This part is about extending the server functionality with user-defined functions, data types, triggers, etc. These are advanced topics which should probably be approached only after all the other user documentation about PostgreSQL has been understood. Later chapters in this part describe the server-side programming languages available in the PostgreSQL distribution as well as general issues concerning server-side programming languages. It is essential to read at least the earlier sections of [**Chapter 37**](https://www.postgresql.org/docs/10/extend.html) (covering functions) before diving into the material about server-side programming languages.

**Chapter 37. Extending SQL**

In the sections that follow, we will discuss how you can extend the PostgreSQL SQL query language by adding:

* functions (starting in [**Section 37.3**](https://www.postgresql.org/docs/10/xfunc.html))
* aggregates (starting in [**Section 37.10**](https://www.postgresql.org/docs/10/xaggr.html))
* data types (starting in [**Section 37.11**](https://www.postgresql.org/docs/10/xtypes.html))
* operators (starting in [**Section 37.12**](https://www.postgresql.org/docs/10/xoper.html))
* operator classes for indexes (starting in [**Section 37.14**](https://www.postgresql.org/docs/10/xindex.html))
* packages of related objects (starting in [**Section 37.15**](https://www.postgresql.org/docs/10/extend-extensions.html))

## 37.1. How Extensibility Works

PostgreSQL is extensible because its operation is catalog-driven. If you are familiar with standard relational database systems, you know that they store information about databases, tables, columns, etc., in what are commonly known as system catalogs. (Some systems call this the data dictionary.) The catalogs appear to the user as tables like any other, but the DBMS stores its internal bookkeeping in them. One key difference between PostgreSQL and standard relational database systems is that PostgreSQL stores much more information in its catalogs: not only information about tables and columns, but also information about data types, functions, access methods, and so on. These tables can be modified by the user, and since PostgreSQL bases its operation on these tables, this means that PostgreSQL can be extended by users. By comparison, conventional database systems can only be extended by changing hardcoded procedures in the source code or by loading modules specially written by the DBMS vendor.

The PostgreSQL server can moreover incorporate user-written code into itself through dynamic loading. That is, the user can specify an object code file (e.g., a shared library) that implements a new type or function, and PostgreSQL will load it as required. Code written in SQL is even more trivial to add to the server. This ability to modify its operation “on the fly” makes PostgreSQL uniquely suited for rapid prototyping of new applications and storage structures.

## © 37.2. The PostgreSQL Type System

PostgreSQL data types are divided into base types, composite types, domains, and pseudo-types.

### 37.2.1. Base Types

Base types are those, like int4, that are implemented below the level of the SQL language (typically in a low-level language such as C). They generally correspond to what are often known as abstract data types. PostgreSQL can only operate on such types through functions provided by the user and only understands the behavior of such types to the extent that the user describes them. Base types are further subdivided into scalar and array types. For each scalar type, a corresponding array type is automatically created that can hold variable-size arrays of that scalar type.

### 37.2.2. Composite Types

Composite types, or row types, are created whenever the user creates a table. It is also possible to use [**CREATE TYPE**](https://www.postgresql.org/docs/10/sql-createtype.html) to define a “stand-alone” composite type with no associated table. A composite type is simply a list of types with associated field names. A value of a composite type is a row or record of field values. The user can access the component fields from SQL queries. Refer to [**Section 8.16**](https://www.postgresql.org/docs/10/rowtypes.html) for more information on composite types.

### 37.2.3. Domains

A domain is based on a particular base type and for many purposes is interchangeable with its base type. However, a domain can have constraints that restrict its valid values to a subset of what the underlying base type would allow.

Domains can be created using the SQL command [**CREATE DOMAIN**](https://www.postgresql.org/docs/10/sql-createdomain.html). Their creation and use is not discussed in this chapter.

### 37.2.4. Pseudo-Types

There are a few “pseudo-types” for special purposes. Pseudo-types cannot appear as columns of tables or attributes of composite types, but they can be used to declare the argument and result types of functions. This provides a mechanism within the type system to identify special classes of functions. [**Table 8.25**](https://www.postgresql.org/docs/10/datatype-pseudo.html#DATATYPE-PSEUDOTYPES-TABLE) lists the existing pseudo-types.

### 37.2.5. Polymorphic Types

Five pseudo-types of special interest are anyelement, anyarray, anynonarray, anyenum, and anyrange, which are collectively called polymorphic types. Any function declared using these types is said to be a polymorphic function. A polymorphic function can operate on many different data types, with the specific data type(s) being determined by the data types actually passed to it in a particular call.

Polymorphic arguments and results are tied to each other and are resolved to a specific data type when a query calling a polymorphic function is parsed. Each position (either argument or return value) declared as anyelement is allowed to have any specific actual data type, but in any given call they must all be the same actual type. Each position declared as anyarray can have any array data type, but similarly they must all be the same type. And similarly, positions declared as anyrange must all be the same range type. Furthermore, if there are positions declared anyarray and others declared anyelement, the actual array type in the anyarray positions must be an array whose elements are the same type appearing in the anyelement positions. Similarly, if there are positions declared anyrange and others declared anyelement, the actual range type in the anyrange positions must be a range whose subtype is the same type appearing in the anyelement positions. anynonarray is treated exactly the same as anyelement, but adds the additional constraint that the actual type must not be an array type. anyenum is treated exactly the same as anyelement, but adds the additional constraint that the actual type must be an enum type.

Thus, when more than one argument position is declared with a polymorphic type, the net effect is that only certain combinations of actual argument types are allowed. For example, a function declared as equal(anyelement, anyelement) will take any two input values, so long as they are of the same data type.

When the return value of a function is declared as a polymorphic type, there must be at least one argument position that is also polymorphic, and the actual data type supplied as the argument determines the actual result type for that call. For example, if there were not already an array subscripting mechanism, one could define a function that implements subscripting as subscript(anyarray, integer) returns anyelement. This declaration constrains the actual first argument to be an array type, and allows the parser to infer the correct result type from the actual first argument's type. Another example is that a function declared as f(anyarray) returns anyenum will only accept arrays of enum types.

Note that anynonarray and anyenum do not represent separate type variables; they are the same type as anyelement, just with an additional constraint. For example, declaring a function as f(anyelement, anyenum) is equivalent to declaring it as f(anyenum, anyenum): both actual arguments have to be the same enum type.

A variadic function (one taking a variable number of arguments, as in [**Section 37.4.5**](https://www.postgresql.org/docs/10/xfunc-sql.html#XFUNC-SQL-VARIADIC-FUNCTIONS)) can be polymorphic: this is accomplished by declaring its last parameter as VARIADIC anyarray. For purposes of argument matching and determining the actual result type, such a function behaves the same as if you had written the appropriate number of anynonarray parameters.

## 37.3. User-defined Functions

PostgreSQL provides four kinds of functions:

* query language functions (functions written in SQL) ([**Section 37.4**](https://www.postgresql.org/docs/10/xfunc-sql.html))
* procedural language functions (functions written in, for example, PL/pgSQL or PL/Tcl) ([**Section 37.7**](https://www.postgresql.org/docs/10/xfunc-pl.html))
* internal functions ([**Section 37.8**](https://www.postgresql.org/docs/10/xfunc-internal.html))
* C-language functions ([**Section 37.9**](https://www.postgresql.org/docs/10/xfunc-c.html))

Every kind of function can take base types, composite types, or combinations of these as arguments (parameters). In addition, every kind of function can return a base type or a composite type. Functions can also be defined to return sets of base or composite values.

Many kinds of functions can take or return certain pseudo-types (such as polymorphic types), but the available facilities vary. Consult the description of each kind of function for more details.

It's easiest to define SQL functions, so we'll start by discussing those. Most of the concepts presented for SQL functions will carry over to the other types of functions.

Throughout this chapter, it can be useful to look at the reference page of the [**CREATE FUNCTION**](https://www.postgresql.org/docs/10/sql-createfunction.html) command to understand the examples better. Some examples from this chapter can be found in funcs.sql and funcs.c in the src/tutorial directory in the PostgreSQL source distribution.

## 37.4. Query Language (SQL) Functions

SQL functions execute an arbitrary list of SQL statements, returning the result of the last query in the list. In the simple (non-set) case, the first row of the last query's result will be returned. (Bear in mind that “the first row” of a multirow result is not well-defined unless you use ORDER BY.) If the last query happens to return no rows at all, the null value will be returned.

Alternatively, an SQL function can be declared to return a set (that is, multiple rows) by specifying the function's return type as SETOF ***sometype***, or equivalently by declaring it as RETURNS TABLE(***columns***). In this case all rows of the last query's result are returned. Further details appear below.

The body of an SQL function must be a list of SQL statements separated by semicolons. A semicolon after the last statement is optional. Unless the function is declared to return void, the last statement must be a SELECT, or an INSERT, UPDATE, or DELETE that has a RETURNING clause.

Any collection of commands in the SQL language can be packaged together and defined as a function. Besides SELECT queries, the commands can include data modification queries (INSERT, UPDATE, and DELETE), as well as other SQL commands. (You cannot use transaction control commands, e.g. COMMIT, SAVEPOINT, and some utility commands, e.g. VACUUM, in SQL functions.) However, the final command must be a SELECT or have a RETURNING clause that returns whatever is specified as the function's return type. Alternatively, if you want to define a SQL function that performs actions but has no useful value to return, you can define it as returning void. For example, this function removes rows with negative salaries from the emp table:

CREATE FUNCTION clean\_emp() RETURNS void AS '

DELETE FROM emp

WHERE salary < 0;

' LANGUAGE SQL;

SELECT clean\_emp();

clean\_emp

-----------

(1 row)

### Note

The entire body of a SQL function is parsed before any of it is executed. While a SQL function can contain commands that alter the system catalogs (e.g., CREATE TABLE), the effects of such commands will not be visible during parse analysis of later commands in the function. Thus, for example, CREATE TABLE foo (...); INSERT INTO foo VALUES(...);will not work as desired if packaged up into a single SQL function, since foo won't exist yet when the INSERT command is parsed. It's recommended to use PL/pgSQL instead of a SQL function in this type of situation.

The syntax of the CREATE FUNCTION command requires the function body to be written as a string constant. It is usually most convenient to use dollar quoting (see [**Section 4.1.2.4**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-DOLLAR-QUOTING)) for the string constant. If you choose to use regular single-quoted string constant syntax, you must double single quote marks (') and backslashes (\) (assuming escape string syntax) in the body of the function (see [**Section 4.1.2.1**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-STRINGS)).

### 37.4.1. Arguments for SQL Functions

Arguments of a SQL function can be referenced in the function body using either names or numbers. Examples of both methods appear below.

To use a name, declare the function argument as having a name, and then just write that name in the function body. If the argument name is the same as any column name in the current SQL command within the function, the column name will take precedence. To override this, qualify the argument name with the name of the function itself, that is ***function\_name***.***argument\_name***. (If this would conflict with a qualified column name, again the column name wins. You can avoid the ambiguity by choosing a different alias for the table within the SQL command.)

In the older numeric approach, arguments are referenced using the syntax $***n***: $1 refers to the first input argument, $2 to the second, and so on. This will work whether or not the particular argument was declared with a name.

If an argument is of a composite type, then the dot notation, e.g., ***argname***.***fieldname*** or $1.***fieldname***, can be used to access attributes of the argument. Again, you might need to qualify the argument's name with the function name to make the form with an argument name unambiguous.

SQL function arguments can only be used as data values, not as identifiers. Thus for example this is reasonable:

INSERT INTO mytable VALUES ($1);

but this will not work:

INSERT INTO $1 VALUES (42);

### Note

The ability to use names to reference SQL function arguments was added in PostgreSQL9.2. Functions to be used in older servers must use the $***n*** notation.

### 37.4.2. SQL Functions on Base Types

The simplest possible SQL function has no arguments and simply returns a base type, such as integer:

CREATE FUNCTION one() RETURNS integer AS $$

SELECT 1 AS result;

$$ LANGUAGE SQL;

-- Alternative syntax for string literal:

CREATE FUNCTION one() RETURNS integer AS '

SELECT 1 AS result;

' LANGUAGE SQL;

SELECT one();

one

-----

1

Notice that we defined a column alias within the function body for the result of the function (with the name result), but this column alias is not visible outside the function. Hence, the result is labeled one instead of result.

It is almost as easy to define SQL functions that take base types as arguments:

CREATE FUNCTION add\_em(x integer, y integer) RETURNS integer AS $$

SELECT x + y;

$$ LANGUAGE SQL;

SELECT add\_em(1, 2) AS answer;

answer

--------

3

Alternatively, we could dispense with names for the arguments and use numbers:

CREATE FUNCTION add\_em(integer, integer) RETURNS integer AS $$

SELECT $1 + $2;

$$ LANGUAGE SQL;

SELECT add\_em(1, 2) AS answer;

answer

--------

3

Here is a more useful function, which might be used to debit a bank account:

CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$

UPDATE bank

SET balance = balance - debit

WHERE accountno = tf1.accountno;

SELECT 1;

$$ LANGUAGE SQL;

A user could execute this function to debit account 17 by $100.00 as follows:

SELECT tf1(17, 100.0);

In this example, we chose the name accountno for the first argument, but this is the same as the name of a column in the bank table. Within the UPDATE command, accountno refers to the column bank.accountno, so tf1.accountno must be used to refer to the argument. We could of course avoid this by using a different name for the argument.

In practice one would probably like a more useful result from the function than a constant 1, so a more likely definition is:

CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$

UPDATE bank

SET balance = balance - debit

WHERE accountno = tf1.accountno;

SELECT balance FROM bank WHERE accountno = tf1.accountno;

$$ LANGUAGE SQL;

which adjusts the balance and returns the new balance. The same thing could be done in one command using RETURNING:

CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$

UPDATE bank

SET balance = balance - debit

WHERE accountno = tf1.accountno

RETURNING balance;

$$ LANGUAGE SQL;

### 37.4.3. SQL Functions on Composite Types

When writing functions with arguments of composite types, we must not only specify which argument we want but also the desired attribute (field) of that argument. For example, suppose that emp is a table containing employee data, and therefore also the name of the composite type of each row of the table. Here is a function double\_salary that computes what someone's salary would be if it were doubled:

CREATE TABLE emp (

name text,

salary numeric,

age integer,

cubicle point

);

INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

CREATE FUNCTION double\_salary(emp) RETURNS numeric AS $$

SELECT $1.salary \* 2 AS salary;

$$ LANGUAGE SQL;

SELECT name, double\_salary(emp.\*) AS dream

FROM emp

WHERE emp.cubicle ~= point '(2,1)';

name | dream

------+-------

Bill | 8400

Notice the use of the syntax $1.salary to select one field of the argument row value. Also notice how the calling SELECT command uses ***table\_name***.\* to select the entire current row of a table as a composite value. The table row can alternatively be referenced using just the table name, like this:

SELECT name, double\_salary(emp) AS dream

FROM emp

WHERE emp.cubicle ~= point '(2,1)';

but this usage is deprecated since it's easy to get confused. (See [**Section 8.16.5**](https://www.postgresql.org/docs/10/rowtypes.html#ROWTYPES-USAGE) for details about these two notations for the composite value of a table row.)

Sometimes it is handy to construct a composite argument value on-the-fly. This can be done with the ROW construct. For example, we could adjust the data being passed to the function:

SELECT name, double\_salary(ROW(name, salary\*1.1, age, cubicle)) AS dream

FROM emp;

It is also possible to build a function that returns a composite type. This is an example of a function that returns a single emp row:

CREATE FUNCTION new\_emp() RETURNS emp AS $$

SELECT text 'None' AS name,

1000.0 AS salary,

25 AS age,

point '(2,2)' AS cubicle;

$$ LANGUAGE SQL;

In this example we have specified each of the attributes with a constant value, but any computation could have been substituted for these constants.

Note two important things about defining the function:

* The select list order in the query must be exactly the same as that in which the columns appear in the table associated with the composite type. (Naming the columns, as we did above, is irrelevant to the system.)
* You must typecast the expressions to match the definition of the composite type, or you will get errors like this:

ERROR: function declared to return emp returns varchar instead of text at column 1

A different way to define the same function is:

CREATE FUNCTION new\_emp() RETURNS emp AS $$

SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;

$$ LANGUAGE SQL;

Here we wrote a SELECT that returns just a single column of the correct composite type. This isn't really better in this situation, but it is a handy alternative in some cases — for example, if we need to compute the result by calling another function that returns the desired composite value.

We could call this function directly either by using it in a value expression:

SELECT new\_emp();

new\_emp

--------------------------

(None,1000.0,25,"(2,2)")

or by calling it as a table function:

SELECT \* FROM new\_emp();

name | salary | age | cubicle

------+--------+-----+---------

None | 1000.0 | 25 | (2,2)

The second way is described more fully in [**Section 37.4.7**](https://www.postgresql.org/docs/10/xfunc-sql.html#XFUNC-SQL-TABLE-FUNCTIONS).

When you use a function that returns a composite type, you might want only one field (attribute) from its result. You can do that with syntax like this:

SELECT (new\_emp()).name;

name

------

None

The extra parentheses are needed to keep the parser from getting confused. If you try to do it without them, you get something like this:

SELECT new\_emp().name;

ERROR: syntax error at or near "."

LINE 1: SELECT new\_emp().name;

^

Another option is to use functional notation for extracting an attribute:

SELECT name(new\_emp());

name

------

None

As explained in [**Section 8.16.5**](https://www.postgresql.org/docs/10/rowtypes.html#ROWTYPES-USAGE), the field notation and functional notation are equivalent.

Another way to use a function returning a composite type is to pass the result to another function that accepts the correct row type as input:

CREATE FUNCTION getname(emp) RETURNS text AS $$

SELECT $1.name;

$$ LANGUAGE SQL;

SELECT getname(new\_emp());

getname

---------

None

(1 row)

### 37.4.4. SQL Functions with Output Parameters

An alternative way of describing a function's results is to define it with output parameters, as in this example:

CREATE FUNCTION add\_em (IN x int, IN y int, OUT sum int)

AS 'SELECT x + y'

LANGUAGE SQL;

SELECT add\_em(3,7);

add\_em

--------

10

(1 row)

This is not essentially different from the version of add\_em shown in [**Section 37.4.2**](https://www.postgresql.org/docs/10/xfunc-sql.html#XFUNC-SQL-BASE-FUNCTIONS). The real value of output parameters is that they provide a convenient way of defining functions that return several columns. For example,

CREATE FUNCTION sum\_n\_product (x int, y int, OUT sum int, OUT product int)

AS 'SELECT x + y, x \* y'

LANGUAGE SQL;

SELECT \* FROM sum\_n\_product(11,42);

sum | product

-----+---------

53 | 462

(1 row)

What has essentially happened here is that we have created an anonymous composite type for the result of the function. The above example has the same end result as

CREATE TYPE sum\_prod AS (sum int, product int);

CREATE FUNCTION sum\_n\_product (int, int) RETURNS sum\_prod

AS 'SELECT $1 + $2, $1 \* $2'

LANGUAGE SQL;

but not having to bother with the separate composite type definition is often handy. Notice that the names attached to the output parameters are not just decoration, but determine the column names of the anonymous composite type. (If you omit a name for an output parameter, the system will choose a name on its own.)

Notice that output parameters are not included in the calling argument list when invoking such a function from SQL. This is because PostgreSQL considers only the input parameters to define the function's calling signature. That means also that only the input parameters matter when referencing the function for purposes such as dropping it. We could drop the above function with either of

DROP FUNCTION sum\_n\_product (x int, y int, OUT sum int, OUT product int);

DROP FUNCTION sum\_n\_product (int, int);

Parameters can be marked as IN (the default), OUT, INOUT, or VARIADIC. An INOUT parameter serves as both an input parameter (part of the calling argument list) and an output parameter (part of the result record type). VARIADIC parameters are input parameters, but are treated specially as described next.

### 37.4.5. SQL Functions with Variable Numbers of Arguments

SQL functions can be declared to accept variable numbers of arguments, so long as all the “optional” arguments are of the same data type. The optional arguments will be passed to the function as an array. The function is declared by marking the last parameter as VARIADIC; this parameter must be declared as being of an array type. For example:

CREATE FUNCTION mleast(VARIADIC arr numeric[]) RETURNS numeric AS $$

SELECT min($1[i]) FROM generate\_subscripts($1, 1) g(i);

$$ LANGUAGE SQL;

SELECT mleast(10, -1, 5, 4.4);

mleast

--------

-1

(1 row)

Effectively, all the actual arguments at or beyond the VARIADIC position are gathered up into a one-dimensional array, as if you had written

SELECT mleast(ARRAY[10, -1, 5, 4.4]); -- doesn't work

You can't actually write that, though — or at least, it will not match this function definition. A parameter marked VARIADIC matches one or more occurrences of its element type, not of its own type.

Sometimes it is useful to be able to pass an already-constructed array to a variadic function; this is particularly handy when one variadic function wants to pass on its array parameter to another one. Also, this is the only secure way to call a variadic function found in a schema that permits untrusted users to create objects; see [**Section 10.3**](https://www.postgresql.org/docs/10/typeconv-func.html). You can do this by specifying VARIADIC in the call:

SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);

This prevents expansion of the function's variadic parameter into its element type, thereby allowing the array argument value to match normally. VARIADIC can only be attached to the last actual argument of a function call.

Specifying VARIADIC in the call is also the only way to pass an empty array to a variadic function, for example:

SELECT mleast(VARIADIC ARRAY[]::numeric[]);

Simply writing SELECT mleast() does not work because a variadic parameter must match at least one actual argument. (You could define a second function also named mleast, with no parameters, if you wanted to allow such calls.)

The array element parameters generated from a variadic parameter are treated as not having any names of their own. This means it is not possible to call a variadic function using named arguments ([**Section 4.3**](https://www.postgresql.org/docs/10/sql-syntax-calling-funcs.html)), except when you specify VARIADIC. For example, this will work:

SELECT mleast(VARIADIC arr => ARRAY[10, -1, 5, 4.4]);

but not these:

SELECT mleast(arr => 10);

SELECT mleast(arr => ARRAY[10, -1, 5, 4.4]);

### 37.4.6. SQL Functions with Default Values for Arguments

Functions can be declared with default values for some or all input arguments. The default values are inserted whenever the function is called with insufficiently many actual arguments. Since arguments can only be omitted from the end of the actual argument list, all parameters after a parameter with a default value have to have default values as well. (Although the use of named argument notation could allow this restriction to be relaxed, it's still enforced so that positional argument notation works sensibly.) Whether or not you use it, this capability creates a need for precautions when calling functions in databases where some users mistrust other users; see [**Section 10.3**](https://www.postgresql.org/docs/10/typeconv-func.html).

For example:

CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)

RETURNS int

LANGUAGE SQL

AS $$

SELECT $1 + $2 + $3;

$$;

SELECT foo(10, 20, 30);

foo

-----

60

(1 row)

SELECT foo(10, 20);

foo

-----

33

(1 row)

SELECT foo(10);

foo

-----

15

(1 row)

SELECT foo(); -- fails since there is no default for the first argument

ERROR: function foo() does not exist

The = sign can also be used in place of the key word DEFAULT.

### 37.4.7. SQL Functions as Table Sources

All SQL functions can be used in the FROM clause of a query, but it is particularly useful for functions returning composite types. If the function is defined to return a base type, the table function produces a one-column table. If the function is defined to return a composite type, the table function produces a column for each attribute of the composite type.

Here is an example:

CREATE TABLE foo (fooid int, foosubid int, fooname text);

INSERT INTO foo VALUES (1, 1, 'Joe');

INSERT INTO foo VALUES (1, 2, 'Ed');

INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$

SELECT \* FROM foo WHERE fooid = $1;

$$ LANGUAGE SQL;

SELECT \*, upper(fooname) FROM getfoo(1) AS t1;

fooid | foosubid | fooname | upper

-------+----------+---------+-------

1 | 1 | Joe | JOE

(1 row)

As the example shows, we can work with the columns of the function's result just the same as if they were columns of a regular table.

Note that we only got one row out of the function. This is because we did not use SETOF. That is described in the next section.

### 37.4.8. SQL Functions Returning Sets

When an SQL function is declared as returning SETOF ***sometype***, the function's final query is executed to completion, and each row it outputs is returned as an element of the result set.

This feature is normally used when calling the function in the FROM clause. In this case each row returned by the function becomes a row of the table seen by the query. For example, assume that table foo has the same contents as above, and we say:

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$

SELECT \* FROM foo WHERE fooid = $1;

$$ LANGUAGE SQL;

SELECT \* FROM getfoo(1) AS t1;

Then we would get:

fooid | foosubid | fooname

-------+----------+---------

1 | 1 | Joe

1 | 2 | Ed

(2 rows)

It is also possible to return multiple rows with the columns defined by output parameters, like this:

CREATE TABLE tab (y int, z int);

INSERT INTO tab VALUES (1, 2), (3, 4), (5, 6), (7, 8);

CREATE FUNCTION sum\_n\_product\_with\_tab (x int, OUT sum int, OUT product int)

RETURNS SETOF record

AS $$

SELECT $1 + tab.y, $1 \* tab.y FROM tab;

$$ LANGUAGE SQL;

SELECT \* FROM sum\_n\_product\_with\_tab(10);

sum | product

-----+---------

11 | 10

13 | 30

15 | 50

17 | 70

(4 rows)

The key point here is that you must write RETURNS SETOF record to indicate that the function returns multiple rows instead of just one. If there is only one output parameter, write that parameter's type instead of record.

It is frequently useful to construct a query's result by invoking a set-returning function multiple times, with the parameters for each invocation coming from successive rows of a table or subquery. The preferred way to do this is to use the LATERAL key word, which is described in [**Section 7.2.1.5**](https://www.postgresql.org/docs/10/queries-table-expressions.html#QUERIES-LATERAL). Here is an example using a set-returning function to enumerate elements of a tree structure:

SELECT \* FROM nodes;

name | parent

-----------+--------

Top |

Child1 | Top

Child2 | Top

Child3 | Top

SubChild1 | Child1

SubChild2 | Child1

(6 rows)

CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$

SELECT name FROM nodes WHERE parent = $1

$$ LANGUAGE SQL STABLE;

SELECT \* FROM listchildren('Top');

listchildren

--------------

Child1

Child2

Child3

(3 rows)

SELECT name, child FROM nodes, LATERAL listchildren(name) AS child;

name | child

--------+-----------

Top | Child1

Top | Child2

Top | Child3

Child1 | SubChild1

Child1 | SubChild2

(5 rows)

This example does not do anything that we couldn't have done with a simple join, but in more complex calculations the option to put some of the work into a function can be quite convenient.

Functions returning sets can also be called in the select list of a query. For each row that the query generates by itself, the set-returning function is invoked, and an output row is generated for each element of the function's result set. The previous example could also be done with queries like these:

SELECT listchildren('Top');

listchildren

--------------

Child1

Child2

Child3

(3 rows)

SELECT name, listchildren(name) FROM nodes;

name | listchildren

--------+--------------

Top | Child1

Top | Child2

Top | Child3

Child1 | SubChild1

Child1 | SubChild2

(5 rows)

In the last SELECT, notice that no output row appears for Child2, Child3, etc. This happens because listchildren returns an empty set for those arguments, so no result rows are generated. This is the same behavior as we got from an inner join to the function result when using the LATERAL syntax.

PostgreSQL's behavior for a set-returning function in a query's select list is almost exactly the same as if the set-returning function had been written in a LATERAL FROM-clause item instead. For example,

SELECT x, generate\_series(1,5) AS g FROM tab;

is almost equivalent to

SELECT x, g FROM tab, LATERAL generate\_series(1,5) AS g;

It would be exactly the same, except that in this specific example, the planner could choose to put g on the outside of the nestloop join, since g has no actual lateral dependency on tab. That would result in a different output row order. Set-returning functions in the select list are always evaluated as though they are on the inside of a nestloop join with the rest of the FROM clause, so that the function(s) are run to completion before the next row from the FROM clause is considered.

If there is more than one set-returning function in the query's select list, the behavior is similar to what you get from putting the functions into a single LATERAL ROWS FROM( ... ) FROM-clause item. For each row from the underlying query, there is an output row using the first result from each function, then an output row using the second result, and so on. If some of the set-returning functions produce fewer outputs than others, null values are substituted for the missing data, so that the total number of rows emitted for one underlying row is the same as for the set-returning function that produced the most outputs. Thus the set-returning functions run “in lockstep” until they are all exhausted, and then execution continues with the next underlying row.

Set-returning functions can be nested in a select list, although that is not allowed in FROM-clause items. In such cases, each level of nesting is treated separately, as though it were a separate LATERAL ROWS FROM( ... ) item. For example, in

SELECT srf1(srf2(x), srf3(y)), srf4(srf5(z)) FROM tab;

the set-returning functions srf2, srf3, and srf5 would be run in lockstep for each row of tab, and then srf1 and srf4 would be applied in lockstep to each row produced by the lower functions.

Set-returning functions cannot be used within conditional-evaluation constructs, such as CASE or COALESCE. For example, consider

SELECT x, CASE WHEN x > 0 THEN generate\_series(1, 5) ELSE 0 END FROM tab;

It might seem that this should produce five repetitions of input rows that have x > 0, and a single repetition of those that do not; but actually, because generate\_series(1, 5) would be run in an implicit LATERAL FROM item before the CASE expression is ever evaluated, it would produce five repetitions of every input row. To reduce confusion, such cases produce a parse-time error instead.

### Note

If a function's last command is INSERT, UPDATE, or DELETE with RETURNING, that command will always be executed to completion, even if the function is not declared with SETOF or the calling query does not fetch all the result rows. Any extra rows produced by the RETURNINGclause are silently dropped, but the commanded table modifications still happen (and are all completed before returning from the function).

### Note

Before PostgreSQL 10, putting more than one set-returning function in the same select list did not behave very sensibly unless they always produced equal numbers of rows. Otherwise, what you got was a number of output rows equal to the least common multiple of the numbers of rows produced by the set-returning functions. Also, nested set-returning functions did not work as described above; instead, a set-returning function could have at most one set-returning argument, and each nest of set-returning functions was run independently. Also, conditional execution (set-returning functions inside CASEetc) was previously allowed, complicating things even more. Use of the LATERAL syntax is recommended when writing queries that need to work in older PostgreSQL versions, because that will give consistent results across different versions. If you have a query that is relying on conditional execution of a set-returning function, you may be able to fix it by moving the conditional test into a custom set-returning function. For example,

SELECT x, CASE WHEN y > 0 THEN generate\_series(1, z) ELSE 5 END FROM tab;

could become

CREATE FUNCTION case\_generate\_series(cond bool, start int, fin int, els int)

RETURNS SETOF int AS $$

BEGIN

IF cond THEN

RETURN QUERY SELECT generate\_series(start, fin);

ELSE

RETURN QUERY SELECT els;

END IF;

END$$ LANGUAGE plpgsql;

SELECT x, case\_generate\_series(y > 0, 1, z, 5) FROM tab;

This formulation will work the same in all versions of PostgreSQL.

### 37.4.9. SQL Functions Returning TABLE

There is another way to declare a function as returning a set, which is to use the syntax RETURNS TABLE(***columns***). This is equivalent to using one or more OUT parameters plus marking the function as returning SETOF record (or SETOF a single output parameter's type, as appropriate). This notation is specified in recent versions of the SQL standard, and thus may be more portable than using SETOF.

For example, the preceding sum-and-product example could also be done this way:

CREATE FUNCTION sum\_n\_product\_with\_tab (x int)

RETURNS TABLE(sum int, product int) AS $$

SELECT $1 + tab.y, $1 \* tab.y FROM tab;

$$ LANGUAGE SQL;

It is not allowed to use explicit OUT or INOUT parameters with the RETURNS TABLE notation — you must put all the output columns in the TABLE list.

### 37.4.10. Polymorphic SQL Functions

SQL functions can be declared to accept and return the polymorphic types anyelement, anyarray, anynonarray, anyenum, and anyrange. See [**Section 37.2.5**](https://www.postgresql.org/docs/10/extend-type-system.html#EXTEND-TYPES-POLYMORPHIC) for a more detailed explanation of polymorphic functions. Here is a polymorphic function make\_array that builds up an array from two arbitrary data type elements:

CREATE FUNCTION make\_array(anyelement, anyelement) RETURNS anyarray AS $$

SELECT ARRAY[$1, $2];

$$ LANGUAGE SQL;

SELECT make\_array(1, 2) AS intarray, make\_array('a'::text, 'b') AS textarray;

intarray | textarray

----------+-----------

{1,2} | {a,b}

(1 row)

Notice the use of the typecast 'a'::text to specify that the argument is of type text. This is required if the argument is just a string literal, since otherwise it would be treated as type unknown, and array of unknown is not a valid type. Without the typecast, you will get errors like this:

ERROR: could not determine polymorphic type because input has type "unknown"

It is permitted to have polymorphic arguments with a fixed return type, but the converse is not. For example:

CREATE FUNCTION is\_greater(anyelement, anyelement) RETURNS boolean AS $$

SELECT $1 > $2;

$$ LANGUAGE SQL;

SELECT is\_greater(1, 2);

is\_greater

------------

f

(1 row)

CREATE FUNCTION invalid\_func() RETURNS anyelement AS $$

SELECT 1;

$$ LANGUAGE SQL;

ERROR: cannot determine result data type

DETAIL: A function returning a polymorphic type must have at least one polymorphic argument.

Polymorphism can be used with functions that have output arguments. For example:

CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)

AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT \* FROM dup(22);

f2 | f3

----+---------

22 | {22,22}

(1 row)

Polymorphism can also be used with variadic functions. For example:

CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$

SELECT min($1[i]) FROM generate\_subscripts($1, 1) g(i);

$$ LANGUAGE SQL;

SELECT anyleast(10, -1, 5, 4);

anyleast

----------

-1

(1 row)

SELECT anyleast('abc'::text, 'def');

anyleast

----------

abc

(1 row)

CREATE FUNCTION concat\_values(text, VARIADIC anyarray) RETURNS text AS $$

SELECT array\_to\_string($2, $1);

$$ LANGUAGE SQL;

SELECT concat\_values('|', 1, 4, 2);

concat\_values

---------------

1|4|2

(1 row)

### 37.4.11. SQL Functions with Collations

When a SQL function has one or more parameters of collatable data types, a collation is identified for each function call depending on the collations assigned to the actual arguments, as described in [**Section 23.2**](https://www.postgresql.org/docs/10/collation.html). If a collation is successfully identified (i.e., there are no conflicts of implicit collations among the arguments) then all the collatable parameters are treated as having that collation implicitly. This will affect the behavior of collation-sensitive operations within the function. For example, using the anyleast function described above, the result of

SELECT anyleast('abc'::text, 'ABC');

will depend on the database's default collation. In C locale the result will be ABC, but in many other locales it will be abc. The collation to use can be forced by adding a COLLATE clause to any of the arguments, for example

SELECT anyleast('abc'::text, 'ABC' COLLATE "C");

Alternatively, if you wish a function to operate with a particular collation regardless of what it is called with, insert COLLATE clauses as needed in the function definition. This version of anyleast would always use en\_US locale to compare strings:

CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$

SELECT min($1[i] COLLATE "en\_US") FROM generate\_subscripts($1, 1) g(i);

$$ LANGUAGE SQL;

But note that this will throw an error if applied to a non-collatable data type.

If no common collation can be identified among the actual arguments, then a SQL function treats its parameters as having their data types' default collation (which is usually the database's default collation, but could be different for parameters of domain types).

The behavior of collatable parameters can be thought of as a limited form of polymorphism, applicable only to textual data types.

## 37.5. Function Overloading

More than one function can be defined with the same SQL name, so long as the arguments they take are different. In other words, function names can be overloaded. Whether or not you use it, this capability entails security precautions when calling functions in databases where some users mistrust other users; see [**Section 10.3**](https://www.postgresql.org/docs/10/typeconv-func.html). When a query is executed, the server will determine which function to call from the data types and the number of the provided arguments. Overloading can also be used to simulate functions with a variable number of arguments, up to a finite maximum number.

When creating a family of overloaded functions, one should be careful not to create ambiguities. For instance, given the functions:

CREATE FUNCTION test(int, real) RETURNS ...

CREATE FUNCTION test(smallint, double precision) RETURNS ...

it is not immediately clear which function would be called with some trivial input like test(1, 1.5). The currently implemented resolution rules are described in [**Chapter 10**](https://www.postgresql.org/docs/10/typeconv.html), but it is unwise to design a system that subtly relies on this behavior.

A function that takes a single argument of a composite type should generally not have the same name as any attribute (field) of that type. Recall that ***attribute***(***table***) is considered equivalent to ***table***.***attribute***. In the case that there is an ambiguity between a function on a composite type and an attribute of the composite type, the attribute will always be used. It is possible to override that choice by schema-qualifying the function name (that is, ***schema***.***func***(***table***) ) but it's better to avoid the problem by not choosing conflicting names.

Another possible conflict is between variadic and non-variadic functions. For instance, it is possible to create both foo(numeric) and foo(VARIADIC numeric[]). In this case it is unclear which one should be matched to a call providing a single numeric argument, such as foo(10.1). The rule is that the function appearing earlier in the search path is used, or if the two functions are in the same schema, the non-variadic one is preferred.

When overloading C-language functions, there is an additional constraint: The C name of each function in the family of overloaded functions must be different from the C names of all other functions, either internal or dynamically loaded. If this rule is violated, the behavior is not portable. You might get a run-time linker error, or one of the functions will get called (usually the internal one). The alternative form of the AS clause for the SQL CREATE FUNCTION command decouples the SQL function name from the function name in the C source code. For instance:

CREATE FUNCTION test(int) RETURNS int

AS '***filename***', 'test\_1arg'

LANGUAGE C;

CREATE FUNCTION test(int, int) RETURNS int

AS '***filename***', 'test\_2arg'

LANGUAGE C;

The names of the C functions here reflect one of many possible conventions.

## 37.6. Function Volatility Categories

Every function has a volatility classification, with the possibilities being VOLATILE, STABLE, or IMMUTABLE. VOLATILE is the default if the [**CREATE FUNCTION**](https://www.postgresql.org/docs/10/sql-createfunction.html) command does not specify a category. The volatility category is a promise to the optimizer about the behavior of the function:

* A VOLATILE function can do anything, including modifying the database. It can return different results on successive calls with the same arguments. The optimizer makes no assumptions about the behavior of such functions. A query using a volatile function will re-evaluate the function at every row where its value is needed.
* A STABLE function cannot modify the database and is guaranteed to return the same results given the same arguments for all rows within a single statement. This category allows the optimizer to optimize multiple calls of the function to a single call. In particular, it is safe to use an expression containing such a function in an index scan condition. (Since an index scan will evaluate the comparison value only once, not once at each row, it is not valid to use a VOLATILE function in an index scan condition.)
* An IMMUTABLE function cannot modify the database and is guaranteed to return the same results given the same arguments forever. This category allows the optimizer to pre-evaluate the function when a query calls it with constant arguments. For example, a query like SELECT ... WHERE x = 2 + 2 can be simplified on sight to SELECT ... WHERE x = 4, because the function underlying the integer addition operator is marked IMMUTABLE.

For best optimization results, you should label your functions with the strictest volatility category that is valid for them.

Any function with side-effects must be labeled VOLATILE, so that calls to it cannot be optimized away. Even a function with no side-effects needs to be labeled VOLATILE if its value can change within a single query; some examples are random(), currval(), timeofday().

Another important example is that the current\_timestamp family of functions qualify as STABLE, since their values do not change within a transaction.

There is relatively little difference between STABLE and IMMUTABLE categories when considering simple interactive queries that are planned and immediately executed: it doesn't matter a lot whether a function is executed once during planning or once during query execution startup. But there is a big difference if the plan is saved and reused later. Labeling a function IMMUTABLE when it really isn't might allow it to be prematurely folded to a constant during planning, resulting in a stale value being re-used during subsequent uses of the plan. This is a hazard when using prepared statements or when using function languages that cache plans (such as PL/pgSQL).

For functions written in SQL or in any of the standard procedural languages, there is a second important property determined by the volatility category, namely the visibility of any data changes that have been made by the SQL command that is calling the function. A VOLATILE function will see such changes, a STABLE or IMMUTABLE function will not. This behavior is implemented using the snapshotting behavior of MVCC (see [**Chapter 13**](https://www.postgresql.org/docs/10/mvcc.html)): STABLE and IMMUTABLE functions use a snapshot established as of the start of the calling query, whereas VOLATILE functions obtain a fresh snapshot at the start of each query they execute.

### Note

Functions written in C can manage snapshots however they want, but it's usually a good idea to make C functions work this way too.

Because of this snapshotting behavior, a function containing only SELECT commands can safely be marked STABLE, even if it selects from tables that might be undergoing modifications by concurrent queries. PostgreSQL will execute all commands of a STABLE function using the snapshot established for the calling query, and so it will see a fixed view of the database throughout that query.

The same snapshotting behavior is used for SELECT commands within IMMUTABLE functions. It is generally unwise to select from database tables within an IMMUTABLE function at all, since the immutability will be broken if the table contents ever change. However, PostgreSQL does not enforce that you do not do that.

A common error is to label a function IMMUTABLE when its results depend on a configuration parameter. For example, a function that manipulates timestamps might well have results that depend on the [**TimeZone**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-TIMEZONE) setting. For safety, such functions should be labeled STABLE instead.

### Note

PostgreSQL requires that STABLE and IMMUTABLE functions contain no SQL commands other than SELECT to prevent data modification. (This is not a completely bulletproof test, since such functions could still call VOLATILE functions that modify the database. If you do that, you will find that the STABLE or IMMUTABLE function does not notice the database changes applied by the called function, since they are hidden from its snapshot.)

## 37.7. Procedural Language Functions

PostgreSQL allows user-defined functions to be written in other languages besides SQL and C. These other languages are generically called procedural languages (PLs). Procedural languages aren't built into the PostgreSQL server; they are offered by loadable modules. See [**Chapter 41**](https://www.postgresql.org/docs/10/xplang.html) and following chapters for more information.

## 37.8. Internal Functions

Internal functions are functions written in C that have been statically linked into the PostgreSQL server. The “body” of the function definition specifies the C-language name of the function, which need not be the same as the name being declared for SQL use. (For reasons of backward compatibility, an empty body is accepted as meaning that the C-language function name is the same as the SQL name.)

Normally, all internal functions present in the server are declared during the initialization of the database cluster (see [**Section 18.2**](https://www.postgresql.org/docs/10/creating-cluster.html)), but a user could use CREATE FUNCTION to create additional alias names for an internal function. Internal functions are declared in CREATE FUNCTION with language name internal. For instance, to create an alias for the sqrt function:

CREATE FUNCTION square\_root(double precision) RETURNS double precision

AS 'dsqrt'

LANGUAGE internal

STRICT;

(Most internal functions expect to be declared “strict”.)

### Note

Not all “predefined” functions are “internal” in the above sense. Some predefined functions are written in SQL.

## 37.9. C-Language Functions

User-defined functions can be written in C (or a language that can be made compatible with C, such as C++). Such functions are compiled into dynamically loadable objects (also called shared libraries) and are loaded by the server on demand. The dynamic loading feature is what distinguishes “C language” functions from “internal” functions — the actual coding conventions are essentially the same for both. (Hence, the standard internal function library is a rich source of coding examples for user-defined C functions.)

Currently only one calling convention is used for C functions (“version 1”). Support for that calling convention is indicated by writing a PG\_FUNCTION\_INFO\_V1() macro call for the function, as illustrated below.

### 37.9.1. Dynamic Loading

The first time a user-defined function in a particular loadable object file is called in a session, the dynamic loader loads that object file into memory so that the function can be called. The CREATE FUNCTION for a user-defined C function must therefore specify two pieces of information for the function: the name of the loadable object file, and the C name (link symbol) of the specific function to call within that object file. If the C name is not explicitly specified then it is assumed to be the same as the SQL function name.

The following algorithm is used to locate the shared object file based on the name given in the CREATE FUNCTION command:

1. If the name is an absolute path, the given file is loaded.
2. If the name starts with the string $libdir, that part is replaced by the PostgreSQL package library directory name, which is determined at build time.
3. If the name does not contain a directory part, the file is searched for in the path specified by the configuration variable [**dynamic\_library\_path**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-DYNAMIC-LIBRARY-PATH).
4. Otherwise (the file was not found in the path, or it contains a non-absolute directory part), the dynamic loader will try to take the name as given, which will most likely fail. (It is unreliable to depend on the current working directory.)

If this sequence does not work, the platform-specific shared library file name extension (often .so) is appended to the given name and this sequence is tried again. If that fails as well, the load will fail.

It is recommended to locate shared libraries either relative to $libdir or through the dynamic library path. This simplifies version upgrades if the new installation is at a different location. The actual directory that $libdir stands for can be found out with the command pg\_config --pkglibdir.

The user ID the PostgreSQL server runs as must be able to traverse the path to the file you intend to load. Making the file or a higher-level directory not readable and/or not executable by the postgres user is a common mistake.

In any case, the file name that is given in the CREATE FUNCTION command is recorded literally in the system catalogs, so if the file needs to be loaded again the same procedure is applied.

### Note

PostgreSQL will not compile a C function automatically. The object file must be compiled before it is referenced in a CREATE FUNCTION command. See [**Section 37.9.5**](https://www.postgresql.org/docs/10/xfunc-c.html#DFUNC) for additional information.

To ensure that a dynamically loaded object file is not loaded into an incompatible server, PostgreSQL checks that the file contains a “magic block” with the appropriate contents. This allows the server to detect obvious incompatibilities, such as code compiled for a different major version of PostgreSQL. A magic block is required as of PostgreSQL 8.2. To include a magic block, write this in one (and only one) of the module source files, after having included the header fmgr.h:

#ifdef PG\_MODULE\_MAGIC

PG\_MODULE\_MAGIC;

#endif

The #ifdef test can be omitted if the code doesn't need to compile against pre-8.2 PostgreSQL releases.

After it is used for the first time, a dynamically loaded object file is retained in memory. Future calls in the same session to the function(s) in that file will only incur the small overhead of a symbol table lookup. If you need to force a reload of an object file, for example after recompiling it, begin a fresh session.

Optionally, a dynamically loaded file can contain initialization and finalization functions. If the file includes a function named \_PG\_init, that function will be called immediately after loading the file. The function receives no parameters and should return void. If the file includes a function named \_PG\_fini, that function will be called immediately before unloading the file. Likewise, the function receives no parameters and should return void. Note that \_PG\_fini will only be called during an unload of the file, not during process termination. (Presently, unloads are disabled and will never occur, but this may change in the future.)

### 37.9.2. Base Types in C-Language Functions

To know how to write C-language functions, you need to know how PostgreSQL internally represents base data types and how they can be passed to and from functions. Internally, PostgreSQLregards a base type as a “blob of memory”. The user-defined functions that you define over a type in turn define the way that PostgreSQL can operate on it. That is, PostgreSQL will only store and retrieve the data from disk and use your user-defined functions to input, process, and output the data.

Base types can have one of three internal formats:

* pass by value, fixed-length
* pass by reference, fixed-length
* pass by reference, variable-length

By-value types can only be 1, 2, or 4 bytes in length (also 8 bytes, if sizeof(Datum) is 8 on your machine). You should be careful to define your types such that they will be the same size (in bytes) on all architectures. For example, the long type is dangerous because it is 4 bytes on some machines and 8 bytes on others, whereas int type is 4 bytes on most Unix machines. A reasonable implementation of the int4 type on Unix machines might be:

/\* 4-byte integer, passed by value \*/

typedef int int4;

(The actual PostgreSQL C code calls this type int32, because it is a convention in C that int***XX*** means ***XX*** bits. Note therefore also that the C type int8 is 1 byte in size. The SQL type int8 is called int64in C. See also [**Table 37.1**](https://www.postgresql.org/docs/10/xfunc-c.html#XFUNC-C-TYPE-TABLE).)

On the other hand, fixed-length types of any size can be passed by-reference. For example, here is a sample implementation of a PostgreSQL type:

/\* 16-byte structure, passed by reference \*/

typedef struct

{

double x, y;

} Point;

Only pointers to such types can be used when passing them in and out of PostgreSQL functions. To return a value of such a type, allocate the right amount of memory with palloc, fill in the allocated memory, and return a pointer to it. (Also, if you just want to return the same value as one of your input arguments that's of the same data type, you can skip the extra palloc and just return the pointer to the input value.)

Finally, all variable-length types must also be passed by reference. All variable-length types must begin with an opaque length field of exactly 4 bytes, which will be set by SET\_VARSIZE; never set this field directly! All data to be stored within that type must be located in the memory immediately following that length field. The length field contains the total length of the structure, that is, it includes the size of the length field itself.

Another important point is to avoid leaving any uninitialized bits within data type values; for example, take care to zero out any alignment padding bytes that might be present in structs. Without this, logically-equivalent constants of your data type might be seen as unequal by the planner, leading to inefficient (though not incorrect) plans.

### Warning

Never modify the contents of a pass-by-reference input value. If you do so you are likely to corrupt on-disk data, since the pointer you are given might point directly into a disk buffer. The sole exception to this rule is explained in [**Section 37.10**](https://www.postgresql.org/docs/10/xaggr.html).

As an example, we can define the type text as follows:

typedef struct {

int32 length;

char data[FLEXIBLE\_ARRAY\_MEMBER];

} text;

The [FLEXIBLE\_ARRAY\_MEMBER] notation means that the actual length of the data part is not specified by this declaration.

When manipulating variable-length types, we must be careful to allocate the correct amount of memory and set the length field correctly. For example, if we wanted to store 40 bytes in a textstructure, we might use a code fragment like this:

#include "postgres.h"

...

char buffer[40]; /\* our source data \*/

...

text \*destination = (text \*) palloc(VARHDRSZ + 40);

SET\_VARSIZE(destination, VARHDRSZ + 40);

memcpy(destination->data, buffer, 40);

...

VARHDRSZ is the same as sizeof(int32), but it's considered good style to use the macro VARHDRSZ to refer to the size of the overhead for a variable-length type. Also, the length field must be set using the SET\_VARSIZE macro, not by simple assignment.

[**Table 37.1**](https://www.postgresql.org/docs/10/xfunc-c.html#XFUNC-C-TYPE-TABLE) specifies which C type corresponds to which SQL type when writing a C-language function that uses a built-in type of PostgreSQL. The “Defined In” column gives the header file that needs to be included to get the type definition. (The actual definition might be in a different file that is included by the listed file. It is recommended that users stick to the defined interface.) Note that you should always include postgres.h first in any source file, because it declares a number of things that you will need anyway.

**Table 37.1. Equivalent C Types for Built-in SQL Types**

| **SQL Type** | **C Type** | **Defined In** |
| --- | --- | --- |
| abstime | AbsoluteTime | utils/nabstime.h |
| bigint (int8) | int64 | postgres.h |
| boolean | bool | postgres.h (maybe compiler built-in) |
| box | BOX\* | utils/geo\_decls.h |
| bytea | bytea\* | postgres.h |
| "char" | char | (compiler built-in) |
| character | BpChar\* | postgres.h |
| cid | CommandId | postgres.h |
| date | DateADT | utils/date.h |
| smallint (int2) | int16 | postgres.h |
| int2vector | int2vector\* | postgres.h |
| integer (int4) | int32 | postgres.h |
| real (float4) | float4\* | postgres.h |
| double precision (float8) | float8\* | postgres.h |
| interval | Interval\* | datatype/timestamp.h |
| lseg | LSEG\* | utils/geo\_decls.h |
| name | Name | postgres.h |
| oid | Oid | postgres.h |
| oidvector | oidvector\* | postgres.h |
| path | PATH\* | utils/geo\_decls.h |
| point | POINT\* | utils/geo\_decls.h |
| regproc | regproc | postgres.h |
| reltime | RelativeTime | utils/nabstime.h |
| text | text\* | postgres.h |
| tid | ItemPointer | storage/itemptr.h |
| time | TimeADT | utils/date.h |
| time with time zone | TimeTzADT | utils/date.h |
| timestamp | Timestamp\* | datatype/timestamp.h |
| tinterval | TimeInterval | utils/nabstime.h |
| varchar | VarChar\* | postgres.h |
| xid | TransactionId | postgres.h |

Now that we've gone over all of the possible structures for base types, we can show some examples of real functions.

### 37.9.3. Version 1 Calling Conventions

The version-1 calling convention relies on macros to suppress most of the complexity of passing arguments and results. The C declaration of a version-1 function is always:

Datum funcname(PG\_FUNCTION\_ARGS)

In addition, the macro call:

PG\_FUNCTION\_INFO\_V1(funcname);

must appear in the same source file. (Conventionally, it's written just before the function itself.) This macro call is not needed for internal-language functions, since PostgreSQL assumes that all internal functions use the version-1 convention. It is, however, required for dynamically-loaded functions.

In a version-1 function, each actual argument is fetched using a PG\_GETARG\_***xxx***() macro that corresponds to the argument's data type. In non-strict functions there needs to be a previous check about argument null-ness using PG\_ARGNULL\_***xxx***(). The result is returned using a PG\_RETURN\_***xxx***() macro for the return type. PG\_GETARG\_***xxx***() takes as its argument the number of the function argument to fetch, where the count starts at 0. PG\_RETURN\_***xxx***() takes as its argument the actual value to return.

Here are some examples using the version-1 calling convention:

#include "postgres.h"

#include <string.h>

#include "fmgr.h"

#include "utils/geo\_decls.h"

#ifdef PG\_MODULE\_MAGIC

PG\_MODULE\_MAGIC;

#endif

/\* by value \*/

PG\_FUNCTION\_INFO\_V1(add\_one);

Datum

add\_one(PG\_FUNCTION\_ARGS)

{

int32 arg = PG\_GETARG\_INT32(0);

PG\_RETURN\_INT32(arg + 1);

}

/\* by reference, fixed length \*/

PG\_FUNCTION\_INFO\_V1(add\_one\_float8);

Datum

add\_one\_float8(PG\_FUNCTION\_ARGS)

{

/\* The macros for FLOAT8 hide its pass-by-reference nature. \*/

float8 arg = PG\_GETARG\_FLOAT8(0);

PG\_RETURN\_FLOAT8(arg + 1.0);

}

PG\_FUNCTION\_INFO\_V1(makepoint);

Datum

makepoint(PG\_FUNCTION\_ARGS)

{

/\* Here, the pass-by-reference nature of Point is not hidden. \*/

Point \*pointx = PG\_GETARG\_POINT\_P(0);

Point \*pointy = PG\_GETARG\_POINT\_P(1);

Point \*new\_point = (Point \*) palloc(sizeof(Point));

new\_point->x = pointx->x;

new\_point->y = pointy->y;

PG\_RETURN\_POINT\_P(new\_point);

}

/\* by reference, variable length \*/

PG\_FUNCTION\_INFO\_V1(copytext);

Datum

copytext(PG\_FUNCTION\_ARGS)

{

text \*t = PG\_GETARG\_TEXT\_PP(0);

/\*

\* VARSIZE\_ANY\_EXHDR is the size of the struct in bytes, minus the

\* VARHDRSZ or VARHDRSZ\_SHORT of its header. Construct the copy with a

\* full-length header.

