SQL
Functions and Procedures

Copyright
© Postgres Professional, 2017–2025
Authors Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Igor Gnatyuk
Translated by Liudmila Mantrova and Alexander Meleshko
Photo by Oleg Bartunov (Tukuche peak, 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.com

Disclaimer
In no event shall Postgres Professional company be liable for any damages or loss, including loss of profits, that arise from direct or indirect, special or incidental 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.

Topics

Functions and their specifics in databases

Parameters and return values

Passing arguments in a function call

Volatility categories and query planning

Procedures and their differences from functions

Overloading and polymorphism

Functions in databases

The main goal is simplifying development tasks

- interface (parameters) and implementation (function body)
- abstracting from other tasks when implementing a particular function

	Traditional languages	*PostgreSQL*
side effects	global variables	whole database (volatility categories)
modules	own interface and implementation	namespaces, client and server
challenges	overhead related to calls (inlining)	hiding the query from the planner (inlining, subqueries, views)

The main goal of introducing functions in programming is simplifying development tasks by decomposing them into smaller subtasks. Such simplification is possible because you can abstract from the big picture when thinking of a function. For this purpose, the function provides a precise interface to the outside world (parameters and the return value).
Its implementation (the function body) can change; the caller does not see these changes and does not depend on them. This ideal situation can be messed up by the global state (global variables), and you have to keep in mind that in the DB context the whole database constitutes such a state.

In traditional programming languages, functions are often grouped into modules (packages, classes for OOP, etc.), which have their own interface and implementation. This separation into modules can be more or less arbitrary. In PostgreSQL, there is a fixed boundary between the client and the server: the server code deals with the database, while the client code manages transactions. There are no modules (or packages), only namespaces.

The only disadvantage of extensive use of functions in traditional languages is function call overhead. It is sometimes overcome by inlining function code into the calling program. In databases, the consequences can be more serious: if some part of the query is moved into a function, the planner stops seeing the big picture and cannot build a good query plan. In some cases, PostgreSQL can also perform inlining; alternatively, subqueries or views can be used.

Functions overview

A database object

- function declaration is stored in the system catalog

The structure of function declaration

- name
- parameters
- return data type
- body

Can be written in various languages, including SQL

- the code is stored as a string literal
- a function is interpreted when it is called

Is called in the context of an expression

Functions are regular database objects, just like tables or indexes. Function declarations are stored in the system catalog; that's why database functions are called *stored functions*.

PostgreSQL provides a lot of standard functions. Some of them are listed in the "Basic data types and functions" handout.

You can also write your own functions in various programming languages. The information provided in this lecture applies to functions in any programming language, but we will use SQL in all examples.

Predictably, a function declaration consists of a name, optional parameters, a return data type, and a body. What may seem unexpected is that the body is written as a string literal, which contains the code written in the programming language of your choice. It makes function declarations look the same regardless of the used programming language. The body string is stored in the system catalog and is interpreted each time the function is called. Since PostgreSQL 14, SQL code can be pre-parsed. In this case the parse result is stored in the system catalog instead of the code itself. Another way to avoid interpretation is to write a function in the C language, but we are not going to discuss this approach here.

A function is always called within the context of an expression: In the list of expressions of the SELECT statement, in the WHERE clause, in CHECK constraints, etc.

https://postgrespro.com/docs/postgresql/17/sql-createfunction

https://postgrespro.com/docs/postgresql/17/sql-syntax-calling-funcs

Functions without parameters

Here is a simple example of a function with no parameters:

```
=> CREATE FUNCTION hello_world() -- function name and an empty list of parameters
RETURNS text                     -- the type of the return value
AS $$ SELECT 'Hello, world!'; $$ -- function body
LANGUAGE sql;                    -- language specification
```

```
CREATE FUNCTION
```

