create_function

- Description
- Parameters
- Overloading
- Notes
- Examples
- Writing SECURITY DEFINER Functions Safely
- Compatibility
- See Also

CREATE FUNCTION

CREATE FUNCTION

7

SQL - Language Statements

CREATE FUNCTION

define a new function

CREATE [OR REPLACE] FUNCTION

name

([[

argmode

] [

argname

]

argtype

[{ DEFAULT | = }

default_expr

] [, ...]]) [RETURNS

rettype

| RETURNS TABLE (

column_name

column_type

[, ...])] { LANGUAGE

lang_name

| TRANSFORM { FOR TYPE

type_name

} [, ...] | WINDOW | { IMMUTABLE | STABLE | VOLATILE } | [NOT] LEAKPROOF | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT } | { [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER } | PARALLEL { UNSAFE | RESTRICTED | SAFE } | COST

execution_cost

| ROWS

result_rows

| SUPPORT

support_function

| SET

configuration_parameter

{ TO

value

| =

value

| FROM CURRENT } | AS '

definition

' | AS '

obj_file

', '

link_symbol

' |

sql_body

} ...

Description

`CREATE FUNCTION` defines a new function. `CREATE OR REPLACE FUNCTION` will either create a new function, or replace an existing definition. To be able to define a function, the user must have the `USAGE` privilege on the language.

If a schema name is included, then the function is created in the specified schema. Otherwise it is created in the current schema. The name of the new function must not match any existing function or procedure with the same input argument types in the same schema. However, functions and procedures of different argument types can share a name (this is called overloading).

To replace the current definition of an existing function, use `CREATE OR REPLACE FUNCTION`. It is not possible to change the name or argument types of a function this way (if you tried, you would actually be creating a new, distinct function). Also, `CREATE OR REPLACE FUNCTION` will not let you change the return type of an existing function. To do that, you must drop and recreate the function. (When using `OUT` parameters, that means you cannot change the types of any `OUT` parameters except by dropping the function.)

When `CREATE OR REPLACE FUNCTION` is used to replace an existing function, the ownership and permissions of the function do not change. All other function properties are assigned the values specified or implied in the command. You must own the function to replace it (this includes being a member of the owning role).

If you drop and then recreate a function, the new function is not the same entity as the old; you will have to drop existing rules, views, triggers, etc. that refer to the old function. Use `CREATE OR REPLACE FUNCTION` to change a function definition without breaking objects that refer to the function.

Also, `ALTER FUNCTION` can be used to change most of the auxiliary properties of an existing function.

The user that creates the function becomes the owner of the function.

To be able to create a function, you must have `USAGE` privilege on the argument types and the return type.

Refer to ??? for further information on writing functions.

Parameters

<name>
The name (optionally schema-qualified) of the function to create.

<argmode>
The mode of an argument: `IN`, `OUT`, `INOUT`, or `VARIADIC`. If omitted, the default is `IN`. Only `OUT` arguments can follow a `VARIADIC` one. Also, `OUT` and `INOUT` arguments cannot be used together with the `RETURNS TABLE` notation.

<argname>
The name of an argument. Some languages (including SQL and PL/pgSQL) let you use the name in the function body. For other languages the name of an input argument is just extra documentation, so far as the function itself is concerned; but you can use input argument names when calling a function to improve readability (see ???). In any case, the name of an output argument is significant, because it defines the column name in the result row type. (If you omit the name for an output argument, the system will choose a default column name.)

<argtype>
The data type(s) of the function's arguments (optionally schema-qualified), if any. The argument types can be base, composite, or domain types, or can reference the type of a table column.

Depending on the implementation language it might also be allowed to specify “pseudo-types” such as `cstring`. Pseudo-types indicate that the actual argument type is either incompletely specified, or outside the set of ordinary SQL data types.

The type of a column is referenced by writing `table_name.column_name%TYPE`. Using this feature can sometimes help make a function independent of changes to the definition of a table.

<default_expr>
An expression to be used as default value if the parameter is not specified. The expression has to be coercible to the argument type of the parameter. Only input (including `INOUT`) parameters can have a default value. All input parameters following a parameter with a default value must have default values as well.

<rettype>
The return data type (optionally schema-qualified). The return type can be a base, composite, or domain type, or can reference the type of a table column. Depending on the implementation language it might also be allowed to specify “pseudo-types” such as `cstring`. If the function is not supposed to return a value, specify `void` as the return type.

When there are `OUT` or `INOUT` parameters, the `RETURNS` clause can be omitted. If present, it must agree with the result type implied by the output parameters: `RECORD` if there are multiple output parameters, or the same type as the single output parameter.