\*/

text \*new\_t = (text \*) palloc(VARSIZE\_ANY\_EXHDR(t) + VARHDRSZ);

SET\_VARSIZE(new\_t, VARSIZE\_ANY\_EXHDR(t) + VARHDRSZ);

/\*

\* VARDATA is a pointer to the data region of the new struct. The source

\* could be a short datum, so retrieve its data through VARDATA\_ANY.

\*/

memcpy((void \*) VARDATA(new\_t), /\* destination \*/

(void \*) VARDATA\_ANY(t), /\* source \*/

VARSIZE\_ANY\_EXHDR(t)); /\* how many bytes \*/

PG\_RETURN\_TEXT\_P(new\_t);

}

PG\_FUNCTION\_INFO\_V1(concat\_text);

Datum

concat\_text(PG\_FUNCTION\_ARGS)

{

text \*arg1 = PG\_GETARG\_TEXT\_PP(0);

text \*arg2 = PG\_GETARG\_TEXT\_PP(1);

int32 arg1\_size = VARSIZE\_ANY\_EXHDR(arg1);

int32 arg2\_size = VARSIZE\_ANY\_EXHDR(arg2);

int32 new\_text\_size = arg1\_size + arg2\_size + VARHDRSZ;

text \*new\_text = (text \*) palloc(new\_text\_size);

SET\_VARSIZE(new\_text, new\_text\_size);

memcpy(VARDATA(new\_text), VARDATA\_ANY(arg1), arg1\_size);

memcpy(VARDATA(new\_text) + arg1\_size, VARDATA\_ANY(arg2), arg2\_size);

PG\_RETURN\_TEXT\_P(new\_text);

}

Supposing that the above code has been prepared in file funcs.c and compiled into a shared object, we could define the functions to PostgreSQL with commands like this:

CREATE FUNCTION add\_one(integer) RETURNS integer

AS '***DIRECTORY***/funcs', 'add\_one'

LANGUAGE C STRICT;

-- note overloading of SQL function name "add\_one"

CREATE FUNCTION add\_one(double precision) RETURNS double precision

AS '***DIRECTORY***/funcs', 'add\_one\_float8'

LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point

AS '***DIRECTORY***/funcs', 'makepoint'

LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text

AS '***DIRECTORY***/funcs', 'copytext'

LANGUAGE C STRICT;

CREATE FUNCTION concat\_text(text, text) RETURNS text

AS '***DIRECTORY***/funcs', 'concat\_text'

LANGUAGE C STRICT;

Here, ***DIRECTORY*** stands for the directory of the shared library file (for instance the PostgreSQL tutorial directory, which contains the code for the examples used in this section). (Better style would be to use just 'funcs' in the AS clause, after having added ***DIRECTORY*** to the search path. In any case, we can omit the system-specific extension for a shared library, commonly .so.)

Notice that we have specified the functions as “strict”, meaning that the system should automatically assume a null result if any input value is null. By doing this, we avoid having to check for null inputs in the function code. Without this, we'd have to check for null values explicitly, using PG\_ARGISNULL().

At first glance, the version-1 coding conventions might appear to be just pointless obscurantism, over using plain C calling conventions. They do however allow to deal with NULLable arguments/return values, and “toasted” (compressed or out-of-line) values.

The macro PG\_ARGISNULL(***n***) allows a function to test whether each input is null. (Of course, doing this is only necessary in functions not declared “strict”.) As with the PG\_GETARG\_***xxx***() macros, the input arguments are counted beginning at zero. Note that one should refrain from executing PG\_GETARG\_***xxx***() until one has verified that the argument isn't null. To return a null result, execute PG\_RETURN\_NULL(); this works in both strict and nonstrict functions.

Other options provided by the version-1 interface are two variants of the PG\_GETARG\_***xxx***() macros. The first of these, PG\_GETARG\_***xxx***\_COPY(), guarantees to return a copy of the specified argument that is safe for writing into. (The normal macros will sometimes return a pointer to a value that is physically stored in a table, which must not be written to. Using the PG\_GETARG\_***xxx***\_COPY() macros guarantees a writable result.) The second variant consists of the PG\_GETARG\_***xxx***\_SLICE() macros which take three arguments. The first is the number of the function argument (as above). The second and third are the offset and length of the segment to be returned. Offsets are counted from zero, and a negative length requests that the remainder of the value be returned. These macros provide more efficient access to parts of large values in the case where they have storage type “external”. (The storage type of a column can be specified using ALTER TABLE ***tablename*** ALTER COLUMN ***colname***SET STORAGE ***storagetype***. ***storagetype*** is one of plain, external, extended, or main.)

Finally, the version-1 function call conventions make it possible to return set results ([**Section 37.9.8**](https://www.postgresql.org/docs/10/xfunc-c.html#XFUNC-C-RETURN-SET)) and implement trigger functions ([**Chapter 38**](https://www.postgresql.org/docs/10/triggers.html)) and procedural-language call handlers ([**Chapter 55**](https://www.postgresql.org/docs/10/plhandler.html)). For more details see src/backend/utils/fmgr/README in the source distribution.

### 37.9.4. Writing Code

Before we turn to the more advanced topics, we should discuss some coding rules for PostgreSQL C-language functions. While it might be possible to load functions written in languages other than C into PostgreSQL, this is usually difficult (when it is possible at all) because other languages, such as C++, FORTRAN, or Pascal often do not follow the same calling convention as C. That is, other languages do not pass argument and return values between functions in the same way. For this reason, we will assume that your C-language functions are actually written in C.

The basic rules for writing and building C functions are as follows:

* Use pg\_config --includedir-server to find out where the PostgreSQL server header files are installed on your system (or the system that your users will be running on).
* Compiling and linking your code so that it can be dynamically loaded into PostgreSQL always requires special flags. See [**Section 37.9.5**](https://www.postgresql.org/docs/10/xfunc-c.html#DFUNC) for a detailed explanation of how to do it for your particular operating system.
* Remember to define a “magic block” for your shared library, as described in [**Section 37.9.1**](https://www.postgresql.org/docs/10/xfunc-c.html#XFUNC-C-DYNLOAD).
* When allocating memory, use the PostgreSQL functions palloc and pfree instead of the corresponding C library functions malloc and free. The memory allocated by palloc will be freed automatically at the end of each transaction, preventing memory leaks.
* Always zero the bytes of your structures using memset (or allocate them with palloc0 in the first place). Even if you assign to each field of your structure, there might be alignment padding (holes in the structure) that contain garbage values. Without this, it's difficult to support hash indexes or hash joins, as you must pick out only the significant bits of your data structure to compute a hash. The planner also sometimes relies on comparing constants via bitwise equality, so you can get undesirable planning results if logically-equivalent values aren't bitwise equal.
* Most of the internal PostgreSQL types are declared in postgres.h, while the function manager interfaces (PG\_FUNCTION\_ARGS, etc.) are in fmgr.h, so you will need to include at least these two files. For portability reasons it's best to include postgres.h first, before any other system or user header files. Including postgres.h will also include elog.h and palloc.h for you.
* Symbol names defined within object files must not conflict with each other or with symbols defined in the PostgreSQL server executable. You will have to rename your functions or variables if you get error messages to this effect.

### 37.9.5. Compiling and Linking Dynamically-loaded Functions

Before you are able to use your PostgreSQL extension functions written in C, they must be compiled and linked in a special way to produce a file that can be dynamically loaded by the server. To be precise, a shared library needs to be created.

For information beyond what is contained in this section you should read the documentation of your operating system, in particular the manual pages for the C compiler, cc, and the link editor, ld. In addition, the PostgreSQL source code contains several working examples in the contrib directory. If you rely on these examples you will make your modules dependent on the availability of the PostgreSQL source code, however.

Creating shared libraries is generally analogous to linking executables: first the source files are compiled into object files, then the object files are linked together. The object files need to be created as position-independent code (PIC), which conceptually means that they can be placed at an arbitrary location in memory when they are loaded by the executable. (Object files intended for executables are usually not compiled that way.) The command to link a shared library contains special flags to distinguish it from linking an executable (at least in theory — on some systems the practice is much uglier).

In the following examples we assume that your source code is in a file foo.c and we will create a shared library foo.so. The intermediate object file will be called foo.o unless otherwise noted. A shared library can contain more than one object file, but we only use one here.

FreeBSD

The compiler flag to create PIC is -fPIC. To create shared libraries the compiler flag is -shared.

gcc -fPIC -c foo.c

gcc -shared -o foo.so foo.o

This is applicable as of version 3.0 of FreeBSD.

HP-UX

The compiler flag of the system compiler to create PIC is +z. When using GCC it's -fPIC. The linker flag for shared libraries is -b. So:

cc +z -c foo.c

or:

gcc -fPIC -c foo.c

and then:

ld -b -o foo.sl foo.o

HP-UX uses the extension .sl for shared libraries, unlike most other systems.

Linux

The compiler flag to create PIC is -fPIC. The compiler flag to create a shared library is -shared. A complete example looks like this:

cc -fPIC -c foo.c

cc -shared -o foo.so foo.o

macOS

Here is an example. It assumes the developer tools are installed.

cc -c foo.c

cc -bundle -flat\_namespace -undefined suppress -o foo.so foo.o

NetBSD

The compiler flag to create PIC is -fPIC. For ELF systems, the compiler with the flag -shared is used to link shared libraries. On the older non-ELF systems, ld -Bshareable is used.

gcc -fPIC -c foo.c

gcc -shared -o foo.so foo.o

OpenBSD

The compiler flag to create PIC is -fPIC. ld -Bshareable is used to link shared libraries.

gcc -fPIC -c foo.c

ld -Bshareable -o foo.so foo.o

Solaris

The compiler flag to create PIC is -KPIC with the Sun compiler and -fPIC with GCC. To link shared libraries, the compiler option is -G with either compiler or alternatively -shared with GCC.

cc -KPIC -c foo.c

cc -G -o foo.so foo.o

or

gcc -fPIC -c foo.c

gcc -G -o foo.so foo.o

### Tip

If this is too complicated for you, you should consider using [**GNU Libtool**](http://www.gnu.org/software/libtool/), which hides the platform differences behind a uniform interface.

The resulting shared library file can then be loaded into PostgreSQL. When specifying the file name to the CREATE FUNCTION command, one must give it the name of the shared library file, not the intermediate object file. Note that the system's standard shared-library extension (usually .so or .sl) can be omitted from the CREATE FUNCTION command, and normally should be omitted for best portability.

Refer back to [**Section 37.9.1**](https://www.postgresql.org/docs/10/xfunc-c.html#XFUNC-C-DYNLOAD) about where the server expects to find the shared library files.

### 37.9.6. Composite-type Arguments

Composite types do not have a fixed layout like C structures. Instances of a composite type can contain null fields. In addition, composite types that are part of an inheritance hierarchy can have different fields than other members of the same inheritance hierarchy. Therefore, PostgreSQL provides a function interface for accessing fields of composite types from C.

Suppose we want to write a function to answer the query:

SELECT name, c\_overpaid(emp, 1500) AS overpaid

FROM emp

WHERE name = 'Bill' OR name = 'Sam';

Using the version-1 calling conventions, we can define c\_overpaid as:

#include "postgres.h"

#include "executor/executor.h" /\* for GetAttributeByName() \*/

#ifdef PG\_MODULE\_MAGIC

PG\_MODULE\_MAGIC;

#endif

PG\_FUNCTION\_INFO\_V1(c\_overpaid);

Datum

c\_overpaid(PG\_FUNCTION\_ARGS)

{

HeapTupleHeader t = PG\_GETARG\_HEAPTUPLEHEADER(0);

int32 limit = PG\_GETARG\_INT32(1);

bool isnull;

Datum salary;

salary = GetAttributeByName(t, "salary", &isnull);

if (isnull)

PG\_RETURN\_BOOL(false);

/\* Alternatively, we might prefer to do PG\_RETURN\_NULL() for null salary. \*/

PG\_RETURN\_BOOL(DatumGetInt32(salary) > limit);

}

GetAttributeByName is the PostgreSQL system function that returns attributes out of the specified row. It has three arguments: the argument of type HeapTupleHeader passed into the function, the name of the desired attribute, and a return parameter that tells whether the attribute is null. GetAttributeByName returns a Datum value that you can convert to the proper data type by using the appropriate DatumGet***XXX***() macro. Note that the return value is meaningless if the null flag is set; always check the null flag before trying to do anything with the result.

There is also GetAttributeByNum, which selects the target attribute by column number instead of name.

The following command declares the function c\_overpaid in SQL:

CREATE FUNCTION c\_overpaid(emp, integer) RETURNS boolean

AS '***DIRECTORY***/funcs', 'c\_overpaid'

LANGUAGE C STRICT;

Notice we have used STRICT so that we did not have to check whether the input arguments were NULL.

### 37.9.7. Returning Rows (Composite Types)

To return a row or composite-type value from a C-language function, you can use a special API that provides macros and functions to hide most of the complexity of building composite data types. To use this API, the source file must include:

#include "funcapi.h"

There are two ways you can build a composite data value (henceforth a “tuple”): you can build it from an array of Datum values, or from an array of C strings that can be passed to the input conversion functions of the tuple's column data types. In either case, you first need to obtain or construct a TupleDesc descriptor for the tuple structure. When working with Datums, you pass the TupleDesc to BlessTupleDesc, and then call heap\_form\_tuple for each row. When working with C strings, you pass the TupleDesc to TupleDescGetAttInMetadata, and then call BuildTupleFromCStrings for each row. In the case of a function returning a set of tuples, the setup steps can all be done once during the first call of the function.

Several helper functions are available for setting up the needed TupleDesc. The recommended way to do this in most functions returning composite values is to call:

TypeFuncClass get\_call\_result\_type(FunctionCallInfo fcinfo,

Oid \*resultTypeId,

TupleDesc \*resultTupleDesc)

passing the same fcinfo struct passed to the calling function itself. (This of course requires that you use the version-1 calling conventions.) resultTypeId can be specified as NULL or as the address of a local variable to receive the function's result type OID. resultTupleDesc should be the address of a local TupleDesc variable. Check that the result is TYPEFUNC\_COMPOSITE; if so, resultTupleDesc has been filled with the needed TupleDesc. (If it is not, you can report an error along the lines of “function returning record called in context that cannot accept type record”.)

### Tip

get\_call\_result\_type can resolve the actual type of a polymorphic function result; so it is useful in functions that return scalar polymorphic results, not only functions that return composites. The resultTypeId output is primarily useful for functions returning polymorphic scalars.

### Note

get\_call\_result\_type has a sibling get\_expr\_result\_type, which can be used to resolve the expected output type for a function call represented by an expression tree. This can be used when trying to determine the result type from outside the function itself. There is also get\_func\_result\_type, which can be used when only the function's OID is available. However these functions are not able to deal with functions declared to return record, and get\_func\_result\_type cannot resolve polymorphic types, so you should preferentially use get\_call\_result\_type.

Older, now-deprecated functions for obtaining TupleDescs are:

TupleDesc RelationNameGetTupleDesc(const char \*relname)

to get a TupleDesc for the row type of a named relation, and:

TupleDesc TypeGetTupleDesc(Oid typeoid, List \*colaliases)

to get a TupleDesc based on a type OID. This can be used to get a TupleDesc for a base or composite type. It will not work for a function that returns record, however, and it cannot resolve polymorphic types.

Once you have a TupleDesc, call:

TupleDesc BlessTupleDesc(TupleDesc tupdesc)

if you plan to work with Datums, or:

AttInMetadata \*TupleDescGetAttInMetadata(TupleDesc tupdesc)

if you plan to work with C strings. If you are writing a function returning set, you can save the results of these functions in the FuncCallContext structure — use the tuple\_desc or attinmeta field respectively.

When working with Datums, use:

HeapTuple heap\_form\_tuple(TupleDesc tupdesc, Datum \*values, bool \*isnull)

to build a HeapTuple given user data in Datum form.

When working with C strings, use:

HeapTuple BuildTupleFromCStrings(AttInMetadata \*attinmeta, char \*\*values)

to build a HeapTuple given user data in C string form. *values* is an array of C strings, one for each attribute of the return row. Each C string should be in the form expected by the input function of the attribute data type. In order to return a null value for one of the attributes, the corresponding pointer in the *values* array should be set to NULL. This function will need to be called again for each row you return.

Once you have built a tuple to return from your function, it must be converted into a Datum. Use:

HeapTupleGetDatum(HeapTuple tuple)

to convert a HeapTuple into a valid Datum. This Datum can be returned directly if you intend to return just a single row, or it can be used as the current return value in a set-returning function.

An example appears in the next section.

### 37.9.8. Returning Sets

There is also a special API that provides support for returning sets (multiple rows) from a C-language function. A set-returning function must follow the version-1 calling conventions. Also, source files must include funcapi.h, as above.

A set-returning function (SRF) is called once for each item it returns. The SRF must therefore save enough state to remember what it was doing and return the next item on each call. The structure FuncCallContext is provided to help control this process. Within a function, fcinfo->flinfo->fn\_extra is used to hold a pointer to FuncCallContext across calls.

typedef struct FuncCallContext

{

/\*

\* Number of times we've been called before

\*

\* call\_cntr is initialized to 0 for you by SRF\_FIRSTCALL\_INIT(), and

\* incremented for you every time SRF\_RETURN\_NEXT() is called.

\*/

uint64 call\_cntr;

/\*

\* OPTIONAL maximum number of calls

\*

\* max\_calls is here for convenience only and setting it is optional.

\* If not set, you must provide alternative means to know when the

\* function is done.

\*/

uint64 max\_calls;

/\*

\* OPTIONAL pointer to result slot

\*

\* This is obsolete and only present for backward compatibility, viz,

\* user-defined SRFs that use the deprecated TupleDescGetSlot().

\*/

TupleTableSlot \*slot;

/\*

\* OPTIONAL pointer to miscellaneous user-provided context information

\*

\* user\_fctx is for use as a pointer to your own data to retain

\* arbitrary context information between calls of your function.

\*/

void \*user\_fctx;

/\*

\* OPTIONAL pointer to struct containing attribute type input metadata

\*

\* attinmeta is for use when returning tuples (i.e., composite data types)

\* and is not used when returning base data types. It is only needed

\* if you intend to use BuildTupleFromCStrings() to create the return

\* tuple.

\*/

AttInMetadata \*attinmeta;

/\*

\* memory context used for structures that must live for multiple calls

\*

\* multi\_call\_memory\_ctx is set by SRF\_FIRSTCALL\_INIT() for you, and used

\* by SRF\_RETURN\_DONE() for cleanup. It is the most appropriate memory

\* context for any memory that is to be reused across multiple calls

\* of the SRF.

\*/

MemoryContext multi\_call\_memory\_ctx;

/\*

\* OPTIONAL pointer to struct containing tuple description

\*

\* tuple\_desc is for use when returning tuples (i.e., composite data types)

\* and is only needed if you are going to build the tuples with

\* heap\_form\_tuple() rather than with BuildTupleFromCStrings(). Note that

\* the TupleDesc pointer stored here should usually have been run through

\* BlessTupleDesc() first.

\*/

TupleDesc tuple\_desc;

} FuncCallContext;

An SRF uses several functions and macros that automatically manipulate the FuncCallContext structure (and expect to find it via fn\_extra). Use:

SRF\_IS\_FIRSTCALL()

to determine if your function is being called for the first or a subsequent time. On the first call (only) use:

SRF\_FIRSTCALL\_INIT()

to initialize the FuncCallContext. On every function call, including the first, use:

SRF\_PERCALL\_SETUP()

to properly set up for using the FuncCallContext and clearing any previously returned data left over from the previous pass.

If your function has data to return, use:

SRF\_RETURN\_NEXT(funcctx, result)

to return it to the caller. (result must be of type Datum, either a single value or a tuple prepared as described above.) Finally, when your function is finished returning data, use:

SRF\_RETURN\_DONE(funcctx)

to clean up and end the SRF.

The memory context that is current when the SRF is called is a transient context that will be cleared between calls. This means that you do not need to call pfree on everything you allocated using palloc; it will go away anyway. However, if you want to allocate any data structures to live across calls, you need to put them somewhere else. The memory context referenced by multi\_call\_memory\_ctx is a suitable location for any data that needs to survive until the SRF is finished running. In most cases, this means that you should switch into multi\_call\_memory\_ctx while doing the first-call setup.

### Warning

While the actual arguments to the function remain unchanged between calls, if you detoast the argument values (which is normally done transparently by the PG\_GETARG\_***xxx***macro) in the transient context then the detoasted copies will be freed on each cycle. Accordingly, if you keep references to such values in your user\_fctx, you must either copy them into the multi\_call\_memory\_ctx after detoasting, or ensure that you detoast the values only in that context.

A complete pseudo-code example looks like the following:

Datum

my\_set\_returning\_function(PG\_FUNCTION\_ARGS)

{

FuncCallContext \*funcctx;

Datum result;

***further declarations as needed***

if (SRF\_IS\_FIRSTCALL())

{

MemoryContext oldcontext;

funcctx = SRF\_FIRSTCALL\_INIT();

oldcontext = MemoryContextSwitchTo(funcctx->multi\_call\_memory\_ctx);

/\* One-time setup code appears here: \*/

***user code***

***if returning composite***

***build TupleDesc, and perhaps AttInMetadata***

***endif returning composite***

***user code***

MemoryContextSwitchTo(oldcontext);

}

/\* Each-time setup code appears here: \*/

***user code***

funcctx = SRF\_PERCALL\_SETUP();

***user code***

/\* this is just one way we might test whether we are done: \*/

if (funcctx->call\_cntr < funcctx->max\_calls)

{

/\* Here we want to return another item: \*/

***user code***

***obtain result Datum***

SRF\_RETURN\_NEXT(funcctx, result);

}

else

{

/\* Here we are done returning items and just need to clean up: \*/

***user code***

SRF\_RETURN\_DONE(funcctx);

}

}

A complete example of a simple SRF returning a composite type looks like:

PG\_FUNCTION\_INFO\_V1(retcomposite);

Datum

retcomposite(PG\_FUNCTION\_ARGS)

{

FuncCallContext \*funcctx;

int call\_cntr;

int max\_calls;

TupleDesc tupdesc;

AttInMetadata \*attinmeta;

/\* stuff done only on the first call of the function \*/

if (SRF\_IS\_FIRSTCALL())

{

MemoryContext oldcontext;

/\* create a function context for cross-call persistence \*/

funcctx = SRF\_FIRSTCALL\_INIT();

/\* switch to memory context appropriate for multiple function calls \*/

oldcontext = MemoryContextSwitchTo(funcctx->multi\_call\_memory\_ctx);

/\* total number of tuples to be returned \*/

funcctx->max\_calls = PG\_GETARG\_UINT32(0);

/\* Build a tuple descriptor for our result type \*/

if (get\_call\_result\_type(fcinfo, NULL, &tupdesc) != TYPEFUNC\_COMPOSITE)

ereport(ERROR,

(errcode(ERRCODE\_FEATURE\_NOT\_SUPPORTED),

errmsg("function returning record called in context "

"that cannot accept type record")));

/\*

\* generate attribute metadata needed later to produce tuples from raw

\* C strings

\*/

attinmeta = TupleDescGetAttInMetadata(tupdesc);

funcctx->attinmeta = attinmeta;

MemoryContextSwitchTo(oldcontext);

}

/\* stuff done on every call of the function \*/

funcctx = SRF\_PERCALL\_SETUP();

call\_cntr = funcctx->call\_cntr;

max\_calls = funcctx->max\_calls;

attinmeta = funcctx->attinmeta;

if (call\_cntr < max\_calls) /\* do when there is more left to send \*/

{

char \*\*values;

HeapTuple tuple;

Datum result;

/\*

\* Prepare a values array for building the returned tuple.

\* This should be an array of C strings which will

\* be processed later by the type input functions.

\*/

values = (char \*\*) palloc(3 \* sizeof(char \*));

values[0] = (char \*) palloc(16 \* sizeof(char));

values[1] = (char \*) palloc(16 \* sizeof(char));

values[2] = (char \*) palloc(16 \* sizeof(char));

snprintf(values[0], 16, "%d", 1 \* PG\_GETARG\_INT32(1));

snprintf(values[1], 16, "%d", 2 \* PG\_GETARG\_INT32(1));

snprintf(values[2], 16, "%d", 3 \* PG\_GETARG\_INT32(1));

/\* build a tuple \*/

tuple = BuildTupleFromCStrings(attinmeta, values);

/\* make the tuple into a datum \*/

result = HeapTupleGetDatum(tuple);

/\* clean up (this is not really necessary) \*/

pfree(values[0]);

pfree(values[1]);

pfree(values[2]);

pfree(values);

SRF\_RETURN\_NEXT(funcctx, result);

}

else /\* do when there is no more left \*/

{

SRF\_RETURN\_DONE(funcctx);

}

}

One way to declare this function in SQL is:

CREATE TYPE \_\_retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)

RETURNS SETOF \_\_retcomposite

AS '***filename***', 'retcomposite'

LANGUAGE C IMMUTABLE STRICT;

A different way is to use OUT parameters:

CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,

OUT f1 integer, OUT f2 integer, OUT f3 integer)

RETURNS SETOF record

AS '***filename***', 'retcomposite'

LANGUAGE C IMMUTABLE STRICT;

Notice that in this method the output type of the function is formally an anonymous record type.

The directory [contrib/tablefunc](https://www.postgresql.org/docs/10/tablefunc.html) module in the source distribution contains more examples of set-returning functions.

### 37.9.9. Polymorphic Arguments and Return Types

C-language functions can be declared to accept and return the polymorphic types anyelement, anyarray, anynonarray, anyenum, and anyrange. See [**Section 37.2.5**](https://www.postgresql.org/docs/10/extend-type-system.html#EXTEND-TYPES-POLYMORPHIC) for a more detailed explanation of polymorphic functions. When function arguments or return types are defined as polymorphic types, the function author cannot know in advance what data type it will be called with, or need to return. There are two routines provided in fmgr.h to allow a version-1 C function to discover the actual data types of its arguments and the type it is expected to return. The routines are called get\_fn\_expr\_rettype(FmgrInfo \*flinfo) and get\_fn\_expr\_argtype(FmgrInfo \*flinfo, int argnum). They return the result or argument type OID, or InvalidOid if the information is not available. The structure flinfo is normally accessed as fcinfo->flinfo. The parameter argnum is zero based. get\_call\_result\_type can also be used as an alternative to get\_fn\_expr\_rettype. There is also get\_fn\_expr\_variadic, which can be used to find out whether variadic arguments have been merged into an array. This is primarily useful for VARIADIC "any" functions, since such merging will always have occurred for variadic functions taking ordinary array types.

For example, suppose we want to write a function to accept a single element of any type, and return a one-dimensional array of that type:

PG\_FUNCTION\_INFO\_V1(make\_array);

Datum

make\_array(PG\_FUNCTION\_ARGS)

{

ArrayType \*result;

Oid element\_type = get\_fn\_expr\_argtype(fcinfo->flinfo, 0);

Datum element;

bool isnull;

int16 typlen;

bool typbyval;

char typalign;

int ndims;

int dims[MAXDIM];

int lbs[MAXDIM];

if (!OidIsValid(element\_type))

elog(ERROR, "could not determine data type of input");

/\* get the provided element, being careful in case it's NULL \*/

isnull = PG\_ARGISNULL(0);

if (isnull)

element = (Datum) 0;

else

element = PG\_GETARG\_DATUM(0);

/\* we have one dimension \*/

ndims = 1;

/\* and one element \*/

dims[0] = 1;

/\* and lower bound is 1 \*/

lbs[0] = 1;

/\* get required info about the element type \*/

get\_typlenbyvalalign(element\_type, &typlen, &typbyval, &typalign);

/\* now build the array \*/

result = construct\_md\_array(&element, &isnull, ndims, dims, lbs,

element\_type, typlen, typbyval, typalign);

PG\_RETURN\_ARRAYTYPE\_P(result);

}

The following command declares the function make\_array in SQL:

CREATE FUNCTION make\_array(anyelement) RETURNS anyarray

AS '***DIRECTORY***/funcs', 'make\_array'

LANGUAGE C IMMUTABLE;

There is a variant of polymorphism that is only available to C-language functions: they can be declared to take parameters of type "any". (Note that this type name must be double-quoted, since it's also a SQL reserved word.) This works like anyelement except that it does not constrain different "any" arguments to be the same type, nor do they help determine the function's result type. A C-language function can also declare its final parameter to be VARIADIC "any". This will match one or more actual arguments of any type (not necessarily the same type). These arguments will not be gathered into an array as happens with normal variadic functions; they will just be passed to the function separately. The PG\_NARGS() macro and the methods described above must be used to determine the number of actual arguments and their types when using this feature. Also, users of such a function might wish to use the VARIADIC keyword in their function call, with the expectation that the function would treat the array elements as separate arguments. The function itself must implement that behavior if wanted, after using get\_fn\_expr\_variadic to detect that the actual argument was marked with VARIADIC.

### 37.9.10. Transform Functions

Some function calls can be simplified during planning based on properties specific to the function. For example, int4mul(n, 1) could be simplified to just n. To define such function-specific optimizations, write a transform function and place its OID in the protransform field of the primary function's pg\_proc entry. The transform function must have the SQL signature protransform(internal) RETURNS internal. The argument, actually FuncExpr \*, is a dummy node representing a call to the primary function. If the transform function's study of the expression tree proves that a simplified expression tree can substitute for all possible concrete calls represented thereby, build and return that simplified expression. Otherwise, return a NULL pointer (not a SQL null).

We make no guarantee that PostgreSQL will never call the primary function in cases that the transform function could simplify. Ensure rigorous equivalence between the simplified expression and an actual call to the primary function.

Currently, this facility is not exposed to users at the SQL level because of security concerns, so it is only practical to use for optimizing built-in functions.

### 37.9.11. Shared Memory and LWLocks

Add-ins can reserve LWLocks and an allocation of shared memory on server startup. The add-in's shared library must be preloaded by specifying it in [**shared\_preload\_libraries**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-SHARED-PRELOAD-LIBRARIES). Shared memory is reserved by calling:

void RequestAddinShmemSpace(int size)

from your \_PG\_init function.

LWLocks are reserved by calling:

void RequestNamedLWLockTranche(const char \*tranche\_name, int num\_lwlocks)

from \_PG\_init. This will ensure that an array of num\_lwlocks LWLocks is available under the name tranche\_name. Use GetNamedLWLockTranche to get a pointer to this array.

To avoid possible race-conditions, each backend should use the LWLock AddinShmemInitLock when connecting to and initializing its allocation of shared memory, as shown here:

static mystruct \*ptr = NULL;

if (!ptr)

{

bool found;

LWLockAcquire(AddinShmemInitLock, LW\_EXCLUSIVE);

ptr = ShmemInitStruct("my struct name", size, &found);

if (!found)

{

initialize contents of shmem area;

acquire any requested LWLocks using:

ptr->locks = GetNamedLWLockTranche("my tranche name");

}

LWLockRelease(AddinShmemInitLock);

}

### 37.9.12. Using C++ for Extensibility

Although the PostgreSQL backend is written in C, it is possible to write extensions in C++ if these guidelines are followed:

* All functions accessed by the backend must present a C interface to the backend; these C functions can then call C++ functions. For example, extern C linkage is required for backend-accessed functions. This is also necessary for any functions that are passed as pointers between the backend and C++ code.
* Free memory using the appropriate deallocation method. For example, most backend memory is allocated using palloc(), so use pfree() to free it. Using C++ delete in such cases will fail.
* Prevent exceptions from propagating into the C code (use a catch-all block at the top level of all extern C functions). This is necessary even if the C++ code does not explicitly throw any exceptions, because events like out-of-memory can still throw exceptions. Any exceptions must be caught and appropriate errors passed back to the C interface. If possible, compile C++ with -fno-exceptions to eliminate exceptions entirely; in such cases, you must check for failures in your C++ code, e.g. check for NULL returned by new().
* If calling backend functions from C++ code, be sure that the C++ call stack contains only plain old data structures (POD). This is necessary because backend errors generate a distant longjmp() that does not properly unroll a C++ call stack with non-POD objects.

In summary, it is best to place C++ code behind a wall of extern C functions that interface to the backend, and avoid exception, memory, and call stack leakage.

## 37.10. User-defined Aggregates

Aggregate functions in PostgreSQL are defined in terms of state values and state transition functions. That is, an aggregate operates using a state value that is updated as each successive input row is processed. To define a new aggregate function, one selects a data type for the state value, an initial value for the state, and a state transition function. The state transition function takes the previous state value and the aggregate's input value(s) for the current row, and returns a new state value. A final function can also be specified, in case the desired result of the aggregate is different from the data that needs to be kept in the running state value. The final function takes the ending state value and returns whatever is wanted as the aggregate result. In principle, the transition and final functions are just ordinary functions that could also be used outside the context of the aggregate. (In practice, it's often helpful for performance reasons to create specialized transition functions that can only work when called as part of an aggregate.)

Thus, in addition to the argument and result data types seen by a user of the aggregate, there is an internal state-value data type that might be different from both the argument and result types.

If we define an aggregate that does not use a final function, we have an aggregate that computes a running function of the column values from each row. sum is an example of this kind of aggregate. sum starts at zero and always adds the current row's value to its running total. For example, if we want to make a sum aggregate to work on a data type for complex numbers, we only need the addition function for that data type. The aggregate definition would be:

CREATE AGGREGATE sum (complex)

(

sfunc = complex\_add,

stype = complex,

initcond = '(0,0)'

);

which we might use like this:

SELECT sum(a) FROM test\_complex;

sum

-----------

(34,53.9)

(Notice that we are relying on function overloading: there is more than one aggregate named sum, but PostgreSQL can figure out which kind of sum applies to a column of type complex.)

The above definition of sum will return zero (the initial state value) if there are no nonnull input values. Perhaps we want to return null in that case instead — the SQL standard expects sum to behave that way. We can do this simply by omitting the initcond phrase, so that the initial state value is null. Ordinarily this would mean that the sfunc would need to check for a null state-value input. But for sum and some other simple aggregates like max and min, it is sufficient to insert the first nonnull input value into the state variable and then start applying the transition function at the second nonnull input value. PostgreSQL will do that automatically if the initial state value is null and the transition function is marked “strict” (i.e., not to be called for null inputs).

Another bit of default behavior for a “strict” transition function is that the previous state value is retained unchanged whenever a null input value is encountered. Thus, null values are ignored. If you need some other behavior for null inputs, do not declare your transition function as strict; instead code it to test for null inputs and do whatever is needed.

avg (average) is a more complex example of an aggregate. It requires two pieces of running state: the sum of the inputs and the count of the number of inputs. The final result is obtained by dividing these quantities. Average is typically implemented by using an array as the state value. For example, the built-in implementation of avg(float8) looks like:

CREATE AGGREGATE avg (float8)

(

sfunc = float8\_accum,

stype = float8[],

finalfunc = float8\_avg,

initcond = '{0,0,0}'

);

### Note

float8\_accum requires a three-element array, not just two elements, because it accumulates the sum of squares as well as the sum and count of the inputs. This is so that it can be used for some other aggregates as well as avg.

Aggregate function calls in SQL allow DISTINCT and ORDER BY options that control which rows are fed to the aggregate's transition function and in what order. These options are implemented behind the scenes and are not the concern of the aggregate's support functions.

For further details see the [**CREATE AGGREGATE**](https://www.postgresql.org/docs/10/sql-createaggregate.html) command.

### 37.10.1. Moving-Aggregate Mode

Aggregate functions can optionally support moving-aggregate mode, which allows substantially faster execution of aggregate functions within windows with moving frame starting points. (See [**Section 3.5**](https://www.postgresql.org/docs/10/tutorial-window.html) and [**Section 4.2.8**](https://www.postgresql.org/docs/10/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS) for information about use of aggregate functions as window functions.) The basic idea is that in addition to a normal “forward” transition function, the aggregate provides an inverse transition function, which allows rows to be removed from the aggregate's running state value when they exit the window frame. For example a sum aggregate, which uses addition as the forward transition function, would use subtraction as the inverse transition function. Without an inverse transition function, the window function mechanism must recalculate the aggregate from scratch each time the frame starting point moves, resulting in run time proportional to the number of input rows times the average frame length. With an inverse transition function, the run time is only proportional to the number of input rows.

The inverse transition function is passed the current state value and the aggregate input value(s) for the earliest row included in the current state. It must reconstruct what the state value would have been if the given input row had never been aggregated, but only the rows following it. This sometimes requires that the forward transition function keep more state than is needed for plain aggregation mode. Therefore, the moving-aggregate mode uses a completely separate implementation from the plain mode: it has its own state data type, its own forward transition function, and its own final function if needed. These can be the same as the plain mode's data type and functions, if there is no need for extra state.

As an example, we could extend the sum aggregate given above to support moving-aggregate mode like this:

CREATE AGGREGATE sum (complex)

(

sfunc = complex\_add,

stype = complex,

initcond = '(0,0)',

msfunc = complex\_add,

minvfunc = complex\_sub,

mstype = complex,

minitcond = '(0,0)'

);

The parameters whose names begin with m define the moving-aggregate implementation. Except for the inverse transition function minvfunc, they correspond to the plain-aggregate parameters without m.

The forward transition function for moving-aggregate mode is not allowed to return null as the new state value. If the inverse transition function returns null, this is taken as an indication that the inverse function cannot reverse the state calculation for this particular input, and so the aggregate calculation will be redone from scratch for the current frame starting position. This convention allows moving-aggregate mode to be used in situations where there are some infrequent cases that are impractical to reverse out of the running state value. The inverse transition function can “punt”on these cases, and yet still come out ahead so long as it can work for most cases. As an example, an aggregate working with floating-point numbers might choose to punt when a NaN (not a number) input has to be removed from the running state value.

When writing moving-aggregate support functions, it is important to be sure that the inverse transition function can reconstruct the correct state value exactly. Otherwise there might be user-visible differences in results depending on whether the moving-aggregate mode is used. An example of an aggregate for which adding an inverse transition function seems easy at first, yet where this requirement cannot be met is sum over float4 or float8 inputs. A naive declaration of sum(float8) could be

CREATE AGGREGATE unsafe\_sum (float8)

(

stype = float8,

sfunc = float8pl,

mstype = float8,

msfunc = float8pl,

minvfunc = float8mi

);

This aggregate, however, can give wildly different results than it would have without the inverse transition function. For example, consider

SELECT

unsafe\_sum(x) OVER (ORDER BY n ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING)

FROM (VALUES (1, 1.0e20::float8),

(2, 1.0::float8)) AS v (n,x);

This query returns 0 as its second result, rather than the expected answer of 1. The cause is the limited precision of floating-point values: adding 1 to 1e20 results in 1e20 again, and so subtracting 1e20from that yields 0, not 1. Note that this is a limitation of floating-point arithmetic in general, not a limitation of PostgreSQL.

### 37.10.2. Polymorphic and Variadic Aggregates

Aggregate functions can use polymorphic state transition functions or final functions, so that the same functions can be used to implement multiple aggregates. See [**Section 37.2.5**](https://www.postgresql.org/docs/10/extend-type-system.html#EXTEND-TYPES-POLYMORPHIC) for an explanation of polymorphic functions. Going a step further, the aggregate function itself can be specified with polymorphic input type(s) and state type, allowing a single aggregate definition to serve for multiple input data types. Here is an example of a polymorphic aggregate:

CREATE AGGREGATE array\_accum (anyelement)

(

sfunc = array\_append,

stype = anyarray,

initcond = '{}'

);

Here, the actual state type for any given aggregate call is the array type having the actual input type as elements. The behavior of the aggregate is to concatenate all the inputs into an array of that type. (Note: the built-in aggregate array\_agg provides similar functionality, with better performance than this definition would have.)

Here's the output using two different actual data types as arguments:

SELECT attrelid::regclass, array\_accum(attname)

FROM pg\_attribute

WHERE attnum > 0 AND attrelid = 'pg\_tablespace'::regclass

GROUP BY attrelid;

attrelid | array\_accum

---------------+---------------------------------------

pg\_tablespace | {spcname,spcowner,spcacl,spcoptions}

(1 row)

SELECT attrelid::regclass, array\_accum(atttypid::regtype)

FROM pg\_attribute

WHERE attnum > 0 AND attrelid = 'pg\_tablespace'::regclass

GROUP BY attrelid;

attrelid | array\_accum

---------------+---------------------------

pg\_tablespace | {name,oid,aclitem[],text[]}

(1 row)

Ordinarily, an aggregate function with a polymorphic result type has a polymorphic state type, as in the above example. This is necessary because otherwise the final function cannot be declared sensibly: it would need to have a polymorphic result type but no polymorphic argument type, which CREATE FUNCTION will reject on the grounds that the result type cannot be deduced from a call. But sometimes it is inconvenient to use a polymorphic state type. The most common case is where the aggregate support functions are to be written in C and the state type should be declared as internal because there is no SQL-level equivalent for it. To address this case, it is possible to declare the final function as taking extra “dummy” arguments that match the input arguments of the aggregate. Such dummy arguments are always passed as null values since no specific value is available when the final function is called. Their only use is to allow a polymorphic final function's result type to be connected to the aggregate's input type(s). For example, the definition of the built-in aggregate array\_agg is equivalent to

CREATE FUNCTION array\_agg\_transfn(internal, anynonarray)

RETURNS internal ...;

CREATE FUNCTION array\_agg\_finalfn(internal, anynonarray)

RETURNS anyarray ...;

CREATE AGGREGATE array\_agg (anynonarray)

(

sfunc = array\_agg\_transfn,

stype = internal,

finalfunc = array\_agg\_finalfn,

finalfunc\_extra

);

Here, the finalfunc\_extra option specifies that the final function receives, in addition to the state value, extra dummy argument(s) corresponding to the aggregate's input argument(s). The extra anynonarray argument allows the declaration of array\_agg\_finalfn to be valid.

An aggregate function can be made to accept a varying number of arguments by declaring its last argument as a VARIADIC array, in much the same fashion as for regular functions; see [**Section 37.4.5**](https://www.postgresql.org/docs/10/xfunc-sql.html#XFUNC-SQL-VARIADIC-FUNCTIONS). The aggregate's transition function(s) must have the same array type as their last argument. The transition function(s) typically would also be marked VARIADIC, but this is not strictly required.

### Note

Variadic aggregates are easily misused in connection with the ORDER BY option (see [**Section 4.2.7**](https://www.postgresql.org/docs/10/sql-expressions.html#SYNTAX-AGGREGATES)), since the parser cannot tell whether the wrong number of actual arguments have been given in such a combination. Keep in mind that everything to the right of ORDER BY is a sort key, not an argument to the aggregate. For example, in

SELECT myaggregate(a ORDER BY a, b, c) FROM ...

the parser will see this as a single aggregate function argument and three sort keys. However, the user might have intended

SELECT myaggregate(a, b, c ORDER BY a) FROM ...

If myaggregate is variadic, both these calls could be perfectly valid.

For the same reason, it's wise to think twice before creating aggregate functions with the same names and different numbers of regular arguments.

### 37.10.3. Ordered-Set Aggregates

The aggregates we have been describing so far are “normal” aggregates. PostgreSQL also supports ordered-set aggregates, which differ from normal aggregates in two key ways. First, in addition to ordinary aggregated arguments that are evaluated once per input row, an ordered-set aggregate can have “direct” arguments that are evaluated only once per aggregation operation. Second, the syntax for the ordinary aggregated arguments specifies a sort ordering for them explicitly. An ordered-set aggregate is usually used to implement a computation that depends on a specific row ordering, for instance rank or percentile, so that the sort ordering is a required aspect of any call. For example, the built-in definition of percentile\_disc is equivalent to:

CREATE FUNCTION ordered\_set\_transition(internal, anyelement)

RETURNS internal ...;

CREATE FUNCTION percentile\_disc\_final(internal, float8, anyelement)

RETURNS anyelement ...;

CREATE AGGREGATE percentile\_disc (float8 ORDER BY anyelement)

(

sfunc = ordered\_set\_transition,

stype = internal,

finalfunc = percentile\_disc\_final,

finalfunc\_extra

);

This aggregate takes a float8 direct argument (the percentile fraction) and an aggregated input that can be of any sortable data type. It could be used to obtain a median household income like this:

SELECT percentile\_disc(0.5) WITHIN GROUP (ORDER BY income) FROM households;

percentile\_disc

-----------------

50489

Here, 0.5 is a direct argument; it would make no sense for the percentile fraction to be a value varying across rows.

Unlike the case for normal aggregates, the sorting of input rows for an ordered-set aggregate is not done behind the scenes, but is the responsibility of the aggregate's support functions. The typical implementation approach is to keep a reference to a “tuplesort” object in the aggregate's state value, feed the incoming rows into that object, and then complete the sorting and read out the data in the final function. This design allows the final function to perform special operations such as injecting additional “hypothetical” rows into the data to be sorted. While normal aggregates can often be implemented with support functions written in PL/pgSQL or another PL language, ordered-set aggregates generally have to be written in C, since their state values aren't definable as any SQL data type. (In the above example, notice that the state value is declared as type internal — this is typical.)

The state transition function for an ordered-set aggregate receives the current state value plus the aggregated input values for each row, and returns the updated state value. This is the same definition as for normal aggregates, but note that the direct arguments (if any) are not provided. The final function receives the last state value, the values of the direct arguments if any, and (if finalfunc\_extra is specified) null values corresponding to the aggregated input(s). As with normal aggregates, finalfunc\_extra is only really useful if the aggregate is polymorphic; then the extra dummy argument(s) are needed to connect the final function's result type to the aggregate's input type(s).

Currently, ordered-set aggregates cannot be used as window functions, and therefore there is no need for them to support moving-aggregate mode.

### 37.10.4. Partial Aggregation

Optionally, an aggregate function can support partial aggregation. The idea of partial aggregation is to run the aggregate's state transition function over different subsets of the input data independently, and then to combine the state values resulting from those subsets to produce the same state value that would have resulted from scanning all the input in a single operation. This mode can be used for parallel aggregation by having different worker processes scan different portions of a table. Each worker produces a partial state value, and at the end those state values are combined to produce a final state value. (In the future this mode might also be used for purposes such as combining aggregations over local and remote tables; but that is not implemented yet.)

To support partial aggregation, the aggregate definition must provide a combine function, which takes two values of the aggregate's state type (representing the results of aggregating over two subsets of the input rows) and produces a new value of the state type, representing what the state would have been after aggregating over the combination of those sets of rows. It is unspecified what the relative order of the input rows from the two sets would have been. This means that it's usually impossible to define a useful combine function for aggregates that are sensitive to input row order.

As simple examples, MAX and MIN aggregates can be made to support partial aggregation by specifying the combine function as the same greater-of-two or lesser-of-two comparison function that is used as their transition function. SUM aggregates just need an addition function as combine function. (Again, this is the same as their transition function, unless the state value is wider than the input data type.)

The combine function is treated much like a transition function that happens to take a value of the state type, not of the underlying input type, as its second argument. In particular, the rules for dealing with null values and strict functions are similar. Also, if the aggregate definition specifies a non-null initcond, keep in mind that that will be used not only as the initial state for each partial aggregation run, but also as the initial state for the combine function, which will be called to combine each partial result into that state.

If the aggregate's state type is declared as internal, it is the combine function's responsibility that its result is allocated in the correct memory context for aggregate state values. This means in particular that when the first input is NULL it's invalid to simply return the second input, as that value will be in the wrong context and will not have sufficient lifespan.

When the aggregate's state type is declared as internal, it is usually also appropriate for the aggregate definition to provide a serialization function and a deserialization function, which allow such a state value to be copied from one process to another. Without these functions, parallel aggregation cannot be performed, and future applications such as local/remote aggregation will probably not work either.

A serialization function must take a single argument of type internal and return a result of type bytea, which represents the state value packaged up into a flat blob of bytes. Conversely, a deserialization function reverses that conversion. It must take two arguments of types bytea and internal, and return a result of type internal. (The second argument is unused and is always zero, but it is required for type-safety reasons.) The result of the deserialization function should simply be allocated in the current memory context, as unlike the combine function's result, it is not long-lived.

Worth noting also is that for an aggregate to be executed in parallel, the aggregate itself must be marked PARALLEL SAFE. The parallel-safety markings on its support functions are not consulted.

### 37.10.5. Support Functions for Aggregates

A function written in C can detect that it is being called as an aggregate support function by calling AggCheckCallContext, for example:

if (AggCheckCallContext(fcinfo, NULL))

One reason for checking this is that when it is true for a transition function, the first input must be a temporary state value and can therefore safely be modified in-place rather than allocating a new copy. See int8inc() for an example. (This is the only case where it is safe for a function to modify a pass-by-reference input. In particular, final functions for normal aggregates must not modify their inputs in any case, because in some cases they will be re-executed on the same final state value.)

The second argument of AggCheckCallContext can be used to retrieve the memory context in which aggregate state values are being kept. This is useful for transition functions that wish to use “expanded” objects (see [**Section 37.11.1**](https://www.postgresql.org/docs/10/xtypes.html#XTYPES-TOAST)) as their state values. On first call, the transition function should return an expanded object whose memory context is a child of the aggregate state context, and then keep returning the same expanded object on subsequent calls. See array\_append() for an example. (array\_append() is not the transition function of any built-in aggregate, but it is written to behave efficiently when used as transition function of a custom aggregate.)

Another support routine available to aggregate functions written in C is AggGetAggref, which returns the Aggref parse node that defines the aggregate call. This is mainly useful for ordered-set aggregates, which can inspect the substructure of the Aggref node to find out what sort ordering they are supposed to implement. Examples can be found in orderedsetaggs.c in the PostgreSQLsource code.

## 37.11. User-defined Types

As described in [**Section 37.2**](https://www.postgresql.org/docs/10/extend-type-system.html), PostgreSQL can be extended to support new data types. This section describes how to define new base types, which are data types defined below the level of the SQLlanguage. Creating a new base type requires implementing functions to operate on the type in a low-level language, usually C.

The examples in this section can be found in complex.sql and complex.c in the src/tutorial directory of the source distribution. See the README file in that directory for instructions about running the examples.

A user-defined type must always have input and output functions. These functions determine how the type appears in strings (for input by the user and output to the user) and how the type is organized in memory. The input function takes a null-terminated character string as its argument and returns the internal (in memory) representation of the type. The output function takes the internal representation of the type as argument and returns a null-terminated character string. If we want to do anything more with the type than merely store it, we must provide additional functions to implement whatever operations we'd like to have for the type.

Suppose we want to define a type complex that represents complex numbers. A natural way to represent a complex number in memory would be the following C structure:

typedef struct Complex {

double x;

double y;

} Complex;

We will need to make this a pass-by-reference type, since it's too large to fit into a single Datum value.

As the external string representation of the type, we choose a string of the form (x,y).

The input and output functions are usually not hard to write, especially the output function. But when defining the external string representation of the type, remember that you must eventually write a complete and robust parser for that representation as your input function. For instance:

PG\_FUNCTION\_INFO\_V1(complex\_in);

Datum

complex\_in(PG\_FUNCTION\_ARGS)

{

char \*str = PG\_GETARG\_CSTRING(0);

double x,

y;

Complex \*result;

if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)

ereport(ERROR,

(errcode(ERRCODE\_INVALID\_TEXT\_REPRESENTATION),

errmsg("invalid input syntax for complex: \"%s\"",

str)));

result = (Complex \*) palloc(sizeof(Complex));

result->x = x;

result->y = y;

PG\_RETURN\_POINTER(result);

}

The output function can simply be:

PG\_FUNCTION\_INFO\_V1(complex\_out);

Datum

complex\_out(PG\_FUNCTION\_ARGS)

{

Complex \*complex = (Complex \*) PG\_GETARG\_POINTER(0);

char \*result;

result = psprintf("(%g,%g)", complex->x, complex->y);

PG\_RETURN\_CSTRING(result);

}

You should be careful to make the input and output functions inverses of each other. If you do not, you will have severe problems when you need to dump your data into a file and then read it back in. This is a particularly common problem when floating-point numbers are involved.

Optionally, a user-defined type can provide binary input and output routines. Binary I/O is normally faster but less portable than textual I/O. As with textual I/O, it is up to you to define exactly what the external binary representation is. Most of the built-in data types try to provide a machine-independent binary representation. For complex, we will piggy-back on the binary I/O converters for type float8:

PG\_FUNCTION\_INFO\_V1(complex\_recv);

Datum

complex\_recv(PG\_FUNCTION\_ARGS)

{

StringInfo buf = (StringInfo) PG\_GETARG\_POINTER(0);

Complex \*result;

result = (Complex \*) palloc(sizeof(Complex));

result->x = pq\_getmsgfloat8(buf);

result->y = pq\_getmsgfloat8(buf);

PG\_RETURN\_POINTER(result);

}

PG\_FUNCTION\_INFO\_V1(complex\_send);

Datum

complex\_send(PG\_FUNCTION\_ARGS)

{

Complex \*complex = (Complex \*) PG\_GETARG\_POINTER(0);

StringInfoData buf;

pq\_begintypsend(&buf);

pq\_sendfloat8(&buf, complex->x);

pq\_sendfloat8(&buf, complex->y);

PG\_RETURN\_BYTEA\_P(pq\_endtypsend(&buf));

}

Once we have written the I/O functions and compiled them into a shared library, we can define the complex type in SQL. First we declare it as a shell type:

CREATE TYPE complex;

This serves as a placeholder that allows us to reference the type while defining its I/O functions. Now we can define the I/O functions:

CREATE FUNCTION complex\_in(cstring)

RETURNS complex

AS '***filename***'

LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex\_out(complex)

RETURNS cstring

AS '***filename***'

LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex\_recv(internal)

RETURNS complex

AS '***filename***'

LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex\_send(complex)

RETURNS bytea

AS '***filename***'

LANGUAGE C IMMUTABLE STRICT;

Finally, we can provide the full definition of the data type:

CREATE TYPE complex (

internallength = 16,

input = complex\_in,

output = complex\_out,

receive = complex\_recv,

send = complex\_send,

alignment = double

);

When you define a new base type, PostgreSQL automatically provides support for arrays of that type. The array type typically has the same name as the base type with the underscore character (\_) prepended.

