21.367814
```

Now, let's introduce a PostgreSQL environment variable called the `timezone` variable. This variable tells us the current value of the time zone:

```
forumdb=> show timezone;
-[ RECORD 1 ]-----
TimeZone | Etc/UTC
```

In this server, the time zone is set to UTC; if we want to modify this value only on this session, we have to perform the following query:

```
forumdb=> set timezone='CET';
SET
```

Now, the time zone is set to CET:

```
forumdb=> show timezone;
-[ RECORD 1 ]-
TimeZone | CET
```

Now, if we execute the query that we performed previously again, we will see that the field with the time zone has changed its value:

```
forumdb=> select * from new_posts ;
-[ RECORD 1 ]+------------------------------
pk           | 5
title        | Indexing PostgreSQL
created_on_t | 2023-01-23 16:21:55.747463+01
create_on_nt | 2023-01-23 15:21:55.747463
-[ RECORD 2 ]+------------------------------
pk           | 6
title        | Indexing Mysql
created_on_t | 2023-01-23 16:22:02.38953+01
create_on_nt | 2023-01-23 15:22:02.38953
-[ RECORD 3 ]+------------------------------
pk           | 7
title        | A view of  Data types in C++
created_on_t | 2023-01-23 16:26:21.367814+01
create_on_nt | 2023-01-23 15:26:21.367814
```

This shows the difference between a timestamp with a time zone and a timestamp without a time zone. For further information on the topic of date and time, please refer to the official documentation at https://www.postgresql.org/docs/current/datatype-datetime.html.

The NoSQL data type

In this section, we will approach the NoSQL data types that are present in PostgreSQL. Since this book is not specifically focused on NoSQL, we will just take a quick look.

PostgreSQL handles the following NoSQL data types:

- `hstore`
- `xml`
- `json/jsonb`

We will now talk about `hstore` and `json`.

The hstore data type

`hstore` was the first NoSQL data type that was implemented in PostgreSQL. This data type is used for storing key-value pairs in a single value. Before working with the `hstore` data type, we need to enable the `hstore` extension on our server:

```
forumdb=> create extension hstore ;
CREATE EXTENSION
```

Let's look at how we can use the `hstore` data type with an example. Suppose that we want to show all posts with their usernames and their categories:

```
forumdb=> select p.pk,p.title,u.username,c.title as category
from posts p
inner join users u on p.author=u.pk
left join categories c on p.category=c.pk
order by 1;
-[ RECORD 1 ]-------------------------
pk       | 5
title    | Indexing PostgreSQL
username | luca_ferrari
category | Database
-[ RECORD 2 ]-------------------------
pk       | 6
title    | Indexing Mysql
username | luca_ferrari
category | Database
-[ RECORD 3 ]-------------------------
pk       | 7
title    | A view of  Data types in C++
username | enrico_pirozzi
category | Programming Languages
```

Suppose now that the table's posts, users, and categories are huge tables and we would like to store all the information about usernames and categories in a single field stored inside the `posts` table. If we could do this, we would no longer need to join three huge tables. In this case, `hstore` can help us:

```
forumdb=> select p.pk,p.title,hstore(ARRAY['username',u.
username,'category',c.title]) as options
from posts p
inner join users u on p.author=u.pk
left join categories c on p.category=c.pk
order by 1;
-[ RECORD 1 ]-------------------------
pk      | 5
title   | Indexing PostgreSQL
options | "category"=>"Database", "username"=>"luca_ferrari"
-[ RECORD 2 ]-------------------------
pk      | 6
title   | Indexing Mysql
options | "category"=>"Database", "username"=>"luca_ferrari"
-[ RECORD 3 ]-------------------------
pk      | 7
title   | A view of  Data types in C++
options | "category"=>"Programming Languages", "username"=>"enrico_
pirozzi"
```

The preceding query first puts in an array the values of the `username` and `category` fields, and then transforms them into `hstore`. Now, if we want to store the data in a new table called `posts_options`, we have to perform something like the following:

```
forumdb=> create table posts_options as
select p.pk,p.title,hstore(ARRAY['username',u.username,'category',c.
title]) as options
from posts p
inner join users u on p.author=u.pk
left join categories c on p.category=c.pk
order by 1;
SELECT 3
```

We now have a new table with the following structure:

```
forumdb=> \d posts_options
            Table "forum.posts_options"
 Column  |  Type   | Collation | Nullable | Default
---------+---------+-----------+----------+---------
 pk      | integer |           |          |
 title   | text    |           |          |
 options | hstore  |           |          |
```

Next, suppose that we want to search for all the records that have `category = 'Database'`. We would have to execute the following:

```
forumdb=> select * from posts_options where options->'category'
='Database';
-[ RECORD 1 ]-------------------------
pk      | 5
title   | Indexing PostgreSQL
options | "category"=>"Database", "username"=>"luca_ferrari"
-[ RECORD 2 ]-------------------------
pk      | 6
title   | Indexing Mysql
options | "category"=>"Database", "username"=>"luca_ferrari"
```

Since `hstore`, as well as the `json/jsonb` data types, is not a structured data type, we can insert any other key value without defining it first – for example, we can do this:

```
forumdb=> insert into posts_options (pk,title,options) values (7,'my last
post','"enabled"=>"false"') ;
INSERT 0 1
```

The result of the selection on the whole table will be the following:

