

# SQL Functions



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](mailto: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.

Functions and their Specifics in Databases

Parameters and Return Values

Passing Arguments in a Function Call

Volatility Categories and Query Planning

# Functions in Databases

The main goal is simplifying development tasks

interface (parameters) and implementation (function body)

abstracting from the main task when implementing a particular function

	<i>Traditional languages</i>	<i>PostgreSQL</i>
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.

## 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

4

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/16/sql-createfunction>

<https://postgrespro.com/docs/postgresql/16/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 statements, it will return the first row of the last operator's output. For SQL functions with unquoted body, you will need to use the BEGIN ATOMIC ... END construct to return the whole block of statements:

```

=> 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 syntax of SQL functions with unquoted body is different from the traditional text string style:

- no AS construct with the function code as a text,
- the new keyword RETURN can be used to return a value,
- LANGUAGE sql clause 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 conform 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 statements can be used in a function. The following ones are disallowed:

- 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 in the next 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, an argument for an optional parameter can be omitted. In this case, all following optional parameters will also use their default values.

```
=> SELECT hello('Bob', 'Excellent'); -- only the last 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 arguments 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 values

defined by parameters with IN or INOUT modes

## Output value

defined either by the RETURNS clause  
or by parameters with OUT or INOUT modes

if both forms are specified, they must correspond to each other

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!",14:20:36.03188+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/16/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 which 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 random() function is honestly called several times; each result returned by the generate\_series call is compared with a random number and filtered out, if required.

You can see it for yourself:

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

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

```
=> \g
```

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

```
=> \g
```

```
generate_series
-----
2
4
8
9
(4 rows)
```

```
=> \g
```

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

```
=> \g
```

```
generate_series
-----
1
5
8
9
10
(5 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
-----
(0 rows)
```

```
=> \g
```

```
generate_series
-----
(0 rows)
```

```
=> \g
```

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

```
-----
Result
One-Time Filter: false
(2 rows)
```

```
=> \g
```

```
QUERY PLAN
```

```
-----
Result
One-Time Filter: false
(2 rows)
```



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.

# Takeaways



You can create your own functions

Functions can be written in various languages, including SQL

Volatility categories affect optimization opportunities

An SQL function can sometimes be inlined



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 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 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 source code 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.)

## 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: 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

## 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: 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

1. Write a function that returns random timestamps, equally distributed within the specified time range. The lower bound of the range is set by a *timestamptz* value, while the upper bound is either a *timestamptz* or an interval.
2. Sometimes, phonewords are used as a mnemonic equivalent of telephone numbers, like 1-800-TAXICAB. A phoneword is constructed by replacing some digits with the corresponding letters, as seen on the keypad (even on modern phones). Write a function that returns number representation.
3. Write a function that solves a quadratic equation.

In all tasks, make sure to pay attention to volatility categories of the functions.

2. The correspondence between digits and letters is as follows:

2 – abc, 3 – def, 4 – ghi, 5 – jkl, 6 – mno, 7 – pqrs, 8 – tuv, 9 – wxyz

Note that a phoneword can be written in lowercase as well as in uppercase.

3. For the equation  $y = ax^2 + bx + c$  the discriminant is  $D = b^2 - 4ac$ :

- for  $D > 0$ , the two roots are  $x_{1,2} = (-b \pm \sqrt{D}) / 2a$
- for  $D = 0$ , the only root is  $x = -b / 2a$  (null can be returned as  $x_2$ )
- when  $D < 0$  there are no roots (both roots are null)

## 1. Random Timestamp

```
=> CREATE DATABASE sql_func;
```

```
CREATE DATABASE
```

```
=> \c sql_func
```

You are now connected to database "sql\_func" as user "student".

A function with two timestamps:

```
=> CREATE FUNCTION rnd_timestamp(t_start timestamptz, t_end timestamptz)
RETURNS timestamptz
LANGUAGE sql VOLATILE
RETURN t_start + (t_end - t_start) * random();
```

```
CREATE FUNCTION
```

Volatility: volatile. The random function will return different values with the same input.

```
=> SELECT current_timestamp,
        rnd_timestamp(
            current_timestamp,
            current_timestamp + interval '1 hour'
        )
FROM generate_series(1,10);
```

current_timestamp		rnd_timestamp
2025-11-27 14:28:20.104506+03		2025-11-27 15:14:05.637861+03
2025-11-27 14:28:20.104506+03		2025-11-27 14:51:57.641358+03
2025-11-27 14:28:20.104506+03		2025-11-27 14:58:28.707004+03
2025-11-27 14:28:20.104506+03		2025-11-27 14:33:58.232028+03
2025-11-27 14:28:20.104506+03		2025-11-27 14:47:22.947406+03
2025-11-27 14:28:20.104506+03		2025-11-27 14:43:24.923114+03
2025-11-27 14:28:20.104506+03		2025-11-27 14:33:34.110874+03
2025-11-27 14:28:20.104506+03		2025-11-27 14:50:03.860486+03
2025-11-27 14:28:20.104506+03		2025-11-27 15:25:29.804819+03
2025-11-27 14:28:20.104506+03		2025-11-27 14:58:10.923546+03

(10 rows)

The second function (with the interval parameter) can be defined using the first one:

```
=> CREATE FUNCTION rnd_timestamp(t_start timestamptz, t_delta interval)
RETURNS timestamptz
LANGUAGE sql VOLATILE
RETURN rnd_timestamp(t_start, t_start + t_delta);
```

```
CREATE FUNCTION
```

```
=> SELECT rnd_timestamp(current_timestamp, interval '1 hour');
```

rnd_timestamp
2025-11-27 15:06:18.937972+03

(1 row)

## 2. Phonewords

A function to convert a phoneword to number:

```
=> CREATE FUNCTION phoneword_to_number(phoneword text) RETURNS numeric
LANGUAGE sql IMMUTABLE
RETURN translate(
    lower(phoneword),
    'abcdefghijklmnopqrstuvwxyz-',
    '2223334445556667778889999'
)::numeric;
```

```
CREATE FUNCTION
```

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

```
=> SELECT phoneword_to_number('1-800-TAXICAB');
```

```
phoneword_to_number
-----
18008294222
(1 row)
```

### 3. The Roots of a Quadratic Equation

```
=> CREATE FUNCTION square_roots(
    a float,
    b float,
    c float,
    x1 OUT float,
    x2 OUT float
)
LANGUAGE sql IMMUTABLE
RETURN (
WITH discriminant(d) AS (
    SELECT b*b - 4*a*c
)
SELECT (CASE WHEN d >= 0.0 THEN (-b + sqrt(d))/2/a END,
        CASE WHEN d > 0.0 THEN (-b - sqrt(d))/2/a END)
FROM discriminant
);
```

CREATE FUNCTION

Volatility category: immutable. The function always returns the same value for the same input parameters.

```
=> SELECT square_roots(1, 0, -4);

 square_roots
-----
(2,-2)
(1 row)
```

```
=> SELECT square_roots(1, -4, 4);

 square_roots
-----
(2,)
(1 row)
```

```
=> SELECT square_roots(1, 1, 1);

 square_roots
-----
(,)
(1 row)
```

```
=> \c postgres
```

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

```
=> DROP DATABASE sql_func;
```

DROP DATABASE