Once the data type exists, we can declare additional functions to provide useful operations on the data type. Operators can then be defined atop the functions, and if needed, operator classes can be created to support indexing of the data type. These additional layers are discussed in following sections.

If the internal representation of the data type is variable-length, the internal representation must follow the standard layout for variable-length data: the first four bytes must be a char[4] field which is never accessed directly (customarily named vl\_len\_). You must use the SET\_VARSIZE() macro to store the total size of the datum (including the length field itself) in this field and VARSIZE() to retrieve it. (These macros exist because the length field may be encoded depending on platform.)

For further details see the description of the [**CREATE TYPE**](https://www.postgresql.org/docs/10/sql-createtype.html) command.

### 37.11.1. TOAST Considerations

If the values of your data type vary in size (in internal form), it's usually desirable to make the data type TOAST-able (see [**Section 66.2**](https://www.postgresql.org/docs/10/storage-toast.html)). You should do this even if the values are always too small to be compressed or stored externally, because TOAST can save space on small data too, by reducing header overhead.

To support TOAST storage, the C functions operating on the data type must always be careful to unpack any toasted values they are handed by using PG\_DETOAST\_DATUM. (This detail is customarily hidden by defining type-specific GETARG\_DATATYPE\_P macros.) Then, when running the CREATE TYPE command, specify the internal length as variable and select some appropriate storage option other than plain.

If data alignment is unimportant (either just for a specific function or because the data type specifies byte alignment anyway) then it's possible to avoid some of the overhead of PG\_DETOAST\_DATUM. You can use PG\_DETOAST\_DATUM\_PACKED instead (customarily hidden by defining a GETARG\_DATATYPE\_PP macro) and using the macros VARSIZE\_ANY\_EXHDR and VARDATA\_ANY to access a potentially-packed datum. Again, the data returned by these macros is not aligned even if the data type definition specifies an alignment. If the alignment is important you must go through the regular PG\_DETOAST\_DATUMinterface.

### Note

Older code frequently declares vl\_len\_ as an int32 field instead of char[4]. This is OK as long as the struct definition has other fields that have at least int32 alignment. But it is dangerous to use such a struct definition when working with a potentially unaligned datum; the compiler may take it as license to assume the datum actually is aligned, leading to core dumps on architectures that are strict about alignment.

Another feature that's enabled by TOAST support is the possibility of having an expanded in-memory data representation that is more convenient to work with than the format that is stored on disk. The regular or “flat” varlena storage format is ultimately just a blob of bytes; it cannot for example contain pointers, since it may get copied to other locations in memory. For complex data types, the flat format may be quite expensive to work with, so PostgreSQL provides a way to “expand” the flat format into a representation that is more suited to computation, and then pass that format in-memory between functions of the data type.

To use expanded storage, a data type must define an expanded format that follows the rules given in src/include/utils/expandeddatum.h, and provide functions to “expand” a flat varlena value into expanded format and “flatten” the expanded format back to the regular varlena representation. Then ensure that all C functions for the data type can accept either representation, possibly by converting one into the other immediately upon receipt. This does not require fixing all existing functions for the data type at once, because the standard PG\_DETOAST\_DATUM macro is defined to convert expanded inputs into regular flat format. Therefore, existing functions that work with the flat varlena format will continue to work, though slightly inefficiently, with expanded inputs; they need not be converted until and unless better performance is important.

C functions that know how to work with an expanded representation typically fall into two categories: those that can only handle expanded format, and those that can handle either expanded or flat varlena inputs. The former are easier to write but may be less efficient overall, because converting a flat input to expanded form for use by a single function may cost more than is saved by operating on the expanded format. When only expanded format need be handled, conversion of flat inputs to expanded form can be hidden inside an argument-fetching macro, so that the function appears no more complex than one working with traditional varlena input. To handle both types of input, write an argument-fetching function that will detoast external, short-header, and compressed varlena inputs, but not expanded inputs. Such a function can be defined as returning a pointer to a union of the flat varlena format and the expanded format. Callers can use the VARATT\_IS\_EXPANDED\_HEADER()macro to determine which format they received.

The TOAST infrastructure not only allows regular varlena values to be distinguished from expanded values, but also distinguishes “read-write” and “read-only” pointers to expanded values. C functions that only need to examine an expanded value, or will only change it in safe and non-semantically-visible ways, need not care which type of pointer they receive. C functions that produce a modified version of an input value are allowed to modify an expanded input value in-place if they receive a read-write pointer, but must not modify the input if they receive a read-only pointer; in that case they have to copy the value first, producing a new value to modify. A C function that has constructed a new expanded value should always return a read-write pointer to it. Also, a C function that is modifying a read-write expanded value in-place should take care to leave the value in a sane state if it fails partway through.

For examples of working with expanded values, see the standard array infrastructure, particularly src/backend/utils/adt/array\_expanded.c.

## 37.12. User-defined Operators

Every operator is “syntactic sugar” for a call to an underlying function that does the real work; so you must first create the underlying function before you can create the operator. However, an operator is not merely syntactic sugar, because it carries additional information that helps the query planner optimize queries that use the operator. The next section will be devoted to explaining that additional information.

PostgreSQL supports left unary, right unary, and binary operators. Operators can be overloaded; that is, the same operator name can be used for different operators that have different numbers and types of operands. When a query is executed, the system determines the operator to call from the number and types of the provided operands.

Here is an example of creating an operator for adding two complex numbers. We assume we've already created the definition of type complex (see [**Section 37.11**](https://www.postgresql.org/docs/10/xtypes.html)). First we need a function that does the work, then we can define the operator:

CREATE FUNCTION complex\_add(complex, complex)

RETURNS complex

AS '***filename***', 'complex\_add'

LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (

leftarg = complex,

rightarg = complex,

procedure = complex\_add,

commutator = +

);

Now we could execute a query like this:

SELECT (a + b) AS c FROM test\_complex;

c

-----------------

(5.2,6.05)

(133.42,144.95)

We've shown how to create a binary operator here. To create unary operators, just omit one of leftarg (for left unary) or rightarg (for right unary). The procedure clause and the argument clauses are the only required items in CREATE OPERATOR. The commutator clause shown in the example is an optional hint to the query optimizer. Further details about commutator and other optimizer hints appear in the next section.

## 37.13. Operator Optimization Information

A PostgreSQL operator definition can include several optional clauses that tell the system useful things about how the operator behaves. These clauses should be provided whenever appropriate, because they can make for considerable speedups in execution of queries that use the operator. But if you provide them, you must be sure that they are right! Incorrect use of an optimization clause can result in slow queries, subtly wrong output, or other Bad Things. You can always leave out an optimization clause if you are not sure about it; the only consequence is that queries might run slower than they need to.

Additional optimization clauses might be added in future versions of PostgreSQL. The ones described here are all the ones that release 10.10 understands.

### 37.13.1. COMMUTATOR

The COMMUTATOR clause, if provided, names an operator that is the commutator of the operator being defined. We say that operator A is the commutator of operator B if (x A y) equals (y B x) for all possible input values x, y. Notice that B is also the commutator of A. For example, operators < and > for a particular data type are usually each others' commutators, and operator + is usually commutative with itself. But operator - is usually not commutative with anything.

The left operand type of a commutable operator is the same as the right operand type of its commutator, and vice versa. So the name of the commutator operator is all that PostgreSQL needs to be given to look up the commutator, and that's all that needs to be provided in the COMMUTATOR clause.

It's critical to provide commutator information for operators that will be used in indexes and join clauses, because this allows the query optimizer to “flip around” such a clause to the forms needed for different plan types. For example, consider a query with a WHERE clause like tab1.x = tab2.y, where tab1.x and tab2.y are of a user-defined type, and suppose that tab2.y is indexed. The optimizer cannot generate an index scan unless it can determine how to flip the clause around to tab2.y = tab1.x, because the index-scan machinery expects to see the indexed column on the left of the operator it is given. PostgreSQL will not simply assume that this is a valid transformation — the creator of the = operator must specify that it is valid, by marking the operator with commutator information.

When you are defining a self-commutative operator, you just do it. When you are defining a pair of commutative operators, things are a little trickier: how can the first one to be defined refer to the other one, which you haven't defined yet? There are two solutions to this problem:

* One way is to omit the COMMUTATOR clause in the first operator that you define, and then provide one in the second operator's definition. Since PostgreSQL knows that commutative operators come in pairs, when it sees the second definition it will automatically go back and fill in the missing COMMUTATOR clause in the first definition.
* The other, more straightforward way is just to include COMMUTATOR clauses in both definitions. When PostgreSQL processes the first definition and realizes that COMMUTATOR refers to a nonexistent operator, the system will make a dummy entry for that operator in the system catalog. This dummy entry will have valid data only for the operator name, left and right operand types, and result type, since that's all that PostgreSQL can deduce at this point. The first operator's catalog entry will link to this dummy entry. Later, when you define the second operator, the system updates the dummy entry with the additional information from the second definition. If you try to use the dummy operator before it's been filled in, you'll just get an error message.

### 37.13.2. NEGATOR

The NEGATOR clause, if provided, names an operator that is the negator of the operator being defined. We say that operator A is the negator of operator B if both return Boolean results and (x A y) equals NOT (x B y) for all possible inputs x, y. Notice that B is also the negator of A. For example, < and >= are a negator pair for most data types. An operator can never validly be its own negator.

Unlike commutators, a pair of unary operators could validly be marked as each other's negators; that would mean (A x) equals NOT (B x) for all x, or the equivalent for right unary operators.

An operator's negator must have the same left and/or right operand types as the operator to be defined, so just as with COMMUTATOR, only the operator name need be given in the NEGATOR clause.

Providing a negator is very helpful to the query optimizer since it allows expressions like NOT (x = y) to be simplified into x <> y. This comes up more often than you might think, because NOToperations can be inserted as a consequence of other rearrangements.

Pairs of negator operators can be defined using the same methods explained above for commutator pairs.

### 37.13.3. RESTRICT

The RESTRICT clause, if provided, names a restriction selectivity estimation function for the operator. (Note that this is a function name, not an operator name.) RESTRICT clauses only make sense for binary operators that return boolean. The idea behind a restriction selectivity estimator is to guess what fraction of the rows in a table will satisfy a WHERE-clause condition of the form:

column OP constant

for the current operator and a particular constant value. This assists the optimizer by giving it some idea of how many rows will be eliminated by WHERE clauses that have this form. (What happens if the constant is on the left, you might be wondering? Well, that's one of the things that COMMUTATOR is for...)

Writing new restriction selectivity estimation functions is far beyond the scope of this chapter, but fortunately you can usually just use one of the system's standard estimators for many of your own operators. These are the standard restriction estimators:

|  |
| --- |
| eqsel for = |
| neqsel for <> |
| scalarltsel for < or <= |
| scalargtsel for > or >= |

It might seem a little odd that these are the categories, but they make sense if you think about it. = will typically accept only a small fraction of the rows in a table; <> will typically reject only a small fraction. < will accept a fraction that depends on where the given constant falls in the range of values for that table column (which, it just so happens, is information collected by ANALYZE and made available to the selectivity estimator). <= will accept a slightly larger fraction than < for the same comparison constant, but they're close enough to not be worth distinguishing, especially since we're not likely to do better than a rough guess anyhow. Similar remarks apply to > and >=.

You can frequently get away with using either eqsel or neqsel for operators that have very high or very low selectivity, even if they aren't really equality or inequality. For example, the approximate-equality geometric operators use eqsel on the assumption that they'll usually only match a small fraction of the entries in a table.

You can use scalarltsel and scalargtsel for comparisons on data types that have some sensible means of being converted into numeric scalars for range comparisons. If possible, add the data type to those understood by the function convert\_to\_scalar() in src/backend/utils/adt/selfuncs.c. (Eventually, this function should be replaced by per-data-type functions identified through a column of the pg\_type system catalog; but that hasn't happened yet.) If you do not do this, things will still work, but the optimizer's estimates won't be as good as they could be.

There are additional selectivity estimation functions designed for geometric operators in src/backend/utils/adt/geo\_selfuncs.c: areasel, positionsel, and contsel. At this writing these are just stubs, but you might want to use them (or even better, improve them) anyway.

### 37.13.4. JOIN

The JOIN clause, if provided, names a join selectivity estimation function for the operator. (Note that this is a function name, not an operator name.) JOIN clauses only make sense for binary operators that return boolean. The idea behind a join selectivity estimator is to guess what fraction of the rows in a pair of tables will satisfy a WHERE-clause condition of the form:

table1.column1 OP table2.column2

for the current operator. As with the RESTRICT clause, this helps the optimizer very substantially by letting it figure out which of several possible join sequences is likely to take the least work.

As before, this chapter will make no attempt to explain how to write a join selectivity estimator function, but will just suggest that you use one of the standard estimators if one is applicable:

|  |
| --- |
| eqjoinsel for = |
| neqjoinsel for <> |
| scalarltjoinsel for < or <= |
| scalargtjoinsel for > or >= |
| areajoinsel for 2D area-based comparisons |
| positionjoinsel for 2D position-based comparisons |
| contjoinsel for 2D containment-based comparisons |

### 37.13.5. HASHES

The HASHES clause, if present, tells the system that it is permissible to use the hash join method for a join based on this operator. HASHES only makes sense for a binary operator that returns boolean, and in practice the operator must represent equality for some data type or pair of data types.

The assumption underlying hash join is that the join operator can only return true for pairs of left and right values that hash to the same hash code. If two values get put in different hash buckets, the join will never compare them at all, implicitly assuming that the result of the join operator must be false. So it never makes sense to specify HASHES for operators that do not represent some form of equality. In most cases it is only practical to support hashing for operators that take the same data type on both sides. However, sometimes it is possible to design compatible hash functions for two or more data types; that is, functions that will generate the same hash codes for “equal” values, even though the values have different representations. For example, it's fairly simple to arrange this property when hashing integers of different widths.

To be marked HASHES, the join operator must appear in a hash index operator family. This is not enforced when you create the operator, since of course the referencing operator family couldn't exist yet. But attempts to use the operator in hash joins will fail at run time if no such operator family exists. The system needs the operator family to find the data-type-specific hash function(s) for the operator's input data type(s). Of course, you must also create suitable hash functions before you can create the operator family.

Care should be exercised when preparing a hash function, because there are machine-dependent ways in which it might fail to do the right thing. For example, if your data type is a structure in which there might be uninteresting pad bits, you cannot simply pass the whole structure to hash\_any. (Unless you write your other operators and functions to ensure that the unused bits are always zero, which is the recommended strategy.) Another example is that on machines that meet the IEEE floating-point standard, negative zero and positive zero are different values (different bit patterns) but they are defined to compare equal. If a float value might contain negative zero then extra steps are needed to ensure it generates the same hash value as positive zero.

A hash-joinable operator must have a commutator (itself if the two operand data types are the same, or a related equality operator if they are different) that appears in the same operator family. If this is not the case, planner errors might occur when the operator is used. Also, it is a good idea (but not strictly required) for a hash operator family that supports multiple data types to provide equality operators for every combination of the data types; this allows better optimization.

### Note

The function underlying a hash-joinable operator must be marked immutable or stable. If it is volatile, the system will never attempt to use the operator for a hash join.

### Note

If a hash-joinable operator has an underlying function that is marked strict, the function must also be complete: that is, it should return true or false, never null, for any two nonnull inputs. If this rule is not followed, hash-optimization of IN operations might generate wrong results. (Specifically, IN might return false where the correct answer according to the standard would be null; or it might yield an error complaining that it wasn't prepared for a null result.)

### 37.13.6. MERGES

The MERGES clause, if present, tells the system that it is permissible to use the merge-join method for a join based on this operator. MERGES only makes sense for a binary operator that returns boolean, and in practice the operator must represent equality for some data type or pair of data types.

Merge join is based on the idea of sorting the left- and right-hand tables into order and then scanning them in parallel. So, both data types must be capable of being fully ordered, and the join operator must be one that can only succeed for pairs of values that fall at the “same place” in the sort order. In practice this means that the join operator must behave like equality. But it is possible to merge-join two distinct data types so long as they are logically compatible. For example, the smallint-versus-integer equality operator is merge-joinable. We only need sorting operators that will bring both data types into a logically compatible sequence.

To be marked MERGES, the join operator must appear as an equality member of a btree index operator family. This is not enforced when you create the operator, since of course the referencing operator family couldn't exist yet. But the operator will not actually be used for merge joins unless a matching operator family can be found. The MERGES flag thus acts as a hint to the planner that it's worth looking for a matching operator family.

A merge-joinable operator must have a commutator (itself if the two operand data types are the same, or a related equality operator if they are different) that appears in the same operator family. If this is not the case, planner errors might occur when the operator is used. Also, it is a good idea (but not strictly required) for a btree operator family that supports multiple data types to provide equality operators for every combination of the data types; this allows better optimization.

### Note

The function underlying a merge-joinable operator must be marked immutable or stable. If it is volatile, the system will never attempt to use the operator for a merge join.

## 37.14. Interfacing Extensions To Indexes

The procedures described thus far let you define new types, new functions, and new operators. However, we cannot yet define an index on a column of a new data type. To do this, we must define an operator class for the new data type. Later in this section, we will illustrate this concept in an example: a new operator class for the B-tree index method that stores and sorts complex numbers in ascending absolute value order.

Operator classes can be grouped into operator families to show the relationships between semantically compatible classes. When only a single data type is involved, an operator class is sufficient, so we'll focus on that case first and then return to operator families.

### 37.14.1. Index Methods and Operator Classes

The pg\_am table contains one row for every index method (internally known as access method). Support for regular access to tables is built into PostgreSQL, but all index methods are described in pg\_am. It is possible to add a new index access method by writing the necessary code and then creating a row in pg\_am — but that is beyond the scope of this chapter (see [**Chapter 60**](https://www.postgresql.org/docs/10/indexam.html)).

The routines for an index method do not directly know anything about the data types that the index method will operate on. Instead, an operator class identifies the set of operations that the index method needs to use to work with a particular data type. Operator classes are so called because one thing they specify is the set of WHERE-clause operators that can be used with an index (i.e., can be converted into an index-scan qualification). An operator class can also specify some support procedures that are needed by the internal operations of the index method, but do not directly correspond to any WHERE-clause operator that can be used with the index.

It is possible to define multiple operator classes for the same data type and index method. By doing this, multiple sets of indexing semantics can be defined for a single data type. For example, a B-tree index requires a sort ordering to be defined for each data type it works on. It might be useful for a complex-number data type to have one B-tree operator class that sorts the data by complex absolute value, another that sorts by real part, and so on. Typically, one of the operator classes will be deemed most commonly useful and will be marked as the default operator class for that data type and index method.

The same operator class name can be used for several different index methods (for example, both B-tree and hash index methods have operator classes named int4\_ops), but each such class is an independent entity and must be defined separately.

### 37.14.2. Index Method Strategies

The operators associated with an operator class are identified by “strategy numbers”, which serve to identify the semantics of each operator within the context of its operator class. For example, B-trees impose a strict ordering on keys, lesser to greater, and so operators like “less than” and “greater than or equal to” are interesting with respect to a B-tree. Because PostgreSQL allows the user to define operators, PostgreSQL cannot look at the name of an operator (e.g., < or >=) and tell what kind of comparison it is. Instead, the index method defines a set of “strategies”, which can be thought of as generalized operators. Each operator class specifies which actual operator corresponds to each strategy for a particular data type and interpretation of the index semantics.

The B-tree index method defines five strategies, shown in [**Table 37.2**](https://www.postgresql.org/docs/10/xindex.html#XINDEX-BTREE-STRAT-TABLE).

**Table 37.2. B-tree Strategies**

| **Operation** | **Strategy Number** |
| --- | --- |
| less than | 1 |
| less than or equal | 2 |
| equal | 3 |
| greater than or equal | 4 |
| greater than | 5 |

Hash indexes support only equality comparisons, and so they use only one strategy, shown in [**Table 37.3**](https://www.postgresql.org/docs/10/xindex.html#XINDEX-HASH-STRAT-TABLE).

**Table 37.3. Hash Strategies**

| **Operation** | **Strategy Number** |
| --- | --- |
| equal | 1 |

GiST indexes are more flexible: they do not have a fixed set of strategies at all. Instead, the “consistency” support routine of each particular GiST operator class interprets the strategy numbers however it likes. As an example, several of the built-in GiST index operator classes index two-dimensional geometric objects, providing the “R-tree” strategies shown in [**Table 37.4**](https://www.postgresql.org/docs/10/xindex.html#XINDEX-RTREE-STRAT-TABLE). Four of these are true two-dimensional tests (overlaps, same, contains, contained by); four of them consider only the X direction; and the other four provide the same tests in the Y direction.

**Table 37.4. GiST Two-Dimensional “R-tree” Strategies**

| **Operation** | **Strategy Number** |
| --- | --- |
| strictly left of | 1 |
| does not extend to right of | 2 |
| overlaps | 3 |
| does not extend to left of | 4 |
| strictly right of | 5 |
| same | 6 |
| contains | 7 |
| contained by | 8 |
| does not extend above | 9 |
| strictly below | 10 |
| strictly above | 11 |
| does not extend below | 12 |

SP-GiST indexes are similar to GiST indexes in flexibility: they don't have a fixed set of strategies. Instead the support routines of each operator class interpret the strategy numbers according to the operator class's definition. As an example, the strategy numbers used by the built-in operator classes for points are shown in [**Table 37.5**](https://www.postgresql.org/docs/10/xindex.html#XINDEX-SPGIST-POINT-STRAT-TABLE).

**Table 37.5. SP-GiST Point Strategies**

| **Operation** | **Strategy Number** |
| --- | --- |
| strictly left of | 1 |
| strictly right of | 5 |
| same | 6 |
| contained by | 8 |
| strictly below | 10 |
| strictly above | 11 |

GIN indexes are similar to GiST and SP-GiST indexes, in that they don't have a fixed set of strategies either. Instead the support routines of each operator class interpret the strategy numbers according to the operator class's definition. As an example, the strategy numbers used by the built-in operator class for arrays are shown in [**Table 37.6**](https://www.postgresql.org/docs/10/xindex.html#XINDEX-GIN-ARRAY-STRAT-TABLE).

**Table 37.6. GIN Array Strategies**

| **Operation** | **Strategy Number** |
| --- | --- |
| overlap | 1 |
| contains | 2 |
| is contained by | 3 |
| equal | 4 |

BRIN indexes are similar to GiST, SP-GiST and GIN indexes in that they don't have a fixed set of strategies either. Instead the support routines of each operator class interpret the strategy numbers according to the operator class's definition. As an example, the strategy numbers used by the built-in Minmax operator classes are shown in [**Table 37.7**](https://www.postgresql.org/docs/10/xindex.html#XINDEX-BRIN-MINMAX-STRAT-TABLE).

**Table 37.7. BRIN Minmax Strategies**

| **Operation** | **Strategy Number** |
| --- | --- |
| less than | 1 |
| less than or equal | 2 |
| equal | 3 |
| greater than or equal | 4 |
| greater than | 5 |

Notice that all the operators listed above return Boolean values. In practice, all operators defined as index method search operators must return type boolean, since they must appear at the top level of a WHERE clause to be used with an index. (Some index access methods also support ordering operators, which typically don't return Boolean values; that feature is discussed in [**Section 37.14.7**](https://www.postgresql.org/docs/10/xindex.html#XINDEX-ORDERING-OPS).)

### 37.14.3. Index Method Support Routines

Strategies aren't usually enough information for the system to figure out how to use an index. In practice, the index methods require additional support routines in order to work. For example, the B-tree index method must be able to compare two keys and determine whether one is greater than, equal to, or less than the other. Similarly, the hash index method must be able to compute hash codes for key values. These operations do not correspond to operators used in qualifications in SQL commands; they are administrative routines used by the index methods, internally.

Just as with strategies, the operator class identifies which specific functions should play each of these roles for a given data type and semantic interpretation. The index method defines the set of functions it needs, and the operator class identifies the correct functions to use by assigning them to the “support function numbers” specified by the index method.

B-trees require a single support function, and allow a second one to be supplied at the operator class author's option, as shown in [**Table 37.8**](https://www.postgresql.org/docs/10/xindex.html#XINDEX-BTREE-SUPPORT-TABLE).

**Table 37.8. B-tree Support Functions**

| **Function** | **Support Number** |
| --- | --- |
| Compare two keys and return an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second | 1 |
| Return the addresses of C-callable sort support function(s), as documented in utils/sortsupport.h (optional) | 2 |

Hash indexes require one support function, shown in [**Table 37.9**](https://www.postgresql.org/docs/10/xindex.html#XINDEX-HASH-SUPPORT-TABLE).

**Table 37.9. Hash Support Functions**

| **Function** | **Support Number** |
| --- | --- |
| Compute the hash value for a key | 1 |

GiST indexes have nine support functions, two of which are optional, as shown in [**Table 37.10**](https://www.postgresql.org/docs/10/xindex.html#XINDEX-GIST-SUPPORT-TABLE). (For more information see [**Chapter 62**](https://www.postgresql.org/docs/10/gist.html).)

**Table 37.10. GiST Support Functions**

| **Function** | **Description** | **Support Number** |
| --- | --- | --- |
| consistent | determine whether key satisfies the query qualifier | 1 |
| union | compute union of a set of keys | 2 |
| compress | compute a compressed representation of a key or value to be indexed | 3 |
| decompress | compute a decompressed representation of a compressed key | 4 |
| penalty | compute penalty for inserting new key into subtree with given subtree's key | 5 |
| picksplit | determine which entries of a page are to be moved to the new page and compute the union keys for resulting pages | 6 |
| equal | compare two keys and return true if they are equal | 7 |
| distance | determine distance from key to query value (optional) | 8 |
| fetch | compute original representation of a compressed key for index-only scans (optional) | 9 |

SP-GiST indexes require five support functions, as shown in [**Table 37.11**](https://www.postgresql.org/docs/10/xindex.html#XINDEX-SPGIST-SUPPORT-TABLE). (For more information see [**Chapter 63**](https://www.postgresql.org/docs/10/spgist.html).)

**Table 37.11. SP-GiST Support Functions**

| **Function** | **Description** | **Support Number** |
| --- | --- | --- |
| config | provide basic information about the operator class | 1 |
| choose | determine how to insert a new value into an inner tuple | 2 |
| picksplit | determine how to partition a set of values | 3 |
| inner\_consistent | determine which sub-partitions need to be searched for a query | 4 |
| leaf\_consistent | determine whether key satisfies the query qualifier | 5 |

GIN indexes have six support functions, three of which are optional, as shown in [**Table 37.12**](https://www.postgresql.org/docs/10/xindex.html#XINDEX-GIN-SUPPORT-TABLE). (For more information see [**Chapter 64**](https://www.postgresql.org/docs/10/gin.html).)

**Table 37.12. GIN Support Functions**

| **Function** | **Description** | **Support Number** |
| --- | --- | --- |
| compare | compare two keys and return an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second | 1 |
| extractValue | extract keys from a value to be indexed | 2 |
| extractQuery | extract keys from a query condition | 3 |
| consistent | determine whether value matches query condition (Boolean variant) (optional if support function 6 is present) | 4 |
| comparePartial | compare partial key from query and key from index, and return an integer less than zero, zero, or greater than zero, indicating whether GIN should ignore this index entry, treat the entry as a match, or stop the index scan (optional) | 5 |
| triConsistent | determine whether value matches query condition (ternary variant) (optional if support function 4 is present) | 6 |

BRIN indexes have four basic support functions, as shown in [**Table 37.13**](https://www.postgresql.org/docs/10/xindex.html#XINDEX-BRIN-SUPPORT-TABLE); those basic functions may require additional support functions to be provided. (For more information see [**Section 65.3**](https://www.postgresql.org/docs/10/brin-extensibility.html).)

**Table 37.13. BRIN Support Functions**

| **Function** | **Description** | **Support Number** |
| --- | --- | --- |
| opcInfo | return internal information describing the indexed columns' summary data | 1 |
| add\_value | add a new value to an existing summary index tuple | 2 |
| consistent | determine whether value matches query condition | 3 |
| union | compute union of two summary tuples | 4 |

Unlike search operators, support functions return whichever data type the particular index method expects; for example in the case of the comparison function for B-trees, a signed integer. The number and types of the arguments to each support function are likewise dependent on the index method. For B-tree and hash the comparison and hashing support functions take the same input data types as do the operators included in the operator class, but this is not the case for most GiST, SP-GiST, GIN, and BRIN support functions.

### 37.14.4. An Example

Now that we have seen the ideas, here is the promised example of creating a new operator class. (You can find a working copy of this example in src/tutorial/complex.c and src/tutorial/complex.sql in the source distribution.) The operator class encapsulates operators that sort complex numbers in absolute value order, so we choose the name complex\_abs\_ops. First, we need a set of operators. The procedure for defining operators was discussed in [**Section 37.12**](https://www.postgresql.org/docs/10/xoper.html). For an operator class on B-trees, the operators we require are:

* absolute-value less-than (strategy 1)
* absolute-value less-than-or-equal (strategy 2)
* absolute-value equal (strategy 3)
* absolute-value greater-than-or-equal (strategy 4)
* absolute-value greater-than (strategy 5)

The least error-prone way to define a related set of comparison operators is to write the B-tree comparison support function first, and then write the other functions as one-line wrappers around the support function. This reduces the odds of getting inconsistent results for corner cases. Following this approach, we first write:

#define Mag(c) ((c)->x\*(c)->x + (c)->y\*(c)->y)

static int

complex\_abs\_cmp\_internal(Complex \*a, Complex \*b)

{

double amag = Mag(a),

bmag = Mag(b);

if (amag < bmag)

return -1;

if (amag > bmag)

return 1;

return 0;

}

Now the less-than function looks like:

PG\_FUNCTION\_INFO\_V1(complex\_abs\_lt);

Datum

complex\_abs\_lt(PG\_FUNCTION\_ARGS)

{

Complex \*a = (Complex \*) PG\_GETARG\_POINTER(0);

Complex \*b = (Complex \*) PG\_GETARG\_POINTER(1);

PG\_RETURN\_BOOL(complex\_abs\_cmp\_internal(a, b) < 0);

}

The other four functions differ only in how they compare the internal function's result to zero.

Next we declare the functions and the operators based on the functions to SQL:

CREATE FUNCTION complex\_abs\_lt(complex, complex) RETURNS bool

AS '***filename***', 'complex\_abs\_lt'

LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (

leftarg = complex, rightarg = complex, procedure = complex\_abs\_lt,

commutator = > , negator = >= ,

restrict = scalarltsel, join = scalarltjoinsel

);

It is important to specify the correct commutator and negator operators, as well as suitable restriction and join selectivity functions, otherwise the optimizer will be unable to make effective use of the index. Note that the less-than, equal, and greater-than cases should use different selectivity functions.

Other things worth noting are happening here:

* There can only be one operator named, say, = and taking type complex for both operands. In this case we don't have any other operator = for complex, but if we were building a practical data type we'd probably want = to be the ordinary equality operation for complex numbers (and not the equality of the absolute values). In that case, we'd need to use some other operator name for complex\_abs\_eq.
* Although PostgreSQL can cope with functions having the same SQL name as long as they have different argument data types, C can only cope with one global function having a given name. So we shouldn't name the C function something simple like abs\_eq. Usually it's a good practice to include the data type name in the C function name, so as not to conflict with functions for other data types.
* We could have made the SQL name of the function abs\_eq, relying on PostgreSQL to distinguish it by argument data types from any other SQL function of the same name. To keep the example simple, we make the function have the same names at the C level and SQL level.

The next step is the registration of the support routine required by B-trees. The example C code that implements this is in the same file that contains the operator functions. This is how we declare the function:

CREATE FUNCTION complex\_abs\_cmp(complex, complex)

RETURNS integer

AS '***filename***'

LANGUAGE C IMMUTABLE STRICT;

Now that we have the required operators and support routine, we can finally create the operator class:

CREATE OPERATOR CLASS complex\_abs\_ops

DEFAULT FOR TYPE complex USING btree AS

OPERATOR 1 < ,

OPERATOR 2 <= ,

OPERATOR 3 = ,

OPERATOR 4 >= ,

OPERATOR 5 > ,

FUNCTION 1 complex\_abs\_cmp(complex, complex);

And we're done! It should now be possible to create and use B-tree indexes on complex columns.

We could have written the operator entries more verbosely, as in:

OPERATOR 1 < (complex, complex) ,

but there is no need to do so when the operators take the same data type we are defining the operator class for.

The above example assumes that you want to make this new operator class the default B-tree operator class for the complex data type. If you don't, just leave out the word DEFAULT.

### 37.14.5. Operator Classes and Operator Families

So far we have implicitly assumed that an operator class deals with only one data type. While there certainly can be only one data type in a particular index column, it is often useful to index operations that compare an indexed column to a value of a different data type. Also, if there is use for a cross-data-type operator in connection with an operator class, it is often the case that the other data type has a related operator class of its own. It is helpful to make the connections between related classes explicit, because this can aid the planner in optimizing SQL queries (particularly for B-tree operator classes, since the planner contains a great deal of knowledge about how to work with them).

To handle these needs, PostgreSQL uses the concept of an operator family. An operator family contains one or more operator classes, and can also contain indexable operators and corresponding support functions that belong to the family as a whole but not to any single class within the family. We say that such operators and functions are “loose” within the family, as opposed to being bound into a specific class. Typically each operator class contains single-data-type operators while cross-data-type operators are loose in the family.

All the operators and functions in an operator family must have compatible semantics, where the compatibility requirements are set by the index method. You might therefore wonder why bother to single out particular subsets of the family as operator classes; and indeed for many purposes the class divisions are irrelevant and the family is the only interesting grouping. The reason for defining operator classes is that they specify how much of the family is needed to support any particular index. If there is an index using an operator class, then that operator class cannot be dropped without dropping the index — but other parts of the operator family, namely other operator classes and loose operators, could be dropped. Thus, an operator class should be specified to contain the minimum set of operators and functions that are reasonably needed to work with an index on a specific data type, and then related but non-essential operators can be added as loose members of the operator family.

As an example, PostgreSQL has a built-in B-tree operator family integer\_ops, which includes operator classes int8\_ops, int4\_ops, and int2\_ops for indexes on bigint (int8), integer (int4), and smallint(int2) columns respectively. The family also contains cross-data-type comparison operators allowing any two of these types to be compared, so that an index on one of these types can be searched using a comparison value of another type. The family could be duplicated by these definitions:

CREATE OPERATOR FAMILY integer\_ops USING btree;

CREATE OPERATOR CLASS int8\_ops

DEFAULT FOR TYPE int8 USING btree FAMILY integer\_ops AS

-- standard int8 comparisons

OPERATOR 1 < ,

OPERATOR 2 <= ,

OPERATOR 3 = ,

OPERATOR 4 >= ,

OPERATOR 5 > ,

FUNCTION 1 btint8cmp(int8, int8) ,

FUNCTION 2 btint8sortsupport(internal) ;

CREATE OPERATOR CLASS int4\_ops

DEFAULT FOR TYPE int4 USING btree FAMILY integer\_ops AS

-- standard int4 comparisons

OPERATOR 1 < ,

OPERATOR 2 <= ,

OPERATOR 3 = ,

OPERATOR 4 >= ,

OPERATOR 5 > ,

FUNCTION 1 btint4cmp(int4, int4) ,

FUNCTION 2 btint4sortsupport(internal) ;

CREATE OPERATOR CLASS int2\_ops

DEFAULT FOR TYPE int2 USING btree FAMILY integer\_ops AS

-- standard int2 comparisons

OPERATOR 1 < ,

OPERATOR 2 <= ,

OPERATOR 3 = ,

OPERATOR 4 >= ,

OPERATOR 5 > ,

FUNCTION 1 btint2cmp(int2, int2) ,

FUNCTION 2 btint2sortsupport(internal) ;

ALTER OPERATOR FAMILY integer\_ops USING btree ADD

-- cross-type comparisons int8 vs int2

OPERATOR 1 < (int8, int2) ,

OPERATOR 2 <= (int8, int2) ,

OPERATOR 3 = (int8, int2) ,

OPERATOR 4 >= (int8, int2) ,

OPERATOR 5 > (int8, int2) ,

FUNCTION 1 btint82cmp(int8, int2) ,

-- cross-type comparisons int8 vs int4

OPERATOR 1 < (int8, int4) ,

OPERATOR 2 <= (int8, int4) ,

OPERATOR 3 = (int8, int4) ,

OPERATOR 4 >= (int8, int4) ,

OPERATOR 5 > (int8, int4) ,

FUNCTION 1 btint84cmp(int8, int4) ,

-- cross-type comparisons int4 vs int2

OPERATOR 1 < (int4, int2) ,

OPERATOR 2 <= (int4, int2) ,

OPERATOR 3 = (int4, int2) ,

OPERATOR 4 >= (int4, int2) ,

OPERATOR 5 > (int4, int2) ,

FUNCTION 1 btint42cmp(int4, int2) ,

-- cross-type comparisons int4 vs int8

OPERATOR 1 < (int4, int8) ,

OPERATOR 2 <= (int4, int8) ,

OPERATOR 3 = (int4, int8) ,

OPERATOR 4 >= (int4, int8) ,

OPERATOR 5 > (int4, int8) ,

FUNCTION 1 btint48cmp(int4, int8) ,

-- cross-type comparisons int2 vs int8

OPERATOR 1 < (int2, int8) ,

OPERATOR 2 <= (int2, int8) ,

OPERATOR 3 = (int2, int8) ,

OPERATOR 4 >= (int2, int8) ,

OPERATOR 5 > (int2, int8) ,

FUNCTION 1 btint28cmp(int2, int8) ,

-- cross-type comparisons int2 vs int4

OPERATOR 1 < (int2, int4) ,

OPERATOR 2 <= (int2, int4) ,

OPERATOR 3 = (int2, int4) ,

OPERATOR 4 >= (int2, int4) ,

OPERATOR 5 > (int2, int4) ,

FUNCTION 1 btint24cmp(int2, int4) ;

Notice that this definition “overloads” the operator strategy and support function numbers: each number occurs multiple times within the family. This is allowed so long as each instance of a particular number has distinct input data types. The instances that have both input types equal to an operator class's input type are the primary operators and support functions for that operator class, and in most cases should be declared as part of the operator class rather than as loose members of the family.

In a B-tree operator family, all the operators in the family must sort compatibly, meaning that the transitive laws hold across all the data types supported by the family: “if A = B and B = C, then A = C”, and “if A < B and B < C, then A < C”. Moreover, implicit or binary coercion casts between types represented in the operator family must not change the associated sort ordering. For each operator in the family there must be a support function having the same two input data types as the operator. It is recommended that a family be complete, i.e., for each combination of data types, all operators are included. Each operator class should include just the non-cross-type operators and support function for its data type.

To build a multiple-data-type hash operator family, compatible hash support functions must be created for each data type supported by the family. Here compatibility means that the functions are guaranteed to return the same hash code for any two values that are considered equal by the family's equality operators, even when the values are of different types. This is usually difficult to accomplish when the types have different physical representations, but it can be done in some cases. Furthermore, casting a value from one data type represented in the operator family to another data type also represented in the operator family via an implicit or binary coercion cast must not change the computed hash value. Notice that there is only one support function per data type, not one per equality operator. It is recommended that a family be complete, i.e., provide an equality operator for each combination of data types. Each operator class should include just the non-cross-type equality operator and the support function for its data type.

GiST, SP-GiST, and GIN indexes do not have any explicit notion of cross-data-type operations. The set of operators supported is just whatever the primary support functions for a given operator class can handle.

In BRIN, the requirements depends on the framework that provides the operator classes. For operator classes based on minmax, the behavior required is the same as for B-tree operator families: all the operators in the family must sort compatibly, and casts must not change the associated sort ordering.

### Note

Prior to PostgreSQL 8.3, there was no concept of operator families, and so any cross-data-type operators intended to be used with an index had to be bound directly into the index's operator class. While this approach still works, it is deprecated because it makes an index's dependencies too broad, and because the planner can handle cross-data-type comparisons more effectively when both data types have operators in the same operator family.

### 37.14.6. System Dependencies on Operator Classes

PostgreSQL uses operator classes to infer the properties of operators in more ways than just whether they can be used with indexes. Therefore, you might want to create operator classes even if you have no intention of indexing any columns of your data type.

In particular, there are SQL features such as ORDER BY and DISTINCT that require comparison and sorting of values. To implement these features on a user-defined data type, PostgreSQL looks for the default B-tree operator class for the data type. The “equals” member of this operator class defines the system's notion of equality of values for GROUP BY and DISTINCT, and the sort ordering imposed by the operator class defines the default ORDER BY ordering.

Comparison of arrays of user-defined types also relies on the semantics defined by the default B-tree operator class.

If there is no default B-tree operator class for a data type, the system will look for a default hash operator class. But since that kind of operator class only provides equality, in practice it is only enough to support array equality.

When there is no default operator class for a data type, you will get errors like “could not identify an ordering operator” if you try to use these SQL features with the data type.

### Note

In PostgreSQL versions before 7.4, sorting and grouping operations would implicitly use operators named =, <, and >. The new behavior of relying on default operator classes avoids having to make any assumption about the behavior of operators with particular names.

Another important point is that an operator that appears in a hash operator family is a candidate for hash joins, hash aggregation, and related optimizations. The hash operator family is essential here since it identifies the hash function(s) to use.

### 37.14.7. Ordering Operators

Some index access methods (currently, only GiST) support the concept of ordering operators. What we have been discussing so far are search operators. A search operator is one for which the index can be searched to find all rows satisfying WHERE ***indexed\_column*** ***operator*** ***constant***. Note that nothing is promised about the order in which the matching rows will be returned. In contrast, an ordering operator does not restrict the set of rows that can be returned, but instead determines their order. An ordering operator is one for which the index can be scanned to return rows in the order represented by ORDER BY ***indexed\_column*** ***operator*** ***constant***. The reason for defining ordering operators that way is that it supports nearest-neighbor searches, if the operator is one that measures distance. For example, a query like

SELECT \* FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;

finds the ten places closest to a given target point. A GiST index on the location column can do this efficiently because <-> is an ordering operator.

While search operators have to return Boolean results, ordering operators usually return some other type, such as float or numeric for distances. This type is normally not the same as the data type being indexed. To avoid hard-wiring assumptions about the behavior of different data types, the definition of an ordering operator is required to name a B-tree operator family that specifies the sort ordering of the result data type. As was stated in the previous section, B-tree operator families define PostgreSQL's notion of ordering, so this is a natural representation. Since the point <-> operator returns float8, it could be specified in an operator class creation command like this:

OPERATOR 15 <-> (point, point) FOR ORDER BY float\_ops

where float\_ops is the built-in operator family that includes operations on float8. This declaration states that the index is able to return rows in order of increasing values of the <-> operator.

### 37.14.8. Special Features of Operator Classes

There are two special features of operator classes that we have not discussed yet, mainly because they are not useful with the most commonly used index methods.

Normally, declaring an operator as a member of an operator class (or family) means that the index method can retrieve exactly the set of rows that satisfy a WHERE condition using the operator. For example:

SELECT \* FROM table WHERE integer\_column < 4;

can be satisfied exactly by a B-tree index on the integer column. But there are cases where an index is useful as an inexact guide to the matching rows. For example, if a GiST index stores only bounding boxes for geometric objects, then it cannot exactly satisfy a WHERE condition that tests overlap between nonrectangular objects such as polygons. Yet we could use the index to find objects whose bounding box overlaps the bounding box of the target object, and then do the exact overlap test only on the objects found by the index. If this scenario applies, the index is said to be “lossy”for the operator. Lossy index searches are implemented by having the index method return a recheck flag when a row might or might not really satisfy the query condition. The core system will then test the original query condition on the retrieved row to see whether it should be returned as a valid match. This approach works if the index is guaranteed to return all the required rows, plus perhaps some additional rows, which can be eliminated by performing the original operator invocation. The index methods that support lossy searches (currently, GiST, SP-GiST and GIN) allow the support functions of individual operator classes to set the recheck flag, and so this is essentially an operator-class feature.

Consider again the situation where we are storing in the index only the bounding box of a complex object such as a polygon. In this case there's not much value in storing the whole polygon in the index entry — we might as well store just a simpler object of type box. This situation is expressed by the STORAGE option in CREATE OPERATOR CLASS: we'd write something like:

CREATE OPERATOR CLASS polygon\_ops

DEFAULT FOR TYPE polygon USING gist AS

...

STORAGE box;

At present, only the GiST, GIN and BRIN index methods support a STORAGE type that's different from the column data type. The GiST compress and decompress support routines must deal with data-type conversion when STORAGE is used. In GIN, the STORAGE type identifies the type of the “key” values, which normally is different from the type of the indexed column — for example, an operator class for integer-array columns might have keys that are just integers. The GIN extractValue and extractQuery support routines are responsible for extracting keys from indexed values. BRIN is similar to GIN: the STORAGE type identifies the type of the stored summary values, and operator classes' support procedures are responsible for interpreting the summary values correctly.

## 37.15. Packaging Related Objects into an Extension

A useful extension to PostgreSQL typically includes multiple SQL objects; for example, a new data type will require new functions, new operators, and probably new index operator classes. It is helpful to collect all these objects into a single package to simplify database management. PostgreSQL calls such a package an extension. To define an extension, you need at least a script file that contains the SQL commands to create the extension's objects, and a control file that specifies a few basic properties of the extension itself. If the extension includes C code, there will typically also be a shared library file into which the C code has been built. Once you have these files, a simple [**CREATE EXTENSION**](https://www.postgresql.org/docs/10/sql-createextension.html) command loads the objects into your database.

The main advantage of using an extension, rather than just running the SQL script to load a bunch of “loose” objects into your database, is that PostgreSQL will then understand that the objects of the extension go together. You can drop all the objects with a single [**DROP EXTENSION**](https://www.postgresql.org/docs/10/sql-dropextension.html) command (no need to maintain a separate “uninstall” script). Even more useful, pg\_dump knows that it should not dump the individual member objects of the extension — it will just include a CREATE EXTENSION command in dumps, instead. This vastly simplifies migration to a new version of the extension that might contain more or different objects than the old version. Note however that you must have the extension's control, script, and other files available when loading such a dump into a new database.

PostgreSQL will not let you drop an individual object contained in an extension, except by dropping the whole extension. Also, while you can change the definition of an extension member object (for example, via CREATE OR REPLACE FUNCTION for a function), bear in mind that the modified definition will not be dumped by pg\_dump. Such a change is usually only sensible if you concurrently make the same change in the extension's script file. (But there are special provisions for tables containing configuration data; see [**Section 37.15.4**](https://www.postgresql.org/docs/10/extend-extensions.html#EXTEND-EXTENSIONS-CONFIG-TABLES).) In production situations, it's generally better to create an extension update script to perform changes to extension member objects.

The extension script may set privileges on objects that are part of the extension via GRANT and REVOKE statements. The final set of privileges for each object (if any are set) will be stored in the [pg\_init\_privs](https://www.postgresql.org/docs/10/catalog-pg-init-privs.html) system catalog. When pg\_dump is used, the CREATE EXTENSION command will be included in the dump, followed by the set of GRANT and REVOKE statements necessary to set the privileges on the objects to what they were at the time the dump was taken.

PostgreSQL does not currently support extension scripts issuing CREATE POLICY or SECURITY LABEL statements. These are expected to be set after the extension has been created. All RLS policies and security labels on extension objects will be included in dumps created by pg\_dump.

The extension mechanism also has provisions for packaging modification scripts that adjust the definitions of the SQL objects contained in an extension. For example, if version 1.1 of an extension adds one function and changes the body of another function compared to 1.0, the extension author can provide an update script that makes just those two changes. The ALTER EXTENSION UPDATEcommand can then be used to apply these changes and track which version of the extension is actually installed in a given database.

The kinds of SQL objects that can be members of an extension are shown in the description of [**ALTER EXTENSION**](https://www.postgresql.org/docs/10/sql-alterextension.html). Notably, objects that are database-cluster-wide, such as databases, roles, and tablespaces, cannot be extension members since an extension is only known within one database. (Although an extension script is not prohibited from creating such objects, if it does so they will not be tracked as part of the extension.) Also notice that while a table can be a member of an extension, its subsidiary objects such as indexes are not directly considered members of the extension. Another important point is that schemas can belong to extensions, but not vice versa: an extension as such has an unqualified name and does not exist “within” any schema. The extension's member objects, however, will belong to schemas whenever appropriate for their object types. It may or may not be appropriate for an extension to own the schema(s) its member objects are within.

If an extension's script creates any temporary objects (such as temp tables), those objects are treated as extension members for the remainder of the current session, but are automatically dropped at session end, as any temporary object would be. This is an exception to the rule that extension member objects cannot be dropped without dropping the whole extension.

### 37.15.1. Defining Extension Objects

Widely-distributed extensions should assume little about the database they occupy. In particular, unless you issued SET search\_path = pg\_temp, assume each unqualified name could resolve to an object that a malicious user has defined. Beware of constructs that depend on search\_path implicitly: IN and CASE ***expression*** WHEN always select an operator using the search path. In their place, use OPERATOR(***schema***.=) ANY and CASE WHEN ***expression***.

### 37.15.2. Extension Files

The [**CREATE EXTENSION**](https://www.postgresql.org/docs/10/sql-createextension.html) command relies on a control file for each extension, which must be named the same as the extension with a suffix of .control, and must be placed in the installation's SHAREDIR/extension directory. There must also be at least one SQL script file, which follows the naming pattern ***extension***--***version***.sql (for example, foo--1.0.sql for version 1.0 of extension foo). By default, the script file(s) are also placed in the SHAREDIR/extension directory; but the control file can specify a different directory for the script file(s).

The file format for an extension control file is the same as for the postgresql.conf file, namely a list of ***parameter\_name*** = ***value*** assignments, one per line. Blank lines and comments introduced by #are allowed. Be sure to quote any value that is not a single word or number.

A control file can set the following parameters:

directory (string)

The directory containing the extension's SQL script file(s). Unless an absolute path is given, the name is relative to the installation's SHAREDIR directory. The default behavior is equivalent to specifying directory = 'extension'.

default\_version (string)

The default version of the extension (the one that will be installed if no version is specified in CREATE EXTENSION). Although this can be omitted, that will result in CREATE EXTENSION failing if no VERSION option appears, so you generally don't want to do that.

comment (string)

A comment (any string) about the extension. The comment is applied when initially creating an extension, but not during extension updates (since that might override user-added comments). Alternatively, the extension's comment can be set by writing a [**COMMENT**](https://www.postgresql.org/docs/10/sql-comment.html) command in the script file.

encoding (string)

The character set encoding used by the script file(s). This should be specified if the script files contain any non-ASCII characters. Otherwise the files will be assumed to be in the database encoding.

module\_pathname (string)

The value of this parameter will be substituted for each occurrence of MODULE\_PATHNAME in the script file(s). If it is not set, no substitution is made. Typically, this is set to $libdir/***shared\_library\_name*** and then MODULE\_PATHNAME is used in CREATE FUNCTION commands for C-language functions, so that the script files do not need to hard-wire the name of the shared library.

requires (string)

A list of names of extensions that this extension depends on, for example requires = 'foo, bar'. Those extensions must be installed before this one can be installed.

superuser (boolean)

If this parameter is true (which is the default), only superusers can create the extension or update it to a new version. If it is set to false, just the privileges required to execute the commands in the installation or update script are required.

relocatable (boolean)

An extension is relocatable if it is possible to move its contained objects into a different schema after initial creation of the extension. The default is false, i.e. the extension is not relocatable. See [**Section 37.15.3**](https://www.postgresql.org/docs/10/extend-extensions.html#EXTEND-EXTENSIONS-RELOCATION) for more information.

schema (string)

This parameter can only be set for non-relocatable extensions. It forces the extension to be loaded into exactly the named schema and not any other. The schema parameter is consulted only when initially creating an extension, not during extension updates. See [**Section 37.15.3**](https://www.postgresql.org/docs/10/extend-extensions.html#EXTEND-EXTENSIONS-RELOCATION) for more information.

In addition to the primary control file ***extension***.control, an extension can have secondary control files named in the style ***extension***--***version***.control. If supplied, these must be located in the script file directory. Secondary control files follow the same format as the primary control file. Any parameters set in a secondary control file override the primary control file when installing or updating to that version of the extension. However, the parameters directory and default\_version cannot be set in a secondary control file.

An extension's SQL script files can contain any SQL commands, except for transaction control commands (BEGIN, COMMIT, etc) and commands that cannot be executed inside a transaction block (such as VACUUM). This is because the script files are implicitly executed within a transaction block.

An extension's SQL script files can also contain lines beginning with \echo, which will be ignored (treated as comments) by the extension mechanism. This provision is commonly used to throw an error if the script file is fed to psql rather than being loaded via CREATE EXTENSION (see example script in [**Section 37.15.7**](https://www.postgresql.org/docs/10/extend-extensions.html#EXTEND-EXTENSIONS-EXAMPLE)). Without that, users might accidentally load the extension's contents as “loose”objects rather than as an extension, a state of affairs that's a bit tedious to recover from.

While the script files can contain any characters allowed by the specified encoding, control files should contain only plain ASCII, because there is no way for PostgreSQL to know what encoding a control file is in. In practice this is only an issue if you want to use non-ASCII characters in the extension's comment. Recommended practice in that case is to not use the control file commentparameter, but instead use COMMENT ON EXTENSION within a script file to set the comment.