It is convenient to write the body as a dollar-quoted string, as shown in the example above. Otherwise, you have to take care of escaping quotes, which are sure to appear in the function body. Compare the following strings:

```
=> SELECT ' SELECT ''Hello, world!''; ';
```

```
         ?column?
--------------------------
  SELECT 'Hello, world!';
(1 row)
```

```
=> SELECT $$ SELECT 'Hello, world!'; $$;
```

```
         ?column?
--------------------------
  SELECT 'Hello, world!';
(1 row)
```

If required, dollar quoting can be nested. It is achieved by using different text strings between dollars in each pair of quotes:

```
=> SELECT $func$ SELECT $$Hello, world!$$; $func$;
```

```
          ?column?
----------------------------
  SELECT $$Hello, world!$$;
(1 row)
```

A function is called in the context of an expression. For example:

```
=> SELECT hello_world(); -- empty brackets are mandatory
```

```
  hello_world
---------------
 Hello, world!
(1 row)
```

Let's have a look at how the body of a function is stored in the system catalog.

```
=> SELECT proname, prosrc, prosqlbody FROM pg_proc
WHERE proname = 'hello_world' \gx
```

```
-[ RECORD 1 ]------------------------
proname    | hello_world
prosrc     |  SELECT 'Hello, world!';
prosqlbody |
```

The function body is stored as-is in a text string.

Let’s go the modern way and recreate the function in accordance with the SQL standard. Here, the body of the function will be just RETURN <expression> (so-called unquoted SQL function body):

```
=> CREATE OR REPLACE FUNCTION hello_world() RETURNS text
LANGUAGE sql
RETURN 'Hello, world!';
```

```
CREATE FUNCTION
```

Check the system catalog again: the function body is stored differently now.

```
=> SELECT proname, prosrc, left(prosqlbody, 100) AS body
FROM pg_proc
WHERE proname = 'hello_world' \gx
```

```
-[ RECORD 1 ]-------------------------------------------------------------------
proname | hello_world
prosrc  |
body    | {QUERY :commandType 1 :querySource 0 :canSetTag true :utilityStmt <>
:resultRelation 0 :hasAggs fals
```

This time, the source code is not stored here. You can get it with the \sf command:

```
=> \sf hello_world
```

```
CREATE OR REPLACE FUNCTION public.hello_world()
 RETURNS text
 LANGUAGE sql
RETURN 'Hello, world!'::text
```

If a function body contains multiple SQL operators, it will return the first row of the last operator's output. If the function code is in the SQL standard format, you will need to use the BEGIN ATOMIC ... END construct to return the whole block of operators:

```
=> CREATE OR REPLACE FUNCTION hello_world() RETURNS text
LANGUAGE sql
BEGIN ATOMIC
  SELECT 'First Line';
  SELECT 'Second Line';
END;
```

```
CREATE FUNCTION
```

Let's call the function:

```
=> SELECT hello_world();
```

```
 hello_world
-------------
 Second Line
(1 row)
```

Note how the SQL standard-style syntax is different from the regular single-line style:

- no AS construct with the function code as a text,
- the new keyword RETURN can be used to return a value,
- "LANGUAGE sql" is optional,
- function code is parsed and the parse result is stored in pg_proc.prosqlbody, while the source code itself is not stored in pg_proc.prosrc, unlike with the traditional notation.

Not only does this confirm to the standard better, but also improves compatibility with other SQL implementations. Now, when a function is called, its commands don't need to go through interpretation again, and the parsed function body is used.

Not all SQL operators can be used in a function. The following ones are forbidden:

- transaction control commands (BEGIN, COMMIT, ROLLBACK, etc.);
- service commands (such as VACUUM or CREATE INDEX).

Here is an example of an invalid function. We have used the void pseudotype, which indicates that the function returns nothing.

```
=> CREATE FUNCTION do_commit() RETURNS void
LANGUAGE sql
BEGIN ATOMIC COMMIT; END;
```

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

You can use procedures to manage transactions; we will cover this topic later in this lecture.

