

PL/pgSQL Dynamic Commands



16

Copyright

© Postgres Professional, 2017–2025

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Igor Gnatyuk

Translated by: Liudmila Mantrova, Alexander Meleshko, Elena Sharafutdinova

Photo by: Oleg Bartunov (Phu monastery, Bhrikuti summit, Nepal)

Use of Course Materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Objectives

Executing Dynamic Queries

Constructing Dynamic Commands

The text of an SQL query is constructed at run time

Objectives

- provide additional flexibility for the application
- construct several specific queries for optimization purposes
- instead of using a single query that covers all possible cases

Trade-off

- statements are not prepared
- the risk of SQL injection rises
- maintenance gets more complicated

An SQL command is considered dynamic if its text is constructed and then executed within PL/pgSQL routine blocks or in anonymous blocks.

In most cases, you can do without dynamic commands, but sometimes they can provide additional flexibility. For example, an application can have a built-in capability to execute commands provided via system settings. Instead of being hard-coded by developers, these settings can be tuned by the support team in production.

When creating reports with a large number of optional parameters, it is sometimes easier to construct the text of a query at run time for the provided arguments only, instead of creating a complex query that includes all possible parameter combinations while developing the application.

The price you pay for using dynamic commands is inability to take advantage of prepared statements, which are used in PL/pgSQL by default. Besides, you have to take care of dynamic commands' security as they are vulnerable to SQL injection.

We should also mention that maintenance gets more complicated. In particular, it will be impossible to scan the source code quickly for executable commands with tools like grep.

EXECUTE construct

- runs a text representation of an SQL query
- allows using parameters
- PL/pgSQL variables do not become implicit parameters

Can be used instead of an SQL query

- independently
- when opening a cursor
- in a loop over a query
- in the RETURN QUERY statement

To run dynamic commands, PL/pgSQL uses the EXECUTE construct which launches the SQL command provided as a text string.

A dynamic query can contain explicit parameters. In the command's text representation, parameters are denoted by \$1, \$2, etc.; their actual values are provided in the USING clause. Parameters are handled in the same way as in prepared statements (which is covered in Architecture. PostgreSQL Fundamentals). However, PL/pgSQL variables do not become implicit parameters, as it happens in the case of regular (as opposed to dynamic) use of SQL in PL/pgSQL.

The EXECUTE construct can be used as an independent statement (it will simply execute a dynamic command). It can also be used in loops over queries, when opening a cursor, or in the RETURN QUERY statement: in all these cases, EXECUTE replaces the SQL statement.

<https://postgrespro.com/docs/postgresql/16/plpgsql-statements#PLPGSQL-STATEMENTS-EXECUTING-DYN>

Note that a procedure cannot perform transaction control if it is called by an EXECUTE construct.

Executing Dynamic Queries

The EXECUTE statement runs SQL commands provided as text.

```
=> DO $$
DECLARE
    cmd CONSTANT text := 'CREATE TABLE city_london(
        name text, architect text, founded integer
    )';
BEGIN
    EXECUTE cmd; -- a table that lists examples of contemporary architecture in London
END
$$;

DO
```

The INTO clause of the EXECUTE statement enables saving a single row of the result (the first returned row) into a variable of a composite type or into several scalar variables.

As with static commands, you can check the result of a dynamic command using GET DIAGNOSTICS (but not the FOUND variable).

```
=> DO $$
DECLARE
    rec record;
    cnt bigint;
BEGIN
    EXECUTE 'INSERT INTO city_london (name, architect, founded) VALUES
        (''The Shard'', ''Renzo Piano'', 2009),
        (''The Scalpel'', ''Kohn Pedersen Fox'', 2018),
        (''London Aquatics Centre'', ''Zaha Hadid'', 2011),
        (''30 St Mary Axe'', ''Norman Foster'', 2001),
        (''London City Hall'', ''Norman Foster'', 2000)
        RETURNING name, architect, founded'
    INTO rec;
    RAISE NOTICE '%', rec;
    GET DIAGNOSTICS cnt := ROW_COUNT;
    RAISE NOTICE 'Added rows: %', cnt;
END
$$;

NOTICE:  ("The Shard","Renzo Piano",2009)
NOTICE:  Added rows: 5
DO
```

You can use STRICT to ensure that the command processes only one row.