### 37.15.3. Extension Relocatability

Users often wish to load the objects contained in an extension into a different schema than the extension's author had in mind. There are three supported levels of relocatability:

* A fully relocatable extension can be moved into another schema at any time, even after it's been loaded into a database. This is done with the ALTER EXTENSION SET SCHEMA command, which automatically renames all the member objects into the new schema. Normally, this is only possible if the extension contains no internal assumptions about what schema any of its objects are in. Also, the extension's objects must all be in one schema to begin with (ignoring objects that do not belong to any schema, such as procedural languages). Mark a fully relocatable extension by setting relocatable = true in its control file.
* An extension might be relocatable during installation but not afterwards. This is typically the case if the extension's script file needs to reference the target schema explicitly, for example in setting search\_path properties for SQL functions. For such an extension, set relocatable = false in its control file, and use @extschema@ to refer to the target schema in the script file. All occurrences of this string will be replaced by the actual target schema's name before the script is executed. The user can set the target schema using the SCHEMA option of CREATE EXTENSION.
* If the extension does not support relocation at all, set relocatable = false in its control file, and also set schema to the name of the intended target schema. This will prevent use of the SCHEMA option of CREATE EXTENSION, unless it specifies the same schema named in the control file. This choice is typically necessary if the extension contains internal assumptions about schema names that can't be replaced by uses of @extschema@. The @extschema@ substitution mechanism is available in this case too, although it is of limited use since the schema name is determined by the control file.

In all cases, the script file will be executed with [**search\_path**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-SEARCH-PATH) initially set to point to the target schema; that is, CREATE EXTENSION does the equivalent of this:

SET LOCAL search\_path TO @extschema@;

This allows the objects created by the script file to go into the target schema. The script file can change search\_path if it wishes, but that is generally undesirable. search\_path is restored to its previous setting upon completion of CREATE EXTENSION.

The target schema is determined by the schema parameter in the control file if that is given, otherwise by the SCHEMA option of CREATE EXTENSION if that is given, otherwise the current default object creation schema (the first one in the caller's search\_path). When the control file schema parameter is used, the target schema will be created if it doesn't already exist, but in the other two cases it must already exist.

If any prerequisite extensions are listed in requires in the control file, their target schemas are appended to the initial setting of search\_path. This allows their objects to be visible to the new extension's script file.

Although a non-relocatable extension can contain objects spread across multiple schemas, it is usually desirable to place all the objects meant for external use into a single schema, which is considered the extension's target schema. Such an arrangement works conveniently with the default setting of search\_path during creation of dependent extensions.

### 37.15.4. Extension Configuration Tables

Some extensions include configuration tables, which contain data that might be added or changed by the user after installation of the extension. Ordinarily, if a table is part of an extension, neither the table's definition nor its content will be dumped by pg\_dump. But that behavior is undesirable for a configuration table; any data changes made by the user need to be included in dumps, or the extension will behave differently after a dump and reload.

To solve this problem, an extension's script file can mark a table or a sequence it has created as a configuration relation, which will cause pg\_dump to include the table's or the sequence's contents (not its definition) in dumps. To do that, call the function pg\_extension\_config\_dump(regclass, text) after creating the table or the sequence, for example

CREATE TABLE my\_config (key text, value text);

CREATE SEQUENCE my\_config\_seq;

SELECT pg\_catalog.pg\_extension\_config\_dump('my\_config', '');

SELECT pg\_catalog.pg\_extension\_config\_dump('my\_config\_seq', '');

Any number of tables or sequences can be marked this way. Sequences associated with serial or bigserial columns can be marked as well.

When the second argument of pg\_extension\_config\_dump is an empty string, the entire contents of the table are dumped by pg\_dump. This is usually only correct if the table is initially empty as created by the extension script. If there is a mixture of initial data and user-provided data in the table, the second argument of pg\_extension\_config\_dump provides a WHERE condition that selects the data to be dumped. For example, you might do

CREATE TABLE my\_config (key text, value text, standard\_entry boolean);

SELECT pg\_catalog.pg\_extension\_config\_dump('my\_config', 'WHERE NOT standard\_entry');

and then make sure that standard\_entry is true only in the rows created by the extension's script.

For sequences, the second argument of pg\_extension\_config\_dump has no effect.

More complicated situations, such as initially-provided rows that might be modified by users, can be handled by creating triggers on the configuration table to ensure that modified rows are marked correctly.

You can alter the filter condition associated with a configuration table by calling pg\_extension\_config\_dump again. (This would typically be useful in an extension update script.) The only way to mark a table as no longer a configuration table is to dissociate it from the extension with ALTER EXTENSION ... DROP TABLE.

Note that foreign key relationships between these tables will dictate the order in which the tables are dumped out by pg\_dump. Specifically, pg\_dump will attempt to dump the referenced-by table before the referencing table. As the foreign key relationships are set up at CREATE EXTENSION time (prior to data being loaded into the tables) circular dependencies are not supported. When circular dependencies exist, the data will still be dumped out but the dump will not be able to be restored directly and user intervention will be required.

Sequences associated with serial or bigserial columns need to be directly marked to dump their state. Marking their parent relation is not enough for this purpose.

### 37.15.5. Extension Updates

One advantage of the extension mechanism is that it provides convenient ways to manage updates to the SQL commands that define an extension's objects. This is done by associating a version name or number with each released version of the extension's installation script. In addition, if you want users to be able to update their databases dynamically from one version to the next, you should provide update scripts that make the necessary changes to go from one version to the next. Update scripts have names following the pattern ***extension***--***oldversion***--***newversion***.sql (for example, foo--1.0--1.1.sql contains the commands to modify version 1.0 of extension foo into version 1.1).

Given that a suitable update script is available, the command ALTER EXTENSION UPDATE will update an installed extension to the specified new version. The update script is run in the same environment that CREATE EXTENSION provides for installation scripts: in particular, search\_path is set up in the same way, and any new objects created by the script are automatically added to the extension. Also, if the script chooses to drop extension member objects, they are automatically dissociated from the extension.

If an extension has secondary control files, the control parameters that are used for an update script are those associated with the script's target (new) version.

The update mechanism can be used to solve an important special case: converting a “loose” collection of objects into an extension. Before the extension mechanism was added to PostgreSQL (in 9.1), many people wrote extension modules that simply created assorted unpackaged objects. Given an existing database containing such objects, how can we convert the objects into a properly packaged extension? Dropping them and then doing a plain CREATE EXTENSION is one way, but it's not desirable if the objects have dependencies (for example, if there are table columns of a data type created by the extension). The way to fix this situation is to create an empty extension, then use ALTER EXTENSION ADD to attach each pre-existing object to the extension, then finally create any new objects that are in the current extension version but were not in the unpackaged release. CREATE EXTENSION supports this case with its FROM ***old\_version*** option, which causes it to not run the normal installation script for the target version, but instead the update script named ***extension***--***old\_version***--***target\_version***.sql. The choice of the dummy version name to use as ***old\_version*** is up to the extension author, though unpackaged is a common convention. If you have multiple prior versions you need to be able to update into extension style, use multiple dummy version names to identify them.

ALTER EXTENSION is able to execute sequences of update script files to achieve a requested update. For example, if only foo--1.0--1.1.sql and foo--1.1--2.0.sql are available, ALTER EXTENSION will apply them in sequence if an update to version 2.0 is requested when 1.0 is currently installed.

PostgreSQL doesn't assume anything about the properties of version names: for example, it does not know whether 1.1 follows 1.0. It just matches up the available version names and follows the path that requires applying the fewest update scripts. (A version name can actually be any string that doesn't contain -- or leading or trailing -.)

Sometimes it is useful to provide “downgrade” scripts, for example foo--1.1--1.0.sql to allow reverting the changes associated with version 1.1. If you do that, be careful of the possibility that a downgrade script might unexpectedly get applied because it yields a shorter path. The risky case is where there is a “fast path” update script that jumps ahead several versions as well as a downgrade script to the fast path's start point. It might take fewer steps to apply the downgrade and then the fast path than to move ahead one version at a time. If the downgrade script drops any irreplaceable objects, this will yield undesirable results.

To check for unexpected update paths, use this command:

SELECT \* FROM pg\_extension\_update\_paths('***extension\_name***');

This shows each pair of distinct known version names for the specified extension, together with the update path sequence that would be taken to get from the source version to the target version, or NULL if there is no available update path. The path is shown in textual form with -- separators. You can use regexp\_split\_to\_array(path,'--') if you prefer an array format.

### 37.15.6. Installing Extensions using Update Scripts

An extension that has been around for awhile will probably exist in several versions, for which the author will need to write update scripts. For example, if you have released a foo extension in versions 1.0, 1.1, and 1.2, there should be update scripts foo--1.0--1.1.sql and foo--1.1--1.2.sql. Before PostgreSQL 10, it was necessary to also create new script files foo--1.1.sql and foo--1.2.sql that directly build the newer extension versions, or else the newer versions could not be installed directly, only by installing 1.0 and then updating. That was tedious and duplicative, but now it's unnecessary, because CREATE EXTENSION can follow update chains automatically. For example, if only the script files foo--1.0.sql, foo--1.0--1.1.sql, and foo--1.1--1.2.sql are available then a request to install version 1.2 is honored by running those three scripts in sequence. The processing is the same as if you'd first installed 1.0 and then updated to 1.2. (As with ALTER EXTENSION UPDATE, if multiple pathways are available then the shortest is preferred.) Arranging an extension's script files in this style can reduce the amount of maintenance effort needed to produce small updates.

If you use secondary (version-specific) control files with an extension maintained in this style, keep in mind that each version needs a control file even if it has no stand-alone installation script, as that control file will determine how the implicit update to that version is performed. For example, if foo--1.0.control specifies requires = 'bar' but foo's other control files do not, the extension's dependency on bar will be dropped when updating from 1.0 to another version.

### 37.15.7. Extension Example

Here is a complete example of an SQL-only extension, a two-element composite type that can store any type of value in its slots, which are named “k” and “v”. Non-text values are automatically coerced to text for storage.

The script file pair--1.0.sql looks like this:

-- complain if script is sourced in psql, rather than via CREATE EXTENSION

\echo Use "CREATE EXTENSION pair" to load this file. \quit

CREATE TYPE pair AS ( k text, v text );

CREATE OR REPLACE FUNCTION pair(text, text)

RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::@extschema@.pair;';

CREATE OPERATOR ~> (LEFTARG = text, RIGHTARG = text, PROCEDURE = pair);

-- "SET search\_path" is easy to get right, but qualified names perform better.

CREATE OR REPLACE FUNCTION lower(pair)

RETURNS pair LANGUAGE SQL

AS 'SELECT ROW(lower($1.k), lower($1.v))::@extschema@.pair;'

SET search\_path = pg\_temp;

CREATE OR REPLACE FUNCTION pair\_concat(pair, pair)

RETURNS pair LANGUAGE SQL

AS 'SELECT ROW($1.k OPERATOR(pg\_catalog.||) $2.k,

$1.v OPERATOR(pg\_catalog.||) $2.v)::@extschema@.pair;';

The control file pair.control looks like this:

# pair extension

comment = 'A key/value pair data type'

default\_version = '1.0'

relocatable = false

While you hardly need a makefile to install these two files into the correct directory, you could use a Makefile containing this:

EXTENSION = pair

DATA = pair--1.0.sql

PG\_CONFIG = pg\_config

PGXS := $(shell $(PG\_CONFIG) --pgxs)

include $(PGXS)

This makefile relies on PGXS, which is described in [**Section 37.16**](https://www.postgresql.org/docs/10/extend-pgxs.html). The command make install will install the control and script files into the correct directory as reported by pg\_config.

Once the files are installed, use the [**CREATE EXTENSION**](https://www.postgresql.org/docs/10/sql-createextension.html) command to load the objects into any particular database.

## 37.16. Extension Building Infrastructure

If you are thinking about distributing your PostgreSQL extension modules, setting up a portable build system for them can be fairly difficult. Therefore the PostgreSQL installation provides a build infrastructure for extensions, called PGXS, so that simple extension modules can be built simply against an already installed server. PGXS is mainly intended for extensions that include C code, although it can be used for pure-SQL extensions too. Note that PGXS is not intended to be a universal build system framework that can be used to build any software interfacing to PostgreSQL; it simply automates common build rules for simple server extension modules. For more complicated packages, you might need to write your own build system.

To use the PGXS infrastructure for your extension, you must write a simple makefile. In the makefile, you need to set some variables and include the global PGXS makefile. Here is an example that builds an extension module named isbn\_issn, consisting of a shared library containing some C code, an extension control file, a SQL script, and a documentation text file:

MODULES = isbn\_issn

EXTENSION = isbn\_issn

DATA = isbn\_issn--1.0.sql

DOCS = README.isbn\_issn

PG\_CONFIG = pg\_config

PGXS := $(shell $(PG\_CONFIG) --pgxs)

include $(PGXS)

The last three lines should always be the same. Earlier in the file, you assign variables or add custom make rules.

Set one of these three variables to specify what is built:

MODULES

list of shared-library objects to be built from source files with same stem (do not include library suffixes in this list)

MODULE\_big

a shared library to build from multiple source files (list object files in OBJS)

PROGRAM

an executable program to build (list object files in OBJS)

The following variables can also be set:

EXTENSION

extension name(s); for each name you must provide an ***extension***.control file, which will be installed into ***prefix***/share/extension

MODULEDIR

subdirectory of ***prefix***/share into which DATA and DOCS files should be installed (if not set, default is extension if EXTENSION is set, or contrib if not)

DATA

random files to install into ***prefix***/share/$MODULEDIR

DATA\_built

random files to install into ***prefix***/share/$MODULEDIR, which need to be built first

DATA\_TSEARCH

random files to install under ***prefix***/share/tsearch\_data

DOCS

random files to install under ***prefix***/doc/$MODULEDIR

SCRIPTS

script files (not binaries) to install into ***prefix***/bin

SCRIPTS\_built

script files (not binaries) to install into ***prefix***/bin, which need to be built first

REGRESS

list of regression test cases (without suffix), see below

REGRESS\_OPTS

additional switches to pass to pg\_regress

NO\_INSTALLCHECK

don't define an installcheck target, useful e.g. if tests require special configuration, or don't use pg\_regress

EXTRA\_CLEAN

extra files to remove in make clean

PG\_CPPFLAGS

will be prepended to CPPFLAGS

PG\_CFLAGS

will be appended to CFLAGS

PG\_CXXFLAGS

will be appended to CXXFLAGS

PG\_LDFLAGS

will be prepended to LDFLAGS

PG\_LIBS

will be added to PROGRAM link line

SHLIB\_LINK

will be added to MODULE\_big link line

PG\_CONFIG

path to pg\_config program for the PostgreSQL installation to build against (typically just pg\_config to use the first one in your PATH)

Put this makefile as Makefile in the directory which holds your extension. Then you can do make to compile, and then make install to install your module. By default, the extension is compiled and installed for the PostgreSQL installation that corresponds to the first pg\_config program found in your PATH. You can use a different installation by setting PG\_CONFIG to point to its pg\_config program, either within the makefile or on the make command line.

You can also run make in a directory outside the source tree of your extension, if you want to keep the build directory separate. This procedure is also called a VPATH build. Here's how:

mkdir build\_dir

cd build\_dir

make -f /path/to/extension/source/tree/Makefile

make -f /path/to/extension/source/tree/Makefile install

Alternatively, you can set up a directory for a VPATH build in a similar way to how it is done for the core code. One way to do this is using the core script config/prep\_buildtree. Once this has been done you can build by setting the make variable VPATH like this:

make VPATH=/path/to/extension/source/tree

make VPATH=/path/to/extension/source/tree install

This procedure can work with a greater variety of directory layouts.

The scripts listed in the REGRESS variable are used for regression testing of your module, which can be invoked by make installcheck after doing make install. For this to work you must have a running PostgreSQL server. The script files listed in REGRESS must appear in a subdirectory named sql/ in your extension's directory. These files must have extension .sql, which must not be included in the REGRESS list in the makefile. For each test there should also be a file containing the expected output in a subdirectory named expected/, with the same stem and extension .out. make installcheckexecutes each test script with psql, and compares the resulting output to the matching expected file. Any differences will be written to the file regression.diffs in diff -c format. Note that trying to run a test that is missing its expected file will be reported as “trouble”, so make sure you have all expected files.

### Tip

The easiest way to create the expected files is to create empty files, then do a test run (which will of course report differences). Inspect the actual result files found in the results/ directory, then copy them to expected/ if they match what you expect from the test.

## Chapter 38. Triggers

This chapter provides general information about writing trigger functions. Trigger functions can be written in most of the available procedural languages, including PL/pgSQL ([**Chapter 42**](https://www.postgresql.org/docs/10/plpgsql.html)), PL/Tcl([**Chapter 43**](https://www.postgresql.org/docs/10/pltcl.html)), PL/Perl ([**Chapter 44**](https://www.postgresql.org/docs/10/plperl.html)), and PL/Python ([**Chapter 45**](https://www.postgresql.org/docs/10/plpython.html)). After reading this chapter, you should consult the chapter for your favorite procedural language to find out the language-specific details of writing a trigger in it.

It is also possible to write a trigger function in C, although most people find it easier to use one of the procedural languages. It is not currently possible to write a trigger function in the plain SQL function language.

## 38.1. Overview of Trigger Behavior

A trigger is a specification that the database should automatically execute a particular function whenever a certain type of operation is performed. Triggers can be attached to tables (partitioned or not), views, and foreign tables.

On tables and foreign tables, triggers can be defined to execute either before or after any INSERT, UPDATE, or DELETE operation, either once per modified row, or once per SQL statement. UPDATE triggers can moreover be set to fire only if certain columns are mentioned in the SET clause of the UPDATE statement. Triggers can also fire for TRUNCATE statements. If a trigger event occurs, the trigger's function is called at the appropriate time to handle the event.

On views, triggers can be defined to execute instead of INSERT, UPDATE, or DELETE operations. Such INSTEAD OF triggers are fired once for each row that needs to be modified in the view. It is the responsibility of the trigger's function to perform the necessary modifications to the view's underlying base table(s) and, where appropriate, return the modified row as it will appear in the view. Triggers on views can also be defined to execute once per SQL statement, before or after INSERT, UPDATE, or DELETE operations. However, such triggers are fired only if there is also an INSTEAD OFtrigger on the view. Otherwise, any statement targeting the view must be rewritten into a statement affecting its underlying base table(s), and then the triggers that will be fired are the ones attached to the base table(s).

The trigger function must be defined before the trigger itself can be created. The trigger function must be declared as a function taking no arguments and returning type trigger. (The trigger function receives its input through a specially-passed TriggerData structure, not in the form of ordinary function arguments.)

Once a suitable trigger function has been created, the trigger is established with [**CREATE TRIGGER**](https://www.postgresql.org/docs/10/sql-createtrigger.html). The same trigger function can be used for multiple triggers.

PostgreSQL offers both per-row triggers and per-statement triggers. With a per-row trigger, the trigger function is invoked once for each row that is affected by the statement that fired the trigger. In contrast, a per-statement trigger is invoked only once when an appropriate statement is executed, regardless of the number of rows affected by that statement. In particular, a statement that affects zero rows will still result in the execution of any applicable per-statement triggers. These two types of triggers are sometimes called row-level triggers and statement-level triggers, respectively. Triggers on TRUNCATE may only be defined at statement level, not per-row.

Triggers are also classified according to whether they fire before, after, or instead of the operation. These are referred to as BEFORE triggers, AFTER triggers, and INSTEAD OF triggers respectively. Statement-level BEFORE triggers naturally fire before the statement starts to do anything, while statement-level AFTER triggers fire at the very end of the statement. These types of triggers may be defined on tables, views, or foreign tables. Row-level BEFORE triggers fire immediately before a particular row is operated on, while row-level AFTER triggers fire at the end of the statement (but before any statement-level AFTER triggers). These types of triggers may only be defined on non-partitioned tables and foreign tables, not views. INSTEAD OF triggers may only be defined on views, and only at row level; they fire immediately as each row in the view is identified as needing to be operated on.

A statement that targets a parent table in an inheritance or partitioning hierarchy does not cause the statement-level triggers of affected child tables to be fired; only the parent table's statement-level triggers are fired. However, row-level triggers of any affected child tables will be fired.

If an INSERT contains an ON CONFLICT DO UPDATE clause, it is possible that the effects of row-level BEFORE INSERT triggers and row-level BEFORE UPDATE triggers can both be applied in a way that is apparent from the final state of the updated row, if an EXCLUDED column is referenced. There need not be an EXCLUDED column reference for both sets of row-level BEFORE triggers to execute, though. The possibility of surprising outcomes should be considered when there are both BEFORE INSERT and BEFORE UPDATE row-level triggers that change a row being inserted/updated (this can be problematic even if the modifications are more or less equivalent, if they're not also idempotent). Note that statement-level UPDATE triggers are executed when ON CONFLICT DO UPDATE is specified, regardless of whether or not any rows were affected by the UPDATE (and regardless of whether the alternative UPDATE path was ever taken). An INSERT with an ON CONFLICT DO UPDATE clause will execute statement-level BEFORE INSERT triggers first, then statement-level BEFORE UPDATE triggers, followed by statement-level AFTER UPDATE triggers and finally statement-level AFTER INSERT triggers.

Trigger functions invoked by per-statement triggers should always return NULL. Trigger functions invoked by per-row triggers can return a table row (a value of type HeapTuple) to the calling executor, if they choose. A row-level trigger fired before an operation has the following choices:

* It can return NULL to skip the operation for the current row. This instructs the executor to not perform the row-level operation that invoked the trigger (the insertion, modification, or deletion of a particular table row).
* For row-level INSERT and UPDATE triggers only, the returned row becomes the row that will be inserted or will replace the row being updated. This allows the trigger function to modify the row being inserted or updated.

A row-level BEFORE trigger that does not intend to cause either of these behaviors must be careful to return as its result the same row that was passed in (that is, the NEW row for INSERT and UPDATEtriggers, the OLD row for DELETE triggers).

A row-level INSTEAD OF trigger should either return NULL to indicate that it did not modify any data from the view's underlying base tables, or it should return the view row that was passed in (the NEWrow for INSERT and UPDATE operations, or the OLD row for DELETE operations). A nonnull return value is used to signal that the trigger performed the necessary data modifications in the view. This will cause the count of the number of rows affected by the command to be incremented. For INSERT and UPDATE operations, the trigger may modify the NEW row before returning it. This will change the data returned by INSERT RETURNING or UPDATE RETURNING, and is useful when the view will not show exactly the same data that was provided.

The return value is ignored for row-level triggers fired after an operation, and so they can return NULL.

If more than one trigger is defined for the same event on the same relation, the triggers will be fired in alphabetical order by trigger name. In the case of BEFORE and INSTEAD OF triggers, the possibly-modified row returned by each trigger becomes the input to the next trigger. If any BEFORE or INSTEAD OF trigger returns NULL, the operation is abandoned for that row and subsequent triggers are not fired (for that row).

A trigger definition can also specify a Boolean WHEN condition, which will be tested to see whether the trigger should be fired. In row-level triggers the WHEN condition can examine the old and/or new values of columns of the row. (Statement-level triggers can also have WHEN conditions, although the feature is not so useful for them.) In a BEFORE trigger, the WHEN condition is evaluated just before the function is or would be executed, so using WHEN is not materially different from testing the same condition at the beginning of the trigger function. However, in an AFTER trigger, the WHEN condition is evaluated just after the row update occurs, and it determines whether an event is queued to fire the trigger at the end of statement. So when an AFTER trigger's WHEN condition does not return true, it is not necessary to queue an event nor to re-fetch the row at end of statement. This can result in significant speedups in statements that modify many rows, if the trigger only needs to be fired for a few of the rows. INSTEAD OF triggers do not support WHEN conditions.

Typically, row-level BEFORE triggers are used for checking or modifying the data that will be inserted or updated. For example, a BEFORE trigger might be used to insert the current time into a timestampcolumn, or to check that two elements of the row are consistent. Row-level AFTER triggers are most sensibly used to propagate the updates to other tables, or make consistency checks against other tables. The reason for this division of labor is that an AFTER trigger can be certain it is seeing the final value of the row, while a BEFORE trigger cannot; there might be other BEFORE triggers firing after it. If you have no specific reason to make a trigger BEFORE or AFTER, the BEFORE case is more efficient, since the information about the operation doesn't have to be saved until end of statement.

If a trigger function executes SQL commands then these commands might fire triggers again. This is known as cascading triggers. There is no direct limitation on the number of cascade levels. It is possible for cascades to cause a recursive invocation of the same trigger; for example, an INSERT trigger might execute a command that inserts an additional row into the same table, causing the INSERT trigger to be fired again. It is the trigger programmer's responsibility to avoid infinite recursion in such scenarios.

When a trigger is being defined, arguments can be specified for it. The purpose of including arguments in the trigger definition is to allow different triggers with similar requirements to call the same function. As an example, there could be a generalized trigger function that takes as its arguments two column names and puts the current user in one and the current time stamp in the other. Properly written, this trigger function would be independent of the specific table it is triggering on. So the same function could be used for INSERT events on any table with suitable columns, to automatically track creation of records in a transaction table for example. It could also be used to track last-update events if defined as an UPDATE trigger.

Each programming language that supports triggers has its own method for making the trigger input data available to the trigger function. This input data includes the type of trigger event (e.g., INSERTor UPDATE) as well as any arguments that were listed in CREATE TRIGGER. For a row-level trigger, the input data also includes the NEW row for INSERT and UPDATE triggers, and/or the OLD row for UPDATE and DELETE triggers.

By default, statement-level triggers do not have any way to examine the individual row(s) modified by the statement. But an AFTER STATEMENT trigger can request that transition tables be created to make the sets of affected rows available to the trigger. AFTER ROW triggers can also request transition tables, so that they can see the total changes in the table as well as the change in the individual row they are currently being fired for. The method for examining the transition tables again depends on the programming language that is being used, but the typical approach is to make the transition tables act like read-only temporary tables that can be accessed by SQL commands issued within the trigger function.

## 38.2. Visibility of Data Changes

If you execute SQL commands in your trigger function, and these commands access the table that the trigger is for, then you need to be aware of the data visibility rules, because they determine whether these SQL commands will see the data change that the trigger is fired for. Briefly:

* Statement-level triggers follow simple visibility rules: none of the changes made by a statement are visible to statement-level BEFORE triggers, whereas all modifications are visible to statement-level AFTER triggers.
* The data change (insertion, update, or deletion) causing the trigger to fire is naturally not visible to SQL commands executed in a row-level BEFORE trigger, because it hasn't happened yet.
* However, SQL commands executed in a row-level BEFORE trigger will see the effects of data changes for rows previously processed in the same outer command. This requires caution, since the ordering of these change events is not in general predictable; a SQL command that affects multiple rows can visit the rows in any order.
* Similarly, a row-level INSTEAD OF trigger will see the effects of data changes made by previous firings of INSTEAD OF triggers in the same outer command.
* When a row-level AFTER trigger is fired, all data changes made by the outer command are already complete, and are visible to the invoked trigger function.

If your trigger function is written in any of the standard procedural languages, then the above statements apply only if the function is declared VOLATILE. Functions that are declared STABLE or IMMUTABLE will not see changes made by the calling command in any case.

Further information about data visibility rules can be found in [**Section 46.4**](https://www.postgresql.org/docs/10/spi-visibility.html). The example in [**Section 38.4**](https://www.postgresql.org/docs/10/trigger-example.html) contains a demonstration of these rules.

## 38.3. Writing Trigger Functions in C

This section describes the low-level details of the interface to a trigger function. This information is only needed when writing trigger functions in C. If you are using a higher-level language then these details are handled for you. In most cases you should consider using a procedural language before writing your triggers in C. The documentation of each procedural language explains how to write a trigger in that language.

Trigger functions must use the “version 1” function manager interface.

When a function is called by the trigger manager, it is not passed any normal arguments, but it is passed a “context” pointer pointing to a TriggerData structure. C functions can check whether they were called from the trigger manager or not by executing the macro:

CALLED\_AS\_TRIGGER(fcinfo)

which expands to:

((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))

If this returns true, then it is safe to cast fcinfo->context to type TriggerData \* and make use of the pointed-to TriggerData structure. The function must not alter the TriggerData structure or any of the data it points to.

struct TriggerData is defined in commands/trigger.h:

typedef struct TriggerData

{

NodeTag type;

TriggerEvent tg\_event;

Relation tg\_relation;

HeapTuple tg\_trigtuple;

HeapTuple tg\_newtuple;

Trigger \*tg\_trigger;

Buffer tg\_trigtuplebuf;

Buffer tg\_newtuplebuf;

Tuplestorestate \*tg\_oldtable;

Tuplestorestate \*tg\_newtable;

} TriggerData;

where the members are defined as follows:

type

Always T\_TriggerData.

tg\_event

Describes the event for which the function is called. You can use the following macros to examine tg\_event:

TRIGGER\_FIRED\_BEFORE(tg\_event)

Returns true if the trigger fired before the operation.

TRIGGER\_FIRED\_AFTER(tg\_event)

Returns true if the trigger fired after the operation.

TRIGGER\_FIRED\_INSTEAD(tg\_event)

Returns true if the trigger fired instead of the operation.

TRIGGER\_FIRED\_FOR\_ROW(tg\_event)

Returns true if the trigger fired for a row-level event.

TRIGGER\_FIRED\_FOR\_STATEMENT(tg\_event)

Returns true if the trigger fired for a statement-level event.

TRIGGER\_FIRED\_BY\_INSERT(tg\_event)

Returns true if the trigger was fired by an INSERT command.

TRIGGER\_FIRED\_BY\_UPDATE(tg\_event)

Returns true if the trigger was fired by an UPDATE command.

TRIGGER\_FIRED\_BY\_DELETE(tg\_event)

Returns true if the trigger was fired by a DELETE command.

TRIGGER\_FIRED\_BY\_TRUNCATE(tg\_event)

Returns true if the trigger was fired by a TRUNCATE command.

tg\_relation

A pointer to a structure describing the relation that the trigger fired for. Look at utils/rel.h for details about this structure. The most interesting things are tg\_relation->rd\_att (descriptor of the relation tuples) and tg\_relation->rd\_rel->relname (relation name; the type is not char\* but NameData; use SPI\_getrelname(tg\_relation) to get a char\* if you need a copy of the name).

tg\_trigtuple

A pointer to the row for which the trigger was fired. This is the row being inserted, updated, or deleted. If this trigger was fired for an INSERT or DELETE then this is what you should return from the function if you don't want to replace the row with a different one (in the case of INSERT) or skip the operation. For triggers on foreign tables, values of system columns herein are unspecified.

tg\_newtuple

A pointer to the new version of the row, if the trigger was fired for an UPDATE, and NULL if it is for an INSERT or a DELETE. This is what you have to return from the function if the event is an UPDATE and you don't want to replace this row by a different one or skip the operation. For triggers on foreign tables, values of system columns herein are unspecified.

tg\_trigger

A pointer to a structure of type Trigger, defined in utils/reltrigger.h:

typedef struct Trigger

{

Oid tgoid;

char \*tgname;

Oid tgfoid;

int16 tgtype;

char tgenabled;

bool tgisinternal;

Oid tgconstrrelid;

Oid tgconstrindid;

Oid tgconstraint;

bool tgdeferrable;

bool tginitdeferred;

int16 tgnargs;

int16 tgnattr;

int16 \*tgattr;

char \*\*tgargs;

char \*tgqual;

char \*tgoldtable;

char \*tgnewtable;

} Trigger;

where tgname is the trigger's name, tgnargs is the number of arguments in tgargs, and tgargs is an array of pointers to the arguments specified in the CREATE TRIGGER statement. The other members are for internal use only.

tg\_trigtuplebuf

The buffer containing tg\_trigtuple, or InvalidBuffer if there is no such tuple or it is not stored in a disk buffer.

tg\_newtuplebuf

The buffer containing tg\_newtuple, or InvalidBuffer if there is no such tuple or it is not stored in a disk buffer.

tg\_oldtable

A pointer to a structure of type Tuplestorestate containing zero or more rows in the format specified by tg\_relation, or a NULL pointer if there is no OLD TABLE transition relation.

tg\_newtable

A pointer to a structure of type Tuplestorestate containing zero or more rows in the format specified by tg\_relation, or a NULL pointer if there is no NEW TABLE transition relation.

To allow queries issued through SPI to reference transition tables, see [**SPI\_register\_trigger\_data**](https://www.postgresql.org/docs/10/spi-spi-register-trigger-data.html).

A trigger function must return either a HeapTuple pointer or a NULL pointer (not an SQL null value, that is, do not set *isNull* true). Be careful to return either tg\_trigtuple or tg\_newtuple, as appropriate, if you don't want to modify the row being operated on.

## 38.4. A Complete Trigger Example

Here is a very simple example of a trigger function written in C. (Examples of triggers written in procedural languages can be found in the documentation of the procedural languages.)

The function trigf reports the number of rows in the table ttest and skips the actual operation if the command attempts to insert a null value into the column x. (So the trigger acts as a not-null constraint but doesn't abort the transaction.)

First, the table definition:

CREATE TABLE ttest (

x integer

);

This is the source code of the trigger function:

#include "postgres.h"

#include "fmgr.h"

#include "executor/spi.h" /\* this is what you need to work with SPI \*/

#include "commands/trigger.h" /\* ... triggers ... \*/

#include "utils/rel.h" /\* ... and relations \*/

PG\_MODULE\_MAGIC;

PG\_FUNCTION\_INFO\_V1(trigf);

Datum

trigf(PG\_FUNCTION\_ARGS)

{

TriggerData \*trigdata = (TriggerData \*) fcinfo->context;

TupleDesc tupdesc;

HeapTuple rettuple;

char \*when;

bool checknull = false;

bool isnull;

int ret, i;

/\* make sure it's called as a trigger at all \*/

if (!CALLED\_AS\_TRIGGER(fcinfo))

elog(ERROR, "trigf: not called by trigger manager");

/\* tuple to return to executor \*/

if (TRIGGER\_FIRED\_BY\_UPDATE(trigdata->tg\_event))

rettuple = trigdata->tg\_newtuple;

else

rettuple = trigdata->tg\_trigtuple;

/\* check for null values \*/

if (!TRIGGER\_FIRED\_BY\_DELETE(trigdata->tg\_event)

&& TRIGGER\_FIRED\_BEFORE(trigdata->tg\_event))

checknull = true;

if (TRIGGER\_FIRED\_BEFORE(trigdata->tg\_event))

when = "before";

else

when = "after ";

tupdesc = trigdata->tg\_relation->rd\_att;

/\* connect to SPI manager \*/

if ((ret = SPI\_connect()) < 0)

elog(ERROR, "trigf (fired %s): SPI\_connect returned %d", when, ret);

/\* get number of rows in table \*/

ret = SPI\_exec("SELECT count(\*) FROM ttest", 0);

if (ret < 0)

elog(ERROR, "trigf (fired %s): SPI\_exec returned %d", when, ret);

/\* count(\*) returns int8, so be careful to convert \*/

i = DatumGetInt64(SPI\_getbinval(SPI\_tuptable->vals[0],

SPI\_tuptable->tupdesc,

1,

&isnull));

elog (INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

SPI\_finish();

if (checknull)

{

SPI\_getbinval(rettuple, tupdesc, 1, &isnull);

if (isnull)

rettuple = NULL;

}

return PointerGetDatum(rettuple);

}

After you have compiled the source code (see [**Section 37.9.5**](https://www.postgresql.org/docs/10/xfunc-c.html#DFUNC)), declare the function and the triggers:

CREATE FUNCTION trigf() RETURNS trigger

AS '***filename***'

LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest

FOR EACH ROW EXECUTE PROCEDURE trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest

FOR EACH ROW EXECUTE PROCEDURE trigf();

Now you can test the operation of the trigger:

=> INSERT INTO ttest VALUES (NULL);

INFO: trigf (fired before): there are 0 rows in ttest

INSERT 0 0

-- Insertion skipped and AFTER trigger is not fired

=> SELECT \* FROM ttest;

x

---

(0 rows)

=> INSERT INTO ttest VALUES (1);

INFO: trigf (fired before): there are 0 rows in ttest

INFO: trigf (fired after ): there are 1 rows in ttest

^^^^^^^^

remember what we said about visibility.

INSERT 167793 1

vac=> SELECT \* FROM ttest;

x

---

1

(1 row)

=> INSERT INTO ttest SELECT x \* 2 FROM ttest;

INFO: trigf (fired before): there are 1 rows in ttest

INFO: trigf (fired after ): there are 2 rows in ttest

^^^^^^

remember what we said about visibility.

INSERT 167794 1

=> SELECT \* FROM ttest;

x

---

1

2

(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;

INFO: trigf (fired before): there are 2 rows in ttest

UPDATE 0

=> UPDATE ttest SET x = 4 WHERE x = 2;

INFO: trigf (fired before): there are 2 rows in ttest

INFO: trigf (fired after ): there are 2 rows in ttest

UPDATE 1

vac=> SELECT \* FROM ttest;

x

---

1

4

(2 rows)

=> DELETE FROM ttest;

INFO: trigf (fired before): there are 2 rows in ttest

INFO: trigf (fired before): there are 1 rows in ttest

INFO: trigf (fired after ): there are 0 rows in ttest

INFO: trigf (fired after ): there are 0 rows in ttest

^^^^^^

remember what we said about visibility.

DELETE 2

=> SELECT \* FROM ttest;

x

---

(0 rows)

There are more complex examples in src/test/regress/regress.c and in [**spi**](https://www.postgresql.org/docs/10/contrib-spi.html).

## Chapter 39. Event Triggers

To supplement the trigger mechanism discussed in [**Chapter 38**](https://www.postgresql.org/docs/10/triggers.html), PostgreSQL also provides event triggers. Unlike regular triggers, which are attached to a single table and capture only DML events, event triggers are global to a particular database and are capable of capturing DDL events.

Like regular triggers, event triggers can be written in any procedural language that includes event trigger support, or in C, but not in plain SQL.

## 39.1. Overview of Event Trigger Behavior

An event trigger fires whenever the event with which it is associated occurs in the database in which it is defined. Currently, the only supported events are ddl\_command\_start, ddl\_command\_end, table\_rewrite and sql\_drop. Support for additional events may be added in future releases.

The ddl\_command\_start event occurs just before the execution of a CREATE, ALTER, DROP, SECURITY LABEL, COMMENT, GRANT or REVOKE command. No check whether the affected object exists or doesn't exist is performed before the event trigger fires. As an exception, however, this event does not occur for DDL commands targeting shared objects — databases, roles, and tablespaces — or for commands targeting event triggers themselves. The event trigger mechanism does not support these object types. ddl\_command\_start also occurs just before the execution of a SELECT INTO command, since this is equivalent to CREATE TABLE AS.

The ddl\_command\_end event occurs just after the execution of this same set of commands. To obtain more details on the DDL operations that took place, use the set-returning function pg\_event\_trigger\_ddl\_commands() from the ddl\_command\_end event trigger code (see [**Section 9.28**](https://www.postgresql.org/docs/10/functions-event-triggers.html)). Note that the trigger fires after the actions have taken place (but before the transaction commits), and thus the system catalogs can be read as already changed.

The sql\_drop event occurs just before the ddl\_command\_end event trigger for any operation that drops database objects. To list the objects that have been dropped, use the set-returning function pg\_event\_trigger\_dropped\_objects() from the sql\_drop event trigger code (see [**Section 9.28**](https://www.postgresql.org/docs/10/functions-event-triggers.html)). Note that the trigger is executed after the objects have been deleted from the system catalogs, so it's not possible to look them up anymore.

The table\_rewrite event occurs just before a table is rewritten by some actions of the commands ALTER TABLE and ALTER TYPE. While other control statements are available to rewrite a table, like CLUSTER and VACUUM, the table\_rewrite event is not triggered by them.

Event triggers (like other functions) cannot be executed in an aborted transaction. Thus, if a DDL command fails with an error, any associated ddl\_command\_end triggers will not be executed. Conversely, if a ddl\_command\_start trigger fails with an error, no further event triggers will fire, and no attempt will be made to execute the command itself. Similarly, if a ddl\_command\_end trigger fails with an error, the effects of the DDL statement will be rolled back, just as they would be in any other case where the containing transaction aborts.

For a complete list of commands supported by the event trigger mechanism, see [**Section 39.2**](https://www.postgresql.org/docs/10/event-trigger-matrix.html).

Event triggers are created using the command [**CREATE EVENT TRIGGER**](https://www.postgresql.org/docs/10/sql-createeventtrigger.html). In order to create an event trigger, you must first create a function with the special return type event\_trigger. This function need not (and may not) return a value; the return type serves merely as a signal that the function is to be invoked as an event trigger.

If more than one event trigger is defined for a particular event, they will fire in alphabetical order by trigger name.

A trigger definition can also specify a WHEN condition so that, for example, a ddl\_command\_start trigger can be fired only for particular commands which the user wishes to intercept. A common use of such triggers is to restrict the range of DDL operations which users may perform.

## 39.2. Event Trigger Firing Matrix

[**Table 39.1**](https://www.postgresql.org/docs/10/event-trigger-matrix.html#EVENT-TRIGGER-BY-COMMAND-TAG) lists all commands for which event triggers are supported.

**Table 39.1. Event Trigger Support by Command Tag**

| **Command Tag** | ddl\_command\_start | ddl\_command\_end | sql\_drop | table\_rewrite | **Notes** |
| --- | --- | --- | --- | --- | --- |
| ALTER AGGREGATE | X | X | - | - |  |
| ALTER COLLATION | X | X | - | - |  |
| ALTER CONVERSION | X | X | - | - |  |
| ALTER DOMAIN | X | X | - | - |  |
| ALTER DEFAULT PRIVILEGES | X | X | - | - |  |
| ALTER EXTENSION | X | X | - | - |  |
| ALTER FOREIGN DATA WRAPPER | X | X | - | - |  |
| ALTER FOREIGN TABLE | X | X | X | - |  |
| ALTER FUNCTION | X | X | - | - |  |
| ALTER LANGUAGE | X | X | - | - |  |
| ALTER LARGE OBJECT | X | X | - | - |  |
| ALTER MATERIALIZED VIEW | X | X | - | - |  |
| ALTER OPERATOR | X | X | - | - |  |
| ALTER OPERATOR CLASS | X | X | - | - |  |
| ALTER OPERATOR FAMILY | X | X | - | - |  |
| ALTER POLICY | X | X | - | - |  |
| ALTER PUBLICATION | X | X | - | - |  |
| ALTER SCHEMA | X | X | - | - |  |
| ALTER SEQUENCE | X | X | - | - |  |
| ALTER SERVER | X | X | - | - |  |
| ALTER STATISTICS | X | X | - | - |  |
| ALTER SUBSCRIPTION | X | X | - | - |  |
| ALTER TABLE | X | X | X | X |  |
| ALTER TEXT SEARCH CONFIGURATION | X | X | - | - |  |
| ALTER TEXT SEARCH DICTIONARY | X | X | - | - |  |
| ALTER TEXT SEARCH PARSER | X | X | - | - |  |
| ALTER TEXT SEARCH TEMPLATE | X | X | - | - |  |
| ALTER TRIGGER | X | X | - | - |  |
| ALTER TYPE | X | X | - | X |  |
| ALTER USER MAPPING | X | X | - | - |  |
| ALTER VIEW | X | X | - | - |  |
| COMMENT | X | X | - | - | Only for local objects |
| CREATE ACCESS METHOD | X | X | - | - |  |
| CREATE AGGREGATE | X | X | - | - |  |
| CREATE CAST | X | X | - | - |  |
| CREATE COLLATION | X | X | - | - |  |
| CREATE CONVERSION | X | X | - | - |  |
| CREATE DOMAIN | X | X | - | - |  |
| CREATE EXTENSION | X | X | - | - |  |
| CREATE FOREIGN DATA WRAPPER | X | X | - | - |  |
| CREATE FOREIGN TABLE | X | X | - | - |  |
| CREATE FUNCTION | X | X | - | - |  |
| CREATE INDEX | X | X | - | - |  |
| CREATE LANGUAGE | X | X | - | - |  |
| CREATE MATERIALIZED VIEW | X | X | - | - |  |
| CREATE OPERATOR | X | X | - | - |  |
| CREATE OPERATOR CLASS | X | X | - | - |  |
| CREATE OPERATOR FAMILY | X | X | - | - |  |
| CREATE POLICY | X | X | - | - |  |
| CREATE PUBLICATION | X | X | - | - |  |
| CREATE RULE | X | X | - | - |  |
| CREATE SCHEMA | X | X | - | - |  |
| CREATE SEQUENCE | X | X | - | - |  |
| CREATE SERVER | X | X | - | - |  |
| CREATE STATISTICS | X | X | - | - |  |
| CREATE SUBSCRIPTION | X | X | - | - |  |
| CREATE TABLE | X | X | - | - |  |
| CREATE TABLE AS | X | X | - | - |  |
| CREATE TEXT SEARCH CONFIGURATION | X | X | - | - |  |
| CREATE TEXT SEARCH DICTIONARY | X | X | - | - |  |
| CREATE TEXT SEARCH PARSER | X | X | - | - |  |
| CREATE TEXT SEARCH TEMPLATE | X | X | - | - |  |
| CREATE TRIGGER | X | X | - | - |  |
| CREATE TYPE | X | X | - | - |  |
| CREATE USER MAPPING | X | X | - | - |  |
| CREATE VIEW | X | X | - | - |  |
| DROP ACCESS METHOD | X | X | X | - |  |
| DROP AGGREGATE | X | X | X | - |  |
| DROP CAST | X | X | X | - |  |
| DROP COLLATION | X | X | X | - |  |
| DROP CONVERSION | X | X | X | - |  |
| DROP DOMAIN | X | X | X | - |  |
| DROP EXTENSION | X | X | X | - |  |
| DROP FOREIGN DATA WRAPPER | X | X | X | - |  |
| DROP FOREIGN TABLE | X | X | X | - |  |
| DROP FUNCTION | X | X | X | - |  |
| DROP INDEX | X | X | X | - |  |
| DROP LANGUAGE | X | X | X | - |  |
| DROP MATERIALIZED VIEW | X | X | X | - |  |
| DROP OPERATOR | X | X | X | - |  |
| DROP OPERATOR CLASS | X | X | X | - |  |
| DROP OPERATOR FAMILY | X | X | X | - |  |
| DROP OWNED | X | X | X | - |  |
| DROP POLICY | X | X | X | - |  |
| DROP PUBLICATION | X | X | X | - |  |
| DROP RULE | X | X | X | - |  |
| DROP SCHEMA | X | X | X | - |  |
| DROP SEQUENCE | X | X | X | - |  |
| DROP SERVER | X | X | X | - |  |
| DROP STATISTICS | X | X | X | - |  |
| DROP SUBSCRIPTION | X | X | X | - |  |
| DROP TABLE | X | X | X | - |  |
| DROP TEXT SEARCH CONFIGURATION | X | X | X | - |  |
| DROP TEXT SEARCH DICTIONARY | X | X | X | - |  |
| DROP TEXT SEARCH PARSER | X | X | X | - |  |
| DROP TEXT SEARCH TEMPLATE | X | X | X | - |  |
| DROP TRIGGER | X | X | X | - |  |
| DROP TYPE | X | X | X | - |  |
| DROP USER MAPPING | X | X | X | - |  |
| DROP VIEW | X | X | X | - |  |
| GRANT | X | X | - | - | Only for local objects |
| IMPORT FOREIGN SCHEMA | X | X | - | - |  |
| REFRESH MATERIALIZED VIEW | X | X | - | - |  |
| REVOKE | X | X | - | - | Only for local objects |
| SECURITY LABEL | X | X | - | - | Only for local objects |
| SELECT INTO | X | X | - | - |  |

## 39.3. Writing Event Trigger Functions in C

This section describes the low-level details of the interface to an event trigger function. This information is only needed when writing event trigger functions in C. If you are using a higher-level language then these details are handled for you. In most cases you should consider using a procedural language before writing your event triggers in C. The documentation of each procedural language explains how to write an event trigger in that language.

Event trigger functions must use the “version 1” function manager interface.

When a function is called by the event trigger manager, it is not passed any normal arguments, but it is passed a “context” pointer pointing to a EventTriggerData structure. C functions can check whether they were called from the event trigger manager or not by executing the macro:

CALLED\_AS\_EVENT\_TRIGGER(fcinfo)

which expands to:

((fcinfo)->context != NULL && IsA((fcinfo)->context, EventTriggerData))

If this returns true, then it is safe to cast fcinfo->context to type EventTriggerData \* and make use of the pointed-to EventTriggerData structure. The function must not alter the EventTriggerDatastructure or any of the data it points to.

struct EventTriggerData is defined in commands/event\_trigger.h:

typedef struct EventTriggerData

{

NodeTag type;

const char \*event; /\* event name \*/

Node \*parsetree; /\* parse tree \*/

const char \*tag; /\* command tag \*/

} EventTriggerData;

where the members are defined as follows:

type

Always T\_EventTriggerData.

event

Describes the event for which the function is called, one of "ddl\_command\_start", "ddl\_command\_end", "sql\_drop", "table\_rewrite". See [**Section 39.1**](https://www.postgresql.org/docs/10/event-trigger-definition.html) for the meaning of these events.

parsetree

A pointer to the parse tree of the command. Check the PostgreSQL source code for details. The parse tree structure is subject to change without notice.

tag

The command tag associated with the event for which the event trigger is run, for example "CREATE FUNCTION".

An event trigger function must return a NULL pointer (not an SQL null value, that is, do not set *isNull* true).

## 39.4. A Complete Event Trigger Example

Here is a very simple example of an event trigger function written in C. (Examples of triggers written in procedural languages can be found in the documentation of the procedural languages.)

The function noddl raises an exception each time it is called. The event trigger definition associated the function with the ddl\_command\_start event. The effect is that all DDL commands (with the exceptions mentioned in [**Section 39.1**](https://www.postgresql.org/docs/10/event-trigger-definition.html)) are prevented from running.

This is the source code of the trigger function:

#include "postgres.h"

#include "commands/event\_trigger.h"

PG\_MODULE\_MAGIC;

PG\_FUNCTION\_INFO\_V1(noddl);

Datum

noddl(PG\_FUNCTION\_ARGS)

{

EventTriggerData \*trigdata;

if (!CALLED\_AS\_EVENT\_TRIGGER(fcinfo)) /\* internal error \*/

elog(ERROR, "not fired by event trigger manager");

trigdata = (EventTriggerData \*) fcinfo->context;

ereport(ERROR,

(errcode(ERRCODE\_INSUFFICIENT\_PRIVILEGE),

errmsg("command \"%s\" denied", trigdata->tag)));

PG\_RETURN\_NULL();

}

After you have compiled the source code (see [**Section 37.9.5**](https://www.postgresql.org/docs/10/xfunc-c.html#DFUNC)), declare the function and the triggers:

CREATE FUNCTION noddl() RETURNS event\_trigger

AS 'noddl' LANGUAGE C;

CREATE EVENT TRIGGER noddl ON ddl\_command\_start

EXECUTE PROCEDURE noddl();

Now you can test the operation of the trigger:

=# \dy

List of event triggers

Name | Event | Owner | Enabled | Procedure | Tags

-------+-------------------+-------+---------+-----------+------

noddl | ddl\_command\_start | dim | enabled | noddl |

(1 row)

=# CREATE TABLE foo(id serial);

ERROR: command "CREATE TABLE" denied

In this situation, in order to be able to run some DDL commands when you need to do so, you have to either drop the event trigger or disable it. It can be convenient to disable the trigger for only the duration of a transaction:

BEGIN;

ALTER EVENT TRIGGER noddl DISABLE;

CREATE TABLE foo (id serial);

ALTER EVENT TRIGGER noddl ENABLE;

COMMIT;

(Recall that DDL commands on event triggers themselves are not affected by event triggers.)

## 39.5. A Table Rewrite Event Trigger Example

Thanks to the table\_rewrite event, it is possible to implement a table rewriting policy only allowing the rewrite in maintenance windows.

Here's an example implementing such a policy.

CREATE OR REPLACE FUNCTION no\_rewrite()

RETURNS event\_trigger

LANGUAGE plpgsql AS

$$

---

--- Implement local Table Rewriting policy:

--- public.foo is not allowed rewriting, ever

--- other tables are only allowed rewriting between 1am and 6am

--- unless they have more than 100 blocks

---

DECLARE

table\_oid oid := pg\_event\_trigger\_table\_rewrite\_oid();

current\_hour integer := extract('hour' from current\_time);

pages integer;

max\_pages integer := 100;

BEGIN

IF pg\_event\_trigger\_table\_rewrite\_oid() = 'public.foo'::regclass

THEN

RAISE EXCEPTION 'you''re not allowed to rewrite the table %',

table\_oid::regclass;

END IF;

SELECT INTO pages relpages FROM pg\_class WHERE oid = table\_oid;

IF pages > max\_pages

THEN

RAISE EXCEPTION 'rewrites only allowed for table with less than % pages',

max\_pages;

END IF;

IF current\_hour NOT BETWEEN 1 AND 6

THEN

RAISE EXCEPTION 'rewrites only allowed between 1am and 6am';

END IF;

END;

$$;

CREATE EVENT TRIGGER no\_rewrite\_allowed

ON table\_rewrite

EXECUTE PROCEDURE no\_rewrite();

## Chapter 40. The Rule System

This chapter discusses the rule system in PostgreSQL. Production rule systems are conceptually simple, but there are many subtle points involved in actually using them.

Some other database systems define active database rules, which are usually stored procedures and triggers. In PostgreSQL, these can be implemented using functions and triggers as well.

The rule system (more precisely speaking, the query rewrite rule system) is totally different from stored procedures and triggers. It modifies queries to take rules into consideration, and then passes the modified query to the query planner for planning and execution. It is very powerful, and can be used for many things such as query language procedures, views, and versions. The theoretical foundations and the power of this rule system are also discussed in [**[ston90b]**](https://www.postgresql.org/docs/10/biblio.html#STON90B) and [**[ong90]**](https://www.postgresql.org/docs/10/biblio.html#ONG90).

## 40.1. The Query Tree

To understand how the rule system works it is necessary to know when it is invoked and what its input and results are.

The rule system is located between the parser and the planner. It takes the output of the parser, one query tree, and the user-defined rewrite rules, which are also query trees with some extra information, and creates zero or more query trees as result. So its input and output are always things the parser itself could have produced and thus, anything it sees is basically representable as an SQL statement.

Now what is a query tree? It is an internal representation of an SQL statement where the single parts that it is built from are stored separately. These query trees can be shown in the server log if you set the configuration parameters debug\_print\_parse, debug\_print\_rewritten, or debug\_print\_plan. The rule actions are also stored as query trees, in the system catalog pg\_rewrite. They are not formatted like the log output, but they contain exactly the same information.

Reading a raw query tree requires some experience. But since SQL representations of query trees are sufficient to understand the rule system, this chapter will not teach how to read them.

