

SQL Procedures



16

Copyright

© Postgres Professional, 2017–2025

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Igor Gnatyuk

Translated by: Liudmila Mantrova, Alexander Meleshko, Elena Sharafutdinova

Photo by: Oleg Bartunov (Phu monastery, Bhrikuti summit, Nepal)

Use of Course Materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Procedures and their Differences from Functions

Input and Output Parameters

Overloading and Polymorphism

Functions

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

Procedures

- called using a CALL statement
- can manage transactions
- can return a result

Procedures were first introduced in PostgreSQL 11. The main reason for their introduction was that functions cannot manage transactions. Functions are called in the context of some expression which 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 a special CALL statement. If this statement 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 (although those with unquoted body may be able to in the future). Therefore, we won't see an example of a procedure that manages transactions until we get to the PL/pgSQL. Query execution section.

Some say that the difference between functions and procedures is that a procedure does not return a result. But it is not true: procedures can also return a result, if required.

An umbrella term for both functions and procedures is *routines*. They share the common namespace.

<https://postgrespro.com/docs/postgresql/16/sql-createprocedure>

<https://postgrespro.com/docs/postgresql/16/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 a CALL statement:

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

CALL

Take a look at the result in the table:

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

```
      a
-----
 0.269508448980468
 0.2572845568619859
 0.7262705855628093
(3 rows)
```

Let's define the procedure again, now with unquoted body:

```
=> CREATE OR REPLACE PROCEDURE fill()
```

```
LANGUAGE sql
```

```
BEGIN ATOMIC
```

```
    DELETE FROM t; -- TRUNCATE is not yet supported in such procedures
```

```
    INSERT INTO t SELECT random() FROM generate_series(1,3);
```

```
END;
```

CREATE PROCEDURE

Check if it works:

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

CALL

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

```
      a
-----
 0.5542507248343103
 0.07157639040029196
 0.8603393100655552
(3 rows)
```

Try committing a transaction within the procedure:

```
=> CREATE OR REPLACE PROCEDURE fill()
```

```
LANGUAGE sql
```

```
BEGIN ATOMIC
```

```
    DELETE FROM t;
```

```
    INSERT INTO t SELECT random() FROM generate_series(1,3);
```

```
    COMMIT;
```

```
END;
```

ERROR: COMMIT is not yet supported in unquoted SQL function body

Note that we get the invalid command error already at the procedure definition stage.

Rename the table the procedure is working with:

```
=> ALTER TABLE t RENAME TO ta;
```

ALTER TABLE

The call below will not result in an error. In the procedure definition in the system catalog, the table is specified not by name but by ID, which was obtained at procedure creation.

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

CALL

Similar behavior can be achieved with a function which returns the output of its last operator. You can define the return type as void if the function does not return anything, or specify the actual result value type.

Let's give the table back its previous name and define the function:

```
=> ALTER TABLE ta RENAME TO t;
```

ALTER TABLE

```
=> CREATE FUNCTION fill_avg() RETURNS float
LANGUAGE sql
BEGIN ATOMIC
    DELETE FROM t;
    INSERT INTO t SELECT random() FROM generate_series(1, 3);
    SELECT avg(a) FROM t;
END;
```

CREATE FUNCTION

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

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

```
      fill_avg
-----
0.6875661870656832
(1 row)
```

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

```
      a
-----
0.8898635547654432
0.3907826147818403
0.7820523916497659
(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)
LANGUAGE sql
BEGIN ATOMIC
    DELETE FROM t;
    INSERT INTO t SELECT random() FROM generate_series(1, nrows);
END;
```

CREATE PROCEDURE

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

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

CALL

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

```

      a
-----
0.7622007860116502
0.9574269589997921
0.16014709786642167
0.9578109049668733
0.1632142083934891
(5 rows)

```

Procedures can also have OUT and INOUT parameters that can be used to return a value.

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

```
DROP PROCEDURE
```

```
=> CREATE PROCEDURE fill(IN nrows integer, OUT average float)
LANGUAGE sql
BEGIN ATOMIC
    DELETE FROM t;
    INSERT INTO t SELECT random() FROM generate_series(1, nrows);
    SELECT avg(a) FROM t; -- like in a function
END;
```

```
CREATE PROCEDURE
```

Let's try it out:

```
=> CALL fill(5, NULL /* the value is not used but has to be specified */);
```

```

      average
-----
0.5597842124557506
(1 row)

```

Several routines with the same name

routine is uniquely identified by a signature:
its name and input parameter types

types of the return value and output parameters are ignored

an appropriate routine is selected during execution based on the argument types

CREATE OR REPLACE command

for new combination of input parameter types,
creates a new overloaded routine

for existing combination 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 name and types of its input parameters form a *routine signature*. When calling a routine, PostgreSQL finds its version that corresponds to the passed arguments. If an appropriate routine cannot be determined unambiguously, a runtime error occurs.

A signature, however, does not include:

- routine type (procedure or function),
- OUT parameter types,
- returned value type.

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, otherwise a matching existing one will be replaced. Besides, when an existing routine is replaced with the CREATE OR REPLACE command, its type, OUT parameter types and return value type may not be changed, but other properties such as the language can be. In some cases, this means you must delete the routine and create it anew to replace it. However, doing so requires you to first delete all dependent objects, such as views, triggers, and other routines (DROP ROUTINE ... CASCADE).

<https://postgrespro.com/docs/postgresql/16/xfunc-overload>

A routine that takes arguments of various types

formal parameters use polymorphic pseudotypes
(such as *anyelement* or *anycompatible*)

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 arguments of any (or almost any) type.

For this purpose, a special *polymorphic pseudotype* is used as the formal parameter type. For now, we will consider just two of them — *anyelement* and *anycompatible* — with more to follow in later sections.

