Authors: Egor Rogov, Pavel Luzanov
Translated by Liudmila Mantrova

Contact Us
Please send your feedback to: edu@postgrespro.ru

Agenda

Procedures and their differences from functions

Input and output parameters

Overloading and polymorphism

Routines

Functions

- are called in the context of an expression
- cannot manage transactions
- return a result

Procedures

- are called using the `CALL` operator
- can manage transactions
- can return a result

Procedures were first introduced in PostgreSQL 11. The main reason for their appearance is that functions cannot manage transactions. Functions are called in the context of some expression that is computed as part of an already started operator (such as `SELECT`) in an already started transaction. It is impossible to complete a transaction and then start a new one while the operator is being executed.

Procedures are always called by the special `CALL` operator. If this operator starts a new transaction (instead of being called from an already started one), then it is possible to use transaction management commands in the called procedure.

Unfortunately, procedures written in SQL cannot use `COMMIT` and `ROLLBACK` commands. So we won't be able to see an example of a procedure that manages transactions until we get to the "PL/pgSQL. Executing Queries" lecture.

Sometimes you can hear that unlike functions, procedures do not return a result. But it is not true: procedures can also return a result, if required.

A generic term for functions and procedures is *routine*.

https://postgrespro.com/docs/postgresql/12/sql-createprocedure

https://postgrespro.com/docs/postgresql/12/sql-call

Procedures without Parameters

Let's start with an example of a simple procedure with no parameters.

```
=> CREATE TABLE t(a float);
```

```
CREATE TABLE
```

```
=> CREATE PROCEDURE fill()
AS $$
    TRUNCATE t;
    INSERT INTO t SELECT random() FROM generate_series(1,3);
$$ LANGUAGE sql;
```

```
CREATE PROCEDURE
```

To call a procedure, you have to use the CALL operator:

```
=> CALL fill();
```

```
CALL
```

Take a look at the result in the table:

```
=> SELECT * FROM t;
```

```
          a
--------------------
  0.9469425380377281
  0.5414317307802392
 0.23874303478227787
(3 rows)
```

You can get the same outcome using a function. Similarly, an SQL function can include several operators (not necessarily SELECT); the return value is determined by the last operator. You can declare the result value void if the function does not have to return anything, or specify the actual result value type:

```
=> CREATE FUNCTION fill_avg() RETURNS float
AS $$
    TRUNCATE t;
    INSERT INTO t SELECT random() FROM generate_series(1,3);
    SELECT avg(a) FROM t;
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

In any case, a function is always called in the context of some expression:

```
=> SELECT fill_avg();
```

```
      fill_avg
--------------------
 0.8195886659782742
(1 row)


=> SELECT * FROM t;

         a
--------------------
 0.8881256130637638
 0.6295244127517456
  0.941115972119313
(3 rows)
```

Functions cannot manage transactions. But SQL procedures do not support it either (although procedures written in other languages do provide such support).

Procedures with Parameters

Let's add an input parameter that defines the number of rows:

```
=> DROP PROCEDURE fill();
```

```
DROP PROCEDURE
```

```
=> CREATE PROCEDURE fill(nrows integer)
AS $$
    TRUNCATE t;
    INSERT INTO t SELECT random() FROM generate_series(1,nrows);
$$ LANGUAGE sql;

CREATE PROCEDURE
```

Just like functions, procedures allow passing arguments by position or by name:

```
=> CALL fill(nrows => 5);

CALL

=> SELECT * FROM t;

          a
----------------------
   0.7425274299857598
   0.8278786326837348
 0.057710084054502175
   0.4762661433963231
   0.6584314240895317
(5 rows)
```

Procedures can also have INOUT parameters that can be used to return a value. OUT parameters are not supported yet (but are likely to appear in PostgreSQL 14).

```
=> DROP PROCEDURE fill(integer);

DROP PROCEDURE

=> CREATE PROCEDURE fill(IN nrows integer, INOUT average float)
AS $$
    TRUNCATE t;
    INSERT INTO t SELECT random() FROM generate_series(1,nrows);
    SELECT avg(a) FROM t; -- like in a function
$$ LANGUAGE sql;

CREATE PROCEDURE
```

Let's try it out:

```
=> CALL fill(5, NULL /* the input parameter is not used */);

      average
--------------------
 0.4654580989594777
(1 row)
```

Overloading

Several routines with the same name

- routines differ in input parameter types;
  types of the return value and output parameters are ignored
- an appropriate routine is selected during execution based on the actual argument types

`CREATE OR REPLACE` command

- for new combinations of input parameter types, creates a new overloaded routine
- for existing combinations of input parameter types, changes the corresponding routine, but not the type of the return value

Overloading is the ability to use one and the same name for several routines (functions or procedures), which differ in types of `IN` and `INOUT` parameters. In other words, a *routine signature* consists of its name and types of its input parameters.

When calling a routine, PostgreSQL finds its version that corresponds to the passed arguments. There might be situations when an appropriate routine cannot be determined unambiguously; in this case, a run-time error occurs.

You have to take overloading into account when executing `CREATE OR REPLACE` (`FUNCTION` or `PROCEDURE`). If input parameter types differ from those used by already existing routines, a new overloaded routine will be created. Besides, when applied to functions, this command does not allow changing types of the output parameters and the return value type.

So you should delete and recreate the routine if required, but it won't be the same routine anymore. When deleting an old function, you also have to delete all views, triggers, and other objects that depend on it (`DROP FUNCTION ... CASCADE`).

https://postgrespro.com/docs/postgresql/12/xfunc-overload

Polymorphism

A routine that takes parameters of various types

- formal parameters use polymorphic pseudotypes (such as `anyelement`)
- the actual data type is selected during execution based on the type of the passed arguments

Instead of having several overloaded routines for different types, it is sometimes more convenient to create a single routine that takes parameters of any (or almost any) type.

For this purpose, a special *polymorphic pseudotype* is specified as the type of the formal parameter. For now, we'll only work with the `anyelement` type, which corresponds to any base type; but later we'll come across some other pseudotypes.

The exact type to be used by the routine is selected at run time based on the type of the passed argument.

If a routine has several polymorphic parameters, the types of the passed arguments must be the same. In other words, in each routine call, `anyelement` stands for some particular data type.

If a function is declared with a polymorphic return value, it must have at least one polymorphic input parameter. The exact type of the return value is also defined by the actual type of the passed input argument.

https://postgrespro.com/docs/postgresql/12/extend-type-system#EXTEND-TYPES-POLYMORPHIC

https://postgrespro.com/docs/postgresql/12/xfunc-sql#id-1.8.3.8.18

Overloaded Routines

Overloading mechanism is the same for both functions and procedures. They have a common namespace.

As an example, let's create a function that compares two integer numbers and returns the largest value. (There is a similar SQL expression called greatest, but we'll write our own function here.)

```
=> CREATE FUNCTION maximum(a integer, b integer) RETURNS integer
AS $$
    SELECT CASE WHEN a > b THEN a ELSE b END;
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

Let's check the result:

```
=> SELECT maximum(10, 20);
```

```
 maximum
---------
      20
(1 row)
```

Suppose we decided to create a similar function for three numbers. Thanks to overloading, we do not need to invent a new name:

```
=> CREATE FUNCTION maximum(a integer, b integer, c integer)
RETURNS integer
AS $$
SELECT CASE
        WHEN a > b THEN maximum(a,c)
        ELSE maximum(b,c)
    END;
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

Now we have two functions with the same name, but a different number of parameters:

```
=> \df maximum
```

```
                               List of functions
 Schema |  Name   | Result data type |       Argument data types       | Type
--------+---------+------------------+---------------------------------+------
 public | maximum | integer          | a integer, b integer            | func
 public | maximum | integer          | a integer, b integer, c integer | func
(2 rows)
```

And both of them work:

```
=> SELECT maximum(10, 20), maximum(10, 20, 30);
```

```
 maximum | maximum
---------+---------
      20 |      30
(1 row)
```

The CREATE OR REPLACE command enables you to create a routine or replace an existing one without deleting it. Since a function with such a signature already exists, it will be replaced:

```
=> CREATE OR REPLACE FUNCTION maximum(a integer, b integer, c integer)
RETURNS integer
AS $$
SELECT CASE
        WHEN a > b THEN
            CASE WHEN a > c THEN a ELSE c END
        ELSE
            CASE WHEN b > c THEN b ELSE c END
    END;
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

Let our function support not only integers, but also real numbers.

How can we implement it? We could define one more function, as follows:

```
=> CREATE FUNCTION maximum(a real, b real) RETURNS real
AS $$
    SELECT CASE WHEN a > b THEN a ELSE b END;
$$ LANGUAGE sql;

CREATE FUNCTION
```

Now we have three functions with the same name:

```
=> \df maximum

                              List of functions
 Schema |  Name   | Result data type |       Argument data types       | Type
--------+---------+------------------+---------------------------------+------
 public | maximum | integer          | a integer, b integer            | func
 public | maximum | integer          | a integer, b integer, c integer | func
 public | maximum | real             | a real, b real                  | func
(3 rows)
```

Two of them have the same number of parameters that differ in types:

```
=> SELECT maximum(10, 20), maximum(1.1, 2.2);

 maximum | maximum
---------+---------
      20 |     2.2
(1 row)
```

But then we would have to define separate functions with exactly the same body for all other data types, and repeat it for the other function with three parameters.