When reading the SQL representations of the query trees in this chapter it is necessary to be able to identify the parts the statement is broken into when it is in the query tree structure. The parts of a query tree are

the command type

This is a simple value telling which command (SELECT, INSERT, UPDATE, DELETE) produced the query tree.

the range table

The range table is a list of relations that are used in the query. In a SELECT statement these are the relations given after the FROM key word.

Every range table entry identifies a table or view and tells by which name it is called in the other parts of the query. In the query tree, the range table entries are referenced by number rather than by name, so here it doesn't matter if there are duplicate names as it would in an SQL statement. This can happen after the range tables of rules have been merged in. The examples in this chapter will not have this situation.

the result relation

This is an index into the range table that identifies the relation where the results of the query go.

SELECT queries don't have a result relation. (The special case of SELECT INTO is mostly identical to CREATE TABLE followed by INSERT ... SELECT, and is not discussed separately here.)

For INSERT, UPDATE, and DELETE commands, the result relation is the table (or view!) where the changes are to take effect.

the target list

The target list is a list of expressions that define the result of the query. In the case of a SELECT, these expressions are the ones that build the final output of the query. They correspond to the expressions between the key words SELECT and FROM. (\* is just an abbreviation for all the column names of a relation. It is expanded by the parser into the individual columns, so the rule system never sees it.)

DELETE commands don't need a normal target list because they don't produce any result. Instead, the planner adds a special CTID entry to the empty target list, to allow the executor to find the row to be deleted. (CTID is added when the result relation is an ordinary table. If it is a view, a whole-row variable is added instead, by the rule system, as described in [**Section 40.2.4**](https://www.postgresql.org/docs/10/rules-views.html#RULES-VIEWS-UPDATE).)

For INSERT commands, the target list describes the new rows that should go into the result relation. It consists of the expressions in the VALUES clause or the ones from the SELECT clause in INSERT ... SELECT. The first step of the rewrite process adds target list entries for any columns that were not assigned to by the original command but have defaults. Any remaining columns (with neither a given value nor a default) will be filled in by the planner with a constant null expression.

For UPDATE commands, the target list describes the new rows that should replace the old ones. In the rule system, it contains just the expressions from the SET column = expression part of the command. The planner will handle missing columns by inserting expressions that copy the values from the old row into the new one. Just as for DELETE, a CTID or whole-row variable is added so that the executor can identify the old row to be updated.

Every entry in the target list contains an expression that can be a constant value, a variable pointing to a column of one of the relations in the range table, a parameter, or an expression tree made of function calls, constants, variables, operators, etc.

the qualification

The query's qualification is an expression much like one of those contained in the target list entries. The result value of this expression is a Boolean that tells whether the operation (INSERT, UPDATE, DELETE, or SELECT) for the final result row should be executed or not. It corresponds to the WHERE clause of an SQL statement.

the join tree

The query's join tree shows the structure of the FROM clause. For a simple query like SELECT ... FROM a, b, c, the join tree is just a list of the FROM items, because we are allowed to join them in any order. But when JOIN expressions, particularly outer joins, are used, we have to join in the order shown by the joins. In that case, the join tree shows the structure of the JOINexpressions. The restrictions associated with particular JOIN clauses (from ON or USING expressions) are stored as qualification expressions attached to those join-tree nodes. It turns out to be convenient to store the top-level WHERE expression as a qualification attached to the top-level join-tree item, too. So really the join tree represents both the FROM and WHERE clauses of a SELECT.

the others

The other parts of the query tree like the ORDER BY clause aren't of interest here. The rule system substitutes some entries there while applying rules, but that doesn't have much to do with the fundamentals of the rule system.

## 40.2. Views and the Rule System

Views in PostgreSQL are implemented using the rule system. In fact, there is essentially no difference between:

CREATE VIEW myview AS SELECT \* FROM mytab;

compared against the two commands:

CREATE TABLE myview (***same column list as mytab***);

CREATE RULE "\_RETURN" AS ON SELECT TO myview DO INSTEAD

SELECT \* FROM mytab;

because this is exactly what the CREATE VIEW command does internally. This has some side effects. One of them is that the information about a view in the PostgreSQL system catalogs is exactly the same as it is for a table. So for the parser, there is absolutely no difference between a table and a view. They are the same thing: relations.

### 40.2.1. How SELECT Rules Work

Rules ON SELECT are applied to all queries as the last step, even if the command given is an INSERT, UPDATE or DELETE. And they have different semantics from rules on the other command types in that they modify the query tree in place instead of creating a new one. So SELECT rules are described first.

Currently, there can be only one action in an ON SELECT rule, and it must be an unconditional SELECT action that is INSTEAD. This restriction was required to make rules safe enough to open them for ordinary users, and it restricts ON SELECT rules to act like views.

The examples for this chapter are two join views that do some calculations and some more views using them in turn. One of the two first views is customized later by adding rules for INSERT, UPDATE, and DELETE operations so that the final result will be a view that behaves like a real table with some magic functionality. This is not such a simple example to start from and this makes things harder to get into. But it's better to have one example that covers all the points discussed step by step rather than having many different ones that might mix up in mind.

For the example, we need a little min function that returns the lower of 2 integer values. We create that as:

CREATE FUNCTION min(integer, integer) RETURNS integer AS $$

SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END

$$ LANGUAGE SQL STRICT;

The real tables we need in the first two rule system descriptions are these:

CREATE TABLE shoe\_data (

shoename text, -- primary key

sh\_avail integer, -- available number of pairs

slcolor text, -- preferred shoelace color

slminlen real, -- minimum shoelace length

slmaxlen real, -- maximum shoelace length

slunit text -- length unit

);

CREATE TABLE shoelace\_data (

sl\_name text, -- primary key

sl\_avail integer, -- available number of pairs

sl\_color text, -- shoelace color

sl\_len real, -- shoelace length

sl\_unit text -- length unit

);

CREATE TABLE unit (

un\_name text, -- primary key

un\_fact real -- factor to transform to cm

);

As you can see, they represent shoe-store data.

The views are created as:

CREATE VIEW shoe AS

SELECT sh.shoename,

sh.sh\_avail,

sh.slcolor,

sh.slminlen,

sh.slminlen \* un.un\_fact AS slminlen\_cm,

sh.slmaxlen,

sh.slmaxlen \* un.un\_fact AS slmaxlen\_cm,

sh.slunit

FROM shoe\_data sh, unit un

WHERE sh.slunit = un.un\_name;

CREATE VIEW shoelace AS

SELECT s.sl\_name,

s.sl\_avail,

s.sl\_color,

s.sl\_len,

s.sl\_unit,

s.sl\_len \* u.un\_fact AS sl\_len\_cm

FROM shoelace\_data s, unit u

WHERE s.sl\_unit = u.un\_name;

CREATE VIEW shoe\_ready AS

SELECT rsh.shoename,

rsh.sh\_avail,

rsl.sl\_name,

rsl.sl\_avail,

min(rsh.sh\_avail, rsl.sl\_avail) AS total\_avail

FROM shoe rsh, shoelace rsl

WHERE rsl.sl\_color = rsh.slcolor

AND rsl.sl\_len\_cm >= rsh.slminlen\_cm

AND rsl.sl\_len\_cm <= rsh.slmaxlen\_cm;

The CREATE VIEW command for the shoelace view (which is the simplest one we have) will create a relation shoelace and an entry in pg\_rewrite that tells that there is a rewrite rule that must be applied whenever the relation shoelace is referenced in a query's range table. The rule has no rule qualification (discussed later, with the non-SELECT rules, since SELECT rules currently cannot have them) and it is INSTEAD. Note that rule qualifications are not the same as query qualifications. The action of our rule has a query qualification. The action of the rule is one query tree that is a copy of the SELECTstatement in the view creation command.

### Note

The two extra range table entries for NEW and OLD that you can see in the pg\_rewrite entry aren't of interest for SELECT rules.

Now we populate unit, shoe\_data and shoelace\_data and run a simple query on a view:

INSERT INTO unit VALUES ('cm', 1.0);

INSERT INTO unit VALUES ('m', 100.0);

INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe\_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');

INSERT INTO shoe\_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');

INSERT INTO shoe\_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');

INSERT INTO shoe\_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace\_data VALUES ('sl1', 5, 'black', 80.0, 'cm');

INSERT INTO shoelace\_data VALUES ('sl2', 6, 'black', 100.0, 'cm');

INSERT INTO shoelace\_data VALUES ('sl3', 0, 'black', 35.0 , 'inch');

INSERT INTO shoelace\_data VALUES ('sl4', 8, 'black', 40.0 , 'inch');

INSERT INTO shoelace\_data VALUES ('sl5', 4, 'brown', 1.0 , 'm');

INSERT INTO shoelace\_data VALUES ('sl6', 0, 'brown', 0.9 , 'm');

INSERT INTO shoelace\_data VALUES ('sl7', 7, 'brown', 60 , 'cm');

INSERT INTO shoelace\_data VALUES ('sl8', 1, 'brown', 40 , 'inch');

SELECT \* FROM shoelace;

sl\_name | sl\_avail | sl\_color | sl\_len | sl\_unit | sl\_len\_cm

-----------+----------+----------+--------+---------+-----------

sl1 | 5 | black | 80 | cm | 80

sl2 | 6 | black | 100 | cm | 100

sl7 | 7 | brown | 60 | cm | 60

sl3 | 0 | black | 35 | inch | 88.9

sl4 | 8 | black | 40 | inch | 101.6

sl8 | 1 | brown | 40 | inch | 101.6

sl5 | 4 | brown | 1 | m | 100

sl6 | 0 | brown | 0.9 | m | 90

(8 rows)

This is the simplest SELECT you can do on our views, so we take this opportunity to explain the basics of view rules. The SELECT \* FROM shoelace was interpreted by the parser and produced the query tree:

SELECT shoelace.sl\_name, shoelace.sl\_avail,

shoelace.sl\_color, shoelace.sl\_len,

shoelace.sl\_unit, shoelace.sl\_len\_cm

FROM shoelace shoelace;

and this is given to the rule system. The rule system walks through the range table and checks if there are rules for any relation. When processing the range table entry for shoelace (the only one up to now) it finds the \_RETURN rule with the query tree:

SELECT s.sl\_name, s.sl\_avail,

s.sl\_color, s.sl\_len, s.sl\_unit,

s.sl\_len \* u.un\_fact AS sl\_len\_cm

FROM shoelace old, shoelace new,

shoelace\_data s, unit u

WHERE s.sl\_unit = u.un\_name;

To expand the view, the rewriter simply creates a subquery range-table entry containing the rule's action query tree, and substitutes this range table entry for the original one that referenced the view. The resulting rewritten query tree is almost the same as if you had typed:

SELECT shoelace.sl\_name, shoelace.sl\_avail,

shoelace.sl\_color, shoelace.sl\_len,

shoelace.sl\_unit, shoelace.sl\_len\_cm

FROM (SELECT s.sl\_name,

s.sl\_avail,

s.sl\_color,

s.sl\_len,

s.sl\_unit,

s.sl\_len \* u.un\_fact AS sl\_len\_cm

FROM shoelace\_data s, unit u

WHERE s.sl\_unit = u.un\_name) shoelace;

There is one difference however: the subquery's range table has two extra entries shoelace old and shoelace new. These entries don't participate directly in the query, since they aren't referenced by the subquery's join tree or target list. The rewriter uses them to store the access privilege check information that was originally present in the range-table entry that referenced the view. In this way, the executor will still check that the user has proper privileges to access the view, even though there's no direct use of the view in the rewritten query.

That was the first rule applied. The rule system will continue checking the remaining range-table entries in the top query (in this example there are no more), and it will recursively check the range-table entries in the added subquery to see if any of them reference views. (But it won't expand old or new — otherwise we'd have infinite recursion!) In this example, there are no rewrite rules for shoelace\_data or unit, so rewriting is complete and the above is the final result given to the planner.

Now we want to write a query that finds out for which shoes currently in the store we have the matching shoelaces (color and length) and where the total number of exactly matching pairs is greater or equal to two.

SELECT \* FROM shoe\_ready WHERE total\_avail >= 2;

shoename | sh\_avail | sl\_name | sl\_avail | total\_avail

----------+----------+---------+----------+-------------

sh1 | 2 | sl1 | 5 | 2

sh3 | 4 | sl7 | 7 | 4

(2 rows)

The output of the parser this time is the query tree:

SELECT shoe\_ready.shoename, shoe\_ready.sh\_avail,

shoe\_ready.sl\_name, shoe\_ready.sl\_avail,

shoe\_ready.total\_avail

FROM shoe\_ready shoe\_ready

WHERE shoe\_ready.total\_avail >= 2;

The first rule applied will be the one for the shoe\_ready view and it results in the query tree:

SELECT shoe\_ready.shoename, shoe\_ready.sh\_avail,

shoe\_ready.sl\_name, shoe\_ready.sl\_avail,

shoe\_ready.total\_avail

FROM (SELECT rsh.shoename,

rsh.sh\_avail,

rsl.sl\_name,

rsl.sl\_avail,

min(rsh.sh\_avail, rsl.sl\_avail) AS total\_avail

FROM shoe rsh, shoelace rsl

WHERE rsl.sl\_color = rsh.slcolor

AND rsl.sl\_len\_cm >= rsh.slminlen\_cm

AND rsl.sl\_len\_cm <= rsh.slmaxlen\_cm) shoe\_ready

WHERE shoe\_ready.total\_avail >= 2;

Similarly, the rules for shoe and shoelace are substituted into the range table of the subquery, leading to a three-level final query tree:

SELECT shoe\_ready.shoename, shoe\_ready.sh\_avail,

shoe\_ready.sl\_name, shoe\_ready.sl\_avail,

shoe\_ready.total\_avail

FROM (SELECT rsh.shoename,

rsh.sh\_avail,

rsl.sl\_name,

rsl.sl\_avail,

min(rsh.sh\_avail, rsl.sl\_avail) AS total\_avail

FROM (SELECT sh.shoename,

sh.sh\_avail,

sh.slcolor,

sh.slminlen,

sh.slminlen \* un.un\_fact AS slminlen\_cm,

sh.slmaxlen,

sh.slmaxlen \* un.un\_fact AS slmaxlen\_cm,

sh.slunit

FROM shoe\_data sh, unit un

WHERE sh.slunit = un.un\_name) rsh,

(SELECT s.sl\_name,

s.sl\_avail,

s.sl\_color,

s.sl\_len,

s.sl\_unit,

s.sl\_len \* u.un\_fact AS sl\_len\_cm

FROM shoelace\_data s, unit u

WHERE s.sl\_unit = u.un\_name) rsl

WHERE rsl.sl\_color = rsh.slcolor

AND rsl.sl\_len\_cm >= rsh.slminlen\_cm

AND rsl.sl\_len\_cm <= rsh.slmaxlen\_cm) shoe\_ready

WHERE shoe\_ready.total\_avail > 2;

It turns out that the planner will collapse this tree into a two-level query tree: the bottommost SELECT commands will be “pulled up” into the middle SELECT since there's no need to process them separately. But the middle SELECT will remain separate from the top, because it contains aggregate functions. If we pulled those up it would change the behavior of the topmost SELECT, which we don't want. However, collapsing the query tree is an optimization that the rewrite system doesn't have to concern itself with.

### 40.2.2. View Rules in Non-SELECT Statements

Two details of the query tree aren't touched in the description of view rules above. These are the command type and the result relation. In fact, the command type is not needed by view rules, but the result relation may affect the way in which the query rewriter works, because special care needs to be taken if the result relation is a view.

There are only a few differences between a query tree for a SELECT and one for any other command. Obviously, they have a different command type and for a command other than a SELECT, the result relation points to the range-table entry where the result should go. Everything else is absolutely the same. So having two tables t1 and t2 with columns a and b, the query trees for the two statements:

SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;

are nearly identical. In particular:

* The range tables contain entries for the tables t1 and t2.
* The target lists contain one variable that points to column b of the range table entry for table t2.
* The qualification expressions compare the columns a of both range-table entries for equality.
* The join trees show a simple join between t1 and t2.

The consequence is, that both query trees result in similar execution plans: They are both joins over the two tables. For the UPDATE the missing columns from t1 are added to the target list by the planner and the final query tree will read as:

UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;

and thus the executor run over the join will produce exactly the same result set as:

SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;

But there is a little problem in UPDATE: the part of the executor plan that does the join does not care what the results from the join are meant for. It just produces a result set of rows. The fact that one is a SELECT command and the other is an UPDATE is handled higher up in the executor, where it knows that this is an UPDATE, and it knows that this result should go into table t1. But which of the rows that are there has to be replaced by the new row?

To resolve this problem, another entry is added to the target list in UPDATE (and also in DELETE) statements: the current tuple ID (CTID). This is a system column containing the file block number and position in the block for the row. Knowing the table, the CTID can be used to retrieve the original row of t1 to be updated. After adding the CTID to the target list, the query actually looks like:

SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;

Now another detail of PostgreSQL enters the stage. Old table rows aren't overwritten, and this is why ROLLBACK is fast. In an UPDATE, the new result row is inserted into the table (after stripping the CTID) and in the row header of the old row, which the CTID pointed to, the cmax and xmax entries are set to the current command counter and current transaction ID. Thus the old row is hidden, and after the transaction commits the vacuum cleaner can eventually remove the dead row.

Knowing all that, we can simply apply view rules in absolutely the same way to any command. There is no difference.

### 40.2.3. The Power of Views in PostgreSQL

The above demonstrates how the rule system incorporates view definitions into the original query tree. In the second example, a simple SELECT from one view created a final query tree that is a join of 4 tables (unit was used twice with different names).

The benefit of implementing views with the rule system is, that the planner has all the information about which tables have to be scanned plus the relationships between these tables plus the restrictive qualifications from the views plus the qualifications from the original query in one single query tree. And this is still the situation when the original query is already a join over views. The planner has to decide which is the best path to execute the query, and the more information the planner has, the better this decision can be. And the rule system as implemented in PostgreSQLensures, that this is all information available about the query up to that point.

### 40.2.4. Updating a View

What happens if a view is named as the target relation for an INSERT, UPDATE, or DELETE? Doing the substitutions described above would give a query tree in which the result relation points at a subquery range-table entry, which will not work. There are several ways in which PostgreSQL can support the appearance of updating a view, however.

If the subquery selects from a single base relation and is simple enough, the rewriter can automatically replace the subquery with the underlying base relation so that the INSERT, UPDATE, or DELETE is applied to the base relation in the appropriate way. Views that are “simple enough” for this are called automatically updatable. For detailed information on the kinds of view that can be automatically updated, see [**CREATE VIEW**](https://www.postgresql.org/docs/10/sql-createview.html).

Alternatively, the operation may be handled by a user-provided INSTEAD OF trigger on the view. Rewriting works slightly differently in this case. For INSERT, the rewriter does nothing at all with the view, leaving it as the result relation for the query. For UPDATE and DELETE, it's still necessary to expand the view query to produce the “old” rows that the command will attempt to update or delete. So the view is expanded as normal, but another unexpanded range-table entry is added to the query to represent the view in its capacity as the result relation.

The problem that now arises is how to identify the rows to be updated in the view. Recall that when the result relation is a table, a special CTID entry is added to the target list to identify the physical locations of the rows to be updated. This does not work if the result relation is a view, because a view does not have any CTID, since its rows do not have actual physical locations. Instead, for an UPDATE or DELETE operation, a special wholerow entry is added to the target list, which expands to include all columns from the view. The executor uses this value to supply the “old” row to the INSTEAD OF trigger. It is up to the trigger to work out what to update based on the old and new row values.

Another possibility is for the user to define INSTEAD rules that specify substitute actions for INSERT, UPDATE, and DELETE commands on a view. These rules will rewrite the command, typically into a command that updates one or more tables, rather than views. That is the topic of [**Section 40.4**](https://www.postgresql.org/docs/10/rules-update.html).

Note that rules are evaluated first, rewriting the original query before it is planned and executed. Therefore, if a view has INSTEAD OF triggers as well as rules on INSERT, UPDATE, or DELETE, then the rules will be evaluated first, and depending on the result, the triggers may not be used at all.

Automatic rewriting of an INSERT, UPDATE, or DELETE query on a simple view is always tried last. Therefore, if a view has rules or triggers, they will override the default behavior of automatically updatable views.

If there are no INSTEAD rules or INSTEAD OF triggers for the view, and the rewriter cannot automatically rewrite the query as an update on the underlying base relation, an error will be thrown because the executor cannot update a view as such.

## 40.3. Materialized Views

Materialized views in PostgreSQL use the rule system like views do, but persist the results in a table-like form. The main differences between:

CREATE MATERIALIZED VIEW mymatview AS SELECT \* FROM mytab;

and:

CREATE TABLE mymatview AS SELECT \* FROM mytab;

are that the materialized view cannot subsequently be directly updated and that the query used to create the materialized view is stored in exactly the same way that a view's query is stored, so that fresh data can be generated for the materialized view with:

REFRESH MATERIALIZED VIEW mymatview;

The information about a materialized view in the PostgreSQL system catalogs is exactly the same as it is for a table or view. So for the parser, a materialized view is a relation, just like a table or a view. When a materialized view is referenced in a query, the data is returned directly from the materialized view, like from a table; the rule is only used for populating the materialized view.

While access to the data stored in a materialized view is often much faster than accessing the underlying tables directly or through a view, the data is not always current; yet sometimes current data is not needed. Consider a table which records sales:

CREATE TABLE invoice (

invoice\_no integer PRIMARY KEY,

seller\_no integer, -- ID of salesperson

invoice\_date date, -- date of sale

invoice\_amt numeric(13,2) -- amount of sale

);

If people want to be able to quickly graph historical sales data, they might want to summarize, and they may not care about the incomplete data for the current date:

CREATE MATERIALIZED VIEW sales\_summary AS

SELECT

seller\_no,

invoice\_date,

sum(invoice\_amt)::numeric(13,2) as sales\_amt

FROM invoice

WHERE invoice\_date < CURRENT\_DATE

GROUP BY

seller\_no,

invoice\_date

ORDER BY

seller\_no,

invoice\_date;

CREATE UNIQUE INDEX sales\_summary\_seller

ON sales\_summary (seller\_no, invoice\_date);

This materialized view might be useful for displaying a graph in the dashboard created for salespeople. A job could be scheduled to update the statistics each night using this SQL statement:

REFRESH MATERIALIZED VIEW sales\_summary;

Another use for a materialized view is to allow faster access to data brought across from a remote system through a foreign data wrapper. A simple example using file\_fdw is below, with timings, but since this is using cache on the local system the performance difference compared to access to a remote system would usually be greater than shown here. Notice we are also exploiting the ability to put an index on the materialized view, whereas file\_fdw does not support indexes; this advantage might not apply for other sorts of foreign data access.

Setup:

CREATE EXTENSION file\_fdw;

CREATE SERVER local\_file FOREIGN DATA WRAPPER file\_fdw;

CREATE FOREIGN TABLE words (word text NOT NULL)

SERVER local\_file

OPTIONS (filename '/usr/share/dict/words');

CREATE MATERIALIZED VIEW wrd AS SELECT \* FROM words;

CREATE UNIQUE INDEX wrd\_word ON wrd (word);

CREATE EXTENSION pg\_trgm;

CREATE INDEX wrd\_trgm ON wrd USING gist (word gist\_trgm\_ops);

VACUUM ANALYZE wrd;

Now let's spell-check a word. Using file\_fdw directly:

SELECT count(\*) FROM words WHERE word = 'caterpiler';

count

-------

0

(1 row)

With EXPLAIN ANALYZE, we see:

Aggregate (cost=21763.99..21764.00 rows=1 width=0) (actual time=188.180..188.181 rows=1 loops=1)

-> Foreign Scan on words (cost=0.00..21761.41 rows=1032 width=0) (actual time=188.177..188.177 rows=0 loops=1)

Filter: (word = 'caterpiler'::text)

Rows Removed by Filter: 479829

Foreign File: /usr/share/dict/words

Foreign File Size: 4953699

Planning time: 0.118 ms

Execution time: 188.273 ms

If the materialized view is used instead, the query is much faster:

Aggregate (cost=4.44..4.45 rows=1 width=0) (actual time=0.042..0.042 rows=1 loops=1)

-> Index Only Scan using wrd\_word on wrd (cost=0.42..4.44 rows=1 width=0) (actual time=0.039..0.039 rows=0 loops=1)

Index Cond: (word = 'caterpiler'::text)

Heap Fetches: 0

Planning time: 0.164 ms

Execution time: 0.117 ms

Either way, the word is spelled wrong, so let's look for what we might have wanted. Again using file\_fdw:

SELECT word FROM words ORDER BY word <-> 'caterpiler' LIMIT 10;

word

---------------

cater

caterpillar

Caterpillar

caterpillars

caterpillar's

Caterpillar's

caterer

caterer's

caters

catered

(10 rows)

Limit (cost=11583.61..11583.64 rows=10 width=32) (actual time=1431.591..1431.594 rows=10 loops=1)

-> Sort (cost=11583.61..11804.76 rows=88459 width=32) (actual time=1431.589..1431.591 rows=10 loops=1)

Sort Key: ((word <-> 'caterpiler'::text))

Sort Method: top-N heapsort Memory: 25kB

-> Foreign Scan on words (cost=0.00..9672.05 rows=88459 width=32) (actual time=0.057..1286.455 rows=479829 loops=1)

Foreign File: /usr/share/dict/words

Foreign File Size: 4953699

Planning time: 0.128 ms

Execution time: 1431.679 ms

Using the materialized view:

Limit (cost=0.29..1.06 rows=10 width=10) (actual time=187.222..188.257 rows=10 loops=1)

-> Index Scan using wrd\_trgm on wrd (cost=0.29..37020.87 rows=479829 width=10) (actual time=187.219..188.252 rows=10 loops=1)

Order By: (word <-> 'caterpiler'::text)

Planning time: 0.196 ms

Execution time: 198.640 ms

If you can tolerate periodic update of the remote data to the local database, the performance benefit can be substantial.

## 40.4. Rules on INSERT, UPDATE, and DELETE

Rules that are defined on INSERT, UPDATE, and DELETE are significantly different from the view rules described in the previous section. First, their CREATE RULE command allows more:

* They are allowed to have no action.
* They can have multiple actions.
* They can be INSTEAD or ALSO (the default).
* The pseudorelations NEW and OLD become useful.
* They can have rule qualifications.

Second, they don't modify the query tree in place. Instead they create zero or more new query trees and can throw away the original one.

### Caution

In many cases, tasks that could be performed by rules on INSERT/UPDATE/DELETE are better done with triggers. Triggers are notationally a bit more complicated, but their semantics are much simpler to understand. Rules tend to have surprising results when the original query contains volatile functions: volatile functions may get executed more times than expected in the process of carrying out the rules.

Also, there are some cases that are not supported by these types of rules at all, notably including WITH clauses in the original query and multiple-assignment sub-SELECTs in the SETlist of UPDATE queries. This is because copying these constructs into a rule query would result in multiple evaluations of the sub-query, contrary to the express intent of the query's author.

### 40.4.1. How Update Rules Work

Keep the syntax:

CREATE [ OR REPLACE ] RULE ***name*** AS ON ***event***

TO ***table*** [ WHERE ***condition*** ]

DO [ ALSO | INSTEAD ] { NOTHING | ***command*** | ( ***command*** ; ***command*** ... ) }

in mind. In the following, update rules means rules that are defined on INSERT, UPDATE, or DELETE.

Update rules get applied by the rule system when the result relation and the command type of a query tree are equal to the object and event given in the CREATE RULE command. For update rules, the rule system creates a list of query trees. Initially the query-tree list is empty. There can be zero (NOTHING key word), one, or multiple actions. To simplify, we will look at a rule with one action. This rule can have a qualification or not and it can be INSTEAD or ALSO (the default).

What is a rule qualification? It is a restriction that tells when the actions of the rule should be done and when not. This qualification can only reference the pseudorelations NEW and/or OLD, which basically represent the relation that was given as object (but with a special meaning).

So we have three cases that produce the following query trees for a one-action rule.

No qualification, with either ALSO or INSTEAD

the query tree from the rule action with the original query tree's qualification added

Qualification given and ALSO

the query tree from the rule action with the rule qualification and the original query tree's qualification added

Qualification given and INSTEAD

the query tree from the rule action with the rule qualification and the original query tree's qualification; and the original query tree with the negated rule qualification added

Finally, if the rule is ALSO, the unchanged original query tree is added to the list. Since only qualified INSTEAD rules already add the original query tree, we end up with either one or two output query trees for a rule with one action.

For ON INSERT rules, the original query (if not suppressed by INSTEAD) is done before any actions added by rules. This allows the actions to see the inserted row(s). But for ON UPDATE and ON DELETErules, the original query is done after the actions added by rules. This ensures that the actions can see the to-be-updated or to-be-deleted rows; otherwise, the actions might do nothing because they find no rows matching their qualifications.

The query trees generated from rule actions are thrown into the rewrite system again, and maybe more rules get applied resulting in more or less query trees. So a rule's actions must have either a different command type or a different result relation than the rule itself is on, otherwise this recursive process will end up in an infinite loop. (Recursive expansion of a rule will be detected and reported as an error.)

The query trees found in the actions of the pg\_rewrite system catalog are only templates. Since they can reference the range-table entries for NEW and OLD, some substitutions have to be made before they can be used. For any reference to NEW, the target list of the original query is searched for a corresponding entry. If found, that entry's expression replaces the reference. Otherwise, NEW means the same as OLD (for an UPDATE) or is replaced by a null value (for an INSERT). Any reference to OLD is replaced by a reference to the range-table entry that is the result relation.

After the system is done applying update rules, it applies view rules to the produced query tree(s). Views cannot insert new update actions so there is no need to apply update rules to the output of view rewriting.

#### 40.4.1.1. A First Rule Step By Step

Say we want to trace changes to the sl\_avail column in the shoelace\_data relation. So we set up a log table and a rule that conditionally writes a log entry when an UPDATE is performed on shoelace\_data.

CREATE TABLE shoelace\_log (

sl\_name text, -- shoelace changed

sl\_avail integer, -- new available value

log\_who text, -- who did it

log\_when timestamp -- when

);

CREATE RULE log\_shoelace AS ON UPDATE TO shoelace\_data

WHERE NEW.sl\_avail <> OLD.sl\_avail

DO INSERT INTO shoelace\_log VALUES (

NEW.sl\_name,

NEW.sl\_avail,

current\_user,

current\_timestamp

);

Now someone does:

UPDATE shoelace\_data SET sl\_avail = 6 WHERE sl\_name = 'sl7';

and we look at the log table:

SELECT \* FROM shoelace\_log;

sl\_name | sl\_avail | log\_who | log\_when

---------+----------+---------+----------------------------------

sl7 | 6 | Al | Tue Oct 20 16:14:45 1998 MET DST

(1 row)

That's what we expected. What happened in the background is the following. The parser created the query tree:

UPDATE shoelace\_data SET sl\_avail = 6

FROM shoelace\_data shoelace\_data

WHERE shoelace\_data.sl\_name = 'sl7';

There is a rule log\_shoelace that is ON UPDATE with the rule qualification expression:

NEW.sl\_avail <> OLD.sl\_avail

and the action:

INSERT INTO shoelace\_log VALUES (

new.sl\_name, new.sl\_avail,

current\_user, current\_timestamp )

FROM shoelace\_data new, shoelace\_data old;

(This looks a little strange since you cannot normally write INSERT ... VALUES ... FROM. The FROM clause here is just to indicate that there are range-table entries in the query tree for new and old. These are needed so that they can be referenced by variables in the INSERT command's query tree.)

The rule is a qualified ALSO rule, so the rule system has to return two query trees: the modified rule action and the original query tree. In step 1, the range table of the original query is incorporated into the rule's action query tree. This results in:

INSERT INTO shoelace\_log VALUES (

new.sl\_name, new.sl\_avail,

current\_user, current\_timestamp )

FROM shoelace\_data new, shoelace\_data old,

**shoelace\_data shoelace\_data**;

In step 2, the rule qualification is added to it, so the result set is restricted to rows where sl\_avail changes:

INSERT INTO shoelace\_log VALUES (

new.sl\_name, new.sl\_avail,

current\_user, current\_timestamp )

FROM shoelace\_data new, shoelace\_data old,

shoelace\_data shoelace\_data

**WHERE new.sl\_avail <> old.sl\_avail**;

(This looks even stranger, since INSERT ... VALUES doesn't have a WHERE clause either, but the planner and executor will have no difficulty with it. They need to support this same functionality anyway for INSERT ... SELECT.)

In step 3, the original query tree's qualification is added, restricting the result set further to only the rows that would have been touched by the original query:

INSERT INTO shoelace\_log VALUES (

new.sl\_name, new.sl\_avail,

current\_user, current\_timestamp )

FROM shoelace\_data new, shoelace\_data old,

shoelace\_data shoelace\_data

WHERE new.sl\_avail <> old.sl\_avail

**AND shoelace\_data.sl\_name = 'sl7'**;

Step 4 replaces references to NEW by the target list entries from the original query tree or by the matching variable references from the result relation:

INSERT INTO shoelace\_log VALUES (

**shoelace\_data.sl\_name**, **6**,

current\_user, current\_timestamp )

FROM shoelace\_data new, shoelace\_data old,

shoelace\_data shoelace\_data

WHERE **6** <> old.sl\_avail

AND shoelace\_data.sl\_name = 'sl7';

Step 5 changes OLD references into result relation references:

INSERT INTO shoelace\_log VALUES (

shoelace\_data.sl\_name, 6,

current\_user, current\_timestamp )

FROM shoelace\_data new, shoelace\_data old,

shoelace\_data shoelace\_data

WHERE 6 <> **shoelace\_data.sl\_avail**

AND shoelace\_data.sl\_name = 'sl7';

That's it. Since the rule is ALSO, we also output the original query tree. In short, the output from the rule system is a list of two query trees that correspond to these statements:

INSERT INTO shoelace\_log VALUES (

shoelace\_data.sl\_name, 6,

current\_user, current\_timestamp )

FROM shoelace\_data

WHERE 6 <> shoelace\_data.sl\_avail

AND shoelace\_data.sl\_name = 'sl7';

UPDATE shoelace\_data SET sl\_avail = 6

WHERE sl\_name = 'sl7';

These are executed in this order, and that is exactly what the rule was meant to do.

The substitutions and the added qualifications ensure that, if the original query would be, say:

UPDATE shoelace\_data SET sl\_color = 'green'

WHERE sl\_name = 'sl7';

no log entry would get written. In that case, the original query tree does not contain a target list entry for sl\_avail, so NEW.sl\_avail will get replaced by shoelace\_data.sl\_avail. Thus, the extra command generated by the rule is:

INSERT INTO shoelace\_log VALUES (

shoelace\_data.sl\_name, **shoelace\_data.sl\_avail**,

current\_user, current\_timestamp )

FROM shoelace\_data

WHERE **shoelace\_data.sl\_avail** <> shoelace\_data.sl\_avail

AND shoelace\_data.sl\_name = 'sl7';

and that qualification will never be true.

It will also work if the original query modifies multiple rows. So if someone issued the command:

UPDATE shoelace\_data SET sl\_avail = 0

WHERE sl\_color = 'black';

four rows in fact get updated (sl1, sl2, sl3, and sl4). But sl3 already has sl\_avail = 0. In this case, the original query trees qualification is different and that results in the extra query tree:

INSERT INTO shoelace\_log

SELECT shoelace\_data.sl\_name, 0,

current\_user, current\_timestamp

FROM shoelace\_data

WHERE 0 <> shoelace\_data.sl\_avail

AND **shoelace\_data.sl\_color = 'black'**;

being generated by the rule. This query tree will surely insert three new log entries. And that's absolutely correct.

Here we can see why it is important that the original query tree is executed last. If the UPDATE had been executed first, all the rows would have already been set to zero, so the logging INSERT would not find any row where 0 <> shoelace\_data.sl\_avail.

### 40.4.2. Cooperation with Views

A simple way to protect view relations from the mentioned possibility that someone can try to run INSERT, UPDATE, or DELETE on them is to let those query trees get thrown away. So we could create the rules:

CREATE RULE shoe\_ins\_protect AS ON INSERT TO shoe

DO INSTEAD NOTHING;

CREATE RULE shoe\_upd\_protect AS ON UPDATE TO shoe

DO INSTEAD NOTHING;

CREATE RULE shoe\_del\_protect AS ON DELETE TO shoe

DO INSTEAD NOTHING;

If someone now tries to do any of these operations on the view relation shoe, the rule system will apply these rules. Since the rules have no actions and are INSTEAD, the resulting list of query trees will be empty and the whole query will become nothing because there is nothing left to be optimized or executed after the rule system is done with it.

A more sophisticated way to use the rule system is to create rules that rewrite the query tree into one that does the right operation on the real tables. To do that on the shoelace view, we create the following rules:

CREATE RULE shoelace\_ins AS ON INSERT TO shoelace

DO INSTEAD

INSERT INTO shoelace\_data VALUES (

NEW.sl\_name,

NEW.sl\_avail,

NEW.sl\_color,

NEW.sl\_len,

NEW.sl\_unit

);

CREATE RULE shoelace\_upd AS ON UPDATE TO shoelace

DO INSTEAD

UPDATE shoelace\_data

SET sl\_name = NEW.sl\_name,

sl\_avail = NEW.sl\_avail,

sl\_color = NEW.sl\_color,

sl\_len = NEW.sl\_len,

sl\_unit = NEW.sl\_unit

WHERE sl\_name = OLD.sl\_name;

CREATE RULE shoelace\_del AS ON DELETE TO shoelace

DO INSTEAD

DELETE FROM shoelace\_data

WHERE sl\_name = OLD.sl\_name;

If you want to support RETURNING queries on the view, you need to make the rules include RETURNING clauses that compute the view rows. This is usually pretty trivial for views on a single table, but it's a bit tedious for join views such as shoelace. An example for the insert case is:

CREATE RULE shoelace\_ins AS ON INSERT TO shoelace

DO INSTEAD

INSERT INTO shoelace\_data VALUES (

NEW.sl\_name,

NEW.sl\_avail,

NEW.sl\_color,

NEW.sl\_len,

NEW.sl\_unit

)

RETURNING

shoelace\_data.\*,

(SELECT shoelace\_data.sl\_len \* u.un\_fact

FROM unit u WHERE shoelace\_data.sl\_unit = u.un\_name);

Note that this one rule supports both INSERT and INSERT RETURNING queries on the view — the RETURNING clause is simply ignored for INSERT.

Now assume that once in a while, a pack of shoelaces arrives at the shop and a big parts list along with it. But you don't want to manually update the shoelace view every time. Instead we set up two little tables: one where you can insert the items from the part list, and one with a special trick. The creation commands for these are:

CREATE TABLE shoelace\_arrive (

arr\_name text,

arr\_quant integer

);

CREATE TABLE shoelace\_ok (

ok\_name text,

ok\_quant integer

);

CREATE RULE shoelace\_ok\_ins AS ON INSERT TO shoelace\_ok

DO INSTEAD

UPDATE shoelace

SET sl\_avail = sl\_avail + NEW.ok\_quant

WHERE sl\_name = NEW.ok\_name;

Now you can fill the table shoelace\_arrive with the data from the parts list:

SELECT \* FROM shoelace\_arrive;

arr\_name | arr\_quant

----------+-----------

sl3 | 10

sl6 | 20

sl8 | 20

(3 rows)

Take a quick look at the current data:

SELECT \* FROM shoelace;

sl\_name | sl\_avail | sl\_color | sl\_len | sl\_unit | sl\_len\_cm

----------+----------+----------+--------+---------+-----------

sl1 | 5 | black | 80 | cm | 80

sl2 | 6 | black | 100 | cm | 100

sl7 | 6 | brown | 60 | cm | 60

sl3 | 0 | black | 35 | inch | 88.9

sl4 | 8 | black | 40 | inch | 101.6

sl8 | 1 | brown | 40 | inch | 101.6

sl5 | 4 | brown | 1 | m | 100

sl6 | 0 | brown | 0.9 | m | 90

(8 rows)

Now move the arrived shoelaces in:

INSERT INTO shoelace\_ok SELECT \* FROM shoelace\_arrive;

and check the results:

SELECT \* FROM shoelace ORDER BY sl\_name;

sl\_name | sl\_avail | sl\_color | sl\_len | sl\_unit | sl\_len\_cm

----------+----------+----------+--------+---------+-----------

sl1 | 5 | black | 80 | cm | 80

sl2 | 6 | black | 100 | cm | 100

sl7 | 6 | brown | 60 | cm | 60

sl4 | 8 | black | 40 | inch | 101.6

sl3 | 10 | black | 35 | inch | 88.9

sl8 | 21 | brown | 40 | inch | 101.6

sl5 | 4 | brown | 1 | m | 100

sl6 | 20 | brown | 0.9 | m | 90

(8 rows)

SELECT \* FROM shoelace\_log;

sl\_name | sl\_avail | log\_who| log\_when

---------+----------+--------+----------------------------------

sl7 | 6 | Al | Tue Oct 20 19:14:45 1998 MET DST

sl3 | 10 | Al | Tue Oct 20 19:25:16 1998 MET DST

sl6 | 20 | Al | Tue Oct 20 19:25:16 1998 MET DST

sl8 | 21 | Al | Tue Oct 20 19:25:16 1998 MET DST

(4 rows)

It's a long way from the one INSERT ... SELECT to these results. And the description of the query-tree transformation will be the last in this chapter. First, there is the parser's output:

INSERT INTO shoelace\_ok

SELECT shoelace\_arrive.arr\_name, shoelace\_arrive.arr\_quant

FROM shoelace\_arrive shoelace\_arrive, shoelace\_ok shoelace\_ok;

Now the first rule shoelace\_ok\_ins is applied and turns this into:

UPDATE shoelace

SET sl\_avail = shoelace.sl\_avail + shoelace\_arrive.arr\_quant

FROM shoelace\_arrive shoelace\_arrive, shoelace\_ok shoelace\_ok,

shoelace\_ok old, shoelace\_ok new,

shoelace shoelace

WHERE shoelace.sl\_name = shoelace\_arrive.arr\_name;

and throws away the original INSERT on shoelace\_ok. This rewritten query is passed to the rule system again, and the second applied rule shoelace\_upd produces:

UPDATE shoelace\_data

SET sl\_name = shoelace.sl\_name,

sl\_avail = shoelace.sl\_avail + shoelace\_arrive.arr\_quant,

sl\_color = shoelace.sl\_color,

sl\_len = shoelace.sl\_len,

sl\_unit = shoelace.sl\_unit

FROM shoelace\_arrive shoelace\_arrive, shoelace\_ok shoelace\_ok,

shoelace\_ok old, shoelace\_ok new,

shoelace shoelace, shoelace old,

shoelace new, shoelace\_data shoelace\_data

WHERE shoelace.sl\_name = shoelace\_arrive.arr\_name

AND shoelace\_data.sl\_name = shoelace.sl\_name;

Again it's an INSTEAD rule and the previous query tree is trashed. Note that this query still uses the view shoelace. But the rule system isn't finished with this step, so it continues and applies the \_RETURN rule on it, and we get:

UPDATE shoelace\_data

SET sl\_name = s.sl\_name,

sl\_avail = s.sl\_avail + shoelace\_arrive.arr\_quant,

sl\_color = s.sl\_color,

sl\_len = s.sl\_len,

sl\_unit = s.sl\_unit

FROM shoelace\_arrive shoelace\_arrive, shoelace\_ok shoelace\_ok,

shoelace\_ok old, shoelace\_ok new,

shoelace shoelace, shoelace old,

shoelace new, shoelace\_data shoelace\_data,

shoelace old, shoelace new,

shoelace\_data s, unit u

WHERE s.sl\_name = shoelace\_arrive.arr\_name

AND shoelace\_data.sl\_name = s.sl\_name;

Finally, the rule log\_shoelace gets applied, producing the extra query tree:

INSERT INTO shoelace\_log

SELECT s.sl\_name,

s.sl\_avail + shoelace\_arrive.arr\_quant,

current\_user,

current\_timestamp

FROM shoelace\_arrive shoelace\_arrive, shoelace\_ok shoelace\_ok,

shoelace\_ok old, shoelace\_ok new,

shoelace shoelace, shoelace old,

shoelace new, shoelace\_data shoelace\_data,

shoelace old, shoelace new,

shoelace\_data s, unit u,

shoelace\_data old, shoelace\_data new

shoelace\_log shoelace\_log

WHERE s.sl\_name = shoelace\_arrive.arr\_name

AND shoelace\_data.sl\_name = s.sl\_name

AND (s.sl\_avail + shoelace\_arrive.arr\_quant) <> s.sl\_avail;

After that the rule system runs out of rules and returns the generated query trees.

So we end up with two final query trees that are equivalent to the SQL statements:

INSERT INTO shoelace\_log

SELECT s.sl\_name,

s.sl\_avail + shoelace\_arrive.arr\_quant,

current\_user,

current\_timestamp

FROM shoelace\_arrive shoelace\_arrive, shoelace\_data shoelace\_data,

shoelace\_data s

WHERE s.sl\_name = shoelace\_arrive.arr\_name

AND shoelace\_data.sl\_name = s.sl\_name

AND s.sl\_avail + shoelace\_arrive.arr\_quant <> s.sl\_avail;

UPDATE shoelace\_data

SET sl\_avail = shoelace\_data.sl\_avail + shoelace\_arrive.arr\_quant

FROM shoelace\_arrive shoelace\_arrive,

shoelace\_data shoelace\_data,

shoelace\_data s

WHERE s.sl\_name = shoelace\_arrive.sl\_name

AND shoelace\_data.sl\_name = s.sl\_name;

The result is that data coming from one relation inserted into another, changed into updates on a third, changed into updating a fourth plus logging that final update in a fifth gets reduced into two queries.

There is a little detail that's a bit ugly. Looking at the two queries, it turns out that the shoelace\_data relation appears twice in the range table where it could definitely be reduced to one. The planner does not handle it and so the execution plan for the rule systems output of the INSERT will be

Nested Loop

-> Merge Join

-> Seq Scan

-> Sort

-> Seq Scan on s

-> Seq Scan

-> Sort

-> Seq Scan on shoelace\_arrive

-> Seq Scan on shoelace\_data

while omitting the extra range table entry would result in a

Merge Join

-> Seq Scan

-> Sort

-> Seq Scan on s

-> Seq Scan

-> Sort

-> Seq Scan on shoelace\_arrive

which produces exactly the same entries in the log table. Thus, the rule system caused one extra scan on the table shoelace\_data that is absolutely not necessary. And the same redundant scan is done once more in the UPDATE. But it was a really hard job to make that all possible at all.

Now we make a final demonstration of the PostgreSQL rule system and its power. Say you add some shoelaces with extraordinary colors to your database:

INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);

INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);

We would like to make a view to check which shoelace entries do not fit any shoe in color. The view for this is:

CREATE VIEW shoelace\_mismatch AS

SELECT \* FROM shoelace WHERE NOT EXISTS

(SELECT shoename FROM shoe WHERE slcolor = sl\_color);

Its output is:

SELECT \* FROM shoelace\_mismatch;

sl\_name | sl\_avail | sl\_color | sl\_len | sl\_unit | sl\_len\_cm

---------+----------+----------+--------+---------+-----------

sl9 | 0 | pink | 35 | inch | 88.9

sl10 | 1000 | magenta | 40 | inch | 101.6

Now we want to set it up so that mismatching shoelaces that are not in stock are deleted from the database. To make it a little harder for PostgreSQL, we don't delete it directly. Instead we create one more view:

CREATE VIEW shoelace\_can\_delete AS

SELECT \* FROM shoelace\_mismatch WHERE sl\_avail = 0;

and do it this way:

DELETE FROM shoelace WHERE EXISTS

(SELECT \* FROM shoelace\_can\_delete

WHERE sl\_name = shoelace.sl\_name);

Voilà:

SELECT \* FROM shoelace;

sl\_name | sl\_avail | sl\_color | sl\_len | sl\_unit | sl\_len\_cm

---------+----------+----------+--------+---------+-----------

sl1 | 5 | black | 80 | cm | 80

sl2 | 6 | black | 100 | cm | 100

sl7 | 6 | brown | 60 | cm | 60

sl4 | 8 | black | 40 | inch | 101.6

sl3 | 10 | black | 35 | inch | 88.9

sl8 | 21 | brown | 40 | inch | 101.6

sl10 | 1000 | magenta | 40 | inch | 101.6

sl5 | 4 | brown | 1 | m | 100

sl6 | 20 | brown | 0.9 | m | 90

(9 rows)

A DELETE on a view, with a subquery qualification that in total uses 4 nesting/joined views, where one of them itself has a subquery qualification containing a view and where calculated view columns are used, gets rewritten into one single query tree that deletes the requested data from a real table.

There are probably only a few situations out in the real world where such a construct is necessary. But it makes you feel comfortable that it works.

## 40.5. Rules and Privileges

Due to rewriting of queries by the PostgreSQL rule system, other tables/views than those used in the original query get accessed. When update rules are used, this can include write access to tables.

Rewrite rules don't have a separate owner. The owner of a relation (table or view) is automatically the owner of the rewrite rules that are defined for it. The PostgreSQL rule system changes the behavior of the default access control system. Relations that are used due to rules get checked against the privileges of the rule owner, not the user invoking the rule. This means that users only need the required privileges for the tables/views that are explicitly named in their queries.

For example: A user has a list of phone numbers where some of them are private, the others are of interest for the assistant of the office. The user can construct the following:

CREATE TABLE phone\_data (person text, phone text, private boolean);

CREATE VIEW phone\_number AS

SELECT person, CASE WHEN NOT private THEN phone END AS phone

FROM phone\_data;

GRANT SELECT ON phone\_number TO assistant;

Nobody except that user (and the database superusers) can access the phone\_data table. But because of the GRANT, the assistant can run a SELECT on the phone\_number view. The rule system will rewrite the SELECT from phone\_number into a SELECT from phone\_data. Since the user is the owner of phone\_number and therefore the owner of the rule, the read access to phone\_data is now checked against the user's privileges and the query is permitted. The check for accessing phone\_number is also performed, but this is done against the invoking user, so nobody but the user and the assistant can use it.

The privileges are checked rule by rule. So the assistant is for now the only one who can see the public phone numbers. But the assistant can set up another view and grant access to that to the public. Then, anyone can see the phone\_number data through the assistant's view. What the assistant cannot do is to create a view that directly accesses phone\_data. (Actually the assistant can, but it will not work since every access will be denied during the permission checks.) And as soon as the user notices that the assistant opened their phone\_number view, the user can revoke the assistant's access. Immediately, any access to the assistant's view would fail.

One might think that this rule-by-rule checking is a security hole, but in fact it isn't. But if it did not work this way, the assistant could set up a table with the same columns as phone\_number and copy the data to there once per day. Then it's the assistant's own data and the assistant can grant access to everyone they want. A GRANT command means, “I trust you”. If someone you trust does the thing above, it's time to think it over and then use REVOKE.

Note that while views can be used to hide the contents of certain columns using the technique shown above, they cannot be used to reliably conceal the data in unseen rows unless the security\_barrier flag has been set. For example, the following view is insecure:

CREATE VIEW phone\_number AS

SELECT person, phone FROM phone\_data WHERE phone NOT LIKE '412%';

This view might seem secure, since the rule system will rewrite any SELECT from phone\_number into a SELECT from phone\_data and add the qualification that only entries where phone does not begin with 412 are wanted. But if the user can create their own functions, it is not difficult to convince the planner to execute the user-defined function prior to the NOT LIKE expression. For example:

CREATE FUNCTION tricky(text, text) RETURNS bool AS $$

BEGIN

RAISE NOTICE '% => %', $1, $2;

RETURN true;

END

$$ LANGUAGE plpgsql COST 0.0000000000000000000001;

SELECT \* FROM phone\_number WHERE tricky(person, phone);

Every person and phone number in the phone\_data table will be printed as a NOTICE, because the planner will choose to execute the inexpensive tricky function before the more expensive NOT LIKE. Even if the user is prevented from defining new functions, built-in functions can be used in similar attacks. (For example, most casting functions include their input values in the error messages they produce.)

Similar considerations apply to update rules. In the examples of the previous section, the owner of the tables in the example database could grant the privileges SELECT, INSERT, UPDATE, and DELETE on the shoelace view to someone else, but only SELECT on shoelace\_log. The rule action to write log entries will still be executed successfully, and that other user could see the log entries. But they could not create fake entries, nor could they manipulate or remove existing ones. In this case, there is no possibility of subverting the rules by convincing the planner to alter the order of operations, because the only rule which references shoelace\_log is an unqualified INSERT. This might not be true in more complex scenarios.

When it is necessary for a view to provide row level security, the security\_barrier attribute should be applied to the view. This prevents maliciously-chosen functions and operators from being passed values from rows until after the view has done its work. For example, if the view shown above had been created like this, it would be secure:

CREATE VIEW phone\_number WITH (security\_barrier) AS

SELECT person, phone FROM phone\_data WHERE phone NOT LIKE '412%';

Views created with the security\_barrier may perform far worse than views created without this option. In general, there is no way to avoid this: the fastest possible plan must be rejected if it may compromise security. For this reason, this option is not enabled by default.

The query planner has more flexibility when dealing with functions that have no side effects. Such functions are referred to as LEAKPROOF, and include many simple, commonly used operators, such as many equality operators. The query planner can safely allow such functions to be evaluated at any point in the query execution process, since invoking them on rows invisible to the user will not leak any information about the unseen rows. Further, functions which do not take arguments or which are not passed any arguments from the security barrier view do not have to be marked as LEAKPROOFto be pushed down, as they never receive data from the view. In contrast, a function that might throw an error depending on the values received as arguments (such as one that throws an error in the event of overflow or division by zero) is not leak-proof, and could provide significant information about the unseen rows if applied before the security view's row filters.

It is important to understand that even a view created with the security\_barrier option is intended to be secure only in the limited sense that the contents of the invisible tuples will not be passed to possibly-insecure functions. The user may well have other means of making inferences about the unseen data; for example, they can see the query plan using EXPLAIN, or measure the run time of queries against the view. A malicious attacker might be able to infer something about the amount of unseen data, or even gain some information about the data distribution or most common values (since these things may affect the run time of the plan; or even, since they are also reflected in the optimizer statistics, the choice of plan). If these types of "covert channel" attacks are of concern, it is probably unwise to grant any access to the data at all.

## 40.6. Rules and Command Status

The PostgreSQL server returns a command status string, such as INSERT 149592 1, for each command it receives. This is simple enough when there are no rules involved, but what happens when the query is rewritten by rules?