The result of a dynamic query can be processed in a FOR loop.

```
=> DO $$
DECLARE
    rec record;
BEGIN
    FOR rec IN EXECUTE 'SELECT * FROM city_london WHERE architect = ''Norman Foster'' ORDER BY founded'
    LOOP
        RAISE NOTICE '%', rec;
    END LOOP;
END
$$;

NOTICE:  ("London City Hall","Norman Foster",2000)
NOTICE:  ("30 St Mary Axe","Norman Foster",2001)
DO
```

Here is the same example using a cursor:

```

=> DO $$
DECLARE
    cur refcursor;
    rec record;
BEGIN
    OPEN cur FOR EXECUTE 'SELECT * FROM city_london WHERE architect = ''Norman Foster'' ORDER BY founded';
    LOOP
        FETCH cur INTO rec;
        EXIT WHEN NOT FOUND;
        RAISE NOTICE '%', rec;
    END LOOP;
END
$$;

NOTICE:  ("London City Hall","Norman Foster",2000)
NOTICE:  ("30 St Mary Axe","Norman Foster",2001)
DO

```

The RETURN QUERY statement can also use dynamic queries to return rows from functions. Let's create a function that retrieves all buildings constructed by a particular architect and up to the specified year. We will have to use parameters for this purpose:

```

=> CREATE FUNCTION sel_london(architect text, founded integer DEFAULT NULL)
RETURNS SETOF text
AS $$
DECLARE
    -- parameters are numbered: $1, $2...
    cmd text := '
        SELECT name FROM city_london
        WHERE architect = $1 AND ($2 IS NULL OR founded <= $2)';
BEGIN
    RETURN QUERY
        EXECUTE cmd
        USING architect, founded; -- provide parameters in the order of their declaration
END
$$ LANGUAGE plpgsql;

CREATE FUNCTION

=> SELECT * FROM sel_london('Norman Foster');

    sel_london
-----
30 St Mary Axe
London City Hall
(2 rows)

=> SELECT * FROM sel_london('Norman Foster', 2000);

    sel_london
-----
London City Hall
(1 row)

```

Parameter values binding

- USING clause
- guarantees protection against SQL injection

Escaping values

- identifiers: `format('%I')`, `quote_ident`
- literals: `format('%L')`, `quote_literal`, `quote_nullable`
- SQL injection is impossible if implemented correctly

Regular string functions

- concatenation, etc.
- risk of SQL injection!

Using the EXECUTE construct makes sense if the command is constructed dynamically. The previous examples could also do without EXECUTE.

Since the command is represented by a text string, it can be constructed using regular string functions that perform such operations as concatenation, etc. This should be done with great care as there is a risk of SQL injection.

If the values are passed as parameters in the USING clause, SQL injection is technically impossible.

However, it is not always possible to use parameters: you may have to concatenate specific parts of the query or insert a table name into the query. In this case, you should escape the values received from an unreliable source to protect your application against injections.

Identifiers are generated by either the format function with the %I specifier or the quote_ident function. These functions ensure that identifiers have valid names by double-quoting them and escaping special characters, if required.

To insert literals into the command text, you can use either quote_literal and quote_nullable functions or the format function with the %L specifier.

<https://postgrespro.com/docs/postgresql/16/functions-string>

Dealing with SQL Injection

Let's rewrite the function returning buildings and add one more parameter: the name of the city. The idea is to allow this function to access tables only if their names start with city_.

```
=> CREATE FUNCTION sel_city(
    city_code text,
    architect text,
    founded integer DEFAULT NULL
)
RETURNS SETOF text AS $$
DECLARE
    cmd text := '
        SELECT name FROM city_' || city_code || '
        WHERE architect = $1 AND ($2 IS NULL OR founded < $2)';
BEGIN
    RAISE NOTICE '%', cmd;
    RETURN QUERY
        EXECUTE cmd
        USING architect, founded;
END
$$ LANGUAGE plpgsql;

CREATE FUNCTION
```