The `SETOF` modifier indicates that the function will return a set of items, rather than a single item.

The type of a column is referenced by writing `table_name.column_name%TYPE`.

<column_name>
The name of an output column in the `RETURNS TABLE` syntax. This is effectively another way of declaring a named `OUT` parameter, except that `RETURNS TABLE` also implies `RETURNS SETOF`.

<column_type>
The data type of an output column in the `RETURNS TABLE` syntax.

<lang_name>
The name of the language that the function is implemented in. It can be `sql`, `c`, `internal`, or the name of a user-defined procedural language, e.g., `plpgsql`. The default is `sql` if <sql_body> is specified. Enclosing the name in single quotes is deprecated and requires matching case.

`TRANSFORM { FOR TYPE type_name } [, ... ]`
Lists which transforms a call to the function should apply. Transforms convert between SQL types and language-specific data types; see ???. Procedural language implementations usually have hardcoded knowledge of the built-in types, so those don't need to be listed here. If a procedural language implementation does not know how to handle a type and no transform is supplied, it will fall back to a default behavior for converting data types, but this depends on the implementation.

`WINDOW`
`WINDOW` indicates that the function is a window function rather than a plain function. This is currently only useful for functions written in C. The `WINDOW` attribute cannot be changed when replacing an existing function definition.

`IMMUTABLE`; `STABLE`; `VOLATILE`
These attributes inform the query optimizer about the behavior of the function. At most one choice can be specified. If none of these appear, `VOLATILE` is the default assumption.

`IMMUTABLE` indicates that the function cannot modify the database and always returns the same result when given the same argument values; that is, it does not do database lookups or otherwise use information not directly present in its argument list. If this option is given, any call of the function with all-constant arguments can be immediately replaced with the function value.

`STABLE` indicates that the function cannot modify the database, and that within a single table scan it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for functions whose results depend on database lookups, parameter variables (such as the current time zone), etc. (It is inappropriate for `AFTER` triggers that wish to query rows modified by the current command.) Also note that the `current_timestamp` family of functions qualify as stable, since their values do not change within a transaction.

`VOLATILE` indicates that the function value can change even within a single table scan, so no optimizations can be made. Relatively few database functions are volatile in this sense; some examples are `random()`, `currval()`, `timeofday()`. But note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away; an example is `setval()`.

For additional details see ???.

`LEAKPROOF`
`LEAKPROOF` indicates that the function has no side effects. It reveals no information about its arguments other than by its return value. For example, a function which throws an error message for some argument values but not others, or which includes the argument values in any error message, is not leakproof. This affects how the system executes queries against views created with the `security_barrier` option or tables with row level security enabled. The system will enforce conditions from security policies and security barrier views before any user-supplied conditions from the query itself that contain non-leakproof functions, in order to prevent the inadvertent exposure of data. Functions and operators marked as leakproof are assumed to be trustworthy, and may be executed before conditions from security policies and security barrier views. In addition, functions which do not take arguments or which are not passed any arguments from the security barrier view or table do not have to be marked as leakproof to be executed before security conditions. See ??? and ???. This option can only be set by the superuser.

`CALLED ON NULL INPUT`; `RETURNS NULL ON NULL INPUT`; `STRICT`
`CALLED ON NULL INPUT` (the default) indicates that the function will be called normally when some of its arguments are null. It is then the function author's responsibility to check for null values if necessary and respond appropriately.

`RETURNS NULL ON NULL INPUT` or `STRICT` indicates that the function always returns null whenever any of its arguments are null. If this parameter is specified, the function is not executed when there are null arguments; instead a null result is assumed automatically.

`EXTERNAL SECURITY INVOKER`; `EXTERNAL SECURITY DEFINER`
`SECURITY INVOKER` indicates that the function is to be executed with the privileges of the user that calls it. That is the default. `SECURITY DEFINER` specifies that the function is to be executed with the privileges of the user that owns it. For information on how to write `SECURITY DEFINER` functions safely, see below.

The key word `EXTERNAL` is allowed for SQL conformance, but it is optional since, unlike in SQL, this feature applies to all functions not only external ones.

`PARALLEL`
`PARALLEL UNSAFE` indicates that the function can't be executed in parallel mode; the presence of such a function in an SQL statement forces a serial execution plan. This is the default. `PARALLEL RESTRICTED` indicates that the function can be executed in parallel mode, but only in the parallel group leader process. `PARALLEL SAFE` indicates that the function is safe to run in parallel mode without restriction, including in parallel worker processes.