Rules affect the command status as follows:

* If there is no unconditional INSTEAD rule for the query, then the originally given query will be executed, and its command status will be returned as usual. (But note that if there were any conditional INSTEAD rules, the negation of their qualifications will have been added to the original query. This might reduce the number of rows it processes, and if so the reported status will be affected.)
* If there is any unconditional INSTEAD rule for the query, then the original query will not be executed at all. In this case, the server will return the command status for the last query that was inserted by an INSTEAD rule (conditional or unconditional) and is of the same command type (INSERT, UPDATE, or DELETE) as the original query. If no query meeting those requirements is added by any rule, then the returned command status shows the original query type and zeroes for the row-count and OID fields.

The programmer can ensure that any desired INSTEAD rule is the one that sets the command status in the second case, by giving it the alphabetically last rule name among the active rules, so that it gets applied last.

## 40.7. Rules Versus Triggers

Many things that can be done using triggers can also be implemented using the PostgreSQL rule system. One of the things that cannot be implemented by rules are some kinds of constraints, especially foreign keys. It is possible to place a qualified rule that rewrites a command to NOTHING if the value of a column does not appear in another table. But then the data is silently thrown away and that's not a good idea. If checks for valid values are required, and in the case of an invalid value an error message should be generated, it must be done by a trigger.

In this chapter, we focused on using rules to update views. All of the update rule examples in this chapter can also be implemented using INSTEAD OF triggers on the views. Writing such triggers is often easier than writing rules, particularly if complex logic is required to perform the update.

For the things that can be implemented by both, which is best depends on the usage of the database. A trigger is fired once for each affected row. A rule modifies the query or generates an additional query. So if many rows are affected in one statement, a rule issuing one extra command is likely to be faster than a trigger that is called for every single row and must re-determine what to do many times. However, the trigger approach is conceptually far simpler than the rule approach, and is easier for novices to get right.

Here we show an example of how the choice of rules versus triggers plays out in one situation. There are two tables:

CREATE TABLE computer (

hostname text, -- indexed

manufacturer text -- indexed

);

CREATE TABLE software (

software text, -- indexed

hostname text -- indexed

);

Both tables have many thousands of rows and the indexes on hostname are unique. The rule or trigger should implement a constraint that deletes rows from software that reference a deleted computer. The trigger would use this command:

DELETE FROM software WHERE hostname = $1;

Since the trigger is called for each individual row deleted from computer, it can prepare and save the plan for this command and pass the hostname value in the parameter. The rule would be written as:

CREATE RULE computer\_del AS ON DELETE TO computer

DO DELETE FROM software WHERE hostname = OLD.hostname;

Now we look at different types of deletes. In the case of a:

DELETE FROM computer WHERE hostname = 'mypc.local.net';

the table computer is scanned by index (fast), and the command issued by the trigger would also use an index scan (also fast). The extra command from the rule would be:

DELETE FROM software WHERE computer.hostname = 'mypc.local.net'

AND software.hostname = computer.hostname;

Since there are appropriate indexes set up, the planner will create a plan of

Nestloop

-> Index Scan using comp\_hostidx on computer

-> Index Scan using soft\_hostidx on software

So there would be not that much difference in speed between the trigger and the rule implementation.

With the next delete we want to get rid of all the 2000 computers where the hostname starts with old. There are two possible commands to do that. One is:

DELETE FROM computer WHERE hostname >= 'old'

AND hostname < 'ole'

The command added by the rule will be:

DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname < 'ole'

AND software.hostname = computer.hostname;

with the plan

Hash Join

-> Seq Scan on software

-> Hash

-> Index Scan using comp\_hostidx on computer

The other possible command is:

DELETE FROM computer WHERE hostname ~ '^old';

which results in the following executing plan for the command added by the rule:

Nestloop

-> Index Scan using comp\_hostidx on computer

-> Index Scan using soft\_hostidx on software

This shows, that the planner does not realize that the qualification for hostname in computer could also be used for an index scan on software when there are multiple qualification expressions combined with AND, which is what it does in the regular-expression version of the command. The trigger will get invoked once for each of the 2000 old computers that have to be deleted, and that will result in one index scan over computer and 2000 index scans over software. The rule implementation will do it with two commands that use indexes. And it depends on the overall size of the table software whether the rule will still be faster in the sequential scan situation. 2000 command executions from the trigger over the SPI manager take some time, even if all the index blocks will soon be in the cache.

The last command we look at is:

DELETE FROM computer WHERE manufacturer = 'bim';

Again this could result in many rows to be deleted from computer. So the trigger will again run many commands through the executor. The command generated by the rule will be:

DELETE FROM software WHERE computer.manufacturer = 'bim'

AND software.hostname = computer.hostname;

The plan for that command will again be the nested loop over two index scans, only using a different index on computer:

Nestloop

-> Index Scan using comp\_manufidx on computer

-> Index Scan using soft\_hostidx on software

In any of these cases, the extra commands from the rule system will be more or less independent from the number of affected rows in a command.

The summary is, rules will only be significantly slower than triggers if their actions result in large and badly qualified joins, a situation where the planner fails.

## Chapter 41. Procedural Languages

PostgreSQL allows user-defined functions to be written in other languages besides SQL and C. These other languages are generically called *procedural languages* (PLs). For a function written in a procedural language, the database server has no built-in knowledge about how to interpret the function's source text. Instead, the task is passed to a special handler that knows the details of the language. The handler could either do all the work of parsing, syntax analysis, execution, etc. itself, or it could serve as “glue” between PostgreSQL and an existing implementation of a programming language. The handler itself is a C language function compiled into a shared object and loaded on demand, just like any other C function.

There are currently four procedural languages available in the standard PostgreSQL distribution: PL/pgSQL ([**Chapter 42**](https://www.postgresql.org/docs/10/plpgsql.html)), PL/Tcl ([**Chapter 43**](https://www.postgresql.org/docs/10/pltcl.html)), PL/Perl ([**Chapter 44**](https://www.postgresql.org/docs/10/plperl.html)), and PL/Python ([**Chapter 45**](https://www.postgresql.org/docs/10/plpython.html)). There are additional procedural languages available that are not included in the core distribution. [**Appendix H**](https://www.postgresql.org/docs/10/external-projects.html) has information about finding them. In addition other languages can be defined by users; the basics of developing a new procedural language are covered in [**Chapter 55**](https://www.postgresql.org/docs/10/plhandler.html).

**41.1. Installing Procedural Languages**

A procedural language must be “installed” into each database where it is to be used. But procedural languages installed in the database template1 are automatically available in all subsequently created databases, since their entries in template1 will be copied by CREATE DATABASE. So the database administrator can decide which languages are available in which databases and can make some languages available by default if desired.

For the languages supplied with the standard distribution, it is only necessary to execute CREATE EXTENSION ***language\_name*** to install the language into the current database. The manual procedure described below is only recommended for installing languages that have not been packaged as extensions.

**Manual Procedural Language Installation**

A procedural language is installed in a database in five steps, which must be carried out by a database superuser. In most cases the required SQL commands should be packaged as the installation script of an “extension”, so that CREATE EXTENSION can be used to execute them.

1. The shared object for the language handler must be compiled and installed into an appropriate library directory. This works in the same way as building and installing modules with regular user-defined C functions does; see [**Section 37.9.5**](https://www.postgresql.org/docs/10/xfunc-c.html#DFUNC). Often, the language handler will depend on an external library that provides the actual programming language engine; if so, that must be installed as well.
2. The handler must be declared with the command
3. CREATE FUNCTION ***handler\_function\_name***()
4. RETURNS language\_handler
5. AS '***path-to-shared-object***'

LANGUAGE C;

The special return type of language\_handler tells the database system that this function does not return one of the defined SQL data types and is not directly usable in SQL statements.

1. Optionally, the language handler can provide an “inline” handler function that executes anonymous code blocks ([**DO**](https://www.postgresql.org/docs/10/sql-do.html) commands) written in this language. If an inline handler function is provided by the language, declare it with a command like
2. CREATE FUNCTION ***inline\_function\_name***(internal)
3. RETURNS void
4. AS '***path-to-shared-object***'

LANGUAGE C;

1. Optionally, the language handler can provide a “validator” function that checks a function definition for correctness without actually executing it. The validator function is called by CREATE FUNCTION if it exists. If a validator function is provided by the language, declare it with a command like
2. CREATE FUNCTION ***validator\_function\_name***(oid)
3. RETURNS void
4. AS '***path-to-shared-object***'

LANGUAGE C STRICT;

1. Finally, the PL must be declared with the command
2. CREATE [TRUSTED] [PROCEDURAL] LANGUAGE ***language-name***
3. HANDLER ***handler\_function\_name***
4. [INLINE ***inline\_function\_name***]

[VALIDATOR ***validator\_function\_name***] ;

The optional key word TRUSTED specifies that the language does not grant access to data that the user would not otherwise have. Trusted languages are designed for ordinary database users (those without superuser privilege) and allows them to safely create functions and trigger procedures. Since PL functions are executed inside the database server, the TRUSTED flag should only be given for languages that do not allow access to database server internals or the file system. The languages PL/pgSQL, PL/Tcl, and PL/Perl are considered trusted; the languages PL/TclU, PL/PerlU, and PL/PythonU are designed to provide unlimited functionality and should *not* be marked trusted.

[**Example 41.1**](https://www.postgresql.org/docs/10/xplang-install.html#XPLANG-INSTALL-EXAMPLE) shows how the manual installation procedure would work with the language PL/Perl.

**Example 41.1. Manual Installation of PL/Perl**

The following command tells the database server where to find the shared object for the PL/Perl language's call handler function:

CREATE FUNCTION plperl\_call\_handler() RETURNS language\_handler AS

'$libdir/plperl' LANGUAGE C;

PL/Perl has an inline handler function and a validator function, so we declare those too:

CREATE FUNCTION plperl\_inline\_handler(internal) RETURNS void AS

'$libdir/plperl' LANGUAGE C;

CREATE FUNCTION plperl\_validator(oid) RETURNS void AS

'$libdir/plperl' LANGUAGE C STRICT;

The command:

CREATE TRUSTED PROCEDURAL LANGUAGE plperl

HANDLER plperl\_call\_handler

INLINE plperl\_inline\_handler

VALIDATOR plperl\_validator;

then defines that the previously declared functions should be invoked for functions and trigger procedures where the language attribute is plperl.

In a default PostgreSQL installation, the handler for the PL/pgSQL language is built and installed into the “library” directory; furthermore, the PL/pgSQL language itself is installed in all databases. If Tclsupport is configured in, the handlers for PL/Tcl and PL/TclU are built and installed in the library directory, but the language itself is not installed in any database by default. Likewise, the PL/Perl and PL/PerlU handlers are built and installed if Perl support is configured, and the PL/PythonU handler is installed if Python support is configured, but these languages are not installed by default.

## Chapter 42. PL/pgSQL - SQL Procedural Language

## 42.1. Overview

PL/pgSQL is a loadable procedural language for the PostgreSQL database system. The design goals of PL/pgSQL were to create a loadable procedural language that

* can be used to create functions and trigger procedures,
* adds control structures to the SQL language,
* can perform complex computations,
* inherits all user-defined types, functions, and operators,
* can be defined to be trusted by the server,
* is easy to use.

Functions created with PL/pgSQL can be used anywhere that built-in functions could be used. For example, it is possible to create complex conditional computation functions and later use them to define operators or use them in index expressions.

In PostgreSQL 9.0 and later, PL/pgSQL is installed by default. However it is still a loadable module, so especially security-conscious administrators could choose to remove it.

### 42.1.1. Advantages of Using PL/pgSQL

SQL is the language PostgreSQL and most other relational databases use as query language. It's portable and easy to learn. But every SQL statement must be executed individually by the database server.

That means that your client application must send each query to the database server, wait for it to be processed, receive and process the results, do some computation, then send further queries to the server. All this incurs interprocess communication and will also incur network overhead if your client is on a different machine than the database server.

With PL/pgSQL you can group a block of computation and a series of queries inside the database server, thus having the power of a procedural language and the ease of use of SQL, but with considerable savings of client/server communication overhead.

* Extra round trips between client and server are eliminated
* Intermediate results that the client does not need do not have to be marshaled or transferred between server and client
* Multiple rounds of query parsing can be avoided

This can result in a considerable performance increase as compared to an application that does not use stored functions.

Also, with PL/pgSQL you can use all the data types, operators and functions of SQL.

### 42.1.2. Supported Argument and Result Data Types

Functions written in PL/pgSQL can accept as arguments any scalar or array data type supported by the server, and they can return a result of any of these types. They can also accept or return any composite type (row type) specified by name. It is also possible to declare a PL/pgSQL function as returning record, which means that the result is a row type whose columns are determined by specification in the calling query, as discussed in [**Section 7.2.1.4**](https://www.postgresql.org/docs/10/queries-table-expressions.html#QUERIES-TABLEFUNCTIONS).

PL/pgSQL functions can be declared to accept a variable number of arguments by using the VARIADIC marker. This works exactly the same way as for SQL functions, as discussed in [**Section 37.4.5**](https://www.postgresql.org/docs/10/xfunc-sql.html#XFUNC-SQL-VARIADIC-FUNCTIONS).

PL/pgSQL functions can also be declared to accept and return the polymorphic types anyelement, anyarray, anynonarray, anyenum, and anyrange. The actual data types handled by a polymorphic function can vary from call to call, as discussed in [**Section 37.2.5**](https://www.postgresql.org/docs/10/extend-type-system.html#EXTEND-TYPES-POLYMORPHIC). An example is shown in [**Section 42.3.1**](https://www.postgresql.org/docs/10/plpgsql-declarations.html#PLPGSQL-DECLARATION-PARAMETERS).

PL/pgSQL functions can also be declared to return a “set” (or table) of any data type that can be returned as a single instance. Such a function generates its output by executing RETURN NEXT for each desired element of the result set, or by using RETURN QUERY to output the result of evaluating a query.

Finally, a PL/pgSQL function can be declared to return void if it has no useful return value.

PL/pgSQL functions can also be declared with output parameters in place of an explicit specification of the return type. This does not add any fundamental capability to the language, but it is often convenient, especially for returning multiple values. The RETURNS TABLE notation can also be used in place of RETURNS SETOF.

Specific examples appear in [**Section 42.3.1**](https://www.postgresql.org/docs/10/plpgsql-declarations.html#PLPGSQL-DECLARATION-PARAMETERS) and [**Section 42.6.1**](https://www.postgresql.org/docs/10/plpgsql-control-structures.html#PLPGSQL-STATEMENTS-RETURNING).

## 42.2. Structure of PL/pgSQL

Functions written in PL/pgSQL are defined to the server by executing [**CREATE FUNCTION**](https://www.postgresql.org/docs/10/sql-createfunction.html) commands. Such a command would normally look like, say,

CREATE FUNCTION somefunc(integer, text) RETURNS integer

AS '***function body text***'

LANGUAGE plpgsql;

The function body is simply a string literal so far as CREATE FUNCTION is concerned. It is often helpful to use dollar quoting (see [**Section 4.1.2.4**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-DOLLAR-QUOTING)) to write the function body, rather than the normal single quote syntax. Without dollar quoting, any single quotes or backslashes in the function body must be escaped by doubling them. Almost all the examples in this chapter use dollar-quoted literals for their function bodies.

PL/pgSQL is a block-structured language. The complete text of a function body must be a block. A block is defined as:

[ <<***label***>> ]

[ DECLARE

***declarations*** ]

BEGIN

***statements***

END [ ***label*** ];

Each declaration and each statement within a block is terminated by a semicolon. A block that appears within another block must have a semicolon after END, as shown above; however the final ENDthat concludes a function body does not require a semicolon.

### Tip

A common mistake is to write a semicolon immediately after BEGIN. This is incorrect and will result in a syntax error.

A ***label*** is only needed if you want to identify the block for use in an EXIT statement, or to qualify the names of the variables declared in the block. If a label is given after END, it must match the label at the block's beginning.

All key words are case-insensitive. Identifiers are implicitly converted to lower case unless double-quoted, just as they are in ordinary SQL commands.

Comments work the same way in PL/pgSQL code as in ordinary SQL. A double dash (--) starts a comment that extends to the end of the line. A /\* starts a block comment that extends to the matching occurrence of \*/. Block comments nest.

Any statement in the statement section of a block can be a subblock. Subblocks can be used for logical grouping or to localize variables to a small group of statements. Variables declared in a subblock mask any similarly-named variables of outer blocks for the duration of the subblock; but you can access the outer variables anyway if you qualify their names with their block's label. For example:

CREATE FUNCTION somefunc() RETURNS integer AS $$

<< outerblock >>

DECLARE

quantity integer := 30;

BEGIN

RAISE NOTICE 'Quantity here is %', quantity; -- Prints 30

quantity := 50;

--

-- Create a subblock

--

DECLARE

quantity integer := 80;

BEGIN

RAISE NOTICE 'Quantity here is %', quantity; -- Prints 80

RAISE NOTICE 'Outer quantity here is %', outerblock.quantity; -- Prints 50

END;

RAISE NOTICE 'Quantity here is %', quantity; -- Prints 50

RETURN quantity;

END;

$$ LANGUAGE plpgsql;

### Note

There is actually a hidden “outer block” surrounding the body of any PL/pgSQL function. This block provides the declarations of the function's parameters (if any), as well as some special variables such as FOUND (see [**Section 42.5.5**](https://www.postgresql.org/docs/10/plpgsql-statements.html#PLPGSQL-STATEMENTS-DIAGNOSTICS)). The outer block is labeled with the function's name, meaning that parameters and special variables can be qualified with the function's name.

It is important not to confuse the use of BEGIN/END for grouping statements in PL/pgSQL with the similarly-named SQL commands for transaction control. PL/pgSQL's BEGIN/END are only for grouping; they do not start or end a transaction. Functions and trigger procedures are always executed within a transaction established by an outer query — they cannot start or commit that transaction, since there would be no context for them to execute in. However, a block containing an EXCEPTION clause effectively forms a subtransaction that can be rolled back without affecting the outer transaction. For more about that see [**Section 42.6.6**](https://www.postgresql.org/docs/10/plpgsql-control-structures.html#PLPGSQL-ERROR-TRAPPING).

## 42.3. Declarations

All variables used in a block must be declared in the declarations section of the block. (The only exceptions are that the loop variable of a FOR loop iterating over a range of integer values is automatically declared as an integer variable, and likewise the loop variable of a FOR loop iterating over a cursor's result is automatically declared as a record variable.)

PL/pgSQL variables can have any SQL data type, such as integer, varchar, and char.

Here are some examples of variable declarations:

user\_id integer;

quantity numeric(5);

url varchar;

myrow tablename%ROWTYPE;

myfield tablename.columnname%TYPE;

arow RECORD;

The general syntax of a variable declaration is:

***name*** [ CONSTANT ] ***type*** [ COLLATE ***collation\_name*** ] [ NOT NULL ] [ { DEFAULT | := | = } ***expression*** ];

The DEFAULT clause, if given, specifies the initial value assigned to the variable when the block is entered. If the DEFAULT clause is not given then the variable is initialized to the SQL null value. The CONSTANT option prevents the variable from being assigned to after initialization, so that its value will remain constant for the duration of the block. The COLLATE option specifies a collation to use for the variable (see [**Section 42.3.6**](https://www.postgresql.org/docs/10/plpgsql-declarations.html#PLPGSQL-DECLARATION-COLLATION)). If NOT NULL is specified, an assignment of a null value results in a run-time error. All variables declared as NOT NULL must have a nonnull default value specified. Equal (=) can be used instead of PL/SQL-compliant :=.

A variable's default value is evaluated and assigned to the variable each time the block is entered (not just once per function call). So, for example, assigning now() to a variable of type timestampcauses the variable to have the time of the current function call, not the time when the function was precompiled.

Examples:

quantity integer DEFAULT 32;

url varchar := 'http://mysite.com';

user\_id CONSTANT integer := 10;

### 42.3.1. Declaring Function Parameters

Parameters passed to functions are named with the identifiers $1, $2, etc. Optionally, aliases can be declared for $***n*** parameter names for increased readability. Either the alias or the numeric identifier can then be used to refer to the parameter value.

There are two ways to create an alias. The preferred way is to give a name to the parameter in the CREATE FUNCTION command, for example:

CREATE FUNCTION sales\_tax(subtotal real) RETURNS real AS $$

BEGIN

RETURN subtotal \* 0.06;

END;

$$ LANGUAGE plpgsql;

The other way is to explicitly declare an alias, using the declaration syntax

***name*** ALIAS FOR $***n***;

The same example in this style looks like:

CREATE FUNCTION sales\_tax(real) RETURNS real AS $$

DECLARE

subtotal ALIAS FOR $1;

BEGIN

RETURN subtotal \* 0.06;

END;

$$ LANGUAGE plpgsql;

### Note

These two examples are not perfectly equivalent. In the first case, subtotal could be referenced as sales\_tax.subtotal, but in the second case it could not. (Had we attached a label to the inner block, subtotal could be qualified with that label, instead.)

Some more examples:

CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$

DECLARE

v\_string ALIAS FOR $1;

index ALIAS FOR $2;

BEGIN

-- some computations using v\_string and index here

END;

$$ LANGUAGE plpgsql;

CREATE FUNCTION concat\_selected\_fields(in\_t sometablename) RETURNS text AS $$

BEGIN

RETURN in\_t.f1 || in\_t.f3 || in\_t.f5 || in\_t.f7;

END;

$$ LANGUAGE plpgsql;

When a PL/pgSQL function is declared with output parameters, the output parameters are given $***n*** names and optional aliases in just the same way as the normal input parameters. An output parameter is effectively a variable that starts out NULL; it should be assigned to during the execution of the function. The final value of the parameter is what is returned. For instance, the sales-tax example could also be done this way:

CREATE FUNCTION sales\_tax(subtotal real, OUT tax real) AS $$

BEGIN

tax := subtotal \* 0.06;

END;

$$ LANGUAGE plpgsql;

Notice that we omitted RETURNS real — we could have included it, but it would be redundant.

Output parameters are most useful when returning multiple values. A trivial example is:

CREATE FUNCTION sum\_n\_product(x int, y int, OUT sum int, OUT prod int) AS $$

BEGIN

sum := x + y;

prod := x \* y;

END;

$$ LANGUAGE plpgsql;

As discussed in [**Section 37.4.4**](https://www.postgresql.org/docs/10/xfunc-sql.html#XFUNC-OUTPUT-PARAMETERS), this effectively creates an anonymous record type for the function's results. If a RETURNS clause is given, it must say RETURNS record.

Another way to declare a PL/pgSQL function is with RETURNS TABLE, for example:

CREATE FUNCTION extended\_sales(p\_itemno int)

RETURNS TABLE(quantity int, total numeric) AS $$

BEGIN

RETURN QUERY SELECT s.quantity, s.quantity \* s.price FROM sales AS s

WHERE s.itemno = p\_itemno;

END;

$$ LANGUAGE plpgsql;

This is exactly equivalent to declaring one or more OUT parameters and specifying RETURNS SETOF ***sometype***.

When the return type of a PL/pgSQL function is declared as a polymorphic type (anyelement, anyarray, anynonarray, anyenum, or anyrange), a special parameter $0 is created. Its data type is the actual return type of the function, as deduced from the actual input types (see [**Section 37.2.5**](https://www.postgresql.org/docs/10/extend-type-system.html#EXTEND-TYPES-POLYMORPHIC)). This allows the function to access its actual return type as shown in [**Section 42.3.3**](https://www.postgresql.org/docs/10/plpgsql-declarations.html#PLPGSQL-DECLARATION-TYPE). $0 is initialized to null and can be modified by the function, so it can be used to hold the return value if desired, though that is not required. $0 can also be given an alias. For example, this function works on any data type that has a + operator:

CREATE FUNCTION add\_three\_values(v1 anyelement, v2 anyelement, v3 anyelement)

RETURNS anyelement AS $$

DECLARE

result ALIAS FOR $0;

BEGIN

result := v1 + v2 + v3;

RETURN result;

END;

$$ LANGUAGE plpgsql;

The same effect can be obtained by declaring one or more output parameters as polymorphic types. In this case the special $0 parameter is not used; the output parameters themselves serve the same purpose. For example:

CREATE FUNCTION add\_three\_values(v1 anyelement, v2 anyelement, v3 anyelement,

OUT sum anyelement)

AS $$

BEGIN

sum := v1 + v2 + v3;

END;

$$ LANGUAGE plpgsql;

### 42.3.2. ALIAS

***newname*** ALIAS FOR ***oldname***;

The ALIAS syntax is more general than is suggested in the previous section: you can declare an alias for any variable, not just function parameters. The main practical use for this is to assign a different name for variables with predetermined names, such as NEW or OLD within a trigger procedure.

Examples:

DECLARE

prior ALIAS FOR old;

updated ALIAS FOR new;

Since ALIAS creates two different ways to name the same object, unrestricted use can be confusing. It's best to use it only for the purpose of overriding predetermined names.

### 42.3.3. Copying Types

***variable***%TYPE

%TYPE provides the data type of a variable or table column. You can use this to declare variables that will hold database values. For example, let's say you have a column named user\_id in your userstable. To declare a variable with the same data type as users.user\_id you write:

user\_id users.user\_id%TYPE;

By using %TYPE you don't need to know the data type of the structure you are referencing, and most importantly, if the data type of the referenced item changes in the future (for instance: you change the type of user\_id from integer to real), you might not need to change your function definition.

%TYPE is particularly valuable in polymorphic functions, since the data types needed for internal variables can change from one call to the next. Appropriate variables can be created by applying %TYPEto the function's arguments or result placeholders.

### 42.3.4. Row Types

***name*** ***table\_name***%ROWTYPE;

***name*** ***composite\_type\_name***;

A variable of a composite type is called a row variable (or row-type variable). Such a variable can hold a whole row of a SELECT or FOR query result, so long as that query's column set matches the declared type of the variable. The individual fields of the row value are accessed using the usual dot notation, for example rowvar.field.

A row variable can be declared to have the same type as the rows of an existing table or view, by using the ***table\_name***%ROWTYPE notation; or it can be declared by giving a composite type's name. (Since every table has an associated composite type of the same name, it actually does not matter in PostgreSQL whether you write %ROWTYPE or not. But the form with %ROWTYPE is more portable.)

Parameters to a function can be composite types (complete table rows). In that case, the corresponding identifier $***n*** will be a row variable, and fields can be selected from it, for example $1.user\_id.

Only the user-defined columns of a table row are accessible in a row-type variable, not the OID or other system columns (because the row could be from a view). The fields of the row type inherit the table's field size or precision for data types such as char(***n***).

Here is an example of using composite types. table1 and table2 are existing tables having at least the mentioned fields:

CREATE FUNCTION merge\_fields(t\_row table1) RETURNS text AS $$

DECLARE

t2\_row table2%ROWTYPE;

BEGIN

SELECT \* INTO t2\_row FROM table2 WHERE ... ;

RETURN t\_row.f1 || t2\_row.f3 || t\_row.f5 || t2\_row.f7;

END;

$$ LANGUAGE plpgsql;

SELECT merge\_fields(t.\*) FROM table1 t WHERE ... ;

### 42.3.5. Record Types

***name*** RECORD;

Record variables are similar to row-type variables, but they have no predefined structure. They take on the actual row structure of the row they are assigned during a SELECT or FOR command. The substructure of a record variable can change each time it is assigned to. A consequence of this is that until a record variable is first assigned to, it has no substructure, and any attempt to access a field in it will draw a run-time error.

Note that RECORD is not a true data type, only a placeholder. One should also realize that when a PL/pgSQL function is declared to return type record, this is not quite the same concept as a record variable, even though such a function might use a record variable to hold its result. In both cases the actual row structure is unknown when the function is written, but for a function returning recordthe actual structure is determined when the calling query is parsed, whereas a record variable can change its row structure on-the-fly.

### 42.3.6. Collation of PL/pgSQL Variables

When a PL/pgSQL function has one or more parameters of collatable data types, a collation is identified for each function call depending on the collations assigned to the actual arguments, as described in [**Section 23.2**](https://www.postgresql.org/docs/10/collation.html). If a collation is successfully identified (i.e., there are no conflicts of implicit collations among the arguments) then all the collatable parameters are treated as having that collation implicitly. This will affect the behavior of collation-sensitive operations within the function. For example, consider

CREATE FUNCTION less\_than(a text, b text) RETURNS boolean AS $$

BEGIN

RETURN a < b;

END;

$$ LANGUAGE plpgsql;

SELECT less\_than(text\_field\_1, text\_field\_2) FROM table1;

SELECT less\_than(text\_field\_1, text\_field\_2 COLLATE "C") FROM table1;

The first use of less\_than will use the common collation of text\_field\_1 and text\_field\_2 for the comparison, while the second use will use C collation.

Furthermore, the identified collation is also assumed as the collation of any local variables that are of collatable types. Thus this function would not work any differently if it were written as

CREATE FUNCTION less\_than(a text, b text) RETURNS boolean AS $$

DECLARE

local\_a text := a;

local\_b text := b;

BEGIN

RETURN local\_a < local\_b;

END;

$$ LANGUAGE plpgsql;

If there are no parameters of collatable data types, or no common collation can be identified for them, then parameters and local variables use the default collation of their data type (which is usually the database's default collation, but could be different for variables of domain types).

A local variable of a collatable data type can have a different collation associated with it by including the COLLATE option in its declaration, for example

DECLARE

local\_a text COLLATE "en\_US";

This option overrides the collation that would otherwise be given to the variable according to the rules above.

Also, of course explicit COLLATE clauses can be written inside a function if it is desired to force a particular collation to be used in a particular operation. For example,

CREATE FUNCTION less\_than\_c(a text, b text) RETURNS boolean AS $$

BEGIN

RETURN a < b COLLATE "C";

END;

$$ LANGUAGE plpgsql;

This overrides the collations associated with the table columns, parameters, or local variables used in the expression, just as would happen in a plain SQL command.

## 42.4. Expressions

All expressions used in PL/pgSQL statements are processed using the server's main SQL executor. For example, when you write a PL/pgSQL statement like

IF ***expression*** THEN ...

PL/pgSQL will evaluate the expression by feeding a query like

SELECT ***expression***

to the main SQL engine. While forming the SELECT command, any occurrences of PL/pgSQL variable names are replaced by parameters, as discussed in detail in [**Section 42.10.1**](https://www.postgresql.org/docs/10/plpgsql-implementation.html#PLPGSQL-VAR-SUBST). This allows the query plan for the SELECT to be prepared just once and then reused for subsequent evaluations with different values of the variables. Thus, what really happens on first use of an expression is essentially a PREPARE command. For example, if we have declared two integer variables x and y, and we write

IF x < y THEN ...

what happens behind the scenes is equivalent to

PREPARE ***statement\_name***(integer, integer) AS SELECT $1 < $2;

and then this prepared statement is EXECUTEd for each execution of the IF statement, with the current values of the PL/pgSQL variables supplied as parameter values. Normally these details are not important to a PL/pgSQL user, but they are useful to know when trying to diagnose a problem. More information appears in [**Section 42.10.2**](https://www.postgresql.org/docs/10/plpgsql-implementation.html#PLPGSQL-PLAN-CACHING).

## 42.5. Basic Statements

In this section and the following ones, we describe all the statement types that are explicitly understood by PL/pgSQL. Anything not recognized as one of these statement types is presumed to be an SQL command and is sent to the main database engine to execute, as described in [**Section 42.5.2**](https://www.postgresql.org/docs/10/plpgsql-statements.html#PLPGSQL-STATEMENTS-SQL-NORESULT) and [**Section 42.5.3**](https://www.postgresql.org/docs/10/plpgsql-statements.html#PLPGSQL-STATEMENTS-SQL-ONEROW).

### 42.5.1. Assignment

An assignment of a value to a PL/pgSQL variable is written as:

***variable*** { := | = } ***expression***;

As explained previously, the expression in such a statement is evaluated by means of an SQL SELECT command sent to the main database engine. The expression must yield a single value (possibly a row value, if the variable is a row or record variable). The target variable can be a simple variable (optionally qualified with a block name), a field of a row or record variable, or an element of an array that is a simple variable or field. Equal (=) can be used instead of PL/SQL-compliant :=.

If the expression's result data type doesn't match the variable's data type, the value will be coerced as though by an assignment cast (see [**Section 10.4**](https://www.postgresql.org/docs/10/typeconv-query.html)). If no assignment cast is known for the pair of data types involved, the PL/pgSQL interpreter will attempt to convert the result value textually, that is by applying the result type's output function followed by the variable type's input function. Note that this could result in run-time errors generated by the input function, if the string form of the result value is not acceptable to the input function.

Examples:

tax := subtotal \* 0.06;

my\_record.user\_id := 20;

### 42.5.2. Executing a Command With No Result

For any SQL command that does not return rows, for example INSERT without a RETURNING clause, you can execute the command within a PL/pgSQL function just by writing the command.

Any PL/pgSQL variable name appearing in the command text is treated as a parameter, and then the current value of the variable is provided as the parameter value at run time. This is exactly like the processing described earlier for expressions; for details see [**Section 42.10.1**](https://www.postgresql.org/docs/10/plpgsql-implementation.html#PLPGSQL-VAR-SUBST).

When executing a SQL command in this way, PL/pgSQL may cache and re-use the execution plan for the command, as discussed in [**Section 42.10.2**](https://www.postgresql.org/docs/10/plpgsql-implementation.html#PLPGSQL-PLAN-CACHING).

Sometimes it is useful to evaluate an expression or SELECT query but discard the result, for example when calling a function that has side-effects but no useful result value. To do this in PL/pgSQL, use the PERFORM statement:

PERFORM ***query***;

This executes ***query*** and discards the result. Write the ***query*** the same way you would write an SQL SELECT command, but replace the initial keyword SELECT with PERFORM. For WITH queries, use PERFORMand then place the query in parentheses. (In this case, the query can only return one row.) PL/pgSQL variables will be substituted into the query just as for commands that return no result, and the plan is cached in the same way. Also, the special variable FOUND is set to true if the query produced at least one row, or false if it produced no rows (see [**Section 42.5.5**](https://www.postgresql.org/docs/10/plpgsql-statements.html#PLPGSQL-STATEMENTS-DIAGNOSTICS)).

### Note

One might expect that writing SELECT directly would accomplish this result, but at present the only accepted way to do it is PERFORM. A SQL command that can return rows, such as SELECT, will be rejected as an error unless it has an INTO clause as discussed in the next section.

An example:

PERFORM create\_mv('cs\_session\_page\_requests\_mv', my\_query);

### 42.5.3. Executing a Query with a Single-row Result

The result of a SQL command yielding a single row (possibly of multiple columns) can be assigned to a record variable, row-type variable, or list of scalar variables. This is done by writing the base SQL command and adding an INTO clause. For example,

SELECT ***select\_expressions*** INTO [STRICT] ***target*** FROM ...;

INSERT ... RETURNING ***expressions*** INTO [STRICT] ***target***;

UPDATE ... RETURNING ***expressions*** INTO [STRICT] ***target***;

DELETE ... RETURNING ***expressions*** INTO [STRICT] ***target***;

where ***target*** can be a record variable, a row variable, or a comma-separated list of simple variables and record/row fields. PL/pgSQL variables will be substituted into the rest of the query, and the plan is cached, just as described above for commands that do not return rows. This works for SELECT, INSERT/UPDATE/DELETE with RETURNING, and utility commands that return row-set results (such as EXPLAIN). Except for the INTO clause, the SQL command is the same as it would be written outside PL/pgSQL.

### Tip

Note that this interpretation of SELECT with INTO is quite different from PostgreSQL's regular SELECT INTO command, wherein the INTO target is a newly created table. If you want to create a table from a SELECT result inside a PL/pgSQL function, use the syntax CREATE TABLE ... AS SELECT.

If a row or a variable list is used as target, the query's result columns must exactly match the structure of the target as to number and data types, or else a run-time error occurs. When a record variable is the target, it automatically configures itself to the row type of the query result columns.

The INTO clause can appear almost anywhere in the SQL command. Customarily it is written either just before or just after the list of ***select\_expressions*** in a SELECT command, or at the end of the command for other command types. It is recommended that you follow this convention in case the PL/pgSQL parser becomes stricter in future versions.

If STRICT is not specified in the INTO clause, then ***target*** will be set to the first row returned by the query, or to nulls if the query returned no rows. (Note that “the first row” is not well-defined unless you've used ORDER BY.) Any result rows after the first row are discarded. You can check the special FOUND variable (see [**Section 42.5.5**](https://www.postgresql.org/docs/10/plpgsql-statements.html#PLPGSQL-STATEMENTS-DIAGNOSTICS)) to determine whether a row was returned:

SELECT \* INTO myrec FROM emp WHERE empname = myname;

IF NOT FOUND THEN

RAISE EXCEPTION 'employee % not found', myname;

END IF;

If the STRICT option is specified, the query must return exactly one row or a run-time error will be reported, either NO\_DATA\_FOUND (no rows) or TOO\_MANY\_ROWS (more than one row). You can use an exception block if you wish to catch the error, for example:

BEGIN

SELECT \* INTO STRICT myrec FROM emp WHERE empname = myname;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE EXCEPTION 'employee % not found', myname;

WHEN TOO\_MANY\_ROWS THEN

RAISE EXCEPTION 'employee % not unique', myname;

END;

Successful execution of a command with STRICT always sets FOUND to true.

For INSERT/UPDATE/DELETE with RETURNING, PL/pgSQL reports an error for more than one returned row, even when STRICT is not specified. This is because there is no option such as ORDER BY with which to determine which affected row should be returned.

If print\_strict\_params is enabled for the function, then when an error is thrown because the requirements of STRICT are not met, the DETAIL part of the error message will include information about the parameters passed to the query. You can change the print\_strict\_params setting for all functions by setting plpgsql.print\_strict\_params, though only subsequent function compilations will be affected. You can also enable it on a per-function basis by using a compiler option, for example:

CREATE FUNCTION get\_userid(username text) RETURNS int

AS $$

#print\_strict\_params on

DECLARE

userid int;

BEGIN

SELECT users.userid INTO STRICT userid

FROM users WHERE users.username = get\_userid.username;

RETURN userid;

END

$$ LANGUAGE plpgsql;

On failure, this function might produce an error message such as

ERROR: query returned no rows

DETAIL: parameters: $1 = 'nosuchuser'

CONTEXT: PL/pgSQL function get\_userid(text) line 6 at SQL statement

### Note

The STRICT option matches the behavior of Oracle PL/SQL's SELECT INTO and related statements.

To handle cases where you need to process multiple result rows from a SQL query, see [**Section 42.6.4**](https://www.postgresql.org/docs/10/plpgsql-control-structures.html#PLPGSQL-RECORDS-ITERATING).

### 42.5.4. Executing Dynamic Commands

Oftentimes you will want to generate dynamic commands inside your PL/pgSQL functions, that is, commands that will involve different tables or different data types each time they are executed. PL/pgSQL's normal attempts to cache plans for commands (as discussed in [**Section 42.10.2**](https://www.postgresql.org/docs/10/plpgsql-implementation.html#PLPGSQL-PLAN-CACHING)) will not work in such scenarios. To handle this sort of problem, the EXECUTE statement is provided:

EXECUTE ***command-string*** [ INTO [STRICT] ***target*** ] [ USING ***expression*** [, ... ] ];

where ***command-string*** is an expression yielding a string (of type text) containing the command to be executed. The optional ***target*** is a record variable, a row variable, or a comma-separated list of simple variables and record/row fields, into which the results of the command will be stored. The optional USING expressions supply values to be inserted into the command.

No substitution of PL/pgSQL variables is done on the computed command string. Any required variable values must be inserted in the command string as it is constructed; or you can use parameters as described below.

Also, there is no plan caching for commands executed via EXECUTE. Instead, the command is always planned each time the statement is run. Thus the command string can be dynamically created within the function to perform actions on different tables and columns.

The INTO clause specifies where the results of a SQL command returning rows should be assigned. If a row or variable list is provided, it must exactly match the structure of the query's results (when a record variable is used, it will configure itself to match the result structure automatically). If multiple rows are returned, only the first will be assigned to the INTO variable. If no rows are returned, NULL is assigned to the INTO variable(s). If no INTO clause is specified, the query results are discarded.

If the STRICT option is given, an error is reported unless the query produces exactly one row.

The command string can use parameter values, which are referenced in the command as $1, $2, etc. These symbols refer to values supplied in the USING clause. This method is often preferable to inserting data values into the command string as text: it avoids run-time overhead of converting the values to text and back, and it is much less prone to SQL-injection attacks since there is no need for quoting or escaping. An example is:

EXECUTE 'SELECT count(\*) FROM mytable WHERE inserted\_by = $1 AND inserted <= $2'

INTO c

USING checked\_user, checked\_date;

Note that parameter symbols can only be used for data values — if you want to use dynamically determined table or column names, you must insert them into the command string textually. For example, if the preceding query needed to be done against a dynamically selected table, you could do this:

EXECUTE 'SELECT count(\*) FROM '

|| quote\_ident(tabname)

|| ' WHERE inserted\_by = $1 AND inserted <= $2'

INTO c

USING checked\_user, checked\_date;

A cleaner approach is to use format()'s %I specification for table or column names (strings separated by a newline are concatenated):

EXECUTE format('SELECT count(\*) FROM %I '

'WHERE inserted\_by = $1 AND inserted <= $2', tabname)

INTO c

USING checked\_user, checked\_date;

Another restriction on parameter symbols is that they only work in SELECT, INSERT, UPDATE, and DELETE commands. In other statement types (generically called utility statements), you must insert values textually even if they are just data values.

An EXECUTE with a simple constant command string and some USING parameters, as in the first example above, is functionally equivalent to just writing the command directly in PL/pgSQL and allowing replacement of PL/pgSQL variables to happen automatically. The important difference is that EXECUTE will re-plan the command on each execution, generating a plan that is specific to the current parameter values; whereas PL/pgSQL may otherwise create a generic plan and cache it for re-use. In situations where the best plan depends strongly on the parameter values, it can be helpful to use EXECUTE to positively ensure that a generic plan is not selected.

SELECT INTO is not currently supported within EXECUTE; instead, execute a plain SELECT command and specify INTO as part of the EXECUTE itself.

### Note

The PL/pgSQL EXECUTE statement is not related to the [**EXECUTE**](https://www.postgresql.org/docs/10/sql-execute.html) SQL statement supported by the PostgreSQL server. The server's EXECUTE statement cannot be used directly within PL/pgSQL functions (and is not needed).

**Example 42.1. Quoting Values In Dynamic Queries**

When working with dynamic commands you will often have to handle escaping of single quotes. The recommended method for quoting fixed text in your function body is dollar quoting. (If you have legacy code that does not use dollar quoting, please refer to the overview in [**Section 42.11.1**](https://www.postgresql.org/docs/10/plpgsql-development-tips.html#PLPGSQL-QUOTE-TIPS), which can save you some effort when translating said code to a more reasonable scheme.)

Dynamic values require careful handling since they might contain quote characters. An example using format() (this assumes that you are dollar quoting the function body so quote marks need not be doubled):

EXECUTE format('UPDATE tbl SET %I = $1 '

'WHERE key = $2', colname) USING newvalue, keyvalue;

It is also possible to call the quoting functions directly:

EXECUTE 'UPDATE tbl SET '

|| quote\_ident(colname)

|| ' = '

|| quote\_literal(newvalue)

|| ' WHERE key = '

|| quote\_literal(keyvalue);

This example demonstrates the use of the quote\_ident and quote\_literal functions (see [**Section 9.4**](https://www.postgresql.org/docs/10/functions-string.html)). For safety, expressions containing column or table identifiers should be passed through quote\_ident before insertion in a dynamic query. Expressions containing values that should be literal strings in the constructed command should be passed through quote\_literal. These functions take the appropriate steps to return the input text enclosed in double or single quotes respectively, with any embedded special characters properly escaped.

Because quote\_literal is labeled STRICT, it will always return null when called with a null argument. In the above example, if newvalue or keyvalue were null, the entire dynamic query string would become null, leading to an error from EXECUTE. You can avoid this problem by using the quote\_nullable function, which works the same as quote\_literal except that when called with a null argument it returns the string NULL. For example,

EXECUTE 'UPDATE tbl SET '

|| quote\_ident(colname)

|| ' = '

|| quote\_nullable(newvalue)

|| ' WHERE key = '

|| quote\_nullable(keyvalue);

If you are dealing with values that might be null, you should usually use quote\_nullable in place of quote\_literal.

As always, care must be taken to ensure that null values in a query do not deliver unintended results. For example the WHERE clause

'WHERE key = ' || quote\_nullable(keyvalue)

will never succeed if keyvalue is null, because the result of using the equality operator = with a null operand is always null. If you wish null to work like an ordinary key value, you would need to rewrite the above as

'WHERE key IS NOT DISTINCT FROM ' || quote\_nullable(keyvalue)

(At present, IS NOT DISTINCT FROM is handled much less efficiently than =, so don't do this unless you must. See [**Section 9.2**](https://www.postgresql.org/docs/10/functions-comparison.html) for more information on nulls and IS DISTINCT.)

Note that dollar quoting is only useful for quoting fixed text. It would be a very bad idea to try to write this example as:

EXECUTE 'UPDATE tbl SET '

|| quote\_ident(colname)

|| ' = $$'

|| newvalue

|| '$$ WHERE key = '

|| quote\_literal(keyvalue);

because it would break if the contents of newvalue happened to contain $$. The same objection would apply to any other dollar-quoting delimiter you might pick. So, to safely quote text that is not known in advance, you must use quote\_literal, quote\_nullable, or quote\_ident, as appropriate.

Dynamic SQL statements can also be safely constructed using the format function (see [**Section 9.4**](https://www.postgresql.org/docs/10/functions-string.html)). For example:

EXECUTE format('UPDATE tbl SET %I = %L '

'WHERE key = %L', colname, newvalue, keyvalue);

%I is equivalent to quote\_ident, and %L is equivalent to quote\_nullable. The format function can be used in conjunction with the USING clause:

EXECUTE format('UPDATE tbl SET %I = $1 WHERE key = $2', colname)

USING newvalue, keyvalue;

This form is better because the variables are handled in their native data type format, rather than unconditionally converting them to text and quoting them via %L. It is also more efficient.

A much larger example of a dynamic command and EXECUTE can be seen in [**Example 42.10**](https://www.postgresql.org/docs/10/plpgsql-porting.html#PLPGSQL-PORTING-EX2), which builds and executes a CREATE FUNCTION command to define a new function.

### 42.5.5. Obtaining the Result Status

There are several ways to determine the effect of a command. The first method is to use the GET DIAGNOSTICS command, which has the form:

GET [ CURRENT ] DIAGNOSTICS ***variable*** { = | := } ***item*** [ , ... ];

This command allows retrieval of system status indicators. CURRENT is a noise word (but see also GET STACKED DIAGNOSTICS in [**Section 42.6.6.1**](https://www.postgresql.org/docs/10/plpgsql-control-structures.html#PLPGSQL-EXCEPTION-DIAGNOSTICS)). Each ***item*** is a key word identifying a status value to be assigned to the specified ***variable*** (which should be of the right data type to receive it). The currently available status items are shown in [**Table 42.1**](https://www.postgresql.org/docs/10/plpgsql-statements.html#PLPGSQL-CURRENT-DIAGNOSTICS-VALUES). Colon-equal (:=) can be used instead of the SQL-standard = token. An example:

GET DIAGNOSTICS integer\_var = ROW\_COUNT;

**Table 42.1. Available Diagnostics Items**

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| ROW\_COUNT | bigint | the number of rows processed by the most recent SQL command |
| RESULT\_OID | oid | the OID of the last row inserted by the most recent SQL command (only useful after an INSERT command into a table having OIDs) |
| PG\_CONTEXT | text | line(s) of text describing the current call stack (see [**Section 42.6.7**](https://www.postgresql.org/docs/10/plpgsql-control-structures.html#PLPGSQL-CALL-STACK)) |

The second method to determine the effects of a command is to check the special variable named FOUND, which is of type boolean. FOUND starts out false within each PL/pgSQL function call. It is set by each of the following types of statements:

* A SELECT INTO statement sets FOUND true if a row is assigned, false if no row is returned.
* A PERFORM statement sets FOUND true if it produces (and discards) one or more rows, false if no row is produced.
* UPDATE, INSERT, and DELETE statements set FOUND true if at least one row is affected, false if no row is affected.
* A FETCH statement sets FOUND true if it returns a row, false if no row is returned.
* A MOVE statement sets FOUND true if it successfully repositions the cursor, false otherwise.
* A FOR or FOREACH statement sets FOUND true if it iterates one or more times, else false. FOUND is set this way when the loop exits; inside the execution of the loop, FOUND is not modified by the loop statement, although it might be changed by the execution of other statements within the loop body.
* RETURN QUERY and RETURN QUERY EXECUTE statements set FOUND true if the query returns at least one row, false if no row is returned.

Other PL/pgSQL statements do not change the state of FOUND. Note in particular that EXECUTE changes the output of GET DIAGNOSTICS, but does not change FOUND.

FOUND is a local variable within each PL/pgSQL function; any changes to it affect only the current function.

### 42.5.6. Doing Nothing At All

Sometimes a placeholder statement that does nothing is useful. For example, it can indicate that one arm of an if/then/else chain is deliberately empty. For this purpose, use the NULL statement:

NULL;

For example, the following two fragments of code are equivalent:

BEGIN

y := x / 0;

EXCEPTION

WHEN division\_by\_zero THEN

NULL; -- ignore the error

END;

BEGIN

y := x / 0;

EXCEPTION

WHEN division\_by\_zero THEN -- ignore the error

END;

Which is preferable is a matter of taste.

### Note

In Oracle's PL/SQL, empty statement lists are not allowed, and so NULL statements are required for situations such as this. PL/pgSQL allows you to just write nothing, instead.

## 42.6. Control Structures

Control structures are probably the most useful (and important) part of PL/pgSQL. With PL/pgSQL's control structures, you can manipulate PostgreSQL data in a very flexible and powerful way.

### 42.6.1. Returning From a Function

There are two commands available that allow you to return data from a function: RETURN and RETURN NEXT.

#### 42.6.1.1. RETURN

RETURN ***expression***;

RETURN with an expression terminates the function and returns the value of ***expression*** to the caller. This form is used for PL/pgSQL functions that do not return a set.

In a function that returns a scalar type, the expression's result will automatically be cast into the function's return type as described for assignments. But to return a composite (row) value, you must write an expression delivering exactly the requested column set. This may require use of explicit casting.

If you declared the function with output parameters, write just RETURN with no expression. The current values of the output parameter variables will be returned.

If you declared the function to return void, a RETURN statement can be used to exit the function early; but do not write an expression following RETURN.

The return value of a function cannot be left undefined. If control reaches the end of the top-level block of the function without hitting a RETURN statement, a run-time error will occur. This restriction does not apply to functions with output parameters and functions returning void, however. In those cases a RETURN statement is automatically executed if the top-level block finishes.

Some examples:

-- functions returning a scalar type

RETURN 1 + 2;

RETURN scalar\_var;

-- functions returning a composite type

RETURN composite\_type\_var;

RETURN (1, 2, 'three'::text); -- must cast columns to correct types

#### 42.6.1.2. RETURN NEXT And RETURN QUERY

RETURN NEXT ***expression***;

RETURN QUERY ***query***;

RETURN QUERY EXECUTE ***command-string*** [ USING ***expression*** [, ... ] ];

When a PL/pgSQL function is declared to return SETOF ***sometype***, the procedure to follow is slightly different. In that case, the individual items to return are specified by a sequence of RETURN NEXT or RETURN QUERY commands, and then a final RETURN command with no argument is used to indicate that the function has finished executing. RETURN NEXT can be used with both scalar and composite data types; with a composite result type, an entire “table” of results will be returned. RETURN QUERY appends the results of executing a query to the function's result set. RETURN NEXT and RETURN QUERYcan be freely intermixed in a single set-returning function, in which case their results will be concatenated.

RETURN NEXT and RETURN QUERY do not actually return from the function — they simply append zero or more rows to the function's result set. Execution then continues with the next statement in the PL/pgSQL function. As successive RETURN NEXT or RETURN QUERY commands are executed, the result set is built up. A final RETURN, which should have no argument, causes control to exit the function (or you can just let control reach the end of the function).

RETURN QUERY has a variant RETURN QUERY EXECUTE, which specifies the query to be executed dynamically. Parameter expressions can be inserted into the computed query string via USING, in just the same way as in the EXECUTE command.

If you declared the function with output parameters, write just RETURN NEXT with no expression. On each execution, the current values of the output parameter variable(s) will be saved for eventual return as a row of the result. Note that you must declare the function as returning SETOF record when there are multiple output parameters, or SETOF ***sometype*** when there is just one output parameter of type ***sometype***, in order to create a set-returning function with output parameters.

Here is an example of a function using RETURN NEXT:

CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);

INSERT INTO foo VALUES (1, 2, 'three');

INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION get\_all\_foo() RETURNS SETOF foo AS

$BODY$

DECLARE

r foo%rowtype;

BEGIN

FOR r IN

SELECT \* FROM foo WHERE fooid > 0

LOOP

-- can do some processing here

RETURN NEXT r; -- return current row of SELECT

END LOOP;

RETURN;

END

$BODY$

LANGUAGE plpgsql;

SELECT \* FROM get\_all\_foo();

Here is an example of a function using RETURN QUERY:

CREATE FUNCTION get\_available\_flightid(date) RETURNS SETOF integer AS

$BODY$

BEGIN

RETURN QUERY SELECT flightid

FROM flight

WHERE flightdate >= $1

AND flightdate < ($1 + 1);

-- Since execution is not finished, we can check whether rows were returned

-- and raise exception if not.

IF NOT FOUND THEN

RAISE EXCEPTION 'No flight at %.', $1;

END IF;

RETURN;

END

$BODY$

LANGUAGE plpgsql;

-- Returns available flights or raises exception if there are no

-- available flights.

SELECT \* FROM get\_available\_flightid(CURRENT\_DATE);

### Note

The current implementation of RETURN NEXT and RETURN QUERY stores the entire result set before returning from the function, as discussed above. That means that if a PL/pgSQLfunction produces a very large result set, performance might be poor: data will be written to disk to avoid memory exhaustion, but the function itself will not return until the entire result set has been generated. A future version of PL/pgSQL might allow users to define set-returning functions that do not have this limitation. Currently, the point at which data begins being written to disk is controlled by the [**work\_mem**](https://www.postgresql.org/docs/10/runtime-config-resource.html#GUC-WORK-MEM) configuration variable. Administrators who have sufficient memory to store larger result sets in memory should consider increasing this parameter.