```
forumdb=>  select * from posts_options;
-[ RECORD 1 ]-------------------------
pk      | 5
title   | Indexing PostgreSQL
options | "category"=>"Database", "username"=>"luca_ferrari"
-[ RECORD 2 ]-------------------------
pk      | 6
```

```
title   | Indexing Mysql
options | "category"=>"Database", "username"=>"luca_ferrari"
-[ RECORD 3 ]-------------------------
pk      | 7
title   | A view of  Data types in C++
options | "category"=>"Programming Languages", "username"=>"enrico_
pirozzi"
-[ RECORD 4 ]-------------------------
pk      | 7
title   | my last post
options | "enabled"=>"false"
```

As we said at the beginning of this section, NoSQL is not the subject of this book, but it is worth briefly going over it. For further information about the hstore data type, please refer to the official documentation at `https://www.postgresql.org/docs/current/hstore.html`.

The JSON data type

In this section, we'll take a brief look at the JSON data type. **JSON** stands for **JavaScript Object Notation**. JSON is an open standard format, and it is formed of key-value pairs. PostgreSQL supports the JSON data type natively. It provides many functions and operators used for manipulating JSON data. PostgreSQL, in addition to the `json` data type, also supports the `jsonb` data type. The difference between these two data types is that the first is internally represented as text, whereas the second is internally represented in a binary and indexable manner. Let's look at how we can use the `json/jsonb` data types with an example.

Suppose that we want to show all the posts and tags that we have in our `forumdb` database. Working in a classic relational SQL way, we should write something like the following:

```
forumdb=> \x
Expanded display is off.
forumdb=> select p.pk,p.title,t.tag
from posts p
left join j_posts_tags jpt on p.pk=jpt.post_pk
left join tags t on jpt.tag_pk=t.pk
order by 1;
 pk |            title             |        tag
----+------------------------------+-------------------
```

```
   5 | Indexing PostgreSQL          | Operating Systems
   5 | Indexing PostgreSQL          | Database
   6 | Indexing Mysql               | Database
   6 | Indexing Mysql               | Operating Systems
   7 | A view of  Data types in C++ | Database
(5 rows)
```

Suppose now that we want to have a result like the following:

pk	title	tag
5	Indexing PostgreSQL	Operating Systems,Database
6	Indexing PostgreSQL	Database,Operating Systems
7	A view of Data types in C++	Database

In a relational way, we have to aggregate data using the first two fields and perform something like the following:

```
forumdb=> \x
Expanded display is on.
forumdb=> select p.pk,p.title,string_agg(t.tag,',') as tag
from posts p
left join j_posts_tags jpt on p.pk=jpt.post_pk
left join tags t on jpt.tag_pk=t.pk
group by 1,2
order by 1;
-[ RECORD 1 ]----------------------
pk    | 5
title | Indexing PostgreSQL
tag   | Operating Systems,Database
-[ RECORD 2 ]----------------------
pk    | 6
title | Indexing Mysql
tag   | Database,Operating Systems
-[ RECORD 3 ]----------------------
pk    | 7
```

```
title | A view of  Data types in C++
tag   | Database
```

Now, imagine that we want to generate a simple JSON structure; we would execute the following query:

```
forumdb=> select row_to_json(q) as json_data from (
 select p.pk,p.title,string_agg(t.tag,',') as tag
 from posts p
 left join j_posts_tags jpt on p.pk=jpt.post_pk
 left join tags t on jpt.tag_pk=t.pk
group by 1,2 order by 1) Q;
-[ RECORD 1 ]----------------------
json_data | {"pk":5,"title":"Indexing PostgreSQL","tag":"Operating
Systems,Database"}
-[ RECORD 2 ]----------------------
json_data | {"pk":6,"title":"Indexing Mysql","tag":"Database,Operating
Systems"}
-[ RECORD 3 ]----------------------
json_data | {"pk":7,"title":"A view of  Data types in
C++","tag":"Database"}
```

As we can see, with a simple query, it is possible to switch from a classic SQL representation to a NoSQL representation. Now, let's create a new table called `post_json`. This table will have only one `jsonb` field, called `jsondata`:

```
forumdb=> create table post_json (jsondata jsonb);
CREATE TABLE
forumdb=> \d post_json
              Table "forum.post_json"
  Column   | Type  | Collation | Nullable | Default
----------+-------+-----------+----------+---------
 jsondata | jsonb |           |          |
```

Now, let's insert some data into the `post_json` table:

```
forumdb=> insert into post_json(jsondata)
select row_to_json(q) as json_data from (
  select p.pk,p.title,string_agg(t.tag,',') as tag
  from posts p
```

```
  left join j_posts_tags jpt on p.pk=jpt.post_pk
  left join tags t on jpt.tag_pk=t.pk
group by 1,2 order by 1) Q;
INSERT 0 3
```

Now, the `post_json` table has the following records:

```
forumdb=> select jsonb_pretty(jsondata) from post_json;
-[ RECORD 1 ]+-----------------------
jsonb_pretty | {                                     +
             |     "pk": 5,                          +
             |     "tag": "Operating Systems,Database",   +
             |     "title": "Indexing PostgreSQL"         +
             | }
-[ RECORD 2 ]+-----------------------
jsonb_pretty | {                                     +
             |     "pk": 6,                          +
             |     "tag": "Database,Operating Systems",   +
             |     "title": "Indexing Mysql"             +
             | }
-[ RECORD 3 ]+-----------------------
jsonb_pretty | {                                     +
             |     "pk": 7,                          +
             |     "tag": "Database",                +
             |     "title": "A view of  Data types in C++"+
             | }
```

If we wanted to search for all data that has `tag = "Database"`, we could use the `@> jsonb` operator. This operator checks whether the left JSON value contains the right JSON path/value entries at the top level; the following query makes this search possible:

```
forumdb=> select jsonb_pretty(jsondata) from post_json where jsondata @>
'{"tag":"Database"}';

-[ RECORD 1 ]+-----------------------
jsonb_pretty | {                                     +
             |     "pk": 7,                          +
             |     "tag": "Database",                +
```

```
               |     "title": "A view of  Data types in C++"+
               | }
```

What we have just written is just a small taste of what can be done through the NoSQL data model. JSON is widely used when working with large tables and when a data structure is needed that minimizes the number of joins to be done during the research phase. A detailed discussion of the NoSQL world is beyond the scope of this book, but we wanted to describe briefly how powerful PostgreSQL is in the approach to unstructured data as well. For more information, please look at the official documentation at `https://www.postgresql.org/docs/current/functions-json.html`.

After understanding what data types are and which data types can be used in PostgreSQL, in the next section, we will see how to use data types within functions.

Exploring functions and languages

PostgreSQL is capable of executing server-side code. There are many ways to provide PostgreSQL with the code to be executed. For example, the user can create functions in different programming languages. The main languages supported by PostgreSQL are as follows:

- SQL
- PL/pgSQL
- C

These listed languages are the built-in languages; there are also other languages that PostgreSQL can manage, but before using them, we need to install them on our system. Some of these other supported languages are as follows:

- PL/Python
- PL/Perl
- PL/tcl
- PL/Java

In this section, we'll talk about SQL and PL/pgSQL functions.

Functions

The command structure with which a function is defined is as follows:

```
CREATE FUNCTION function_name(p1 type, p2 type,p3 type, ....., pn type)
 RETURNS type AS
```

```
BEGIN
 -- function logic
END;
LANGUAGE language_name
```

The following steps always apply to any type of function we want to create:

1. Specify the name of the function after the `CREATE FUNCTION` keywords.
2. Make a list of parameters separated by commas.
3. Specify the return data type after the `RETURNS` keyword.
4. For the PL/pgSQL language, put some code between the `BEGIN` and `END` blocks.
5. For the PL/pgSQL language, the function has to end with the `END` keyword followed by a semicolon.
6. Define the language in which the function was written – for example, `sql` or `plpgsql`, `plperl`, `plpython`, and so on.

This is the basic scheme to which we will refer later in the chapter; this scheme may have small variations in some specific cases.

SQL functions

SQL functions are the easiest way to write functions in PostgreSQL, and we can use any SQL command inside them.

Basic functions

This section will show how to take your first steps into the SQL functions world. For example, the following function carries out a sum between two numbers:

```
forumdb=> CREATE OR REPLACE FUNCTION my_sum(x integer, y integer) RETURNS
integer AS $$
 SELECT x + y;
$$ LANGUAGE SQL;
CREATE FUNCTION


forumdb=> select my_sum(1,2);
 my_sum
--------
```

```
      3
(1 row)
```

As we can see in the preceding example, the code function is placed between `$$`; we can consider `$$` as labels. The function can be called using the `SELECT` statement without using any `FROM` clauses. The arguments of a SQL function can be referenced in the function body using either numbers (the old way) or their names (the new way). For example, we could write the same function in this way:

```
CREATE OR REPLACE FUNCTION my_sum(integer, integer) RETURNS integer AS $$
 SELECT $1 + $2;
$$ LANGUAGE SQL;
```

In the preceding function, we can see the old way to reference the parameter inside the function. In the old way, the parameters were referenced positionally, so the value `$1` corresponds to the first parameter of the function, `$2` to the second, and so on. In the code of the SQL functions, we can use all the SQL commands, including those seen in previous chapters.

SQL functions returning a set of elements

In this section, we will look at how to make a SQL function that returns a result set of a data type. For example, suppose that we want to write a function that takes `p_title` as a parameter and deletes all the records that have `title=p_title`, as well as returning all the keys of the deleted records. The following function would make this possible:

```
forumdb=> CREATE OR REPLACE FUNCTION delete_posts(p_title text) returns
setof integer as $$
delete from posts where title=p_title returning pk;
$$
LANGUAGE SQL;
CREATE FUNCTION
```

This is the situation before we called the `delete_posts` function:

```
forumdb=> select pk,title from posts order by pk;
 pk |           title
----+------------------------------
  5 | Indexing PostgreSQL
  6 | Indexing Mysql
  7 | A view of  Data types in C++
(3 rows)
```

Now, suppose that we want to delete the record that has the field title equal to `A view of  Data types in C++`. The table posts has the `pk` field as the primary key, and for the record `A view of  Data types in C++`, the value of `pk` is equal to `7`; so first of all, let's delete the records from the `j_posts_tags` table for which the value `post_pk=7`. This is because there is a foreign key that links the posts and `j_posts_tags` tables:

```
forumdb=> delete from j_posts_tags where post_pk = 7;
DELETE 1
```

Now let's call the `delete_posts` function using `A view of  Data types in C++` as the parameter. This is the situation after we called the `delete_posts` function:

```
forumdb=> select delete_posts('A view of  Data types in C++');
 delete_posts
--------------
            7
(1 row)
forumdb=> select pk,title from posts order by pk;
 pk |        title
----+---------------------
  5 | Indexing PostgreSQL
  6 | Indexing Mysql
(2 rows)
```

In this function, we've introduced a new kind of data type – the `setof` data type. The `setof` directive simply defines a result set of a data type. For example, the `delete_posts` function is defined to return a set of integers, so its result will be an integer dataset. We can use the `setof` directive with any type of data.

SQL functions returning a table

In the previous section, we saw how to write a function that returns a result set of a single data type; however, it is possible that there will be cases where we need our function to return a result set of multiple fields. For example, let's consider the same function as before, but this time, we want the `pk`, `title` pair to be returned as a result, so our function becomes the following:

```
forumdb=> create or replace function delete_posts_table (p_title text)
returns table (ret_key integer,ret_title text) AS $$
delete from posts where title=p_title returning pk,title;
$$
```

```
language SQL;
CREATE FUNCTION
```

The only difference between this and the previous function is that now the function returns a `table` type; inside the `table` type, we have to specify the name and the type of the fields. As we have seen before, this is the situation before calling the function:

```
forumdb=> select pk,title from posts order by pk;
 pk |        title
----+---------------------
  5 | Indexing PostgreSQL
  6 | Indexing Mysql
(2 rows)
```

Let's now insert a new record:

```
forumdb=> insert into posts(title,author,category) values ('My new
post',1,1);
INSERT 0 1
```

Now let's call the `delete_posts_table` function. The correct way to call the function is:

```
forumdb=> select * from  delete_posts_table('My new post');
 ret_key |  ret_title
---------+-------------
       9 | My new post
(1 row)
)
```

This is the situation after calling the function:

```
forumdb=> select pk,title from posts order by pk;
 pk |        title
----+---------------------
  5 | Indexing PostgreSQL
  6 | Indexing Mysql
(2 rows)
```

The functions that return a table can be treated as real tables, in the sense that we can use them with the `in`, `exists`, `join`, and so on options.

Polymorphic SQL functions

In this section, we will briefly talk about polymorphic SQL functions.

Polymorphic functions are useful for DBAs when we need to write a function that has to work with different types of data. To better understand polymorphic functions, let's start with an example. Suppose we want to recreate something that looks like the Oracle NVL function – in other words, we want to create a function that accepts two parameters and replaces the first parameter with the second one if the first parameter is NULL. The problem is that we want to write a single function that is valid for all types of data (integer, real, text, and so on).

The following function makes this possible:

```
forumdb=> create or replace function nvl ( anyelement,anyelement) returns
anyelement as $$
select coalesce($1,$2);
$$
language SQL;
CREATE FUNCTION
```

This is how to call it:

```
forumdb=> select nvl(NULL::int,1);
 nvl
-----
   1
(1 row)

forumdb=> select nvl(''::text,'n'::text);
 nvl
-----

(1 row)

forumdb=> select nvl('a'::text,'n'::text);
 nvl
-----
 a
(1 row)
```

For further information, see the official documentation at `https://www.postgresql.org/docs/current/extend-type-system.html`.

PL/pgSQL functions

In this section, we'll talk about the PL/pgSQL language. The PL/pgSQL language is the default built-in procedural language for PostgreSQL. As described in the official documentation, the design goals with PL/pgSQL were to create a loadable procedural language that can do the following:

- Can be used to create functions and trigger procedures (we'll talk about triggers in the next chapter).
- Add new control structures.
- Add new data types to the SQL language.

It is very similar to Oracle PL/SQL and supports the following:

- Variable declarations
- Expressions
- Control structures as conditional structures or loop structures
- Cursors

First overview

As we saw at the beginning of the *SQL functions* section, the prototype for writing functions in PostgreSQL is as follows:

```
CREATE FUNCTION function_name(p1 type, p2 type,p3 type, ....., pn type)
 RETURNS type AS
BEGIN
 -- function logic
END;
LANGUAGE language_name
```

Now, suppose that we want to recreate the `my_sum` function using the PL/pgSQL language:

```
forumdb=> CREATE OR REPLACE FUNCTION my_sum(x integer, y integer) RETURNS
integer AS
$BODY$
DECLARE
 ret integer;
BEGIN
```

```
  ret := x + y;
  return ret;
END;
$BODY$
language 'plpgsql';
CREATE FUNCTION
forumdb=> select my_sum(2,3);
 my_sum
--------
      5
(1 row)
```

The preceding query provides the same results as the query seen at the beginning of the chapter. Now, let's examine it in more detail:

1. The following is the function header; here, you define the name of the function, the input parameters, and the return value:

   ```
   CREATE OR REPLACE FUNCTION my_sum(x integer, y integer) RETURNS
   integer AS
   ```

2. The following is a label indicating the beginning of the code. We can put any string in between the `$$` characters; the important thing is that the same label is present at the end of the function:

   ```
   $BODY$
   ```

3. In the following section, we can define our variables; it is important that each declaration or statement ends with a semicolon:

   ```
   DECLARE
     ret integer;
   ```

4. With the `BEGIN` statement, we tell PostgreSQL that we want to start to write our logic:

   ```
   BEGIN
     ret := x + y;
     return ret;
   ```

> Caution: Do not write a semicolon after `BEGIN` – it's not correct and it will generate a syntax error.

5. Between the `BEGIN` statement and the `END` statement, we can put our own code:

```
END;
```

6. The `END` instruction indicates that our code has ended:

```
$BODY$
```

7. This label closes the first label and at last, the `language` statement specifies PostgreSQL, in which the function is written:

```
language 'plpgsql';
```

Dropping functions

To drop a function, we have to execute the `DROP FUNCTION` command followed by the name of the function and its parameters. For example, to drop the `my_sum` function, we have to execute:

```
forumdb=> DROP FUNCTION my_sum(integer,integer);
DROP FUNCTION
```

Declaring function parameters

After learning about how to write a simple PL/pgSQL function, let's go into a little more detail about the single aspects seen in the preceding section. Let's start with the declaration of the parameters. In the next two examples, we'll see how to define, in two different ways, the `my_sum` function that we have seen before.

The first example is as follows:

```
forumdb=> CREATE OR REPLACE FUNCTION my_sum(integer, integer) RETURNS
integer AS
$BODY$
DECLARE
 x alias for $1;
 y alias for $2;
```

```
 ret integer;
BEGIN
 ret := x + y;
 return ret;
END;
$BODY$
language 'plpgsql';
CREATE FUNCTION
```

The second example is as follows:

```
forumdb=> CREATE OR REPLACE FUNCTION my_sum(integer, integer) RETURNS
integer AS
$BODY$
DECLARE
 ret integer;
BEGIN
 ret := $1 + $2;
 return ret;
END;
$BODY$
language 'plpgsql';
CREATE FUNCTION
```

In the first example, we used `alias`; the syntax of `alias` is, in general, the following:

```
newname ALIAS FOR oldname;
```

In our specific case, we used the positional variable `$1` as the `oldname` value. In the second example, we used the positional approach exactly as we did in the case of SQL functions.

IN/OUT parameters

In the preceding example, we used the `RETURNS` clause in the first row of the function definition; however, there is another way to reach the same goal. In PL/pgSQL, we can define all parameters as input parameters, output parameters, or input/output parameters. For example, say we write the following:

```
forumdb=> CREATE OR REPLACE FUNCTION my_sum_3_params(IN x integer,IN y
integer, OUT z integer) AS
$BODY$
```

```
BEGIN
 z := x+y;
END;
$BODY$
language 'plpgsql';
CREATE FUNCTION
```

We have defined a new function called `my_sum_3_params`, which accepts two input parameters (x and y) and has an output of parameter z. As there are two input parameters, the function will be called with only two parameters, exactly as in the last function:

```
forumdb=> select my_sum_3_params(2,3);
 my_sum_3_params
-----------------
               5
(1 row)
```

With this kind of parameter definition, we can have functions that have multiple variables as a result. For example, if we want a function that, given two integer values, computes their sum and their product, we can write something like this:

```
forumdb=> CREATE OR REPLACE FUNCTION my_sum_mul(IN x integer,IN y
integer,OUT w integer, OUT z integer) AS
$BODY$
BEGIN
 z := x+y;
 w := x*y;
END;
$BODY$
language 'plpgsql';
CREATE FUNCTION
```

The strange thing is that if we invoke the function as we did before, we will have the following result:

```
forumdb=> select my_sum_mul(2,3);
 my_sum_mul
------------
 (6,5)
(1 row)
```

This result seems to be a little bit strange because the result is not a scalar value but a record, which is a custom type. To cause the output to be separated as columns, we have to use the following syntax:

```
forumdb=> select * from my_sum_mul(2,3);
 w | z
---+---
 6 | 5
(1 row)
```

We can use the result of the function exactly as if it were a result of a table and write, for example, the following:

```
forumdb=> select * from my_sum_mul(2,3) where w=6;
 w | z
---+---
 6 | 5
(1 row)
```

We can define the parameters as follows:

- `IN`: Input parameters (if omitted, this is the default option)
- `OUT`: Output parameters
- `INOUT`: Input/output parameters

Function volatility categories

In PostgreSQL, each function can be defined as `VOLATILE`, `STABLE`, or `IMMUTABLE`. If we do not specify anything, the default value is `VOLATILE`. The difference between these three possible definitions is well described in the official documentation (`https://www.postgresql.org/docs/current/xfunc-volatility.html`):

> A `VOLATILE` function can do everything, including modifying the database. It can return different results on successive calls with the same arguments. The optimizer makes no assumptions about the behavior of such functions. A query using a volatile function will re-evaluate the function at every row where its value is needed. If a function is marked as `VOLATILE`, it can return different results if we call it multiple times using the same input parameters.
>
> A `STABLE` function cannot modify the database and is guaranteed to return the same results given the same arguments for all rows within a single statement. This category allows the optimizer to optimize multiple calls of the function to a single call. In particular, it is safe to use an expression containing such a function in an index scan condition. If a function is marked as `STABLE`, the function will return the same result given the same parameters within the same transaction.
>
> An `IMMUTABLE` function cannot modify the database and is guaranteed to return the same results given the same arguments forever. This category allows the optimizer to pre-evaluate the function when a query calls it with constant arguments.

In the following pages of this chapter, we will only be focusing on examples of volatile functions; however, here we will briefly look at one example of a stable function and one example of an immutable function:

1. Let's start with a stable function – for example, the `now()` function is a stable function. The `now()` function returns the current date and time that we have at the beginning of the transaction, as we can see here:

```
forumdb=> begin ;
BEGIN

forumdb=*> select now();
             now
------------------------------
```

```
 2023-03-17 13:25:25.37224+00
(1 row)

forumdb=*> select now();
             now
------------------------------
 2023-03-17 13:25:25.37224+00
(1 row)


forumdb=*> commit;
COMMIT


forumdb=> begin ;
BEGIN

forumdb=*> select now();
              now
-------------------------------
 2023-03-17 13:27:02.012632+00
(1 row)

forumdb=*> commit ;
COMMIT
```

Note: In PostgreSQL 16, when psql shows us a prompt like `*>`, it means that we are inside a transaction block.

2. Now, let's look at an immutable function – for example, the `lower(string_expression)` function. The `lower` function accepts a string and converts it into a lowercase format. As we can see, if the input parameters are the same, the `lower` function always returns the same result, even if it is performed in different transactions:

```
forumdb=> begin;
BEGIN

forumdb=*> select now();
```

```
              now
-------------------------------
 2023-03-17 13:33:39.586388+00
(1 row)

forumdb=*> select lower('MICKY MOUSE');
    lower
-------------
 micky mouse
(1 row)

forumdb=*> commit;
COMMIT

forumdb=> begin;
BEGIN

forumdb=*> select now();
              now
-------------------------------
 2023-03-17 13:34:56.491773+00
(1 row)


forumdb=*> select lower('MICKY MOUSE');
    lower
-------------
 micky mouse
(1 row)

forumdb=*> commit;
COMMIT
```

Control structure

PL/pgSQL has the ability to manage control structures such as the following:

- Conditional statements

- Loop statements
- Exception handler statements

Conditional statements

The PL/pgSQL language can manage `IF`-type conditional statements and `CASE`-type conditional statements.

IF statements

In PL/pgSQL, the syntax of an `IF` statement is as follows:

```
IF boolean-expression THEN
 statements
[ ELSIF boolean-expression THEN
 statements
[ ELSIF boolean-expression THEN
 statements
 ...
]
]
[ ELSE
 statements ]
END IF;
```

For example, say we want to write a function that, when given the two input values, `x` and `y`, returns the following:

- *first parameter is greater than second parameter* if `x > y`
- *second parameter is greater than first parameter* if `x < y`
- *the 2 parameters are equals if* `x = y`

We have to write the following function:

```
forumdb=> CREATE OR REPLACE FUNCTION my_check(x integer default 0, y
integer default 0) RETURNS text AS
$BODY$
BEGIN
 IF x > y THEN
 return 'first parameter is greater than second parameter';
```

```
 ELSIF x < y THEN
 return 'second parameter is greater than first parameter';
 ELSE
 return 'the 2 parameters are equals';
 END IF;
END;
$BODY$
language 'plpgsql';
CREATE FUNCTION
```

In this example, we have seen the `IF` construct in its largest form: `IF [...] THEN[...] ELSIF [...] ELSE[...] ENDIF;`

However, shorter forms also exist, as follows:

- `IF [...] THEN[...] ELSE[...] ENDIF;`
- `IF [...] THEN[...] ENDIF;`

Some examples of the results provided by the previously defined function are as follows:

```
forumdb=> select my_check(1,2);
                    my_check
--------------------------------------------------
 second parameter is higher than first parameter
(1 row)


forumdb=> select my_check(2,1);
                    my_check
--------------------------------------------------
 first parameter is higher than second parameter
(1 row)

forumdb=>  select my_check(1,1);
          my_check
-----------------------------
 the 2 parameters are equals
(1 row)
```

CASE statements

In PL/pgSQL, it is also possible to use the `CASE` statement. The `CASE` statement can have the following two syntaxes.

The following is a simple `CASE` statement:

```
CASE search-expression
 WHEN expression [, expression [ ... ]] THEN
 statements
 [ WHEN expression [, expression [ ... ]] THEN
 statements
 ... ]
 [ ELSE
 statements ]
END CASE;
```

The following is a searched `CASE` statement:

```
CASE
 WHEN boolean-expression THEN
 statements
 [ WHEN boolean-expression THEN
 statements
 ... ]
 [ ELSE
 statements ]
END CASE;
```

Now, we will perform the following operations:

- We will use the first one, the simple `CASE` syntax, if we have to make a choice from a list of values.
- We will use the second one when we have to choose from a range of values.

Let's start with the first syntax:

```
forumdb=> CREATE OR REPLACE FUNCTION my_check_value(x integer default 0)
RETURNS text AS
$BODY$
BEGIN
```

```
 CASE x
 WHEN 1 THEN return 'value = 1';
 WHEN 2 THEN return 'value = 2';
 ELSE return 'value >= 3 ';
 END CASE;
END;
$BODY$
language 'plpgsql';
CREATE FUNCTION
```

The preceding `my_check_value` function returns the following:

- `value = 1` if x = 1
- `value = 2` if x = 2
- `value >= 3` if x >= 3

We can see this to be true here:

```
forumdb=> select my_check_value(1);
 my_check_value
----------------
 value = 1
(1 row)

forumdb=> select my_check_value(2);
 my_check_value
----------------
 value = 2
(1 row)

forumdb=> select my_check_value(3);
 my_check_value
----------------
 value >= 3
(1 row)
```

Now, let's see an example of the searched `CASE` syntax:

```
forumdb=> CREATE OR REPLACE FUNCTION my_check_case(x integer default 0, y
integer default 0) RETURNS text AS
```