Functions with input parameters

Here is a function with a single parameter:

```
=> CREATE FUNCTION hello(name text) -- a formal parameter
RETURNS text
LANGUAGE sql
RETURN 'Hello, ' || name || '!';
```

```
CREATE FUNCTION
```

When calling this function, we have to specify the actual value that corresponds to the formal parameter:

```
=> SELECT hello('Alice');
```

```
     hello
---------------
 Hello, Alice!
(1 row)
```

When specifying parameter types, you can add a modifier (such as varchar(10)), but it will be ignored.

You can define a function parameter without a name; then the function body will have to refer to it by its position number. Let's delete this function and create a new one:

```
=> DROP FUNCTION hello(text); -- it is enough to specify the parameter type

DROP FUNCTION

=> CREATE FUNCTION hello(text)
RETURNS text
LANGUAGE sql
RETURN 'Hello, ' || $1 || '!'; -- a number instead of the name

CREATE FUNCTION

=> SELECT hello('Alice');

     hello
---------------
 Hello, Alice!
(1 row)
```

But this approach is inconvenient and should be avoided.

Let's delete and recreate the function again, now adding two more parameters: a greeting and the title of a person.

```
=> DROP FUNCTION hello(text);

DROP FUNCTION
```

Here we have used an optional IN keyword, which means the input parameter. The DEFAULT clause is used to define the default parameter value:

```
=> CREATE FUNCTION hello(IN name text, IN greet text DEFAULT 'Dear', IN title text DEFAULT 'Mr')
RETURNS text
LANGUAGE sql
RETURN 'Hello, ' || greet || ' ' || title || ' ' || name || '!';

CREATE FUNCTION

=> SELECT hello('Alice', 'Charming', 'Mrs'); -- the second and the third parameter are specified

           hello
----------------------------
 Hello, Charming Mrs Alice!
(1 row)
```

Note that parameters with default values must be at the end of the list. When calling a function, if some default parameters are omitted and use their actual values, all following default-able parameters will also use their default values,

```
=> SELECT hello('Bob', 'Excellent'); -- only the first parameter gets the default value

          hello
--------------------------
 Hello, Excellent Mr Bob!
(1 row)


=> SELECT hello('Bob'); -- both parameters with default values are omitted

        hello
--------------------
 Hello, Dear Mr Bob!
(1 row)
```

So far, we have provided function parameters as positional ones, in the order they were specified in the function declaration. In many standard functions, parameter names are not set, so it is the only way possible.

But if the formal parameters are named, you can use these names when providing their actual values. In this case, parameters can be specified in any order:

```
=> SELECT hello(title => 'Dr.', name => 'Alice');
```

```
         hello
-----------------------
 Hello, Dear Dr. Alice!
(1 row)
```

This approach is convenient if the order of parameters is not quite obvious, especially if there are a lot of them.

You can combine both conventions: provide some parameters by position (starting from the first one) and specify the rest by name:

```
=> SELECT hello('Alice', title => 'Dr.');
```

```
         hello
-----------------------
 Hello, Dear Dr. Alice!
(1 row)
```

If the function must return NULL when at least one of its input parameters is NULL, it can be declared STRICT. In this case, the function body will not be executed at all.

```
=> DROP FUNCTION hello(text, text, text);
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(IN name text, IN title text DEFAULT 'Mr')
RETURNS text
LANGUAGE sql STRICT
RETURN 'Hello, ' || title || ' ' || name || '!';
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice', NULL);
```

```
 hello
-------

(1 row)
```

Input and output

Input values

are defined by parameters with IN or INOUT modes

Output value

is defined either by the RETURNS clause
or by parameters with IN or INOUT modes

if both forms are specified, they must be logically equivalent

Formal parameters that have IN or INOUT modes are *input parameters*. Their actual values must be specified in the function call (or the default values must be defined).

There are two ways to define the return value:

- use the RETURNS clause to specify the return data type,
- define *output parameters* using INOUT or OUT modes.

These two approaches are equivalent. For example, a function with the RETURNS integer clause and a function with the OUT integer parameter both return an integer number.

You can combine these two approaches. In this case, the function will also return *one* integer number. But note that the types of the output parameters and the RETURNS clause must not contradict each other.

