CREATE PROCEDURE

CREATE PROCEDURE — define a new procedure

Synopsis

```
CREATE [ OR REPLACE ] PROCEDURE
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
    | sql_body
  } ...
```

Description

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

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

To replace the current definition of an existing procedure, use `CREATE OR REPLACE PROCEDURE`. It is not possible to change the name or argument types of a procedure this way (if you tried, you would actually be creating a new, distinct procedure).

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

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

To be able to create a procedure, you must have `USAGE` privilege on the argument types.

Refer to Section 38.4 for further information on writing procedures.

Parameters

name

The name (optionally schema-qualified) of the procedure to create.

argmode

The mode of an argument: `IN`, `OUT`, `INOUT`, or `VARIADIC`. If omitted, the default is `IN`.

argname

The name of an argument.

argtype

The data type(s) of the procedure'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 procedure 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. All input parameters following a parameter with a default value must have default values as well.

lang_name

The name of the language that the procedure 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 procedure should apply. Transforms convert between SQL types and language-specific data types; see CREATE TRANSFORM. 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.

```
[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER
```

`SECURITY INVOKER` indicates that the procedure is to be executed with the privileges of the user that calls it. That is the default. `SECURITY DEFINER` specifies that the procedure is to be executed with the privileges of the user that owns it.

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

A `SECURITY DEFINER` procedure cannot execute transaction control statements (for example, `COMMIT` and `ROLLBACK`, depending on the language).

configuration_parameter
value

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

If a `SET` clause is attached to a procedure, then the effects of a `SET LOCAL` command executed inside the procedure for the same variable are restricted to the procedure: the configuration parameter's prior value is still restored at procedure 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 procedure exit, unless the current transaction is rolled back.

If a `SET` clause is attached to a procedure, then that procedure cannot execute transaction control statements (for example, `COMMIT` and `ROLLBACK`, depending on the language).

See SET and Chapter 20 for more information about allowed parameter names and values.

definition

A string constant defining the procedure; the meaning depends on the language. It can be an internal procedure 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 Section 4.1.2.4) to write the procedure definition string, rather than the normal single quote syntax. Without dollar quoting, any single quotes or backslashes in the procedure definition must be escaped by doubling them.

obj_file, ***link_symbol***

This form of the `AS` clause is used for dynamically loadable C language procedures when the procedure name in the C language source code is not the same as the name of the SQL procedure. The string ***obj_file*** is the name of the shared library file containing the compiled C procedure, and is interpreted as for the LOAD command. The string ***link_symbol*** is the procedure's link symbol, that is, the name of the procedure 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 procedure being defined.

When repeated `CREATE PROCEDURE` 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` procedure. This should be a block

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

This is similar to writing the text of the procedure 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 procedure 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 procedure definition time. This form tracks dependencies between the procedure and objects used in the procedure body, so `DROP ... CASCADE` will work correctly, whereas the form using string literals may leave dangling procedures. Finally, this form is more compatible with the SQL standard and other SQL implementations.

Notes

See CREATE FUNCTION for more details on function creation that also apply to procedures.

Use CALL to execute a procedure.

Examples

```
CREATE PROCEDURE insert_data(a integer, b integer)
LANGUAGE SQL
AS $$
INSERT INTO tbl VALUES (a);
INSERT INTO tbl VALUES (b);
$$;
```

or

```
CREATE PROCEDURE insert_data(a integer, b integer)
LANGUAGE SQL
BEGIN ATOMIC
  INSERT INTO tbl VALUES (a);
  INSERT INTO tbl VALUES (b);
END;
```

and call like this:

```
CALL insert_data(1, 2);
```

Compatibility

A `CREATE PROCEDURE` command is defined in the SQL standard. The PostgreSQL implementation can be used in a compatible way but has many extensions. For details see also CREATE FUNCTION.

See Also

ALTER PROCEDURE, DROP PROCEDURE, CALL, CREATE FUNCTION

Submit correction

If you see anything in the documentation that is not correct, does not match your experience with the particular feature or requires further clarification, please use this form to report a documentation issue.