Functions should be labeled parallel unsafe if they modify any database state, change the transaction state (other than by using a subtransaction for error recovery), access sequences (e.g., by calling `currval`) or make persistent changes to settings. They should be labeled parallel restricted if they access temporary tables, client connection state, cursors, prepared statements, or miscellaneous backend-local state which the system cannot synchronize in parallel mode (e.g., `setseed` cannot be executed other than by the group leader because a change made by another process would not be reflected in the leader). In general, if a function is labeled as being safe when it is restricted or unsafe, or if it is labeled as being restricted when it is in fact unsafe, it may throw errors or produce wrong answers when used in a parallel query. C-language functions could in theory exhibit totally undefined behavior if mislabeled, since there is no way for the system to protect itself against arbitrary C code, but in most likely cases the result will be no worse than for any other function. If in doubt, functions should be labeled as `UNSAFE`, which is the default.

`COST` <execution_cost>
A positive number giving the estimated execution cost for the function, in units of ???. If the function returns a set, this is the cost per returned row. If the cost is not specified, 1 unit is assumed for C-language and internal functions, and 100 units for functions in all other languages. Larger values cause the planner to try to avoid evaluating the function more often than necessary.

`ROWS` <result_rows>
A positive number giving the estimated number of rows that the planner should expect the function to return. This is only allowed when the function is declared to return a set. The default assumption is 1000 rows.

`SUPPORT` <support_function>
The name (optionally schema-qualified) of a planner support function to use for this function. See ??? for details. You must be superuser to use this option.

<configuration_parameter>; <value>
The `SET` clause causes the specified configuration parameter to be set to the specified value when the function is entered, and then restored to its prior value when the function exits. `SET FROM CURRENT` saves the value of the parameter that is current when `CREATE FUNCTION` is executed as the value to be applied when the function is entered.

If a `SET` clause is attached to a function, then the effects of a `SET LOCAL` command executed inside the function for the same variable are restricted to the function: the configuration parameter's prior

value is still restored at function exit. However, an ordinary `SET` command (without `LOCAL`) overrides the `SET` clause, much as it would do for a previous `SET LOCAL` command: the effects of such a command will persist after function exit, unless the current transaction is rolled back.

See ??? and ??? for more information about allowed parameter names and values.

<definition>
A string constant defining the function; the meaning depends on the language. It can be an internal function name, the path to an object file, an SQL command, or text in a procedural language.

It is often helpful to use dollar quoting (see ???) to write the function definition string, rather than the normal single quote syntax. Without dollar quoting, any single quotes or backslashes in the function definition must be escaped by doubling them.

`obj_file, link_symbol`
This form of the `AS` clause is used for dynamically loadable C language functions when the function name in the C language source code is not the same as the name of the SQL function. The string <obj_file> is the name of the shared library file containing the compiled C function, and is interpreted as for the `LOAD` command. The string <link_symbol> is the function's link symbol, that is, the name of the function in the C language source code. If the link symbol is omitted, it is assumed to be the same as the name of the SQL function being defined. The C names of all functions must be different, so you must give overloaded C functions different C names (for example, use the argument types as part of the C names).

When repeated `CREATE FUNCTION` calls refer to the same object file, the file is only loaded once per session. To unload and reload the file (perhaps during development), start a new session.

<sql_body>
The body of a `LANGUAGE SQL` function. This can either be a single statement

```
RETURN expression
```

or a block

```
BEGIN ATOMIC
  statement;
  statement;
  ...
  statement;
END
```

This is similar to writing the text of the function body as a string constant (see <definition> above), but there are some differences: This form only works for `LANGUAGE SQL`, the string constant form works for all languages. This form is parsed at function definition time, the string constant form is

parsed at execution time; therefore this form cannot support polymorphic argument types and other constructs that are not resolvable at function definition time. This form tracks dependencies between the function and objects used in the function body, so `DROP ... CASCADE` will work correctly, whereas the form using string literals may leave dangling functions. Finally, this form is more compatible with the SQL standard and other SQL implementations.

Overloading

PostgreSQL allows function overloading; that is, the same name can be used for several different functions so long as they have distinct input argument types. Whether or not you use it, this capability entails security precautions when calling functions in databases where some users mistrust other users; see ???.

Two functions are considered the same if they have the same names and *input* argument types, ignoring any `OUT` parameters. Thus for example these declarations conflict:

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, OUT text) ...
```

Functions that have different argument type lists will not be considered to conflict at creation time, but if defaults are provided they might conflict in use. For example, consider

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, int default 42) ...
```

A call `foo(10)` will fail due to the ambiguity about which function should be called.

Notes

The full SQL type syntax is allowed for declaring a function's arguments and return value. However, parenthesized type modifiers (e.g., the precision field for type `numeric`) are discarded by `CREATE FUNCTION`. Thus for example `CREATE FUNCTION foo (varchar(10)) ...` is exactly the same as `CREATE FUNCTION foo (varchar) ...`.