Thus, unlike in many traditional programming languages, you cannot write a function that returns one value while passing another value into the OUT parameter.

Functions with output parameters

An alternative way to return a value is to use an output parameter.

```
=> DROP FUNCTION hello(text, text);
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(
    IN name text,
    OUT text -- you can omit the parameter name if it is not required
)
LANGUAGE sql
RETURN 'Hello, ' || name || '!';
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice');
```

```
     hello
---------------
 Hello, Alice!
(1 row)
```

The result is the same.

You can use the RETURNS clause and the OUT parameter together: the result will be the same anyway:

```
=> DROP FUNCTION hello(text); -- OUT parameters are omitted
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(IN name text, OUT text)
RETURNS text
LANGUAGE sql
RETURN 'Hello, ' || name || '!';
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice');
```

```
     hello
---------------
 Hello, Alice!
(1 row)
```

Or even use an INOUT parameter:

```
=> DROP FUNCTION hello(text);
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(INOUT name text)
LANGUAGE sql
RETURN 'Hello, ' || name || '!';
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice');
```

```
     hello
---------------
 Hello, Alice!
(1 row)
```

Note that the actual value passed to the SQL function in an INOUT parameter is not modified: we pass an input value, and the output value is returned as a result (SQL differs from many other programming languages in this respect). That's why we can pass a constant, although other languages would require a variable.

While the RETURNS clause can take only one value, there can be several output parameters. For example:

```
=> DROP FUNCTION hello(text);
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(
    IN name text,
    OUT greeting text,
    OUT clock timetz)
LANGUAGE sql
RETURN ('Hello, ' || name || '!', current_time);
```

```
CREATE FUNCTION
```

Here, the expression after RETURN has to be in parentheses.

```
=> SELECT hello('Alice');
```

```
                hello
-------------------------------------
 ("Hello, Alice!",21:39:43.010241+03)
(1 row)
```

Indeed, our function has returned not just one but several values at once.

We will provide more details about this feature and composite types in the “SQL. Composite Types” lecture.

Volatility categories

Volatile

- may return different values for the same input arguments
- is used by default

Stable

- the return value cannot change within a single SQL operator
- the function cannot change the database state

Immutable

- the return value cannot change, the function is deterministic
- the function cannot change the database state

Each function is mapped to a particular volatility category, which defines the properties of the return value for the same input arguments.

The *volatile* category means that the return value can change randomly. Such functions will be executed each time they are called. If the function is declared without a category specification, it is assumed to be volatile.

The *stable* category is used for functions that always return the same value within a single SQL operator. In particular, such functions cannot change the state of the database. PostgreSQL *could* execute such a function only once during the query and then use the computed value.

The *immutable* category is even more strict: the return value always remains the same. Such a function *could* be executed at the planning stage, before the query is actually executed.

It does not mean that it happens so all the time, but the planner has the right to perform such optimizations. In some (simple) cases, the planner makes its own assumptions about function volatility, regardless of the explicitly provided category.

https://postgrespro.com/docs/postgresql/17/xfunc-volatility

Volatility categories and isolation

In general, using functions within queries does not violate the isolation level of the transaction, but there are two points worth knowing.

First, volatile functions can cause data inconsistency within the query when used at the Read Committed level.

Let's create a function that returns the number of rows in a table:

```
=> CREATE TABLE t(n integer);
```

```
CREATE TABLE
```

```
=> CREATE FUNCTION cnt() RETURNS bigint
LANGUAGE sql VOLATILE
RETURN (SELECT count(*) FROM t);
```

```
CREATE FUNCTION
```

Now let's call it several times with a delay and insert a row into the table in a parallel session.

```
=> BEGIN ISOLATION LEVEL READ COMMITTED;
```

```
BEGIN
```

```
=> SELECT (SELECT count(*) FROM t), cnt(), pg_sleep(1)
FROM generate_series(1,4);
```

```
  => INSERT INTO t VALUES (1);
```

```
  INSERT 0 1
```

```
 count | cnt | pg_sleep
-------+-----+----------
     0 |   0 |
     0 |   0 |
     0 |   1 |
     0 |   1 |
(4 rows)
```

```
=> END;
```

```
COMMIT
```