### 42.6.2. Conditionals

IF and CASE statements let you execute alternative commands based on certain conditions. PL/pgSQL has three forms of IF:

* IF ... THEN ... END IF
* IF ... THEN ... ELSE ... END IF
* IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF

and two forms of CASE:

* CASE ... WHEN ... THEN ... ELSE ... END CASE
* CASE WHEN ... THEN ... ELSE ... END CASE

#### 42.6.2.1. IF-THEN

IF ***boolean-expression*** THEN

***statements***

END IF;

IF-THEN statements are the simplest form of IF. The statements between THEN and END IF will be executed if the condition is true. Otherwise, they are skipped.

Example:

IF v\_user\_id <> 0 THEN

UPDATE users SET email = v\_email WHERE user\_id = v\_user\_id;

END IF;

#### 42.6.2.2. IF-THEN-ELSE

IF ***boolean-expression*** THEN

***statements***

ELSE

***statements***

END IF;

IF-THEN-ELSE statements add to IF-THEN by letting you specify an alternative set of statements that should be executed if the condition is not true. (Note this includes the case where the condition evaluates to NULL.)

Examples:

IF parentid IS NULL OR parentid = ''

THEN

RETURN fullname;

ELSE

RETURN hp\_true\_filename(parentid) || '/' || fullname;

END IF;

IF v\_count > 0 THEN

INSERT INTO users\_count (count) VALUES (v\_count);

RETURN 't';

ELSE

RETURN 'f';

END IF;

#### 42.6.2.3. IF-THEN-ELSIF

IF ***boolean-expression*** THEN

***statements***

[ ELSIF ***boolean-expression*** THEN

***statements***

[ ELSIF ***boolean-expression*** THEN

***statements***

...]]

[ ELSE

***statements*** ]

END IF;

Sometimes there are more than just two alternatives. IF-THEN-ELSIF provides a convenient method of checking several alternatives in turn. The IF conditions are tested successively until the first one that is true is found. Then the associated statement(s) are executed, after which control passes to the next statement after END IF. (Any subsequent IF conditions are not tested.) If none of the IFconditions is true, then the ELSE block (if any) is executed.

Here is an example:

IF number = 0 THEN

result := 'zero';

ELSIF number > 0 THEN

result := 'positive';

ELSIF number < 0 THEN

result := 'negative';

ELSE

-- hmm, the only other possibility is that number is null

result := 'NULL';

END IF;

The key word ELSIF can also be spelled ELSEIF.

An alternative way of accomplishing the same task is to nest IF-THEN-ELSE statements, as in the following example:

IF demo\_row.sex = 'm' THEN

pretty\_sex := 'man';

ELSE

IF demo\_row.sex = 'f' THEN

pretty\_sex := 'woman';

END IF;

END IF;

However, this method requires writing a matching END IF for each IF, so it is much more cumbersome than using ELSIF when there are many alternatives.

#### 42.6.2.4. Simple CASE

CASE ***search-expression***

WHEN ***expression*** [, ***expression*** [ ... ]] THEN

***statements***

[ WHEN ***expression*** [, ***expression*** [ ... ]] THEN

***statements***

... ]

[ ELSE

***statements*** ]

END CASE;

The simple form of CASE provides conditional execution based on equality of operands. The ***search-expression*** is evaluated (once) and successively compared to each ***expression*** in the WHEN clauses. If a match is found, then the corresponding ***statements*** are executed, and then control passes to the next statement after END CASE. (Subsequent WHEN expressions are not evaluated.) If no match is found, the ELSE ***statements*** are executed; but if ELSE is not present, then a CASE\_NOT\_FOUND exception is raised.

Here is a simple example:

CASE x

WHEN 1, 2 THEN

msg := 'one or two';

ELSE

msg := 'other value than one or two';

END CASE;

#### 42.6.2.5. Searched CASE

CASE

WHEN ***boolean-expression*** THEN

***statements***

[ WHEN ***boolean-expression*** THEN

***statements***

... ]

[ ELSE

***statements*** ]

END CASE;

The searched form of CASE provides conditional execution based on truth of Boolean expressions. Each WHEN clause's ***boolean-expression*** is evaluated in turn, until one is found that yields true. Then the corresponding ***statements*** are executed, and then control passes to the next statement after END CASE. (Subsequent WHEN expressions are not evaluated.) If no true result is found, the ELSE***statements*** are executed; but if ELSE is not present, then a CASE\_NOT\_FOUND exception is raised.

Here is an example:

CASE

WHEN x BETWEEN 0 AND 10 THEN

msg := 'value is between zero and ten';

WHEN x BETWEEN 11 AND 20 THEN

msg := 'value is between eleven and twenty';

END CASE;

This form of CASE is entirely equivalent to IF-THEN-ELSIF, except for the rule that reaching an omitted ELSE clause results in an error rather than doing nothing.

### 42.6.3. Simple Loops

With the LOOP, EXIT, CONTINUE, WHILE, FOR, and FOREACH statements, you can arrange for your PL/pgSQL function to repeat a series of commands.

#### 42.6.3.1. LOOP

[ <<***label***>> ]

LOOP

***statements***

END LOOP [ ***label*** ];

LOOP defines an unconditional loop that is repeated indefinitely until terminated by an EXIT or RETURN statement. The optional ***label*** can be used by EXIT and CONTINUE statements within nested loops to specify which loop those statements refer to.

#### 42.6.3.2. EXIT

EXIT [ ***label*** ] [ WHEN ***boolean-expression*** ];

If no ***label*** is given, the innermost loop is terminated and the statement following END LOOP is executed next. If ***label*** is given, it must be the label of the current or some outer level of nested loop or block. Then the named loop or block is terminated and control continues with the statement after the loop's/block's corresponding END.

If WHEN is specified, the loop exit occurs only if ***boolean-expression*** is true. Otherwise, control passes to the statement after EXIT.

EXIT can be used with all types of loops; it is not limited to use with unconditional loops.

When used with a BEGIN block, EXIT passes control to the next statement after the end of the block. Note that a label must be used for this purpose; an unlabeled EXIT is never considered to match a BEGIN block. (This is a change from pre-8.4 releases of PostgreSQL, which would allow an unlabeled EXIT to match a BEGIN block.)

Examples:

LOOP

-- some computations

IF count > 0 THEN

EXIT; -- exit loop

END IF;

END LOOP;

LOOP

-- some computations

EXIT WHEN count > 0; -- same result as previous example

END LOOP;

<<ablock>>

BEGIN

-- some computations

IF stocks > 100000 THEN

EXIT ablock; -- causes exit from the BEGIN block

END IF;

-- computations here will be skipped when stocks > 100000

END;

#### 42.6.3.3. CONTINUE

CONTINUE [ ***label*** ] [ WHEN ***boolean-expression*** ];

If no ***label*** is given, the next iteration of the innermost loop is begun. That is, all statements remaining in the loop body are skipped, and control returns to the loop control expression (if any) to determine whether another loop iteration is needed. If ***label*** is present, it specifies the label of the loop whose execution will be continued.

If WHEN is specified, the next iteration of the loop is begun only if ***boolean-expression*** is true. Otherwise, control passes to the statement after CONTINUE.

CONTINUE can be used with all types of loops; it is not limited to use with unconditional loops.

Examples:

LOOP

-- some computations

EXIT WHEN count > 100;

CONTINUE WHEN count < 50;

-- some computations for count IN [50 .. 100]

END LOOP;

#### 42.6.3.4. WHILE

[ <<***label***>> ]

WHILE ***boolean-expression*** LOOP

***statements***

END LOOP [ ***label*** ];

The WHILE statement repeats a sequence of statements so long as the ***boolean-expression*** evaluates to true. The expression is checked just before each entry to the loop body.

For example:

WHILE amount\_owed > 0 AND gift\_certificate\_balance > 0 LOOP

-- some computations here

END LOOP;

WHILE NOT done LOOP

-- some computations here

END LOOP;

#### 42.6.3.5. FOR (Integer Variant)

[ <<***label***>> ]

FOR ***name*** IN [ REVERSE ] ***expression*** .. ***expression*** [ BY ***expression*** ] LOOP

***statements***

END LOOP [ ***label*** ];

This form of FOR creates a loop that iterates over a range of integer values. The variable ***name*** is automatically defined as type integer and exists only inside the loop (any existing definition of the variable name is ignored within the loop). The two expressions giving the lower and upper bound of the range are evaluated once when entering the loop. If the BY clause isn't specified the iteration step is 1, otherwise it's the value specified in the BY clause, which again is evaluated once on loop entry. If REVERSE is specified then the step value is subtracted, rather than added, after each iteration.

Some examples of integer FOR loops:

FOR i IN 1..10 LOOP

-- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop

END LOOP;

FOR i IN REVERSE 10..1 LOOP

-- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop

END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP

-- i will take on the values 10,8,6,4,2 within the loop

END LOOP;

If the lower bound is greater than the upper bound (or less than, in the REVERSE case), the loop body is not executed at all. No error is raised.

If a ***label*** is attached to the FOR loop then the integer loop variable can be referenced with a qualified name, using that ***label***.

### 42.6.4. Looping Through Query Results

Using a different type of FOR loop, you can iterate through the results of a query and manipulate that data accordingly. The syntax is:

[ <<***label***>> ]

FOR ***target*** IN ***query*** LOOP

***statements***

END LOOP [ ***label*** ];

The ***target*** is a record variable, row variable, or comma-separated list of scalar variables. The ***target*** is successively assigned each row resulting from the ***query*** and the loop body is executed for each row. Here is an example:

CREATE FUNCTION cs\_refresh\_mviews() RETURNS integer AS $$

DECLARE

mviews RECORD;

BEGIN

RAISE NOTICE 'Refreshing materialized views...';

FOR mviews IN SELECT \* FROM cs\_materialized\_views ORDER BY sort\_key LOOP

-- Now "mviews" has one record from cs\_materialized\_views

RAISE NOTICE 'Refreshing materialized view %s ...', quote\_ident(mviews.mv\_name);

EXECUTE format('TRUNCATE TABLE %I', mviews.mv\_name);

EXECUTE format('INSERT INTO %I %s', mviews.mv\_name, mviews.mv\_query);

END LOOP;

RAISE NOTICE 'Done refreshing materialized views.';

RETURN 1;

END;

$$ LANGUAGE plpgsql;

If the loop is terminated by an EXIT statement, the last assigned row value is still accessible after the loop.

The ***query*** used in this type of FOR statement can be any SQL command that returns rows to the caller: SELECT is the most common case, but you can also use INSERT, UPDATE, or DELETE with a RETURNINGclause. Some utility commands such as EXPLAIN will work too.

PL/pgSQL variables are substituted into the query text, and the query plan is cached for possible re-use, as discussed in detail in [**Section 42.10.1**](https://www.postgresql.org/docs/10/plpgsql-implementation.html#PLPGSQL-VAR-SUBST) and [**Section 42.10.2**](https://www.postgresql.org/docs/10/plpgsql-implementation.html#PLPGSQL-PLAN-CACHING).

The FOR-IN-EXECUTE statement is another way to iterate over rows:

[ <<***label***>> ]

FOR ***target*** IN EXECUTE ***text\_expression*** [ USING ***expression*** [, ... ] ] LOOP

***statements***

END LOOP [ ***label*** ];

This is like the previous form, except that the source query is specified as a string expression, which is evaluated and replanned on each entry to the FOR loop. This allows the programmer to choose the speed of a preplanned query or the flexibility of a dynamic query, just as with a plain EXECUTE statement. As with EXECUTE, parameter values can be inserted into the dynamic command via USING.

Another way to specify the query whose results should be iterated through is to declare it as a cursor. This is described in [**Section 42.7.4**](https://www.postgresql.org/docs/10/plpgsql-cursors.html#PLPGSQL-CURSOR-FOR-LOOP).

### 42.6.5. Looping Through Arrays

The FOREACH loop is much like a FOR loop, but instead of iterating through the rows returned by a SQL query, it iterates through the elements of an array value. (In general, FOREACH is meant for looping through components of a composite-valued expression; variants for looping through composites besides arrays may be added in future.) The FOREACH statement to loop over an array is:

[ <<***label***>> ]

FOREACH ***target*** [ SLICE ***number*** ] IN ARRAY ***expression*** LOOP

***statements***

END LOOP [ ***label*** ];

Without SLICE, or if SLICE 0 is specified, the loop iterates through individual elements of the array produced by evaluating the ***expression***. The ***target*** variable is assigned each element value in sequence, and the loop body is executed for each element. Here is an example of looping through the elements of an integer array:

CREATE FUNCTION sum(int[]) RETURNS int8 AS $$

DECLARE

s int8 := 0;

x int;

BEGIN

FOREACH x IN ARRAY $1

LOOP

s := s + x;

END LOOP;

RETURN s;

END;

$$ LANGUAGE plpgsql;

The elements are visited in storage order, regardless of the number of array dimensions. Although the ***target*** is usually just a single variable, it can be a list of variables when looping through an array of composite values (records). In that case, for each array element, the variables are assigned from successive columns of the composite value.

With a positive SLICE value, FOREACH iterates through slices of the array rather than single elements. The SLICE value must be an integer constant not larger than the number of dimensions of the array. The ***target*** variable must be an array, and it receives successive slices of the array value, where each slice is of the number of dimensions specified by SLICE. Here is an example of iterating through one-dimensional slices:

CREATE FUNCTION scan\_rows(int[]) RETURNS void AS $$

DECLARE

x int[];

BEGIN

FOREACH x SLICE 1 IN ARRAY $1

LOOP

RAISE NOTICE 'row = %', x;

END LOOP;

END;

$$ LANGUAGE plpgsql;

SELECT scan\_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE: row = {1,2,3}

NOTICE: row = {4,5,6}

NOTICE: row = {7,8,9}

NOTICE: row = {10,11,12}

### 42.6.6. Trapping Errors

By default, any error occurring in a PL/pgSQL function aborts execution of the function, and indeed of the surrounding transaction as well. You can trap errors and recover from them by using a BEGINblock with an EXCEPTION clause. The syntax is an extension of the normal syntax for a BEGIN block:

[ <<***label***>> ]

[ DECLARE

***declarations*** ]

BEGIN

***statements***

EXCEPTION

WHEN ***condition*** [ OR ***condition*** ... ] THEN

***handler\_statements***

[ WHEN ***condition*** [ OR ***condition*** ... ] THEN

***handler\_statements***

... ]

END;

If no error occurs, this form of block simply executes all the ***statements***, and then control passes to the next statement after END. But if an error occurs within the ***statements***, further processing of the ***statements*** is abandoned, and control passes to the EXCEPTION list. The list is searched for the first ***condition*** matching the error that occurred. If a match is found, the corresponding ***handler\_statements*** are executed, and then control passes to the next statement after END. If no match is found, the error propagates out as though the EXCEPTION clause were not there at all: the error can be caught by an enclosing block with EXCEPTION, or if there is none it aborts processing of the function.

The ***condition*** names can be any of those shown in [**Appendix A**](https://www.postgresql.org/docs/10/errcodes-appendix.html). A category name matches any error within its category. The special condition name OTHERS matches every error type except QUERY\_CANCELED and ASSERT\_FAILURE. (It is possible, but often unwise, to trap those two error types by name.) Condition names are not case-sensitive. Also, an error condition can be specified by SQLSTATE code; for example these are equivalent:

WHEN division\_by\_zero THEN ...

WHEN SQLSTATE '22012' THEN ...

If a new error occurs within the selected ***handler\_statements***, it cannot be caught by this EXCEPTION clause, but is propagated out. A surrounding EXCEPTION clause could catch it.

When an error is caught by an EXCEPTION clause, the local variables of the PL/pgSQL function remain as they were when the error occurred, but all changes to persistent database state within the block are rolled back. As an example, consider this fragment:

INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');

BEGIN

UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';

x := x + 1;

y := x / 0;

EXCEPTION

WHEN division\_by\_zero THEN

RAISE NOTICE 'caught division\_by\_zero';

RETURN x;

END;

When control reaches the assignment to y, it will fail with a division\_by\_zero error. This will be caught by the EXCEPTION clause. The value returned in the RETURN statement will be the incremented value of x, but the effects of the UPDATE command will have been rolled back. The INSERT command preceding the block is not rolled back, however, so the end result is that the database contains Tom Jones not Joe Jones.

### Tip

A block containing an EXCEPTION clause is significantly more expensive to enter and exit than a block without one. Therefore, don't use EXCEPTION without need.

**Example 42.2. Exceptions with**UPDATE**/**INSERT

This example uses exception handling to perform either UPDATE or INSERT, as appropriate. It is recommended that applications use INSERT with ON CONFLICT DO UPDATE rather than actually using this pattern. This example serves primarily to illustrate use of PL/pgSQL control flow structures:

CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge\_db(key INT, data TEXT) RETURNS VOID AS

$$

BEGIN

LOOP

-- first try to update the key

UPDATE db SET b = data WHERE a = key;

IF found THEN

RETURN;

END IF;

-- not there, so try to insert the key

-- if someone else inserts the same key concurrently,

-- we could get a unique-key failure

BEGIN

INSERT INTO db(a,b) VALUES (key, data);

RETURN;

EXCEPTION WHEN unique\_violation THEN

-- Do nothing, and loop to try the UPDATE again.

END;

END LOOP;

END;

$$

LANGUAGE plpgsql;

SELECT merge\_db(1, 'david');

SELECT merge\_db(1, 'dennis');

This coding assumes the unique\_violation error is caused by the INSERT, and not by, say, an INSERT in a trigger function on the table. It might also misbehave if there is more than one unique index on the table, since it will retry the operation regardless of which index caused the error. More safety could be had by using the features discussed next to check that the trapped error was the one expected.

#### 42.6.6.1. Obtaining Information About An Error

Exception handlers frequently need to identify the specific error that occurred. There are two ways to get information about the current exception in PL/pgSQL: special variables and the GET STACKED DIAGNOSTICS command.

Within an exception handler, the special variable SQLSTATE contains the error code that corresponds to the exception that was raised (refer to [**Table A.1**](https://www.postgresql.org/docs/10/errcodes-appendix.html#ERRCODES-TABLE) for a list of possible error codes). The special variable SQLERRM contains the error message associated with the exception. These variables are undefined outside exception handlers.

Within an exception handler, one may also retrieve information about the current exception by using the GET STACKED DIAGNOSTICS command, which has the form:

GET STACKED DIAGNOSTICS ***variable*** { = | := } ***item*** [ , ... ];

Each ***item*** is a key word identifying a status value to be assigned to the specified ***variable*** (which should be of the right data type to receive it). The currently available status items are shown in [**Table 42.2**](https://www.postgresql.org/docs/10/plpgsql-control-structures.html#PLPGSQL-EXCEPTION-DIAGNOSTICS-VALUES).

**Table 42.2. Error Diagnostics Items**

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| RETURNED\_SQLSTATE | text | the SQLSTATE error code of the exception |
| COLUMN\_NAME | text | the name of the column related to exception |
| CONSTRAINT\_NAME | text | the name of the constraint related to exception |
| PG\_DATATYPE\_NAME | text | the name of the data type related to exception |
| MESSAGE\_TEXT | text | the text of the exception's primary message |
| TABLE\_NAME | text | the name of the table related to exception |
| SCHEMA\_NAME | text | the name of the schema related to exception |
| PG\_EXCEPTION\_DETAIL | text | the text of the exception's detail message, if any |
| PG\_EXCEPTION\_HINT | text | the text of the exception's hint message, if any |
| PG\_EXCEPTION\_CONTEXT | text | line(s) of text describing the call stack at the time of the exception (see [**Section 42.6.7**](https://www.postgresql.org/docs/10/plpgsql-control-structures.html#PLPGSQL-CALL-STACK)) |

If the exception did not set a value for an item, an empty string will be returned.

Here is an example:

DECLARE

text\_var1 text;

text\_var2 text;

text\_var3 text;

BEGIN

-- some processing which might cause an exception

...

EXCEPTION WHEN OTHERS THEN

GET STACKED DIAGNOSTICS text\_var1 = MESSAGE\_TEXT,

text\_var2 = PG\_EXCEPTION\_DETAIL,

text\_var3 = PG\_EXCEPTION\_HINT;

END;

### 42.6.7. Obtaining Execution Location Information

The GET DIAGNOSTICS command, previously described in [**Section 42.5.5**](https://www.postgresql.org/docs/10/plpgsql-statements.html#PLPGSQL-STATEMENTS-DIAGNOSTICS), retrieves information about current execution state (whereas the GET STACKED DIAGNOSTICS command discussed above reports information about the execution state as of a previous error). Its PG\_CONTEXT status item is useful for identifying the current execution location. PG\_CONTEXT returns a text string with line(s) of text describing the call stack. The first line refers to the current function and currently executing GET DIAGNOSTICS command. The second and any subsequent lines refer to calling functions further up the call stack. For example:

CREATE OR REPLACE FUNCTION outer\_func() RETURNS integer AS $$

BEGIN

RETURN inner\_func();

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION inner\_func() RETURNS integer AS $$

DECLARE

stack text;

BEGIN

GET DIAGNOSTICS stack = PG\_CONTEXT;

RAISE NOTICE E'--- Call Stack ---\n%', stack;

RETURN 1;

END;

$$ LANGUAGE plpgsql;

SELECT outer\_func();

NOTICE: --- Call Stack ---

PL/pgSQL function inner\_func() line 5 at GET DIAGNOSTICS

PL/pgSQL function outer\_func() line 3 at RETURN

CONTEXT: PL/pgSQL function outer\_func() line 3 at RETURN

outer\_func

------------

1

(1 row)

GET STACKED DIAGNOSTICS ... PG\_EXCEPTION\_CONTEXT returns the same sort of stack trace, but describing the location at which an error was detected, rather than the current location.

## 42.7. Cursors

Rather than executing a whole query at once, it is possible to set up a cursor that encapsulates the query, and then read the query result a few rows at a time. One reason for doing this is to avoid memory overrun when the result contains a large number of rows. (However, PL/pgSQL users do not normally need to worry about that, since FOR loops automatically use a cursor internally to avoid memory problems.) A more interesting usage is to return a reference to a cursor that a function has created, allowing the caller to read the rows. This provides an efficient way to return large row sets from functions.

### 42.7.1. Declaring Cursor Variables

All access to cursors in PL/pgSQL goes through cursor variables, which are always of the special data type refcursor. One way to create a cursor variable is just to declare it as a variable of type refcursor. Another way is to use the cursor declaration syntax, which in general is:

***name*** [ [ NO ] SCROLL ] CURSOR [ ( ***arguments*** ) ] FOR ***query***;

(FOR can be replaced by IS for Oracle compatibility.) If SCROLL is specified, the cursor will be capable of scrolling backward; if NO SCROLL is specified, backward fetches will be rejected; if neither specification appears, it is query-dependent whether backward fetches will be allowed. ***arguments***, if specified, is a comma-separated list of pairs ***name*** ***datatype*** that define names to be replaced by parameter values in the given query. The actual values to substitute for these names will be specified later, when the cursor is opened.

Some examples:

DECLARE

curs1 refcursor;

curs2 CURSOR FOR SELECT \* FROM tenk1;

curs3 CURSOR (key integer) FOR SELECT \* FROM tenk1 WHERE unique1 = key;

All three of these variables have the data type refcursor, but the first can be used with any query, while the second has a fully specified query already bound to it, and the last has a parameterized query bound to it. (key will be replaced by an integer parameter value when the cursor is opened.) The variable curs1 is said to be unbound since it is not bound to any particular query.

### 42.7.2. Opening Cursors

Before a cursor can be used to retrieve rows, it must be opened. (This is the equivalent action to the SQL command DECLARE CURSOR.) PL/pgSQL has three forms of the OPEN statement, two of which use unbound cursor variables while the third uses a bound cursor variable.

### Note

Bound cursor variables can also be used without explicitly opening the cursor, via the FORstatement described in [**Section 42.7.4**](https://www.postgresql.org/docs/10/plpgsql-cursors.html#PLPGSQL-CURSOR-FOR-LOOP).

#### 42.7.2.1. OPEN FOR *Query*

OPEN ***unbound\_cursorvar*** [ [ NO ] SCROLL ] FOR ***query***;

The cursor variable is opened and given the specified query to execute. The cursor cannot be open already, and it must have been declared as an unbound cursor variable (that is, as a simple refcursor variable). The query must be a SELECT, or something else that returns rows (such as EXPLAIN). The query is treated in the same way as other SQL commands in PL/pgSQL: PL/pgSQL variable names are substituted, and the query plan is cached for possible reuse. When a PL/pgSQL variable is substituted into the cursor query, the value that is substituted is the one it has at the time of the OPEN; subsequent changes to the variable will not affect the cursor's behavior. The SCROLL and NO SCROLL options have the same meanings as for a bound cursor.

An example:

OPEN curs1 FOR SELECT \* FROM foo WHERE key = mykey;

#### 42.7.2.2. OPEN FOR EXECUTE

OPEN ***unbound\_cursorvar*** [ [ NO ] SCROLL ] FOR EXECUTE ***query\_string***

[ USING ***expression*** [, ... ] ];

The cursor variable is opened and given the specified query to execute. The cursor cannot be open already, and it must have been declared as an unbound cursor variable (that is, as a simple refcursor variable). The query is specified as a string expression, in the same way as in the EXECUTE command. As usual, this gives flexibility so the query plan can vary from one run to the next (see [**Section 42.10.2**](https://www.postgresql.org/docs/10/plpgsql-implementation.html#PLPGSQL-PLAN-CACHING)), and it also means that variable substitution is not done on the command string. As with EXECUTE, parameter values can be inserted into the dynamic command via format() and USING. The SCROLL and NO SCROLL options have the same meanings as for a bound cursor.

An example:

OPEN curs1 FOR EXECUTE format('SELECT \* FROM %I WHERE col1 = $1',tabname) USING keyvalue;

In this example, the table name is inserted into the query via format(). The comparison value for col1 is inserted via a USING parameter, so it needs no quoting.

#### 42.7.2.3. Opening A Bound Cursor

OPEN ***bound\_cursorvar*** [ ( [ ***argument\_name*** := ] ***argument\_value*** [, ...] ) ];

This form of OPEN is used to open a cursor variable whose query was bound to it when it was declared. The cursor cannot be open already. A list of actual argument value expressions must appear if and only if the cursor was declared to take arguments. These values will be substituted in the query.

The query plan for a bound cursor is always considered cacheable; there is no equivalent of EXECUTE in this case. Notice that SCROLL and NO SCROLL cannot be specified in OPEN, as the cursor's scrolling behavior was already determined.

Argument values can be passed using either positional or named notation. In positional notation, all arguments are specified in order. In named notation, each argument's name is specified using :=to separate it from the argument expression. Similar to calling functions, described in [**Section 4.3**](https://www.postgresql.org/docs/10/sql-syntax-calling-funcs.html), it is also allowed to mix positional and named notation.

Examples (these use the cursor declaration examples above):

OPEN curs2;

OPEN curs3(42);

OPEN curs3(key := 42);

Because variable substitution is done on a bound cursor's query, there are really two ways to pass values into the cursor: either with an explicit argument to OPEN, or implicitly by referencing a PL/pgSQL variable in the query. However, only variables declared before the bound cursor was declared will be substituted into it. In either case the value to be passed is determined at the time of the OPEN. For example, another way to get the same effect as the curs3 example above is

DECLARE

key integer;

curs4 CURSOR FOR SELECT \* FROM tenk1 WHERE unique1 = key;

BEGIN

key := 42;

OPEN curs4;

### 42.7.3. Using Cursors

Once a cursor has been opened, it can be manipulated with the statements described here.

These manipulations need not occur in the same function that opened the cursor to begin with. You can return a refcursor value out of a function and let the caller operate on the cursor. (Internally, a refcursor value is simply the string name of a so-called portal containing the active query for the cursor. This name can be passed around, assigned to other refcursor variables, and so on, without disturbing the portal.)

All portals are implicitly closed at transaction end. Therefore a refcursor value is usable to reference an open cursor only until the end of the transaction.

#### 42.7.3.1. FETCH

FETCH [ ***direction*** { FROM | IN } ] ***cursor*** INTO ***target***;

FETCH retrieves the next row from the cursor into a target, which might be a row variable, a record variable, or a comma-separated list of simple variables, just like SELECT INTO. If there is no next row, the target is set to NULL(s). As with SELECT INTO, the special variable FOUND can be checked to see whether a row was obtained or not.

The ***direction*** clause can be any of the variants allowed in the SQL [**FETCH**](https://www.postgresql.org/docs/10/sql-fetch.html) command except the ones that can fetch more than one row; namely, it can be NEXT, PRIOR, FIRST, LAST, ABSOLUTE ***count***, RELATIVE ***count***, FORWARD, or BACKWARD. Omitting ***direction*** is the same as specifying NEXT. In the forms using a ***count***, the ***count*** can be any integer-valued expression (unlike the SQL FETCH command, which only allows an integer constant). ***direction*** values that require moving backward are likely to fail unless the cursor was declared or opened with the SCROLL option.

***cursor*** must be the name of a refcursor variable that references an open cursor portal.

Examples:

FETCH curs1 INTO rowvar;

FETCH curs2 INTO foo, bar, baz;

FETCH LAST FROM curs3 INTO x, y;

FETCH RELATIVE -2 FROM curs4 INTO x;

#### 42.7.3.2. MOVE

MOVE [ ***direction*** { FROM | IN } ] ***cursor***;

MOVE repositions a cursor without retrieving any data. MOVE works exactly like the FETCH command, except it only repositions the cursor and does not return the row moved to. As with SELECT INTO, the special variable FOUND can be checked to see whether there was a next row to move to.

Examples:

MOVE curs1;

MOVE LAST FROM curs3;

MOVE RELATIVE -2 FROM curs4;

MOVE FORWARD 2 FROM curs4;

#### 42.7.3.3. UPDATE/DELETE WHERE CURRENT OF

UPDATE ***table*** SET ... WHERE CURRENT OF ***cursor***;

DELETE FROM ***table*** WHERE CURRENT OF ***cursor***;

When a cursor is positioned on a table row, that row can be updated or deleted using the cursor to identify the row. There are restrictions on what the cursor's query can be (in particular, no grouping) and it's best to use FOR UPDATE in the cursor. For more information see the [**DECLARE**](https://www.postgresql.org/docs/10/sql-declare.html) reference page.

An example:

UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;

#### 42.7.3.4. CLOSE

CLOSE ***cursor***;

CLOSE closes the portal underlying an open cursor. This can be used to release resources earlier than end of transaction, or to free up the cursor variable to be opened again.

An example:

CLOSE curs1;

#### 42.7.3.5. Returning Cursors

PL/pgSQL functions can return cursors to the caller. This is useful to return multiple rows or columns, especially with very large result sets. To do this, the function opens the cursor and returns the cursor name to the caller (or simply opens the cursor using a portal name specified by or otherwise known to the caller). The caller can then fetch rows from the cursor. The cursor can be closed by the caller, or it will be closed automatically when the transaction closes.

The portal name used for a cursor can be specified by the programmer or automatically generated. To specify a portal name, simply assign a string to the refcursor variable before opening it. The string value of the refcursor variable will be used by OPEN as the name of the underlying portal. However, if the refcursor variable is null, OPEN automatically generates a name that does not conflict with any existing portal, and assigns it to the refcursor variable.

### Note

A bound cursor variable is initialized to the string value representing its name, so that the portal name is the same as the cursor variable name, unless the programmer overrides it by assignment before opening the cursor. But an unbound cursor variable defaults to the null value initially, so it will receive an automatically-generated unique name, unless overridden.

The following example shows one way a cursor name can be supplied by the caller:

CREATE TABLE test (col text);

INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '

BEGIN

OPEN $1 FOR SELECT col FROM test;

RETURN $1;

END;

' LANGUAGE plpgsql;

BEGIN;

SELECT reffunc('funccursor');

FETCH ALL IN funccursor;

COMMIT;

The following example uses automatic cursor name generation:

CREATE FUNCTION reffunc2() RETURNS refcursor AS '

DECLARE

ref refcursor;

BEGIN

OPEN ref FOR SELECT col FROM test;

RETURN ref;

END;

' LANGUAGE plpgsql;

-- need to be in a transaction to use cursors.

BEGIN;

SELECT reffunc2();

reffunc2

--------------------

<unnamed cursor 1>

(1 row)

FETCH ALL IN "<unnamed cursor 1>";

COMMIT;

The following example shows one way to return multiple cursors from a single function:

CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$

BEGIN

OPEN $1 FOR SELECT \* FROM table\_1;

RETURN NEXT $1;

OPEN $2 FOR SELECT \* FROM table\_2;

RETURN NEXT $2;

END;

$$ LANGUAGE plpgsql;

-- need to be in a transaction to use cursors.

BEGIN;

SELECT \* FROM myfunc('a', 'b');

FETCH ALL FROM a;

FETCH ALL FROM b;

COMMIT;

### 42.7.4. Looping Through a Cursor's Result

There is a variant of the FOR statement that allows iterating through the rows returned by a cursor. The syntax is:

[ <<***label***>> ]

FOR ***recordvar*** IN ***bound\_cursorvar*** [ ( [ ***argument\_name*** := ] ***argument\_value*** [, ...] ) ] LOOP

***statements***

END LOOP [ ***label*** ];

The cursor variable must have been bound to some query when it was declared, and it cannot be open already. The FOR statement automatically opens the cursor, and it closes the cursor again when the loop exits. A list of actual argument value expressions must appear if and only if the cursor was declared to take arguments. These values will be substituted in the query, in just the same way as during an OPEN (see [**Section 42.7.2.3**](https://www.postgresql.org/docs/10/plpgsql-cursors.html#PLPGSQL-OPEN-BOUND-CURSOR)).

The variable ***recordvar*** is automatically defined as type record and exists only inside the loop (any existing definition of the variable name is ignored within the loop). Each row returned by the cursor is successively assigned to this record variable and the loop body is executed.

## 42.8. Errors and Messages

### 42.8.1. Reporting Errors and Messages

Use the RAISE statement to report messages and raise errors.

RAISE [ ***level*** ] '***format***' [, ***expression*** [, ... ]] [ USING ***option*** = ***expression*** [, ... ] ];

RAISE [ ***level*** ] ***condition\_name*** [ USING ***option*** = ***expression*** [, ... ] ];

RAISE [ ***level*** ] SQLSTATE '***sqlstate***' [ USING ***option*** = ***expression*** [, ... ] ];

RAISE [ ***level*** ] USING ***option*** = ***expression*** [, ... ];

RAISE ;

The ***level*** option specifies the error severity. Allowed levels are DEBUG, LOG, INFO, NOTICE, WARNING, and EXCEPTION, with EXCEPTION being the default. EXCEPTION raises an error (which normally aborts the current transaction); the other levels only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the [**log\_min\_messages**](https://www.postgresql.org/docs/10/runtime-config-logging.html#GUC-LOG-MIN-MESSAGES) and [**client\_min\_messages**](https://www.postgresql.org/docs/10/runtime-config-client.html#GUC-CLIENT-MIN-MESSAGES) configuration variables. See [**Chapter 19**](https://www.postgresql.org/docs/10/runtime-config.html) for more information.

After ***level*** if any, you can write a ***format*** (which must be a simple string literal, not an expression). The format string specifies the error message text to be reported. The format string can be followed by optional argument expressions to be inserted into the message. Inside the format string, % is replaced by the string representation of the next optional argument's value. Write %% to emit a literal %. The number of arguments must match the number of % placeholders in the format string, or an error is raised during the compilation of the function.

In this example, the value of v\_job\_id will replace the % in the string:

RAISE NOTICE 'Calling cs\_create\_job(%)', v\_job\_id;

You can attach additional information to the error report by writing USING followed by ***option*** = ***expression*** items. Each ***expression*** can be any string-valued expression. The allowed ***option*** key words are:

MESSAGE

Sets the error message text. This option can't be used in the form of RAISE that includes a format string before USING.

DETAIL

Supplies an error detail message.

HINT

Supplies a hint message.

ERRCODE

Specifies the error code (SQLSTATE) to report, either by condition name, as shown in [**Appendix A**](https://www.postgresql.org/docs/10/errcodes-appendix.html), or directly as a five-character SQLSTATE code.

COLUMN  
CONSTRAINT  
DATATYPE  
TABLE  
SCHEMA

Supplies the name of a related object.

This example will abort the transaction with the given error message and hint:

RAISE EXCEPTION 'Nonexistent ID --> %', user\_id

USING HINT = 'Please check your user ID';

These two examples show equivalent ways of setting the SQLSTATE:

RAISE 'Duplicate user ID: %', user\_id USING ERRCODE = 'unique\_violation';

RAISE 'Duplicate user ID: %', user\_id USING ERRCODE = '23505';

There is a second RAISE syntax in which the main argument is the condition name or SQLSTATE to be reported, for example:

RAISE division\_by\_zero;

RAISE SQLSTATE '22012';

In this syntax, USING can be used to supply a custom error message, detail, or hint. Another way to do the earlier example is

RAISE unique\_violation USING MESSAGE = 'Duplicate user ID: ' || user\_id;

Still another variant is to write RAISE USING or RAISE ***level*** USING and put everything else into the USING list.

The last variant of RAISE has no parameters at all. This form can only be used inside a BEGIN block's EXCEPTION clause; it causes the error currently being handled to be re-thrown.

### Note

Before PostgreSQL 9.1, RAISE without parameters was interpreted as re-throwing the error from the block containing the active exception handler. Thus an EXCEPTION clause nested within that handler could not catch it, even if the RAISE was within the nested EXCEPTIONclause's block. This was deemed surprising as well as being incompatible with Oracle's PL/SQL.

If no condition name nor SQLSTATE is specified in a RAISE EXCEPTION command, the default is to use RAISE\_EXCEPTION (P0001). If no message text is specified, the default is to use the condition name or SQLSTATE as message text.

### Note

When specifying an error code by SQLSTATE code, you are not limited to the predefined error codes, but can select any error code consisting of five digits and/or upper-case ASCII letters, other than 00000. It is recommended that you avoid throwing error codes that end in three zeroes, because these are category codes and can only be trapped by trapping the whole category.

### 42.8.2. Checking Assertions

The ASSERT statement is a convenient shorthand for inserting debugging checks into PL/pgSQL functions.

ASSERT ***condition*** [ , ***message*** ];

The ***condition*** is a Boolean expression that is expected to always evaluate to true; if it does, the ASSERT statement does nothing further. If the result is false or null, then an ASSERT\_FAILURE exception is raised. (If an error occurs while evaluating the ***condition***, it is reported as a normal error.)

If the optional ***message*** is provided, it is an expression whose result (if not null) replaces the default error message text “assertion failed”, should the ***condition*** fail. The ***message*** expression is not evaluated in the normal case where the assertion succeeds.

Testing of assertions can be enabled or disabled via the configuration parameter plpgsql.check\_asserts, which takes a Boolean value; the default is on. If this parameter is off then ASSERT statements do nothing.

Note that ASSERT is meant for detecting program bugs, not for reporting ordinary error conditions. Use the RAISE statement, described above, for that.

## 42.9. Trigger Procedures

PL/pgSQL can be used to define trigger procedures on data changes or database events. A trigger procedure is created with the CREATE FUNCTION command, declaring it as a function with no arguments and a return type of trigger (for data change triggers) or event\_trigger (for database event triggers). Special local variables named TG\_***something*** are automatically defined to describe the condition that triggered the call.

### 42.9.1. Triggers on Data Changes

A [**data change trigger**](https://www.postgresql.org/docs/10/triggers.html) is declared as a function with no arguments and a return type of trigger. Note that the function must be declared with no arguments even if it expects to receive some arguments specified in CREATE TRIGGER — such arguments are passed via TG\_ARGV, as described below.

When a PL/pgSQL function is called as a trigger, several special variables are created automatically in the top-level block. They are:

NEW

Data type RECORD; variable holding the new database row for INSERT/UPDATE operations in row-level triggers. This variable is unassigned in statement-level triggers and for DELETE operations.

OLD

Data type RECORD; variable holding the old database row for UPDATE/DELETE operations in row-level triggers. This variable is unassigned in statement-level triggers and for INSERT operations.

TG\_NAME

Data type name; variable that contains the name of the trigger actually fired.

TG\_WHEN

Data type text; a string of BEFORE, AFTER, or INSTEAD OF, depending on the trigger's definition.

TG\_LEVEL

Data type text; a string of either ROW or STATEMENT depending on the trigger's definition.

TG\_OP

Data type text; a string of INSERT, UPDATE, DELETE, or TRUNCATE telling for which operation the trigger was fired.

TG\_RELID

Data type oid; the object ID of the table that caused the trigger invocation.

TG\_RELNAME

Data type name; the name of the table that caused the trigger invocation. This is now deprecated, and could disappear in a future release. Use TG\_TABLE\_NAME instead.

TG\_TABLE\_NAME

Data type name; the name of the table that caused the trigger invocation.

TG\_TABLE\_SCHEMA

Data type name; the name of the schema of the table that caused the trigger invocation.

TG\_NARGS

Data type integer; the number of arguments given to the trigger procedure in the CREATE TRIGGER statement.

TG\_ARGV[]

Data type array of text; the arguments from the CREATE TRIGGER statement. The index counts from 0. Invalid indexes (less than 0 or greater than or equal to tg\_nargs) result in a null value.

A trigger function must return either NULL or a record/row value having exactly the structure of the table the trigger was fired for.

Row-level triggers fired BEFORE can return null to signal the trigger manager to skip the rest of the operation for this row (i.e., subsequent triggers are not fired, and the INSERT/UPDATE/DELETE does not occur for this row). If a nonnull value is returned then the operation proceeds with that row value. Returning a row value different from the original value of NEW alters the row that will be inserted or updated. Thus, if the trigger function wants the triggering action to succeed normally without altering the row value, NEW (or a value equal thereto) has to be returned. To alter the row to be stored, it is possible to replace single values directly in NEW and return the modified NEW, or to build a complete new record/row to return. In the case of a before-trigger on DELETE, the returned value has no direct effect, but it has to be nonnull to allow the trigger action to proceed. Note that NEW is null in DELETE triggers, so returning that is usually not sensible. The usual idiom in DELETE triggers is to return OLD.

INSTEAD OF triggers (which are always row-level triggers, and may only be used on views) can return null to signal that they did not perform any updates, and that the rest of the operation for this row should be skipped (i.e., subsequent triggers are not fired, and the row is not counted in the rows-affected status for the surrounding INSERT/UPDATE/DELETE). Otherwise a nonnull value should be returned, to signal that the trigger performed the requested operation. For INSERT and UPDATE operations, the return value should be NEW, which the trigger function may modify to support INSERT RETURNING and UPDATE RETURNING (this will also affect the row value passed to any subsequent triggers, or passed to a special EXCLUDED alias reference within an INSERT statement with an ON CONFLICT DO UPDATE clause). For DELETE operations, the return value should be OLD.

The return value of a row-level trigger fired AFTER or a statement-level trigger fired BEFORE or AFTER is always ignored; it might as well be null. However, any of these types of triggers might still abort the entire operation by raising an error.

[**Example 42.3**](https://www.postgresql.org/docs/10/plpgsql-trigger.html#PLPGSQL-TRIGGER-EXAMPLE) shows an example of a trigger procedure in PL/pgSQL.

**Example 42.3. A PL/pgSQL Trigger Procedure**

This example trigger ensures that any time a row is inserted or updated in the table, the current user name and time are stamped into the row. And it checks that an employee's name is given and that the salary is a positive value.

CREATE TABLE emp (

empname text,

salary integer,

last\_date timestamp,

last\_user text

);

CREATE FUNCTION emp\_stamp() RETURNS trigger AS $emp\_stamp$

BEGIN

-- Check that empname and salary are given

IF NEW.empname IS NULL THEN

RAISE EXCEPTION 'empname cannot be null';

END IF;

IF NEW.salary IS NULL THEN

RAISE EXCEPTION '% cannot have null salary', NEW.empname;

END IF;

-- Who works for us when they must pay for it?

IF NEW.salary < 0 THEN

RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;

END IF;

-- Remember who changed the payroll when

NEW.last\_date := current\_timestamp;

NEW.last\_user := current\_user;

RETURN NEW;

END;

$emp\_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp\_stamp BEFORE INSERT OR UPDATE ON emp

FOR EACH ROW EXECUTE PROCEDURE emp\_stamp();

Another way to log changes to a table involves creating a new table that holds a row for each insert, update, or delete that occurs. This approach can be thought of as auditing changes to a table. [**Example 42.4**](https://www.postgresql.org/docs/10/plpgsql-trigger.html#PLPGSQL-TRIGGER-AUDIT-EXAMPLE) shows an example of an audit trigger procedure in PL/pgSQL.

**Example 42.4. A PL/pgSQL Trigger Procedure For Auditing**

This example trigger ensures that any insert, update or delete of a row in the emp table is recorded (i.e., audited) in the emp\_audit table. The current time and user name are stamped into the row, together with the type of operation performed on it.

CREATE TABLE emp (

empname text NOT NULL,

salary integer

);

CREATE TABLE emp\_audit(

operation char(1) NOT NULL,

stamp timestamp NOT NULL,

userid text NOT NULL,

empname text NOT NULL,

salary integer

);

CREATE OR REPLACE FUNCTION process\_emp\_audit() RETURNS TRIGGER AS $emp\_audit$

BEGIN

--

-- Create a row in emp\_audit to reflect the operation performed on emp,

-- making use of the special variable TG\_OP to work out the operation.

--

IF (TG\_OP = 'DELETE') THEN

INSERT INTO emp\_audit SELECT 'D', now(), user, OLD.\*;

ELSIF (TG\_OP = 'UPDATE') THEN

INSERT INTO emp\_audit SELECT 'U', now(), user, NEW.\*;

ELSIF (TG\_OP = 'INSERT') THEN

INSERT INTO emp\_audit SELECT 'I', now(), user, NEW.\*;

END IF;

RETURN NULL; -- result is ignored since this is an AFTER trigger

END;

$emp\_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp\_audit

AFTER INSERT OR UPDATE OR DELETE ON emp

FOR EACH ROW EXECUTE PROCEDURE process\_emp\_audit();

A variation of the previous example uses a view joining the main table to the audit table, to show when each entry was last modified. This approach still records the full audit trail of changes to the table, but also presents a simplified view of the audit trail, showing just the last modified timestamp derived from the audit trail for each entry. [**Example 42.5**](https://www.postgresql.org/docs/10/plpgsql-trigger.html#PLPGSQL-VIEW-TRIGGER-AUDIT-EXAMPLE) shows an example of an audit trigger on a view in PL/pgSQL.

**Example 42.5. A PL/pgSQL View Trigger Procedure For Auditing**

This example uses a trigger on the view to make it updatable, and ensure that any insert, update or delete of a row in the view is recorded (i.e., audited) in the emp\_audit table. The current time and user name are recorded, together with the type of operation performed, and the view displays the last modified time of each row.

CREATE TABLE emp (

empname text PRIMARY KEY,

salary integer

);

CREATE TABLE emp\_audit(

operation char(1) NOT NULL,

userid text NOT NULL,

empname text NOT NULL,

salary integer,

stamp timestamp NOT NULL

);

CREATE VIEW emp\_view AS

SELECT e.empname,

e.salary,

max(ea.stamp) AS last\_updated

FROM emp e

LEFT JOIN emp\_audit ea ON ea.empname = e.empname

GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION update\_emp\_view() RETURNS TRIGGER AS $$

BEGIN

--

-- Perform the required operation on emp, and create a row in emp\_audit

-- to reflect the change made to emp.

--

IF (TG\_OP = 'DELETE') THEN

DELETE FROM emp WHERE empname = OLD.empname;

IF NOT FOUND THEN RETURN NULL; END IF;

OLD.last\_updated = now();

INSERT INTO emp\_audit VALUES('D', user, OLD.\*);

RETURN OLD;

ELSIF (TG\_OP = 'UPDATE') THEN

UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;

IF NOT FOUND THEN RETURN NULL; END IF;

NEW.last\_updated = now();

INSERT INTO emp\_audit VALUES('U', user, NEW.\*);

RETURN NEW;

ELSIF (TG\_OP = 'INSERT') THEN

INSERT INTO emp VALUES(NEW.empname, NEW.salary);

NEW.last\_updated = now();

INSERT INTO emp\_audit VALUES('I', user, NEW.\*);

RETURN NEW;

END IF;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER emp\_audit

INSTEAD OF INSERT OR UPDATE OR DELETE ON emp\_view

FOR EACH ROW EXECUTE PROCEDURE update\_emp\_view();

One use of triggers is to maintain a summary table of another table. The resulting summary can be used in place of the original table for certain queries — often with vastly reduced run times. This technique is commonly used in Data Warehousing, where the tables of measured or observed data (called fact tables) might be extremely large. [**Example 42.6**](https://www.postgresql.org/docs/10/plpgsql-trigger.html#PLPGSQL-TRIGGER-SUMMARY-EXAMPLE) shows an example of a trigger procedure in PL/pgSQL that maintains a summary table for a fact table in a data warehouse.

**Example 42.6. A PL/pgSQL Trigger Procedure For Maintaining A Summary Table**

The schema detailed here is partly based on the Grocery Store example from The Data Warehouse Toolkit by Ralph Kimball.

--

-- Main tables - time dimension and sales fact.

--

CREATE TABLE time\_dimension (

time\_key integer NOT NULL,

day\_of\_week integer NOT NULL,

day\_of\_month integer NOT NULL,

month integer NOT NULL,

quarter integer NOT NULL,

year integer NOT NULL

);

CREATE UNIQUE INDEX time\_dimension\_key ON time\_dimension(time\_key);

CREATE TABLE sales\_fact (

time\_key integer NOT NULL,

product\_key integer NOT NULL,

store\_key integer NOT NULL,

amount\_sold numeric(12,2) NOT NULL,

units\_sold integer NOT NULL,

amount\_cost numeric(12,2) NOT NULL

);

CREATE INDEX sales\_fact\_time ON sales\_fact(time\_key);

--

-- Summary table - sales by time.

--

CREATE TABLE sales\_summary\_bytime (

time\_key integer NOT NULL,

amount\_sold numeric(15,2) NOT NULL,

units\_sold numeric(12) NOT NULL,

amount\_cost numeric(15,2) NOT NULL

);

CREATE UNIQUE INDEX sales\_summary\_bytime\_key ON sales\_summary\_bytime(time\_key);

--

-- Function and trigger to amend summarized column(s) on UPDATE, INSERT, DELETE.

--

CREATE OR REPLACE FUNCTION maint\_sales\_summary\_bytime() RETURNS TRIGGER

AS $maint\_sales\_summary\_bytime$

DECLARE

delta\_time\_key integer;

delta\_amount\_sold numeric(15,2);

delta\_units\_sold numeric(12);

delta\_amount\_cost numeric(15,2);

BEGIN

-- Work out the increment/decrement amount(s).

IF (TG\_OP = 'DELETE') THEN

delta\_time\_key = OLD.time\_key;

delta\_amount\_sold = -1 \* OLD.amount\_sold;

delta\_units\_sold = -1 \* OLD.units\_sold;

delta\_amount\_cost = -1 \* OLD.amount\_cost;

ELSIF (TG\_OP = 'UPDATE') THEN

-- forbid updates that change the time\_key -

-- (probably not too onerous, as DELETE + INSERT is how most

-- changes will be made).

IF ( OLD.time\_key != NEW.time\_key) THEN

RAISE EXCEPTION 'Update of time\_key : % -> % not allowed',

OLD.time\_key, NEW.time\_key;

END IF;

delta\_time\_key = OLD.time\_key;

delta\_amount\_sold = NEW.amount\_sold - OLD.amount\_sold;

delta\_units\_sold = NEW.units\_sold - OLD.units\_sold;

delta\_amount\_cost = NEW.amount\_cost - OLD.amount\_cost;

ELSIF (TG\_OP = 'INSERT') THEN

delta\_time\_key = NEW.time\_key;

delta\_amount\_sold = NEW.amount\_sold;

delta\_units\_sold = NEW.units\_sold;

delta\_amount\_cost = NEW.amount\_cost;

END IF;

-- Insert or update the summary row with the new values.

<<insert\_update>>

LOOP

UPDATE sales\_summary\_bytime

SET amount\_sold = amount\_sold + delta\_amount\_sold,

units\_sold = units\_sold + delta\_units\_sold,

amount\_cost = amount\_cost + delta\_amount\_cost

WHERE time\_key = delta\_time\_key;

EXIT insert\_update WHEN found;

BEGIN

INSERT INTO sales\_summary\_bytime (

time\_key,

amount\_sold,

units\_sold,

amount\_cost)

VALUES (

delta\_time\_key,

delta\_amount\_sold,

delta\_units\_sold,

delta\_amount\_cost

);

EXIT insert\_update;

EXCEPTION

WHEN UNIQUE\_VIOLATION THEN

-- do nothing

END;

END LOOP insert\_update;

RETURN NULL;

END;

$maint\_sales\_summary\_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint\_sales\_summary\_bytime

AFTER INSERT OR UPDATE OR DELETE ON sales\_fact

FOR EACH ROW EXECUTE PROCEDURE maint\_sales\_summary\_bytime();

INSERT INTO sales\_fact VALUES(1,1,1,10,3,15);

INSERT INTO sales\_fact VALUES(1,2,1,20,5,35);

INSERT INTO sales\_fact VALUES(2,2,1,40,15,135);

INSERT INTO sales\_fact VALUES(2,3,1,10,1,13);

SELECT \* FROM sales\_summary\_bytime;

DELETE FROM sales\_fact WHERE product\_key = 1;

SELECT \* FROM sales\_summary\_bytime;

UPDATE sales\_fact SET units\_sold = units\_sold \* 2;

SELECT \* FROM sales\_summary\_bytime;

AFTER triggers can also make use of transition tables to inspect the entire set of rows changed by the triggering statement. The CREATE TRIGGER command assigns names to one or both transition tables, and then the function can refer to those names as though they were read-only temporary tables. [**Example 42.7**](https://www.postgresql.org/docs/10/plpgsql-trigger.html#PLPGSQL-TRIGGER-AUDIT-TRANSITION-EXAMPLE) shows an example.