Polymorphic Functions

We can use the polymorphic anyelement type.

Let's delete all the three functions that we have created...

```
=> DROP FUNCTION maximum(integer, integer);

DROP FUNCTION

=> DROP FUNCTION maximum(integer, integer, integer);

DROP FUNCTION

=> DROP FUNCTION maximum(real, real);

DROP FUNCTION
```

...and then create a new one:

```
=> CREATE FUNCTION maximum(a anyelement, b anyelement)
RETURNS anyelement
AS $$
    SELECT CASE WHEN a > b THEN a ELSE b END;
$$ LANGUAGE sql;

CREATE FUNCTION
```

This function should accept any data type (but will work only with those types for which the "greater than" operator is defined).

Will it work?

```
=> SELECT maximum('A', 'B');

ERROR:  could not determine polymorphic type because input has type unknown
```

Unfortunately not. In this case, string literals can be of the char, varchar, or text type; the exact type is unknown. But we can use explicit type casting:

```
=> SELECT maximum('A'::text, 'B'::text);

 maximum
---------
 B
(1 row)
```

Here is another example with a different type:

```
=> SELECT maximum(now(), now() + interval '1 day');
```

```
            maximum
-------------------------------
 2021-10-20 17:01:45.550182+03
(1 row)
```

The type of the result value will always be the same as the parameter type.

It's important that both parameters have the same type; otherwise, an error occurs:

```
=> SELECT maximum(1, 'A');
```

```
ERROR:  invalid input syntax for type integer: "A"
LINE 1: SELECT maximum(1, 'A');
                          ^
```

In this example, such a requirement looks quite natural, but it may turn out to be inconvenient in some other cases.

Now let's create a function with three parameters, so that the third parameter is optional.

```
=> CREATE FUNCTION maximum(
    a anyelement,
    b anyelement,
    c anyelement DEFAULT NULL
) RETURNS anyelement
AS $$
SELECT CASE
        WHEN c IS NULL THEN
            x
        ELSE
            CASE WHEN x > c THEN x ELSE c END
    END
FROM (
    SELECT CASE WHEN a > b THEN a ELSE b END
) max2(x);
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

Let's try it out:

```
=> SELECT maximum(10, 20, 30);
```

```
 maximum
---------
      30
(1 row)
```

It works fine this way. And what about the following query?

```
=> SELECT maximum(10, 20);
```

```
ERROR:  function maximum(integer, integer) is not unique
LINE 1: SELECT maximum(10, 20);
               ^
HINT:  Could not choose a best candidate function. You might need to add explicit type casts.
```

A conflict occurs between two overloaded functions:

```
=> \df maximum
```

```
                                          List of functions
 Schema |  Name   | Result data type |                      Argument data types                       | Type
--------+---------+------------------+----------------------------------------------------------------+------
 public | maximum | anyelement       | a anyelement, b anyelement                                     | func
 public | maximum | anyelement       | a anyelement, b anyelement, c anyelement DEFAULT NULL::unknown | func
(2 rows)
```

It's impossible to understand whether we meant to run the function with two parameters, or simply omitted the third one.

This conflict can be easily resolved: let's delete the first function as it is no longer required.

```
=> DROP FUNCTION maximum(anyelement, anyelement);
```

```
DROP FUNCTION
```

```
=> SELECT maximum(10, 20), maximum(10, 20, 30);
```

```
 maximum | maximum
---------+---------
      20 |      30
(1 row)
```

Now everything works fine. Once we get to the “PL/pgSQL. Arrays” lecture, we will also learn how to define routines with an arbitrary number of parameters.

Summary

You can create and use your own procedures

Unlike functions, procedures are called using the `CALL` operator and can manage transactions

Both procedures and functions support overloading and polymorphism

Practice

1. In the `authors` table, authors' full names must be unique, but this condition is not checked.
   Create a procedure that deletes possible duplicates.
2. To eliminate the need for such a procedure, create a constraint that will not allow entering duplicates in the future.

Task 1. The feature for adding new authors won't appear in the application until we get to the "PL/pgSQL. Executing Queries" lecture. For now, you can insert duplicates manually to check your solution.

Task 1. Eliminating Duplicates

To check the solution, let's add another Pushkin:

```
=> INSERT INTO authors(last_name, first_name, middle_name)
    VALUES ('Pushkin', 'Alexander', 'Sergeyevich');
```

```
INSERT 0 1
```

```
=> SELECT last_name, first_name, middle_name, count(*)
FROM authors
GROUP BY last_name, first_name, middle_name;
```