It won't happen at stricter isolation levels, or if the function is stable or immutable.

```
=> ALTER FUNCTION cnt() STABLE;
```

```
ALTER FUNCTION
```

```
=> TRUNCATE t;
```

```
TRUNCATE TABLE
```

```
=> BEGIN ISOLATION LEVEL READ COMMITTED;
```

```
BEGIN
```

```
=> SELECT (SELECT count(*) FROM t), cnt(), pg_sleep(1)
FROM generate_series(1,4);
```

```
  => INSERT INTO t VALUES (1);
```

```
  INSERT 0 1
```

```
 count | cnt | pg_sleep
-------+-----+----------
     0 |   0 |
     0 |   0 |
     0 |   0 |
     0 |   0 |
(4 rows)
```

```
=> END;
```

```
COMMIT
```

Another point is the visibility of changes made by the same transaction.

Volatile functions can see all the changes, even those made by the current SQL operator that has not been completed yet.

```
=> ALTER FUNCTION cnt() VOLATILE;
```

```
ALTER FUNCTION
```

```
=> TRUNCATE t;
```

```
TRUNCATE TABLE
```

```
=> INSERT INTO t SELECT cnt() FROM generate_series(1,5);
```

```
INSERT 0 5
```

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

```
 n
---
 0
 1
 2
 3
 4
(5 rows)
```

It is true for any isolation level.

Stable and immutable functions see only the changes performed by an already completed operator.

```
=> ALTER FUNCTION cnt() STABLE;
```

```
ALTER FUNCTION
```

```
=> TRUNCATE t;
```

```
TRUNCATE TABLE
```

```
=> INSERT INTO t SELECT cnt() FROM generate_series(1,5);
```

```
INSERT 0 5
```

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

```
 n
---
 0
 0
 0
 0
 0
(5 rows)
```

Volatility categories and query planning

Thanks to the volatility labels that provide additional information about the function behavior, the optimizer can spare some function calls.

To try it out, let's create a function that returns a random number:

```
=> CREATE FUNCTION rnd() RETURNS float
LANGUAGE sql VOLATILE
RETURN random();
```

```
CREATE FUNCTION
```

Let's check the execution plan of the following query:

```
=> EXPLAIN (costs off)
SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;
```

```
                   QUERY PLAN
-------------------------------------------------
 Function Scan on generate_series
   Filter: (random() > '0.5'::double precision)
(2 rows)
```

The query plan shows that the generate_series function is honestly called several times; each result is compared with a random number and is filtered out, if required.

You can see it for yourself:

```
=> SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;

 generate_series
-----------------
               4
               5
               7
               9
              10
(5 rows)

=> \g

 generate_series
-----------------
               2
               3
               6
               9
(4 rows)

=> \g

 generate_series
-----------------
               2
               3
               5
               8
               9
              10
(6 rows)

=> \g

 generate_series
-----------------
               3
               6
               9
              10
(4 rows)

=> \g

 generate_series
-----------------
               1
               6
               9
(3 rows)
```

Here, we randomly get 0 to 10 rows.

A stable function will be called only once, because we have virtually specified that its value cannot change within a single operator:

```
=> ALTER FUNCTION rnd() STABLE;

ALTER FUNCTION

=> EXPLAIN (costs off)
SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;

                       QUERY PLAN
-------------------------------------------------------
 Result
   One-Time Filter: (rnd() > '0.5'::double precision)
   ->  Function Scan on generate_series
(3 rows)
```

The output will be either 0 or 10 rows.

```
=> SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;
```

```
 generate_series
-----------------
               1
               2
               3
               4
               5
               6
               7
               8
               9
              10
(10 rows)


=> \g
 generate_series
-----------------
(0 rows)


=> \g
 generate_series
-----------------
               1
               2
               3
               4
               5
               6
               7
               8
               9
              10
(10 rows)
```