When replacing an existing function with `CREATE OR REPLACE FUNCTION`, there are restrictions on changing parameter names. You cannot change the name already assigned to any input parameter (although you can add names to parameters that had none before). If there is more than one output parameter, you cannot change the names of the output parameters, because that would change the column names of the anonymous composite type that describes the function's result.

These restrictions are made to ensure that existing calls of the function do not stop working when it is replaced.

If a function is declared `STRICT` with a `VARIADIC` argument, the strictness check tests that the variadic array *as a whole* is non-null. The function will still be called if the array has null elements.

Examples

Add two integers using an SQL function:

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'SELECT $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

The same function written in a more SQL-conforming style, using argument names and an unquoted body:

```
CREATE FUNCTION add(a integer, b integer) RETURNS integer
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT
    RETURN a + b;
```

Increment an integer, making use of an argument name, in PL/pgSQL:

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

Return a record containing multiple output parameters:

```
CREATE FUNCTION dup(IN int, OUT f1 int, OUT f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

You can do the same thing more verbosely with an explicitly named composite type:

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

Another way to return multiple columns is to use a `TABLE` function:

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

However, a `TABLE` function is different from the preceding examples, because it actually returns a *set* of records, not just one record.

Writing `SECURITY DEFINER` Functions Safely

search_path

configuration parameter

use in securing functions

createrole_self_grant

configuration parameter

use in securing functions

Because a `SECURITY DEFINER` function is executed with the privileges of the user that owns it, care is needed to ensure that the function cannot be misused. For security, ??? should be set to exclude any schemas writable by untrusted users. This prevents malicious users from creating objects (e.g.,

tables, functions, and operators) that mask objects intended to be used by the function. Particularly important in this regard is the temporary-table schema, which is searched first by default, and is normally writable by anyone. A secure arrangement can be obtained by forcing the temporary schema to be searched last. To do this, write `pg_temp` as the last entry in `search_path`. This function illustrates safe usage:

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN;
BEGIN
        SELECT  (pwd = $2) INTO passed
        FROM    pwds
        WHERE   username = $1;

        RETURN passed;
END;
$$  LANGUAGE plpgsql
    SECURITY DEFINER
    -- Set a secure search_path: trusted schema(s), then 'pg_temp'.
    SET search_path = admin, pg_temp;
```

This function's intention is to access a table admin.pwds. But without the `SET` clause, or with a `SET` clause mentioning only `admin`, the function could be subverted by creating a temporary table named pwds.

If the security definer function intends to create roles, and if it is running as a non-superuser, `createrole_self_grant` should also be set to a known value using the `SET` clause.

Another point to keep in mind is that by default, execute privilege is granted to `PUBLIC` for newly created functions (see ??? for more information). Frequently you will wish to restrict use of a security definer function to only some users. To do that, you must revoke the default `PUBLIC` privileges and then grant execute privilege selectively. To avoid having a window where the new function is accessible to all, create it and set the privileges within a single transaction. For example:

```
BEGIN;
CREATE FUNCTION check_password(uname TEXT, pass TEXT) ... SECURITY DEFINER;
REVOKE ALL ON FUNCTION check_password(uname TEXT, pass TEXT) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION check_password(uname TEXT, pass TEXT) TO admins;
COMMIT;
```

Compatibility

A `CREATE FUNCTION` command is defined in the SQL standard. The PostgreSQL implementation can be used in a compatible way but has many extensions. Conversely, the SQL standard specifies a number of optional features that are not implemented in PostgreSQL.

The following are important compatibility issues:

- `OR REPLACE` is a PostgreSQL extension.
- For compatibility with some other database systems, <argmode> can be written either before or after <argname>. But only the first way is standard-compliant.
- For parameter defaults, the SQL standard specifies only the syntax with the `DEFAULT` key word. The syntax with `=` is used in T-SQL and Firebird.
- The `SETOF` modifier is a PostgreSQL extension.
- Only `SQL` is standardized as a language.
- All other attributes except `CALLED ON NULL INPUT` and `RETURNS NULL ON NULL INPUT` are not standardized.
- For the body of `LANGUAGE SQL` functions, the SQL standard only specifies the <sql_body> form.

Simple `LANGUAGE SQL` functions can be written in a way that is both standard-conforming and portable to other implementations. More complex functions using advanced features, optimization attributes, or other languages will necessarily be specific to PostgreSQL in a significant way.

See Also

□□□ #2
□ □□□ □□□ 2026-10-31 14:09:03 UTC
□ □□□ □□□ 2026-10-31 14:09:04 UTC