```
  last_name  | first_name | middle_name  | count
-------------+------------+--------------+-------
 Jerome      | Jerome     | Klapka       |     1
 Pushkin     | Alexander  | Sergeyevich  |     2
 Gaiman      | Neil       |              |     1
 Swift       | Jonathan   |              |     1
 Bunin       | Ivan       | Alekseyevich |     1
 Shakespeare | William    |              |     1
 Pratchett   | Terry      |              |     1
(7 rows)
```

There are different ways to eliminate duplicates. Here is one example:

```
=> CREATE PROCEDURE authors_dedup()
AS $$
DELETE FROM authors
WHERE author_id IN (
    SELECT author_id
    FROM (
        SELECT author_id,
               row_number() OVER (
                   PARTITION BY first_name, last_name, middle_name
                   ORDER BY author_id
               ) AS rn
        FROM authors
    ) t
    WHERE t.rn > 1
);
$$ LANGUAGE sql;
```

```
CREATE PROCEDURE
```

```
=> CALL authors_dedup();
```

```
CALL
```

```
=> SELECT last_name, first_name, middle_name, count(*)
FROM authors
GROUP BY last_name, first_name, middle_name;
```

```
  last_name  | first_name | middle_name  | count
-------------+------------+--------------+-------
 Jerome      | Jerome     | Klapka       |     1
 Pushkin     | Alexander  | Sergeyevich  |     1
 Gaiman      | Neil       |              |     1
 Swift       | Jonathan   |              |     1
 Bunin       | Ivan       | Alekseyevich |     1
 Shakespeare | William    |              |     1
 Pratchett   | Terry      |              |     1
(7 rows)
```

Task 2. Using Constraints

We cannot define a suitable constraint because a middle name can be NULL. NULL values are considered to be different, so the constraint

```
UNIQUE(first_name, last_name, middle_name)
```

will still allow you to add another Jonathan Swift without a middle name.

The problem can be solved by creating a unique index:

```
=> CREATE UNIQUE INDEX authors_full_name_idx ON authors(
    last_name, first_name, coalesce(middle_name,'')
);

CREATE INDEX
```

Let's check the result:

```
=> INSERT INTO authors(last_name, first_name)
    VALUES ('Swift', 'Jonathan');

ERROR:  duplicate key value violates unique constraint "authors_full_name_idx"
DETAIL:  Key (last_name, first_name, COALESCE(middle_name, ''::text))=(Swift, Jonathan, ) already exists.

=> INSERT INTO authors(last_name, first_name, middle_name)
    VALUES ('Pushkin', 'Alexander', 'Sergeyevich'),
```

Practice

1. Is it possible to create the following objects with the same name that belong to the same schema:
   1) a procedure with one input parameter;
   2) a function with one input parameter of the same type that returns some value?
   Check your response.
2. A table stores real numbers (for example, the results of some measurements). Create a procedure that performs data normalization by multiplying all numbers by a certain factor, so that all values fit the interval between −1 and 1.
   The procedure must return the chosen normalization factor.

Task 2. As a normalization factor, use the maximum absolute value stored in the table.

Task 1. Overloading Functions and Procedures

It won't work because the routine signature includes only its name and types of input parameters (the return value is ignored), while procedures and functions have a common namespace.

```
=> CREATE PROCEDURE test(IN x integer)
AS $$
    SELECT 1;
$$ LANGUAGE sql;
```

```
CREATE PROCEDURE
```

```
=> CREATE FUNCTION test(IN x integer) RETURNS integer
AS $$
    SELECT 1;
$$ LANGUAGE sql;
```

```
ERROR:  function "test" already exists with same argument types
```

Some error messages (like this one in particular) use the word "function" instead of "procedure" because they have much in common.

Task 2. Data Normalization

Create a table with test data:

```
=> CREATE TABLE samples(a float);
```

```
CREATE TABLE
```

```
=> INSERT INTO samples(a)
    SELECT (0.5 - random())*100 FROM generate_series(1,10);
```

```
INSERT 0 10
```

You can create a procedure with one SQL operator:

```
=> CREATE PROCEDURE normalize_samples(INOUT coeff float)
AS $$
    WITH c(coeff) AS (
        SELECT 1/max(abs(a))
        FROM samples
    ),
    upd AS (
        UPDATE samples
        SET a = a * c.coeff
        FROM c
    )
    SELECT coeff FROM c;
$$ LANGUAGE sql;
```

```
CREATE PROCEDURE
```

```
=> CALL normalize_samples(NULL);
```

```
        coeff
---------------------
 0.020797912866554757
(1 row)
```

```
=> SELECT * FROM samples;
```

```
          a
----------------------
 -0.15036745718412478
  -0.6099210680053324
   0.7097958064669586
                    1
  -0.7465966784372502
  -0.0994382955589336
 0.038246337232130626
 -0.25446926346628457
  -0.6832525076445765
   0.9134939829918195
(10 rows)
```