```
 $BODY$
 BEGIN
   CASE
    WHEN x > y THEN return 'first parameter is higher than second
parameter';
    WHEN x < y THEN return 'second parameter is higher than first
parameter';
 ELSE return 'the 2 parameters are equals';
 END CASE;
 END;
 $BODY$
 language 'plpgsql';
CREATE FUNCTION
```

The `my_check_case` function returns the same data as the `my_check` function that we wrote before:

```
forumdb=> select my_check_case(2,1);
                  my_check_case
-------------------------------------------------
 first parameter is higher than second parameter
(1 row)

forumdb=> select my_check_case(1,2);
                  my_check_case
-------------------------------------------------
 second parameter is higher than first parameter
(1 row)

forumdb=> select my_check_case(1,1);
        my_check_case
-----------------------------
 the 2 parameters are equals
(1 row)

forumdb=> select my_check_case();
        my_check_case
-----------------------------
```

```
 the 2 parameters are equals
(1 row)
```

Loop statements

PL/pgSQL can handle loops in many ways. We will look at some examples of how to make a loop next. For further details, we suggest referring to the official documentation at `https://www.postgresql.org/docs/current/plpgsql.html`. What makes PL/pgSQL particularly useful is the fact that it allows us to process data from queries through procedural language. We are going to see now how this is possible.

Suppose that we want to build a PL/pgSQL function that, when given an integer as a parameter, returns a result set of a composite data type. The composite data type that we want it to return is as follows:

ID	pk field	Integer data type
TITLE	Title field	`text` data type
RECORD_DATA	Title field + content field	`hstore` data type

The right way to build a composite data type is as follows:

```
forumdb=> create type my_ret_type as (
 id integer,
 title text,
 record_data hstore
);
CREATE TYPE
```

The preceding statement creates a new data type, a composite data type, which is composed of an integer data type + a `text` data type + an `hstore` data type. Now, if we want to write a function that returns a result set of the `my_ret_type` data type, our first attempt might be as follows:

```
forumdb=> CREATE OR REPLACE FUNCTION my_first_fun (p_id integer) returns
setof my_ret_type as
$$
DECLARE
 rw posts%ROWTYPE; -- declare a rowtype;
 ret my_ret_type;
BEGIN
```

```
    for rw in select * from posts where pk=p_id loop
      ret.id := rw.pk;
      ret.title := rw.title;
      ret.record_data := hstore(ARRAY['title',rw.title,'Title and Content'
                          ,format('%s %s',rw.title,rw.content)]);
     return next ret;
     end loop;
 return;
END;
$$
language 'plpgsql';
CREATE FUNCTION
```

As we can see, many things are concentrated in these few lines of PL/pgSQL code:

1. `rw posts%ROWTYPE`: With this statement, the `rw` variable is defined as a container of a single row of the `posts` table.
2. `for rw in select * from posts where pk=p_id loop`: With this statement, we cycle within the result of the selection, assigning the value returned by the `select` command each time to the `rw` variable. The next three steps assign the values to the `ret` variable.
3. `return next ret;`: This statement returns the value of the `ret` variable and goes to the next record of the `for` cycle.
4. `end loop;`: This statement tells PostgreSQL that the `for` cycle ends here.
5. `return;`: This is the return instruction of the function.

> An important thing to remember is that the PL/pgSQL language is inside the PostgreSQL transaction system. This means that the functions are executed atomically and that the function returns the results not at the execution of the `RETURN NEXT` command but at the execution of the `RETURN` command placed at the end of the function. This may mean that for very large datasets, the PL/pgSQL functions can take a long time before returning results.

The record type

In an example that we used previously, we introduced the `%ROWTYPE` data type. In the PL/pgSQL language, it is possible to generalize this concept. There is a data type called `record` that generalizes the concept of `%ROWTYPE`.

For example, we can rewrite `my_first_fun` in the following way:

```
forumdb=> CREATE OR REPLACE FUNCTION my_second_fun (p_id integer) returns
setof my_ret_type as
$$
DECLARE
   rw record; -- declare a record variable
   ret my_ret_type;
BEGIN
   for rw in select * from posts where pk=p_id loop
   ret.id := rw.pk;
   ret.title := rw.title;
   ret.record_data := hstore(ARRAY['title',rw.title
                    ,'Title and Content',format('%s %s',rw.title,rw.
content)]);
   return next ret;
 end loop;
 return;
END;
$$
language 'plpgsql';
CREATE FUNCTION
```

The only difference between `my_first_fun` and `my_second_fun` is in this definition:

```
rw record; -- declare a record variable
```

This time, the `rw` variable is defined as a `record` data type. This means that the `rw` variable is an object that can be associated with any records of any table. The result of the two functions, `my_first_fun` and `my_second_fun`, is the same:

```
forumdb=> \x
Expanded display is on.
forumdb=> select * from my_first_fun(5);
-[ RECORD 1 ]-----------------------
id          | 5
title       | Indexing PostgreSQL
record_data | "title"=>"Indexing PostgreSQL", "Title and
Content"=>"Indexing PostgreSQL Btree in PostgreSQL is...."
```

Exception handling statements

PL/pgSQL can also handle exceptions. The `BEGIN...END` block of a function allows the `EXCEPTION` option, which works as a catch for exceptions. For example, if we write a function to divide two numbers, we could have a problem with a division by 0:

```
forumdb=> CREATE OR REPLACE FUNCTION my_first_except (x real, y real )
returns real as
$$
DECLARE
 ret real;
BEGIN
 ret := x / y;
 return ret;
END;
$$
language 'plpgsql';
CREATE FUNCTION
```

This function works well if y <> 0, as we can see here:

```
forumdb=> \x
Expanded display is off.
forumdb=> select my_first_except(4,2);
 my_first_except
-----------------
               2
(1 row)
```

However, if y assumes a 0 value, we have a problem:

```
forumdb=> select my_first_except(4,0);
ERROR:  division by zero
CONTEXT:  PL/pgSQL function my_first_except(real,real) line 5 at
assignment
```

To solve this problem, we have to handle the exception. To do this, we have to rewrite our function in the following way:

```
forumdb=> CREATE OR REPLACE FUNCTION my_second_except (x real, y real )
returns real as
```

```
$$
DECLARE
  ret real;
BEGIN
  ret := x / y;
  return ret;
EXCEPTION
  WHEN division_by_zero THEN
     RAISE INFO 'DIVISION BY ZERO';
     RAISE INFO 'Error % %', SQLSTATE, SQLERRM;
     RETURN 0;
END;
$$
language 'plpgsql' ;
CREATE FUNCTION
```

The `SQLSTATE` and `SQLERRM` variables contain the status and message associated with the generated error. Now, if we execute the second function, we no longer get an error from PostgreSQL:

```
forumdb=> select my_second_except(4,0);
INFO:  DIVISION BY ZERO
INFO:  Error 22012 division by zero
 my_second_except
------------------
                0
(1 row)
```

The list of errors that PostgreSQL can manage is available at `https://www.postgresql.org/docs/current/errcodes-appendix.html`.

Security definer

This option allows the user to invoke a function as if they were its owner. It can be useful in all cases where we want to display data to which the average user does not have access.

For example, in PostgreSQL, there is a system view called `pg_stat_activity`, which allows us to view what PostgreSQL is currently doing.

As user `forum`, let's execute this statement:

```
postgres@learn_postgresql:~$ psql -U forum forumdb
```

```
forumdb=>

forumdb=> select pid,query from pg_stat_activity  ;
 pid |                  query
-----+------------------------------------------
  74 | <insufficient privilege>
  75 | <insufficient privilege>
 217 | select pid,query from pg_stat_activity ;
 [..]
```

As we can see above, there are some `<insufficient privilege>` results. Here are the steps to solve this problem:

- Let's connect to the database as user `postgres`:

```
postgres@learn_postgresql:~$ psql forumdb
forumdb=#
```

- Now let's execute the function `my_stat_activity()` written here:

```
forumdb=# create function forum.my_stat_activity()
returns table (pid integer,query text)
as $$
     select pid, query  from pg_stat_activity;
$$ language 'sql'
security definer;
```

- Let's give the execute permission to the `forum` user on the function `my_stat_activity`. We will see this feature in *Chapter 10*, *Granting and Revoking Permissions*:

```
forumdb=# grant execute on function forum.my_stat_activity TO forum;
```

- Let's connect again to the database as user `forum`:

```
postgres@learn_postgresql:~$ psql -U forum forumdb
forumdb=>
```

- Now let's execute the query written below:

```
forumdb=> select * from my_stat_activity();
 pid |               query
-----+-----------------------------------
```

```
  74 |
  75 |
 271 | select * from my_stat_activity();
 [..]
```

We no longer have the problem we had before. This is because the `security definer` allows the `forum.my_stat_activity()` function to be executed with the permissions of the user who created it, and in this case, the user who created it is the `postgres` user.

Summary

In this chapter, we introduced the world of server-side programming. The topic is so vast that there are specific books dedicated just to it. We have tried to give you a better understanding of the main concepts of server-side programming. We talked about the main data types managed by PostgreSQL, then we saw how it is possible to create new ones using composite data types. We also mentioned SQL functions and polymorphic functions, and finally, we provided some information about the PL/pgSQL language.

In the next chapter, we will use these concepts to introduce event management in PostgreSQL. We will talk about event management through the use of triggers and the functions associated with them.

Verify your knowledge

- Is it possible to extend Is it possible to extend features and data types in postgresql?

  Yes it is, we can extend PostgreSQL in terms of data types and in terms of functions.

  See the *The concept of extensibility* section for more details.

- Does PostgreSQL support only relational databases?

  No, PostgreSQL supports NoSQL databases too.

  See the *The NoSql data type* section for more details.

- Does PostgreSQL support SQL functions?

  Yes it does, we can write any kind of SQL function.

  See the *SQL functions* section for more details.

- Does PostgreSQL have a default built-in procedural language ?

  Yes PostgreSQL has a default built-in procedural language called PL/pgSQL.

  See the *PL/pgSQL functions* section for more details.

- As a user without administrative privileges, can we read a table that requires administrative permissions in order to be read?

  Yes we can; as an administrator user let's create a function that reads the table, let's define the function using the security definer clause, and let's give the execution permissions of the function to the non-administrator user.

  See the *Security definer* section for more details.

References

- PostgreSQL – data types official documentation: `https://www.postgresql.org/docs/current/datatype.html`
- PostgreSQL – SQL functions official documentation: `https://www.postgresql.org/docs/current/xfunc-sql.html`
- PostgreSQL – PL/pgSQL official documentation: `https://www.postgresql.org/docs/current/plpgsql.html`
- PostgreSQL 11 Server Side Programming Quick Start Guide: `https://subscription.packtpub.com/book/data/9781789342222/1`

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

`https://discord.gg/jYWCjF6Tku`