A routine defined with polymorphic pseudotypes as input parameters may take any data type as input. 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 is defined with multiple polymorphic parameters of the *anyelement* type, all passed arguments will be implicitly converted to the type of the first parameter. On the other hand, if a routine is defined with multiple polymorphic parameters of the *anycompatible* type, all passed arguments will be converted to some common 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. For SQL routines with unquoted body, there is no way to use polymorphic data types for arguments.

<https://postgrespro.com/docs/postgresql/16/extend-type-system#EXTEND-TYPES-POLYMORPHIC>

<https://postgrespro.com/docs/postgresql/16/xfunc-sql#XFUNC-SQL-POLYMORPHIC-FUNCTIONS>

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
LANGUAGE sql
RETURN CASE WHEN a > b THEN a ELSE b END;
```

CREATE FUNCTION

Let's check:

```
=> 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
LANGUAGE sql
RETURN CASE
    WHEN a > b THEN maximum(a, c)
    ELSE maximum(b, c)
END;
```

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
LANGUAGE sql
RETURN 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;
```

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
LANGUAGE sql
RETURN CASE WHEN a > b THEN a ELSE b END;
```

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)

If a routine is overloaded multiple times, you can output information on specific overloads in \df by specifying parameter types:

```
=> \df maximum real
```

List of functions				
Schema	Name	Result data type	Argument data types	Type
public	maximum	real	a real, b real	func

(1 row)

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

Polymorphic Routines

We can use the polymorphic types anyelement and anycompatible. These are pseudotypes, and when a function is called and interpreted, they are substituted with actual argument types. Naturally, if a routine is defined with unquoted body, its code is parsed at creation, preventing the use of pseudotypes.

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
-----
2025-11-28 14:21:06.269648+03
(1 row)

```

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

But we could go further and make polymorphic routines take not just the same types but compatible ones: those that can be implicitly converted into each other. This is where the polymorphic pseudotype `anycompatible` comes in.

Let's recreate our function:

```
=> DROP FUNCTION maximum;
```

DROP FUNCTION

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

CREATE FUNCTION

Try the literals again:

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

```

maximum
-----
B
(1 row)

```

It works!

But if the types are neither the same nor compatible, we get an error:

```
=> 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 anycompatible,
    b anycompatible,
    c anycompatible DEFAULT NULL
) RETURNS anycompatible
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

```
=> SELECT maximum(10, 11.21, 3e3);
```

```

maximum
-----
      3000
(1 row)

```

It works. And what about the following query?

```
=> SELECT maximum(10, 11.21);
```

ERROR: function maximum(integer, numeric) is not unique

```
LINE 1: SELECT maximum(10, 11.21);
                  ^
```

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
```

Schema	Name	Result data type Type	Argument data types
public	maximum	anycompatible func	a anycompatible, b anycompatible
public	maximum	anycompatible func	a anycompatible, b anycompatible, c anycompatible

DEFAULT NULL::unknown
(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(anycompatible, anycompatible);
```

```
DROP FUNCTION
```

```
=> SELECT maximum(10, 11.21), maximum(10, 11.21, 3e3);
```

```

maximum | maximum
-----+-----
      11.21 |      3000
(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.

Takeaways



You can create and use custom procedures

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

Both procedures and functions support overloading and polymorphism



1. In the authors table, authors' first, last and middle 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.

1. The feature for adding new authors won't appear in the application until we get to the PL/pgSQL. Query execution section. For now, you can insert duplicates manually to check if your solution works.

Task 1. Eliminating Duplicates

To test the solution, 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 several ways to eliminate duplicates. One example is:

```
=> CREATE PROCEDURE authors_dedup()
LANGUAGE sql
BEGIN ATOMIC
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
);
END;
```

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');
```

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

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?
 - 3) a procedure with one input and one output parameter?Try and see.
2. A table stores *real* numbers (for example, measurements of some sort). 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 selected normalization factor.

2. Take the maximum absolute value from the table as the normalization factor.

1. Procedure and function overloading

```
=> CREATE DATABASE sql_proc;
```

CREATE DATABASE

```
=> \c sql_proc
```

You are now connected to database "sql_proc" as user "student".

This will not work, because the signature of a routine includes only the name and types of the input parameters (the return value is ignored), and the procedures and functions share a common namespace.

```
=> CREATE PROCEDURE test(IN x integer)
LANGUAGE sql
RETURN 1;
```

CREATE PROCEDURE

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

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

In some messages, as in this one, “function” is used instead of the word “procedure”, because they are similar in many ways.

```
=> CREATE OR REPLACE PROCEDURE test(IN x integer, OUT y integer)
LANGUAGE sql
RETURN x;
```

ERROR: cannot change whether a procedure has output parameters
HINT: Use DROP PROCEDURE test(integer) first.

This also doesn't work, since there is already a procedure with the same signature, and it is also forbidden to change the output parameters (and their presence) for an existing routine. The routine must be deleted before recreating it.

2. Data Normalization

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

Just a single SQL statement is enough:

```
=> CREATE PROCEDURE normalize_samples(INOUT coeff float)
LANGUAGE sql
BEGIN ATOMIC
    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;
END;
```

CREATE PROCEDURE

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

```
      coeff
-----
0.021707473843965406
(1 row)
```

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

```
      a
-----
-0.5538052060243341
      1
-0.7623533652039323
-0.9010737067043512
-0.3807898477398218
 0.10349324186333168
   0.504165881570581
   0.8243917598543103
   0.0863366693457439
-0.15227989829493432
(10 rows)
```

```
=> \c postgres
```

You are now connected to database "postgres" as user "student".

```
=> DROP DATABASE sql_proc;
```

```
DROP DATABASE
```