```
               |     "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