Finally, immutable functions are computed at the planning stage, so we do not need any filters during execution:

```
=> ALTER FUNCTION rnd() IMMUTABLE;

ALTER FUNCTION

=> EXPLAIN (costs off)
SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;

            QUERY PLAN
---------------------------------
 Function Scan on generate_series
(1 row)


=> \g
            QUERY PLAN
---------------------------------
 Function Scan on generate_series
(1 row)


=> \g
        QUERY PLAN
--------------------------
 Result
   One-Time Filter: false
(2 rows)


=> \g
            QUERY PLAN
---------------------------------
 Function Scan on generate_series
(1 row)
```

The plan for immutable is random!

It is the developer's responsibility to provide the correct information.

Function inlining

In some (very simple) cases, a function can be inlined: the function body written in SQL can be inserted right into the main SQL operator while the query is being parsed. In this case, we can save some time on the function call.

Roughly speaking, the following conditions should be met:

- The function body contains only one SELECT operator.
- The function does not access any tables.
- There are no subqueries, grouping operations, etc.
- There must be only one return value.
- The called functions must not violate the specified volatility category.

We have already seen such an example: the rnd() function, which is declared volatile.

Let's take another look.

```
=> ALTER FUNCTION rnd() VOLATILE;
```

```
ALTER FUNCTION
```

```
=> EXPLAIN (costs off)
SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;
```

```
                   QUERY PLAN
-------------------------------------------------
 Function Scan on generate_series
   Filter: (random() > '0.5'::double precision)
(2 rows)
```

The Filter does not mention the rnd() function, only random() is present; it will be called directly, without using the rnd() wrapper.

Procedures overview

The same structure of declaration

- except for return data type
- can return a result via OUT parameters

Is called using the CALL statement

Can manage transactions

- except for SQL language

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 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 (although those written in accordance with the new SQL standard 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/17/sql-createprocedure

https://postgrespro.com/docs/postgresql/17/sql-call

Procedures without parameters

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

```
=> CREATE PROCEDURE fill()
AS $$
    TRUNCATE t;
    INSERT INTO t SELECT random(1,100) FROM generate_series(1,3);
$$ 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;
```

```
 n
----
 21
 76
 40
(3 rows)
```

Let's define the procedure again, now in the SQL standard style:

```
=> 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(1,100) FROM generate_series(1,3);
END;
```

```
CREATE PROCEDURE
```

Check if it works:

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

```
CALL
```

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

```
 n
----
 81
 42
 69
(3 rows)
```

Try to commit a transaction within the procedure:

```
=> CREATE OR REPLACE PROCEDURE fill()
LANGUAGE sql
BEGIN ATOMIC
    DELETE FROM t;
    INSERT INTO t SELECT random(1,100) 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 as early as 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 that returns the output of its last operator. You can define the return type as void if the function does not return anything.

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(1,100) FROM generate_series(1, 3);
    SELECT avg(n) FROM t;
END;
```

```
CREATE FUNCTION
```

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

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

```
 fill_avg
----------
       42
(1 row)
```

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

```
 n
----
  7
 53
 66
(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(1,100) 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;
```

```
 n
----
 36
 83
  2
  8
 39
(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(1,100) FROM generate_series(1, nrows);
    SELECT avg(a) FROM t; -- like in a function
END;

ERROR:  column "a" does not exist
LINE 6:     SELECT avg(a) FROM t; -- like in a function
                       ^
```

Let's try it out:

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

ERROR:  procedure fill(integer, unknown) does not exist
LINE 1: CALL fill(5, NULL /* the input parameter is not used but has...
             ^
HINT:  No procedure matches the given name and argument types. You might need to add
explicit type casts.
```

Overloading

Several routines with the same name

- routines differ in names 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 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 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/17/xfunc-overload

Polymorphism

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 use just two of them — *anyelement* and *anycompatible* — with more to follow in later sections.

A routine defined with polymorphic pesudotypes 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 standard-style routines, there is no way to use polymorphic data types for arguments.

https://postgrespro.com/docs/postgresql/17/extend-type-system#EXTEND-TYPES-POLYMORPHIC

https://postgrespro.com/docs/postgresql/17/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 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
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, which 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 in SQL standard style, 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
-------------------------------
 2025-04-17 21:39:54.825624+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
```

```
                                                 List of functions
 Schema |  Name   | Result data type |                              Argument data types
                    | Type
--------+---------+------------------+-----------------------------------------------------
---------------------+------
 public | maximum | anycompatible    | a anycompatible, b anycompatible
                    | func
 public | maximum | anycompatible    | a anycompatible, b anycompatible, c anycompatible
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(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 your own routines (functions and procedures)

Routines can be written in various languages, including SQL

Routines support overloading and polymorphism

Functions volatility categories affect optimization opportunities

An SQL function can sometimes be inlined

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

Practice

1. Create a function *author_name* to construct author names. The function takes three parameters (last_name, first_name, and middle_name) and returns the full name, with the middle name abbreviated to its initial.

   Use this function in the *authors_v* view.

2. Create a function *book_name* to construct book titles. The function takes two parameters (book ID and the title) and returns a concatenation of the book title and the list of authors, ordered by seq_num. The name of each author is produced by the *author_name* function.

   Use this function in the *catalog_v* view.

Check the changes in the application.

Reminder: all the required functions are listed in the “Basic data types and functions” handout.

1.
```
FUNCTION author_name(
 last_name text, first_name text, middle_name text
)
RETURNS text
```

For example:
```
author_name('Alexander','Sergeyevich', 'Pushkin')→
→ 'Alexander S. Pushkin'
```

3.
```
FUNCTION book_name(book_id integer, title text)
RETURNS text
```

For example:
```
book_name(3,'Good Omens') →
→ 'Good Omens. Terry Pratchett, Neil Gaiman'
```

Stored functions can be edited directly. For example, psql provides the `\ef` command that opens the function body in an editor and saves the changes in the database.

You should avoid using this capability (or at least do not overuse it). A properly set up development process requires that all the code is stored in files under version control. If a function has to be changed, the file is modified and executed (using psql or an IDE). Function modifications made directly in the database can be easily lost. (In fact, setting up development processes is much more complex, but we are not going to cover it in this course.)

Task 1. The author_name function

```
=> CREATE FUNCTION author_name(
    last_name text,
    first_name text,
    middle_name text
) RETURNS text
LANGUAGE sql IMMUTABLE
RETURN first_name ||
       CASE WHEN middle_name != '' -- NOT NULL is implied
           THEN ' ' || left(middle_name, 1) || '.'
           ELSE ''
       END || ' ' ||
       last_name;
```

```
CREATE FUNCTION
```

Volatility category: immutable. The function always returns the same value given the same input arguments.

```
=> CREATE OR REPLACE VIEW authors_v AS
SELECT a.author_id,
       author_name(a.last_name, a.first_name, a.middle_name) AS display_name
FROM   authors a
ORDER BY display_name;
```

```
CREATE VIEW
```

Task 2. The book_name function

```
=> CREATE FUNCTION book_name(book_id integer, title text)
RETURNS text
LANGUAGE sql STABLE
RETURN (
SELECT title || '. ' ||
       string_agg(
           author_name(a.last_name, a.first_name, a.middle_name), ', '
           ORDER BY ash.seq_num
       )
FROM   authors a
       JOIN authorship ash ON a.author_id = ash.author_id
WHERE  ash.book_id = book_name.book_id
);
```

```
CREATE FUNCTION
```

Volatility category: stable. The function returns the same value given the same input arguments, but only within a single SQL query.

```
=> CREATE OR REPLACE VIEW catalog_v AS
SELECT b.book_id,
       book_name(b.book_id, b.title) AS display_name
FROM   books b
ORDER BY display_name;
```

```
CREATE VIEW
```