**Example 42.7. Auditing with Transition Tables**

This example produces the same results as [**Example 42.4**](https://www.postgresql.org/docs/10/plpgsql-trigger.html#PLPGSQL-TRIGGER-AUDIT-EXAMPLE), but instead of using a trigger that fires for every row, it uses a trigger that fires once per statement, after collecting the relevant information in a transition table. This can be significantly faster than the row-trigger approach when the invoking statement has modified many rows. Notice that we must make a separate trigger declaration for each kind of event, since the REFERENCING clauses must be different for each case. But this does not stop us from using a single trigger function if we choose. (In practice, it might be better to use three separate functions and avoid the run-time tests on TG\_OP.)

CREATE TABLE emp (

empname text NOT NULL,

salary integer

);

CREATE TABLE emp\_audit(

operation char(1) NOT NULL,

stamp timestamp NOT NULL,

userid text NOT NULL,

empname text NOT NULL,

salary integer

);

CREATE OR REPLACE FUNCTION process\_emp\_audit() RETURNS TRIGGER AS $emp\_audit$

BEGIN

--

-- Create rows in emp\_audit to reflect the operations performed on emp,

-- making use of the special variable TG\_OP to work out the operation.

--

IF (TG\_OP = 'DELETE') THEN

INSERT INTO emp\_audit

SELECT 'D', now(), user, o.\* FROM old\_table o;

ELSIF (TG\_OP = 'UPDATE') THEN

INSERT INTO emp\_audit

SELECT 'U', now(), user, n.\* FROM new\_table n;

ELSIF (TG\_OP = 'INSERT') THEN

INSERT INTO emp\_audit

SELECT 'I', now(), user, n.\* FROM new\_table n;

END IF;

RETURN NULL; -- result is ignored since this is an AFTER trigger

END;

$emp\_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp\_audit\_ins

AFTER INSERT ON emp

REFERENCING NEW TABLE AS new\_table

FOR EACH STATEMENT EXECUTE PROCEDURE process\_emp\_audit();

CREATE TRIGGER emp\_audit\_upd

AFTER UPDATE ON emp

REFERENCING OLD TABLE AS old\_table NEW TABLE AS new\_table

FOR EACH STATEMENT EXECUTE PROCEDURE process\_emp\_audit();

CREATE TRIGGER emp\_audit\_del

AFTER DELETE ON emp

REFERENCING OLD TABLE AS old\_table

FOR EACH STATEMENT EXECUTE PROCEDURE process\_emp\_audit();

### 42.9.2. Triggers on Events

PL/pgSQL can be used to define [**event triggers**](https://www.postgresql.org/docs/10/event-triggers.html). PostgreSQL requires that a procedure that is to be called as an event trigger must be declared as a function with no arguments and a return type of event\_trigger.

When a PL/pgSQL function is called as an event trigger, several special variables are created automatically in the top-level block. They are:

TG\_EVENT

Data type text; a string representing the event the trigger is fired for.

TG\_TAG

Data type text; variable that contains the command tag for which the trigger is fired.

[**Example 42.8**](https://www.postgresql.org/docs/10/plpgsql-trigger.html#PLPGSQL-EVENT-TRIGGER-EXAMPLE) shows an example of an event trigger procedure in PL/pgSQL.

**Example 42.8. A PL/pgSQL Event Trigger Procedure**

This example trigger simply raises a NOTICE message each time a supported command is executed.

CREATE OR REPLACE FUNCTION snitch() RETURNS event\_trigger AS $$

BEGIN

RAISE NOTICE 'snitch: % %', tg\_event, tg\_tag;

END;

$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER snitch ON ddl\_command\_start EXECUTE PROCEDURE snitch();

## 42.10. PL/pgSQL Under the Hood

This section discusses some implementation details that are frequently important for PL/pgSQL users to know.

### 42.10.1. Variable Substitution

SQL statements and expressions within a PL/pgSQL function can refer to variables and parameters of the function. Behind the scenes, PL/pgSQL substitutes query parameters for such references. Parameters will only be substituted in places where a parameter or column reference is syntactically allowed. As an extreme case, consider this example of poor programming style:

INSERT INTO foo (foo) VALUES (foo);

The first occurrence of foo must syntactically be a table name, so it will not be substituted, even if the function has a variable named foo. The second occurrence must be the name of a column of the table, so it will not be substituted either. Only the third occurrence is a candidate to be a reference to the function's variable.

### Note

PostgreSQL versions before 9.0 would try to substitute the variable in all three cases, leading to syntax errors.

Since the names of variables are syntactically no different from the names of table columns, there can be ambiguity in statements that also refer to tables: is a given name meant to refer to a table column, or a variable? Let's change the previous example to

INSERT INTO dest (col) SELECT foo + bar FROM src;

Here, dest and src must be table names, and col must be a column of dest, but foo and bar might reasonably be either variables of the function or columns of src.

By default, PL/pgSQL will report an error if a name in a SQL statement could refer to either a variable or a table column. You can fix such a problem by renaming the variable or column, or by qualifying the ambiguous reference, or by telling PL/pgSQL which interpretation to prefer.

The simplest solution is to rename the variable or column. A common coding rule is to use a different naming convention for PL/pgSQL variables than you use for column names. For example, if you consistently name function variables v\_***something*** while none of your column names start with v\_, no conflicts will occur.

Alternatively you can qualify ambiguous references to make them clear. In the above example, src.foo would be an unambiguous reference to the table column. To create an unambiguous reference to a variable, declare it in a labeled block and use the block's label (see [**Section 42.2**](https://www.postgresql.org/docs/10/plpgsql-structure.html)). For example,

<<block>>

DECLARE

foo int;

BEGIN

foo := ...;

INSERT INTO dest (col) SELECT block.foo + bar FROM src;

Here block.foo means the variable even if there is a column foo in src. Function parameters, as well as special variables such as FOUND, can be qualified by the function's name, because they are implicitly declared in an outer block labeled with the function's name.

Sometimes it is impractical to fix all the ambiguous references in a large body of PL/pgSQL code. In such cases you can specify that PL/pgSQL should resolve ambiguous references as the variable (which is compatible with PL/pgSQL's behavior before PostgreSQL 9.0), or as the table column (which is compatible with some other systems such as Oracle).

To change this behavior on a system-wide basis, set the configuration parameter plpgsql.variable\_conflict to one of error, use\_variable, or use\_column (where error is the factory default). This parameter affects subsequent compilations of statements in PL/pgSQL functions, but not statements already compiled in the current session. Because changing this setting can cause unexpected changes in the behavior of PL/pgSQL functions, it can only be changed by a superuser.

You can also set the behavior on a function-by-function basis, by inserting one of these special commands at the start of the function text:

#variable\_conflict error

#variable\_conflict use\_variable

#variable\_conflict use\_column

These commands affect only the function they are written in, and override the setting of plpgsql.variable\_conflict. An example is

CREATE FUNCTION stamp\_user(id int, comment text) RETURNS void AS $$

#variable\_conflict use\_variable

DECLARE

curtime timestamp := now();

BEGIN

UPDATE users SET last\_modified = curtime, comment = comment

WHERE users.id = id;

END;

$$ LANGUAGE plpgsql;

In the UPDATE command, curtime, comment, and id will refer to the function's variable and parameters whether or not users has columns of those names. Notice that we had to qualify the reference to users.id in the WHERE clause to make it refer to the table column. But we did not have to qualify the reference to comment as a target in the UPDATE list, because syntactically that must be a column of users. We could write the same function without depending on the variable\_conflict setting in this way:

CREATE FUNCTION stamp\_user(id int, comment text) RETURNS void AS $$

<<fn>>

DECLARE

curtime timestamp := now();

BEGIN

UPDATE users SET last\_modified = fn.curtime, comment = stamp\_user.comment

WHERE users.id = stamp\_user.id;

END;

$$ LANGUAGE plpgsql;

Variable substitution does not happen in the command string given to EXECUTE or one of its variants. If you need to insert a varying value into such a command, do so as part of constructing the string value, or use USING, as illustrated in [**Section 42.5.4**](https://www.postgresql.org/docs/10/plpgsql-statements.html#PLPGSQL-STATEMENTS-EXECUTING-DYN).

Variable substitution currently works only in SELECT, INSERT, UPDATE, and DELETE commands, because the main SQL engine allows query parameters only in these commands. To use a non-constant name or value in other statement types (generically called utility statements), you must construct the utility statement as a string and EXECUTE it.

### 42.10.2. Plan Caching

The PL/pgSQL interpreter parses the function's source text and produces an internal binary instruction tree the first time the function is called (within each session). The instruction tree fully translates the PL/pgSQL statement structure, but individual SQL expressions and SQL commands used in the function are not translated immediately.

As each expression and SQL command is first executed in the function, the PL/pgSQL interpreter parses and analyzes the command to create a prepared statement, using the SPI manager's SPI\_prepare function. Subsequent visits to that expression or command reuse the prepared statement. Thus, a function with conditional code paths that are seldom visited will never incur the overhead of analyzing those commands that are never executed within the current session. A disadvantage is that errors in a specific expression or command cannot be detected until that part of the function is reached in execution. (Trivial syntax errors will be detected during the initial parsing pass, but anything deeper will not be detected until execution.)

PL/pgSQL (or more precisely, the SPI manager) can furthermore attempt to cache the execution plan associated with any particular prepared statement. If a cached plan is not used, then a fresh execution plan is generated on each visit to the statement, and the current parameter values (that is, PL/pgSQL variable values) can be used to optimize the selected plan. If the statement has no parameters, or is executed many times, the SPI manager will consider creating a generic plan that is not dependent on specific parameter values, and caching that for re-use. Typically this will happen only if the execution plan is not very sensitive to the values of the PL/pgSQL variables referenced in it. If it is, generating a plan each time is a net win. See [**PREPARE**](https://www.postgresql.org/docs/10/sql-prepare.html) for more information about the behavior of prepared statements.

Because PL/pgSQL saves prepared statements and sometimes execution plans in this way, SQL commands that appear directly in a PL/pgSQL function must refer to the same tables and columns on every execution; that is, you cannot use a parameter as the name of a table or column in an SQL command. To get around this restriction, you can construct dynamic commands using the PL/pgSQLEXECUTE statement — at the price of performing new parse analysis and constructing a new execution plan on every execution.

The mutable nature of record variables presents another problem in this connection. When fields of a record variable are used in expressions or statements, the data types of the fields must not change from one call of the function to the next, since each expression will be analyzed using the data type that is present when the expression is first reached. EXECUTE can be used to get around this problem when necessary.

If the same function is used as a trigger for more than one table, PL/pgSQL prepares and caches statements independently for each such table — that is, there is a cache for each trigger function and table combination, not just for each function. This alleviates some of the problems with varying data types; for instance, a trigger function will be able to work successfully with a column named keyeven if it happens to have different types in different tables.

Likewise, functions having polymorphic argument types have a separate statement cache for each combination of actual argument types they have been invoked for, so that data type differences do not cause unexpected failures.

Statement caching can sometimes have surprising effects on the interpretation of time-sensitive values. For example there is a difference between what these two functions do:

CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$

BEGIN

INSERT INTO logtable VALUES (logtxt, 'now');

END;

$$ LANGUAGE plpgsql;

and:

CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$

DECLARE

curtime timestamp;

BEGIN

curtime := 'now';

INSERT INTO logtable VALUES (logtxt, curtime);

END;

$$ LANGUAGE plpgsql;

In the case of logfunc1, the PostgreSQL main parser knows when analyzing the INSERT that the string 'now' should be interpreted as timestamp, because the target column of logtable is of that type. Thus, 'now' will be converted to a timestamp constant when the INSERT is analyzed, and then used in all invocations of logfunc1 during the lifetime of the session. Needless to say, this isn't what the programmer wanted. A better idea is to use the now() or current\_timestamp function.

In the case of logfunc2, the PostgreSQL main parser does not know what type 'now' should become and therefore it returns a data value of type text containing the string now. During the ensuing assignment to the local variable curtime, the PL/pgSQL interpreter casts this string to the timestamp type by calling the text\_out and timestamp\_in functions for the conversion. So, the computed time stamp is updated on each execution as the programmer expects. Even though this happens to work as expected, it's not terribly efficient, so use of the now() function would still be a better idea.

## 42.11. Tips for Developing in PL/pgSQL

One good way to develop in PL/pgSQL is to use the text editor of your choice to create your functions, and in another window, use psql to load and test those functions. If you are doing it this way, it is a good idea to write the function using CREATE OR REPLACE FUNCTION. That way you can just reload the file to update the function definition. For example:

CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$

....

$$ LANGUAGE plpgsql;

While running psql, you can load or reload such a function definition file with:

\i filename.sql

and then immediately issue SQL commands to test the function.

Another good way to develop in PL/pgSQL is with a GUI database access tool that facilitates development in a procedural language. One example of such a tool is pgAdmin, although others exist. These tools often provide convenient features such as escaping single quotes and making it easier to recreate and debug functions.

### 42.11.1. Handling of Quotation Marks

The code of a PL/pgSQL function is specified in CREATE FUNCTION as a string literal. If you write the string literal in the ordinary way with surrounding single quotes, then any single quotes inside the function body must be doubled; likewise any backslashes must be doubled (assuming escape string syntax is used). Doubling quotes is at best tedious, and in more complicated cases the code can become downright incomprehensible, because you can easily find yourself needing half a dozen or more adjacent quote marks. It's recommended that you instead write the function body as a “dollar-quoted” string literal (see [**Section 4.1.2.4**](https://www.postgresql.org/docs/10/sql-syntax-lexical.html#SQL-SYNTAX-DOLLAR-QUOTING)). In the dollar-quoting approach, you never double any quote marks, but instead take care to choose a different dollar-quoting delimiter for each level of nesting you need. For example, you might write the CREATE FUNCTION command as:

CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$

....

$PROC$ LANGUAGE plpgsql;

Within this, you might use quote marks for simple literal strings in SQL commands and $$ to delimit fragments of SQL commands that you are assembling as strings. If you need to quote text that includes $$, you could use $Q$, and so on.

The following chart shows what you have to do when writing quote marks without dollar quoting. It might be useful when translating pre-dollar quoting code into something more comprehensible.

1 quotation mark

To begin and end the function body, for example:

CREATE FUNCTION foo() RETURNS integer AS '

....

' LANGUAGE plpgsql;

Anywhere within a single-quoted function body, quote marks must appear in pairs.

2 quotation marks

For string literals inside the function body, for example:

a\_output := ''Blah'';

SELECT \* FROM users WHERE f\_name=''foobar'';

In the dollar-quoting approach, you'd just write:

a\_output := 'Blah';

SELECT \* FROM users WHERE f\_name='foobar';

which is exactly what the PL/pgSQL parser would see in either case.

4 quotation marks

When you need a single quotation mark in a string constant inside the function body, for example:

a\_output := a\_output || '' AND name LIKE ''''foobar'''' AND xyz''

The value actually appended to a\_output would be: AND name LIKE 'foobar' AND xyz.

In the dollar-quoting approach, you'd write:

a\_output := a\_output || $$ AND name LIKE 'foobar' AND xyz$$

being careful that any dollar-quote delimiters around this are not just $$.

6 quotation marks

When a single quotation mark in a string inside the function body is adjacent to the end of that string constant, for example:

a\_output := a\_output || '' AND name LIKE ''''foobar''''''

The value appended to a\_output would then be: AND name LIKE 'foobar'.

In the dollar-quoting approach, this becomes:

a\_output := a\_output || $$ AND name LIKE 'foobar'$$

10 quotation marks

When you want two single quotation marks in a string constant (which accounts for 8 quotation marks) and this is adjacent to the end of that string constant (2 more). You will probably only need that if you are writing a function that generates other functions, as in [**Example 42.10**](https://www.postgresql.org/docs/10/plpgsql-porting.html#PLPGSQL-PORTING-EX2). For example:

a\_output := a\_output || '' if v\_'' ||

referrer\_keys.kind || '' like ''''''''''

|| referrer\_keys.key\_string || ''''''''''

then return '''''' || referrer\_keys.referrer\_type

|| ''''''; end if;'';

The value of a\_output would then be:

if v\_... like ''...'' then return ''...''; end if;

In the dollar-quoting approach, this becomes:

a\_output := a\_output || $$ if v\_$$ || referrer\_keys.kind || $$ like '$$

|| referrer\_keys.key\_string || $$'

then return '$$ || referrer\_keys.referrer\_type

|| $$'; end if;$$;

where we assume we only need to put single quote marks into a\_output, because it will be re-quoted before use.

### 42.11.2. Additional Compile-time Checks

To aid the user in finding instances of simple but common problems before they cause harm, PL/pgSQL provides additional ***checks***. When enabled, depending on the configuration, they can be used to emit either a WARNING or an ERROR during the compilation of a function. A function which has received a WARNING can be executed without producing further messages, so you are advised to test in a separate development environment.

These additional checks are enabled through the configuration variables plpgsql.extra\_warnings for warnings and plpgsql.extra\_errors for errors. Both can be set either to a comma-separated list of checks, "none" or "all". The default is "none". Currently the list of available checks includes only one:

shadowed\_variables

Checks if a declaration shadows a previously defined variable.

The following example shows the effect of plpgsql.extra\_warnings set to shadowed\_variables:

SET plpgsql.extra\_warnings TO 'shadowed\_variables';

CREATE FUNCTION foo(f1 int) RETURNS int AS $$

DECLARE

f1 int;

BEGIN

RETURN f1;

END

$$ LANGUAGE plpgsql;

WARNING: variable "f1" shadows a previously defined variable

LINE 3: f1 int;

^

CREATE FUNCTION

## 42.12. Porting from Oracle PL/SQL 和Oracle的PL/SQL的区别

This section explains differences between PostgreSQL's PL/pgSQL language and Oracle's PL/SQL language, to help developers who port applications from Oracle® to PostgreSQL.

PL/pgSQL is similar to PL/SQL in many aspects. It is a block-structured, imperative language, and all variables have to be declared. Assignments, loops, and conditionals are similar. The main differences you should keep in mind when porting from PL/SQL to PL/pgSQL are:

* If a name used in a SQL command could be either a column name of a table or a reference to a variable of the function, PL/SQL treats it as a column name. This corresponds to PL/pgSQL's plpgsql.variable\_conflict = use\_column behavior, which is not the default, as explained in [**Section 42.10.1**](https://www.postgresql.org/docs/10/plpgsql-implementation.html#PLPGSQL-VAR-SUBST). It's often best to avoid such ambiguities in the first place, but if you have to port a large amount of code that depends on this behavior, setting variable\_conflict may be the best solution.
* In PostgreSQL the function body must be written as a string literal. Therefore you need to use dollar quoting or escape single quotes in the function body. (See [**Section 42.11.1**](https://www.postgresql.org/docs/10/plpgsql-development-tips.html#PLPGSQL-QUOTE-TIPS).)
* Data type names often need translation. For example, in Oracle string values are commonly declared as being of type varchar2, which is a non-SQL-standard type. In PostgreSQL, use type varchar or text instead. Similarly, replace type number with numeric, or use some other numeric data type if there's a more appropriate one.
* Instead of packages, use schemas to organize your functions into groups.
* Since there are no packages, there are no package-level variables either. This is somewhat annoying. You can keep per-session state in temporary tables instead.
* Integer FOR loops with REVERSE work differently: PL/SQL counts down from the second number to the first, while PL/pgSQL counts down from the first number to the second, requiring the loop bounds to be swapped when porting. This incompatibility is unfortunate but is unlikely to be changed. (See [**Section 42.6.3.5**](https://www.postgresql.org/docs/10/plpgsql-control-structures.html#PLPGSQL-INTEGER-FOR).)
* FOR loops over queries (other than cursors) also work differently: the target variable(s) must have been declared, whereas PL/SQL always declares them implicitly. An advantage of this is that the variable values are still accessible after the loop exits.
* There are various notational differences for the use of cursor variables.

### 42.12.1. Porting Examples

[**Example 42.9**](https://www.postgresql.org/docs/10/plpgsql-porting.html#PGSQL-PORTING-EX1) shows how to port a simple function from PL/SQL to PL/pgSQL.

**Example 42.9. Porting a Simple Function from PL/SQL to PL/pgSQL**

Here is an Oracle PL/SQL function:

CREATE OR REPLACE FUNCTION cs\_fmt\_browser\_version(v\_name varchar2,

v\_version varchar2)

RETURN varchar2 IS

BEGIN

IF v\_version IS NULL THEN

RETURN v\_name;

END IF;

RETURN v\_name || '/' || v\_version;

END;

/

show errors;

Let's go through this function and see the differences compared to PL/pgSQL:

* The type name varchar2 has to be changed to varchar or text. In the examples in this section, we'll use varchar, but text is often a better choice if you do not need specific string length limits.
* The RETURN key word in the function prototype (not the function body) becomes RETURNS in PostgreSQL. Also, IS becomes AS, and you need to add a LANGUAGE clause because PL/pgSQL is not the only possible function language.
* In PostgreSQL, the function body is considered to be a string literal, so you need to use quote marks or dollar quotes around it. This substitutes for the terminating / in the Oracle approach.
* The show errors command does not exist in PostgreSQL, and is not needed since errors are reported automatically.

This is how this function would look when ported to PostgreSQL:

CREATE OR REPLACE FUNCTION cs\_fmt\_browser\_version(v\_name varchar,

v\_version varchar)

RETURNS varchar AS $$

BEGIN

IF v\_version IS NULL THEN

RETURN v\_name;

END IF;

RETURN v\_name || '/' || v\_version;

END;

$$ LANGUAGE plpgsql;

[**Example 42.10**](https://www.postgresql.org/docs/10/plpgsql-porting.html#PLPGSQL-PORTING-EX2) shows how to port a function that creates another function and how to handle the ensuing quoting problems.

**Example 42.10. Porting a Function that Creates Another Function from PL/SQL to PL/pgSQL**

The following procedure grabs rows from a SELECT statement and builds a large function with the results in IF statements, for the sake of efficiency.

This is the Oracle version:

CREATE OR REPLACE PROCEDURE cs\_update\_referrer\_type\_proc IS

CURSOR referrer\_keys IS

SELECT \* FROM cs\_referrer\_keys

ORDER BY try\_order;

func\_cmd VARCHAR(4000);

BEGIN

func\_cmd := 'CREATE OR REPLACE FUNCTION cs\_find\_referrer\_type(v\_host IN VARCHAR2,

v\_domain IN VARCHAR2, v\_url IN VARCHAR2) RETURN VARCHAR2 IS BEGIN';

FOR referrer\_key IN referrer\_keys LOOP

func\_cmd := func\_cmd ||

' IF v\_' || referrer\_key.kind

|| ' LIKE ''' || referrer\_key.key\_string

|| ''' THEN RETURN ''' || referrer\_key.referrer\_type

|| '''; END IF;';

END LOOP;

func\_cmd := func\_cmd || ' RETURN NULL; END;';

EXECUTE IMMEDIATE func\_cmd;

END;

/

show errors;

Here is how this function would end up in PostgreSQL:

CREATE OR REPLACE FUNCTION cs\_update\_referrer\_type\_proc() RETURNS void AS $func$

DECLARE

referrer\_keys CURSOR IS

SELECT \* FROM cs\_referrer\_keys

ORDER BY try\_order;

func\_body text;

func\_cmd text;

BEGIN

func\_body := 'BEGIN';

FOR referrer\_key IN referrer\_keys LOOP

func\_body := func\_body ||

' IF v\_' || referrer\_key.kind

|| ' LIKE ' || quote\_literal(referrer\_key.key\_string)

|| ' THEN RETURN ' || quote\_literal(referrer\_key.referrer\_type)

|| '; END IF;' ;

END LOOP;

func\_body := func\_body || ' RETURN NULL; END;';

func\_cmd :=

'CREATE OR REPLACE FUNCTION cs\_find\_referrer\_type(v\_host varchar,

v\_domain varchar,

v\_url varchar)

RETURNS varchar AS '

|| quote\_literal(func\_body)

|| ' LANGUAGE plpgsql;' ;

EXECUTE func\_cmd;

END;

$func$ LANGUAGE plpgsql;

Notice how the body of the function is built separately and passed through quote\_literal to double any quote marks in it. This technique is needed because we cannot safely use dollar quoting for defining the new function: we do not know for sure what strings will be interpolated from the referrer\_key.key\_string field. (We are assuming here that referrer\_key.kind can be trusted to always be host, domain, or url, but referrer\_key.key\_string might be anything, in particular it might contain dollar signs.) This function is actually an improvement on the Oracle original, because it will not generate broken code when referrer\_key.key\_string or referrer\_key.referrer\_type contain quote marks.

[**Example 42.11**](https://www.postgresql.org/docs/10/plpgsql-porting.html#PLPGSQL-PORTING-EX3) shows how to port a function with OUT parameters and string manipulation. PostgreSQL does not have a built-in instr function, but you can create one using a combination of other functions. In [**Section 42.12.3**](https://www.postgresql.org/docs/10/plpgsql-porting.html#PLPGSQL-PORTING-APPENDIX) there is a PL/pgSQL implementation of instr that you can use to make your porting easier.

**Example 42.11. Porting a Procedure With String Manipulation and**OUT**Parameters from PL/SQL to PL/pgSQL**

The following Oracle PL/SQL procedure is used to parse a URL and return several elements (host, path, and query).

This is the Oracle version:

CREATE OR REPLACE PROCEDURE cs\_parse\_url(

v\_url IN VARCHAR2,

v\_host OUT VARCHAR2, -- This will be passed back

v\_path OUT VARCHAR2, -- This one too

v\_query OUT VARCHAR2) -- And this one

IS

a\_pos1 INTEGER;

a\_pos2 INTEGER;

BEGIN

v\_host := NULL;

v\_path := NULL;

v\_query := NULL;

a\_pos1 := instr(v\_url, '//');

IF a\_pos1 = 0 THEN

RETURN;

END IF;

a\_pos2 := instr(v\_url, '/', a\_pos1 + 2);

IF a\_pos2 = 0 THEN

v\_host := substr(v\_url, a\_pos1 + 2);

v\_path := '/';

RETURN;

END IF;

v\_host := substr(v\_url, a\_pos1 + 2, a\_pos2 - a\_pos1 - 2);

a\_pos1 := instr(v\_url, '?', a\_pos2 + 1);

IF a\_pos1 = 0 THEN

v\_path := substr(v\_url, a\_pos2);

RETURN;

END IF;

v\_path := substr(v\_url, a\_pos2, a\_pos1 - a\_pos2);

v\_query := substr(v\_url, a\_pos1 + 1);

END;

/

show errors;

Here is a possible translation into PL/pgSQL:

CREATE OR REPLACE FUNCTION cs\_parse\_url(

v\_url IN VARCHAR,

v\_host OUT VARCHAR, -- This will be passed back

v\_path OUT VARCHAR, -- This one too

v\_query OUT VARCHAR) -- And this one

AS $$

DECLARE

a\_pos1 INTEGER;

a\_pos2 INTEGER;

BEGIN

v\_host := NULL;

v\_path := NULL;

v\_query := NULL;

a\_pos1 := instr(v\_url, '//');

IF a\_pos1 = 0 THEN

RETURN;

END IF;

a\_pos2 := instr(v\_url, '/', a\_pos1 + 2);

IF a\_pos2 = 0 THEN

v\_host := substr(v\_url, a\_pos1 + 2);

v\_path := '/';

RETURN;

END IF;

v\_host := substr(v\_url, a\_pos1 + 2, a\_pos2 - a\_pos1 - 2);

a\_pos1 := instr(v\_url, '?', a\_pos2 + 1);

IF a\_pos1 = 0 THEN

v\_path := substr(v\_url, a\_pos2);

RETURN;

END IF;

v\_path := substr(v\_url, a\_pos2, a\_pos1 - a\_pos2);

v\_query := substr(v\_url, a\_pos1 + 1);

END;

$$ LANGUAGE plpgsql;

This function could be used like this:

SELECT \* FROM cs\_parse\_url('http://foobar.com/query.cgi?baz');

[**Example 42.12**](https://www.postgresql.org/docs/10/plpgsql-porting.html#PLPGSQL-PORTING-EX4) shows how to port a procedure that uses numerous features that are specific to Oracle.

**Example 42.12. Porting a Procedure from PL/SQL to PL/pgSQL**

The Oracle version:

CREATE OR REPLACE PROCEDURE cs\_create\_job(v\_job\_id IN INTEGER) IS

a\_running\_job\_count INTEGER;

PRAGMA AUTONOMOUS\_TRANSACTION; -- (1)

BEGIN

LOCK TABLE cs\_jobs IN EXCLUSIVE MODE; -- (2)

SELECT count(\*) INTO a\_running\_job\_count FROM cs\_jobs WHERE end\_stamp IS NULL;

IF a\_running\_job\_count > 0 THEN

COMMIT; -- free lock (3)

raise\_application\_error(-20000,

'Unable to create a new job: a job is currently running.');

END IF;

DELETE FROM cs\_active\_job;

INSERT INTO cs\_active\_job(job\_id) VALUES (v\_job\_id);

BEGIN

INSERT INTO cs\_jobs (job\_id, start\_stamp) VALUES (v\_job\_id, sysdate);

EXCEPTION

WHEN dup\_val\_on\_index THEN NULL; -- don't worry if it already exists

END;

COMMIT;

END;

/

show errors

Procedures like this can easily be converted into PostgreSQL functions returning void. This procedure in particular is interesting because it can teach us some things:

|  |  |
| --- | --- |
| [**(1)**](https://www.postgresql.org/docs/10/plpgsql-porting.html#co.plpgsql-porting-pragma) | There is no PRAGMA statement in PostgreSQL. |
| [**(2)**](https://www.postgresql.org/docs/10/plpgsql-porting.html#co.plpgsql-porting-locktable) | If you do a LOCK TABLE in PL/pgSQL, the lock will not be released until the calling transaction is finished. |
| [**(3)**](https://www.postgresql.org/docs/10/plpgsql-porting.html#co.plpgsql-porting-commit) | You cannot issue COMMIT in a PL/pgSQL function. The function is running within some outer transaction and so COMMIT would imply terminating the function's execution. However, in this particular case it is not necessary anyway, because the lock obtained by the LOCK TABLE will be released when we raise an error. |

This is how we could port this procedure to PL/pgSQL:

CREATE OR REPLACE FUNCTION cs\_create\_job(v\_job\_id integer) RETURNS void AS $$

DECLARE

a\_running\_job\_count integer;

BEGIN

LOCK TABLE cs\_jobs IN EXCLUSIVE MODE;

SELECT count(\*) INTO a\_running\_job\_count FROM cs\_jobs WHERE end\_stamp IS NULL;

IF a\_running\_job\_count > 0 THEN

RAISE EXCEPTION 'Unable to create a new job: a job is currently running'; -- (1)

END IF;

DELETE FROM cs\_active\_job;

INSERT INTO cs\_active\_job(job\_id) VALUES (v\_job\_id);

BEGIN

INSERT INTO cs\_jobs (job\_id, start\_stamp) VALUES (v\_job\_id, now());

EXCEPTION

WHEN unique\_violation THEN -- (2)

-- don't worry if it already exists

END;

END;

$$ LANGUAGE plpgsql;

|  |  |
| --- | --- |
| [**(1)**](https://www.postgresql.org/docs/10/plpgsql-porting.html#co.plpgsql-porting-raise) | The syntax of RAISE is considerably different from Oracle's statement, although the basic case RAISE ***exception\_name*** works similarly. |
| [**(2)**](https://www.postgresql.org/docs/10/plpgsql-porting.html#co.plpgsql-porting-exception) | The exception names supported by PL/pgSQL are different from Oracle's. The set of built-in exception names is much larger (see [**Appendix A**](https://www.postgresql.org/docs/10/errcodes-appendix.html)). There is not currently a way to declare user-defined exception names, although you can throw user-chosen SQLSTATE values instead. |

The main functional difference between this procedure and the Oracle equivalent is that the exclusive lock on the cs\_jobs table will be held until the calling transaction completes. Also, if the caller later aborts (for example due to an error), the effects of this procedure will be rolled back.

### 42.12.2. Other Things to Watch For

This section explains a few other things to watch for when porting Oracle PL/SQL functions to PostgreSQL.

#### 42.12.2.1. Implicit Rollback After Exceptions

In PL/pgSQL, when an exception is caught by an EXCEPTION clause, all database changes since the block's BEGIN are automatically rolled back. That is, the behavior is equivalent to what you'd get in Oracle with:

BEGIN

SAVEPOINT s1;

... code here ...

EXCEPTION

WHEN ... THEN

ROLLBACK TO s1;

... code here ...

WHEN ... THEN

ROLLBACK TO s1;

... code here ...

END;

If you are translating an Oracle procedure that uses SAVEPOINT and ROLLBACK TO in this style, your task is easy: just omit the SAVEPOINT and ROLLBACK TO. If you have a procedure that uses SAVEPOINT and ROLLBACK TO in a different way then some actual thought will be required.

#### 42.12.2.2. EXECUTE

The PL/pgSQL version of EXECUTE works similarly to the PL/SQL version, but you have to remember to use quote\_literal and quote\_ident as described in [**Section 42.5.4**](https://www.postgresql.org/docs/10/plpgsql-statements.html#PLPGSQL-STATEMENTS-EXECUTING-DYN). Constructs of the type EXECUTE 'SELECT \* FROM $1'; will not work reliably unless you use these functions.

#### 42.12.2.3. Optimizing PL/PgSQL Functions

PostgreSQL gives you two function creation modifiers to optimize execution: “volatility” (whether the function always returns the same result when given the same arguments) and “strictness”(whether the function returns null if any argument is null). Consult the [**CREATE FUNCTION**](https://www.postgresql.org/docs/10/sql-createfunction.html) reference page for details.

When making use of these optimization attributes, your CREATE FUNCTION statement might look something like this:

CREATE FUNCTION foo(...) RETURNS integer AS $$

...

$$ LANGUAGE plpgsql STRICT IMMUTABLE;

### 42.12.3. Appendix

This section contains the code for a set of Oracle-compatible instr functions that you can use to simplify your porting efforts.

--

-- instr functions that mimic Oracle's counterpart

-- Syntax: instr(string1, string2 [, n [, m]])

-- where [] denotes optional parameters.

--

-- Search string1, beginning at the nth character, for the mth occurrence

-- of string2. If n is negative, search backwards, starting at the abs(n)'th

-- character from the end of string1.

-- If n is not passed, assume 1 (search starts at first character).

-- If m is not passed, assume 1 (find first occurrence).

-- Returns starting index of string2 in string1, or 0 if string2 is not found.

--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$

BEGIN

RETURN instr($1, $2, 1);

END;

$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE FUNCTION instr(string varchar, string\_to\_search\_for varchar,

beg\_index integer)

RETURNS integer AS $$

DECLARE

pos integer NOT NULL DEFAULT 0;

temp\_str varchar;

beg integer;

length integer;

ss\_length integer;

BEGIN

IF beg\_index > 0 THEN

temp\_str := substring(string FROM beg\_index);

pos := position(string\_to\_search\_for IN temp\_str);

IF pos = 0 THEN

RETURN 0;

ELSE

RETURN pos + beg\_index - 1;

END IF;

ELSIF beg\_index < 0 THEN

ss\_length := char\_length(string\_to\_search\_for);

length := char\_length(string);

beg := length + 1 + beg\_index;

WHILE beg > 0 LOOP

temp\_str := substring(string FROM beg FOR ss\_length);

IF string\_to\_search\_for = temp\_str THEN

RETURN beg;

END IF;

beg := beg - 1;

END LOOP;

RETURN 0;

ELSE

RETURN 0;

END IF;

END;

$$ LANGUAGE plpgsql STRICT IMMUTABLE;

CREATE FUNCTION instr(string varchar, string\_to\_search\_for varchar,

beg\_index integer, occur\_index integer)

RETURNS integer AS $$

DECLARE

pos integer NOT NULL DEFAULT 0;

occur\_number integer NOT NULL DEFAULT 0;

temp\_str varchar;

beg integer;

i integer;

length integer;

ss\_length integer;

BEGIN

IF occur\_index <= 0 THEN

RAISE 'argument ''%'' is out of range', occur\_index

USING ERRCODE = '22003';

END IF;

IF beg\_index > 0 THEN

beg := beg\_index - 1;

FOR i IN 1..occur\_index LOOP

temp\_str := substring(string FROM beg + 1);

pos := position(string\_to\_search\_for IN temp\_str);

IF pos = 0 THEN

RETURN 0;

END IF;

beg := beg + pos;

END LOOP;

RETURN beg;

ELSIF beg\_index < 0 THEN

ss\_length := char\_length(string\_to\_search\_for);

length := char\_length(string);

beg := length + 1 + beg\_index;

WHILE beg > 0 LOOP

temp\_str := substring(string FROM beg FOR ss\_length);

IF string\_to\_search\_for = temp\_str THEN

occur\_number := occur\_number + 1;

IF occur\_number = occur\_index THEN

RETURN beg;

END IF;

END IF;

beg := beg - 1;

END LOOP;

RETURN 0;

ELSE

RETURN 0;

END IF;

END;

$$ LANGUAGE plpgsql STRICT IMMUTABLE;

## Chapter 43. PL/Tcl - Tcl Procedural Language

PL/Tcl is a loadable procedural language for the PostgreSQL database system that enables the [**Tcl language**](http://www.tcl.tk/) to be used to write functions and trigger procedures.

## 43.1. Overview

PL/Tcl offers most of the capabilities a function writer has in the C language, with a few restrictions, and with the addition of the powerful string processing libraries that are available for Tcl.

One compelling good restriction is that everything is executed from within the safety of the context of a Tcl interpreter. In addition to the limited command set of safe Tcl, only a few commands are available to access the database via SPI and to raise messages via elog(). PL/Tcl provides no way to access internals of the database server or to gain OS-level access under the permissions of the PostgreSQL server process, as a C function can do. Thus, unprivileged database users can be trusted to use this language; it does not give them unlimited authority.

The other notable implementation restriction is that Tcl functions cannot be used to create input/output functions for new data types.

Sometimes it is desirable to write Tcl functions that are not restricted to safe Tcl. For example, one might want a Tcl function that sends email. To handle these cases, there is a variant of PL/Tcl called PL/TclU (for untrusted Tcl). This is exactly the same language except that a full Tcl interpreter is used. If *PL/TclU* is used, it must be installed as an untrusted procedural language so that only database superusers can create functions in it. The writer of a PL/TclU function must take care that the function cannot be used to do anything unwanted, since it will be able to do anything that could be done by a user logged in as the database administrator.

The shared object code for the PL/Tcl and PL/TclU call handlers is automatically built and installed in the PostgreSQL library directory if Tcl support is specified in the configuration step of the installation procedure. To install PL/Tcl and/or PL/TclU in a particular database, use the CREATE EXTENSION command, for example CREATE EXTENSION pltcl or CREATE EXTENSION pltclu.

<https://www.postgresql.org/docs/10/pltcl-overview.html>

## Chapter 44. PL/Perl - Perl Procedural Language

<https://www.postgresql.org/docs/10/plperl.html>

PL/Perl is a loadable procedural language that enables you to write PostgreSQL functions in the [**Perl programming language**](http://www.perl.org/).

The main advantage to using PL/Perl is that this allows use, within stored functions, of the manyfold “string munging” operators and functions available for Perl. Parsing complex strings might be easier using Perl than it is with the string functions and control structures provided in PL/pgSQL.

To install PL/Perl in a particular database, use CREATE EXTENSION plperl.

### Tip

If a language is installed into template1, all subsequently created databases will have the language installed automatically.

### Note

Users of source packages must specially enable the build of PL/Perl during the installation process. (Refer to [**Chapter 16**](https://www.postgresql.org/docs/10/installation.html) for more information.) Users of binary packages might find PL/Perl in a separate subpackage.

## Chapter 45. PL/Python - Python Procedural Language

<https://www.postgresql.org/docs/10/plpython.html>

The PL/Python procedural language allows PostgreSQL functions to be written in the [**Python language**](http://www.python.org/).

To install PL/Python in a particular database, use CREATE EXTENSION plpythonu (but see also [**Section 45.1**](https://www.postgresql.org/docs/10/plpython-python23.html)).

## Chapter 46. Server Programming Interface

<https://www.postgresql.org/docs/10/spi.html>

The Server Programming Interface (SPI) gives writers of user-defined C functions the ability to run SQL commands inside their functions. SPI is a set of interface functions to simplify access to the parser, planner, and executor. SPI also does some memory management.

### Note

The available procedural languages provide various means to execute SQL commands from procedures. Most of these facilities are based on SPI, so this documentation might be of use for users of those languages as well.

To avoid misunderstanding we'll use the term “function” when we speak of SPI interface functions and “procedure” for a user-defined C-function that is using SPI.

Note that if a command invoked via SPI fails, then control will not be returned to your procedure. Rather, the transaction or subtransaction in which your procedure executes will be rolled back. (This might seem surprising given that the SPI functions mostly have documented error-return conventions. Those conventions only apply for errors detected within the SPI functions themselves, however.) It is possible to recover control after an error by establishing your own subtransaction surrounding SPI calls that might fail.

SPI functions return a nonnegative result on success (either via a returned integer value or in the global variable SPI\_result, as described below). On error, a negative result or NULL will be returned.

Source code files that use SPI must include the header file executor/spi.h.

## Chapter 47. Background Worker Processes

PostgreSQL can be extended to run user-supplied code in separate processes. Such processes are started, stopped and monitored by postgres, which permits them to have a lifetime closely linked to the server's status. These processes have the option to attach to PostgreSQL's shared memory area and to connect to databases internally; they can also run multiple transactions serially, just like a regular client-connected server process. Also, by linking to libpq they can connect to the server and behave like a regular client application.

### Warning

There are considerable robustness and security risks in using background worker processes because, being written in the C language, they have unrestricted access to data. Administrators wishing to enable modules that include background worker process should exercise extreme caution. Only carefully audited modules should be permitted to run background worker processes.

Background workers can be initialized at the time that PostgreSQL is started by including the module name in shared\_preload\_libraries. A module wishing to run a background worker can register it by calling RegisterBackgroundWorker(BackgroundWorker \*worker) from its \_PG\_init(). Background workers can also be started after the system is up and running by calling the function RegisterDynamicBackgroundWorker(BackgroundWorker \*worker, BackgroundWorkerHandle \*\*handle). Unlike RegisterBackgroundWorker, which can only be called from within the postmaster, RegisterDynamicBackgroundWorker must be called from a regular backend.

The structure BackgroundWorker is defined thus:

typedef void (\*bgworker\_main\_type)(Datum main\_arg);

typedef struct BackgroundWorker

{

char bgw\_name[BGW\_MAXLEN];

int bgw\_flags;

BgWorkerStartTime bgw\_start\_time;

int bgw\_restart\_time; /\* in seconds, or BGW\_NEVER\_RESTART \*/

char bgw\_library\_name[BGW\_MAXLEN];

char bgw\_function\_name[BGW\_MAXLEN];

Datum bgw\_main\_arg;

char bgw\_extra[BGW\_EXTRALEN];

int bgw\_notify\_pid;

} BackgroundWorker;

bgw\_name is a string to be used in log messages, process listings and similar contexts.

bgw\_flags is a bitwise-or'd bit mask indicating the capabilities that the module wants. Possible values are:

BGWORKER\_SHMEM\_ACCESS

Requests shared memory access. Workers without shared memory access cannot access any of PostgreSQL's shared data structures, such as heavyweight or lightweight locks, shared buffers, or any custom data structures which the worker itself may wish to create and use.

BGWORKER\_BACKEND\_DATABASE\_CONNECTION

Requests the ability to establish a database connection through which it can later run transactions and queries. A background worker using BGWORKER\_BACKEND\_DATABASE\_CONNECTION to connect to a database must also attach shared memory using BGWORKER\_SHMEM\_ACCESS, or worker start-up will fail.

bgw\_start\_time is the server state during which postgres should start the process; it can be one of BgWorkerStart\_PostmasterStart (start as soon as postgres itself has finished its own initialization; processes requesting this are not eligible for database connections), BgWorkerStart\_ConsistentState (start as soon as a consistent state has been reached in a hot standby, allowing processes to connect to databases and run read-only queries), and BgWorkerStart\_RecoveryFinished (start as soon as the system has entered normal read-write state). Note the last two values are equivalent in a server that's not a hot standby. Note that this setting only indicates when the processes are to be started; they do not stop when a different state is reached.

bgw\_restart\_time is the interval, in seconds, that postgres should wait before restarting the process, in case it crashes. It can be any positive value, or BGW\_NEVER\_RESTART, indicating not to restart the process in case of a crash.

bgw\_library\_name is the name of a library in which the initial entry point for the background worker should be sought. The named library will be dynamically loaded by the worker process and bgw\_function\_name will be used to identify the function to be called. If loading a function from the core code, this must be set to "postgres".

bgw\_function\_name is the name of a function in a dynamically loaded library which should be used as the initial entry point for a new background worker.

bgw\_main\_arg is the Datum argument to the background worker main function. This main function should take a single argument of type Datum and return void. bgw\_main\_arg will be passed as the argument. In addition, the global variable MyBgworkerEntry points to a copy of the BackgroundWorker structure passed at registration time; the worker may find it helpful to examine this structure.

On Windows (and anywhere else where EXEC\_BACKEND is defined) or in dynamic background workers it is not safe to pass a Datum by reference, only by value. If an argument is required, it is safest to pass an int32 or other small value and use that as an index into an array allocated in shared memory. If a value like a cstring or text is passed then the pointer won't be valid from the new background worker process.

bgw\_extra can contain extra data to be passed to the background worker. Unlike bgw\_main\_arg, this data is not passed as an argument to the worker's main function, but it can be accessed via MyBgworkerEntry, as discussed above.

bgw\_notify\_pid is the PID of a PostgreSQL backend process to which the postmaster should send SIGUSR1 when the process is started or exits. It should be 0 for workers registered at postmaster startup time, or when the backend registering the worker does not wish to wait for the worker to start up. Otherwise, it should be initialized to MyProcPid.

Once running, the process can connect to a database by calling BackgroundWorkerInitializeConnection(*char \*dbname*, *char \*username*) or BackgroundWorkerInitializeConnectionByOid(*Oid dboid*, *Oid useroid*). This allows the process to run transactions and queries using the SPI interface. If dbname is NULL or dboid is InvalidOid, the session is not connected to any particular database, but shared catalogs can be accessed. If username is NULL or useroid is InvalidOid, the process will run as the superuser created during initdb. A background worker can only call one of these two functions, and only once. It is not possible to switch databases.

Signals are initially blocked when control reaches the background worker's main function, and must be unblocked by it; this is to allow the process to customize its signal handlers, if necessary. Signals can be unblocked in the new process by calling BackgroundWorkerUnblockSignals and blocked by calling BackgroundWorkerBlockSignals.

If bgw\_restart\_time for a background worker is configured as BGW\_NEVER\_RESTART, or if it exits with an exit code of 0 or is terminated by TerminateBackgroundWorker, it will be automatically unregistered by the postmaster on exit. Otherwise, it will be restarted after the time period configured via bgw\_restart\_time, or immediately if the postmaster reinitializes the cluster due to a backend failure. Backends which need to suspend execution only temporarily should use an interruptible sleep rather than exiting; this can be achieved by calling WaitLatch(). Make sure the WL\_POSTMASTER\_DEATH flag is set when calling that function, and verify the return code for a prompt exit in the emergency case that postgres itself has terminated.

When a background worker is registered using the RegisterDynamicBackgroundWorker function, it is possible for the backend performing the registration to obtain information regarding the status of the worker. Backends wishing to do this should pass the address of a BackgroundWorkerHandle \* as the second argument to RegisterDynamicBackgroundWorker. If the worker is successfully registered, this pointer will be initialized with an opaque handle that can subsequently be passed to GetBackgroundWorkerPid(*BackgroundWorkerHandle \**, *pid\_t \**) or TerminateBackgroundWorker(*BackgroundWorkerHandle \**). GetBackgroundWorkerPid can be used to poll the status of the worker: a return value of BGWH\_NOT\_YET\_STARTED indicates that the worker has not yet been started by the postmaster; BGWH\_STOPPED indicates that it has been started but is no longer running; and BGWH\_STARTED indicates that it is currently running. In this last case, the PID will also be returned via the second argument. TerminateBackgroundWorker causes the postmaster to send SIGTERM to the worker if it is running, and to unregister it as soon as it is not.

In some cases, a process which registers a background worker may wish to wait for the worker to start up. This can be accomplished by initializing bgw\_notify\_pid to MyProcPid and then passing the BackgroundWorkerHandle \* obtained at registration time to WaitForBackgroundWorkerStartup(*BackgroundWorkerHandle \*handle*, *pid\_t \**) function. This function will block until the postmaster has attempted to start the background worker, or until the postmaster dies. If the background runner is running, the return value will BGWH\_STARTED, and the PID will be written to the provided address. Otherwise, the return value will be BGWH\_STOPPED or BGWH\_POSTMASTER\_DIED.

If a background worker sends asynchronous notifications with the NOTIFY command via the Server Programming Interface (SPI), it should call ProcessCompletedNotifies explicitly after committing the enclosing transaction so that any notifications can be delivered. If a background worker registers to receive asynchronous notifications with the LISTEN through SPI, the worker will log those notifications, but there is no programmatic way for the worker to intercept and respond to those notifications.

The src/test/modules/worker\_spi module contains a working example, which demonstrates some useful techniques.

The maximum number of registered background workers is limited by [**max\_worker\_processes**](https://www.postgresql.org/docs/10/runtime-config-resource.html#GUC-MAX-WORKER-PROCESSES).

## Chapter 48. Logical Decoding

<https://www.postgresql.org/docs/10/logicaldecoding-example.html>

PostgreSQL provides infrastructure to stream the modifications performed via SQL to external consumers. This functionality can be used for a variety of purposes, including replication solutions and auditing.

Changes are sent out in streams identified by logical replication slots.

The format in which those changes are streamed is determined by the output plugin used. An example plugin is provided in the PostgreSQL distribution. Additional plugins can be written to extend the choice of available formats without modifying any core code. Every output plugin has access to each individual new row produced by INSERT and the new row version created by UPDATE. Availability of old row versions for UPDATE and DELETE depends on the configured replica identity (see [REPLICA IDENTITY](https://www.postgresql.org/docs/10/sql-altertable.html#SQL-CREATETABLE-REPLICA-IDENTITY)).

Changes can be consumed either using the streaming replication protocol (see [**Section 52.4**](https://www.postgresql.org/docs/10/protocol-replication.html) and [**Section 48.3**](https://www.postgresql.org/docs/10/logicaldecoding-walsender.html)), or by calling functions via SQL (see [**Section 48.4**](https://www.postgresql.org/docs/10/logicaldecoding-sql.html)). It is also possible to write additional methods of consuming the output of a replication slot without modifying core code (see [**Section 48.7**](https://www.postgresql.org/docs/10/logicaldecoding-writer.html)).

## Chapter 49. Replication Progress Tracking

<https://www.postgresql.org/docs/10/replication-origins.html>

Replication origins are intended to make it easier to implement logical replication solutions on top of [**logical decoding**](https://www.postgresql.org/docs/10/logicaldecoding.html). They provide a solution to two common problems:

* How to safely keep track of replication progress
* How to change replication behavior based on the origin of a row; for example, to prevent loops in bi-directional replication setups

Replication origins have just two properties, a name and an OID. The name, which is what should be used to refer to the origin across systems, is free-form text. It should be used in a way that makes conflicts between replication origins created by different replication solutions unlikely; e.g. by prefixing the replication solution's name to it. The OID is used only to avoid having to store the long version in situations where space efficiency is important. It should never be shared across systems.

Replication origins can be created using the function [pg\_replication\_origin\_create()](https://www.postgresql.org/docs/10/functions-admin.html#PG-REPLICATION-ORIGIN-CREATE); dropped using [pg\_replication\_origin\_drop()](https://www.postgresql.org/docs/10/functions-admin.html#PG-REPLICATION-ORIGIN-DROP); and seen in the [pg\_replication\_origin](https://www.postgresql.org/docs/10/catalog-pg-replication-origin.html) system catalog.

One nontrivial part of building a replication solution is to keep track of replay progress in a safe manner. When the applying process, or the whole cluster, dies, it needs to be possible to find out up to where data has successfully been replicated. Naive solutions to this, such as updating a row in a table for every replayed transaction, have problems like run-time overhead and database bloat.

Using the replication origin infrastructure a session can be marked as replaying from a remote node (using the [pg\_replication\_origin\_session\_setup()](https://www.postgresql.org/docs/10/functions-admin.html#PG-REPLICATION-ORIGIN-SESSION-SETUP) function). Additionally the LSN and commit time stamp of every source transaction can be configured on a per transaction basis using [pg\_replication\_origin\_xact\_setup()](https://www.postgresql.org/docs/10/functions-admin.html#PG-REPLICATION-ORIGIN-XACT-SETUP). If that's done replication progress will persist in a crash safe manner. Replay progress for all replication origins can be seen in the [pg\_replication\_origin\_status](https://www.postgresql.org/docs/10/view-pg-replication-origin-status.html) view. An individual origin's progress, e.g. when resuming replication, can be acquired using [pg\_replication\_origin\_progress()](https://www.postgresql.org/docs/10/functions-admin.html#PG-REPLICATION-ORIGIN-PROGRESS) for any origin or [pg\_replication\_origin\_session\_progress()](https://www.postgresql.org/docs/10/functions-admin.html#PG-REPLICATION-ORIGIN-SESSION-PROGRESS) for the origin configured in the current session.

In replication topologies more complex than replication from exactly one system to one other system, another problem can be that it is hard to avoid replicating replayed rows again. That can lead both to cycles in the replication and inefficiencies. Replication origins provide an optional mechanism to recognize and prevent that. When configured using the functions referenced in the previous paragraph, every change and transaction passed to output plugin callbacks (see [**Section 48.6**](https://www.postgresql.org/docs/10/logicaldecoding-output-plugin.html)) generated by the session is tagged with the replication origin of the generating session. This allows treating them differently in the output plugin, e.g. ignoring all but locally-originating rows. Additionally the [filter\_by\_origin\_cb](https://www.postgresql.org/docs/10/logicaldecoding-output-plugin.html#LOGICALDECODING-OUTPUT-PLUGIN-FILTER-ORIGIN) callback can be used to filter the logical decoding change stream based on the source. While less flexible, filtering via that callback is considerably more efficient than doing it in the output plugin.