The function works fine if its parameter values are “correct”:

```
=> SELECT * FROM sel_city('london', 'Renzo Piano');

NOTICE:
    SELECT name FROM city_london
    WHERE architect = $1 AND ($2 IS NULL OR founded < $2)
 sel_city
-----
The Shard
(1 row)
```

But a malicious user can pick a value that will change the syntactic structure of the query and enable unauthorized access to data:

```
=> SELECT * FROM sel_city('london WHERE false
    UNION ALL
    SELECT username FROM pg_user
    UNION ALL
    SELECT name FROM city_london', '');

NOTICE:
    SELECT name FROM city_london WHERE false
    UNION ALL
    SELECT username FROM pg_user
    UNION ALL
    SELECT name FROM city_london
    WHERE architect = $1 AND ($2 IS NULL OR founded < $2)
 sel_city
-----
student
postgres
(2 rows)
```

When you are using prepared statements or dynamic commands with parameters, such a situation is technically impossible because the structure of the SQL query is locked while the statement is parsed. An expression will always remain an expression; it is impossible to convert it, say, into a table name.

Constructing a Dynamic Command

It is impossible to provide the names of objects (such as tables or columns) as parameters of the USING clause in a dynamic command. Such identifiers must be escaped, so that it is impossible to modify the query structure:

```
=> SELECT format('%I', 'foo'),
    format('%I', 'foo bar'),
    format('%I', 'foo"bar');
```


format		format		format
foo		"foo bar"		"foo"bar"

(1 row)

The following function does the same thing:

```
=> SELECT quote_ident('foo'),
         quote_ident('foo bar'),
         quote_ident('foo"bar');
```

quote_ident		quote_ident		quote_ident
foo		"foo bar"		"foo"bar"

(1 row)

Here is an example of creating a table:

```
=> DO $$
DECLARE
    cmd CONSTANT text := 'CREATE TABLE %I(
        name text, architect text, founded integer
    )';
BEGIN
    EXECUTE format(cmd, 'city_paris'); -- a table for Paris
    EXECUTE format(cmd, 'city_milan'); -- a table for Milan
END
$$;

DO
```

Instead of using parameters, you can insert literals into a string. It also requires escaping, but in a bit different way:

```
=> SELECT format('%L', 'foo bar'),
         format('%L', 'foo''bar'),
         format('%L', NULL);
```

format		format		format
'foo bar'		'foo''bar'		NULL

(1 row)

The quote_nullable function also does the same thing:

```
=> SELECT quote_nullable('foo bar'),
         quote_nullable('foo''bar'),
         quote_nullable(NULL);
```

quote_nullable		quote_nullable		quote_nullable
'foo bar'		'foo''bar'		NULL

(1 row)

The quote_literal function is quite similar, but it does not convert NULL values into literals:

```
=> SELECT quote_literal(NULL);
```

quote_literal

(1 row)

As an example, let's rewrite the function that returns the list of buildings of a particular city, so that it does not use any parameters, but still remains safe.

```

=> CREATE OR REPLACE FUNCTION sel_city(
    city_code text,
    architect text,
    founded integer DEFAULT NULL
)
RETURNS SETOF text
AS $$
DECLARE
    cmd text := '
        SELECT name FROM %I
        WHERE architect = %L AND (%L IS NULL OR founded < %L::integer)';
BEGIN
    RETURN QUERY EXECUTE format(
        cmd, 'city_'||city_code, architect, founded, founded
    );
END
$$ LANGUAGE plpgsql;

CREATE FUNCTION

```

Note that we perform two extra type casting operations: first, the integer parameter is converted into a string, and then it is cast back to integer at run time (this can be avoided with USING parameters):

```

=> SELECT * FROM sel_city('london', 'Renzo Piano', 2009);

 sel_city
-----
(0 rows)

```

An attempt to pass an invalid value will not succeed:

```

=> SELECT * FROM sel_city('london WHERE false
    UNION ALL
    SELECT username FROM pg_user
    UNION ALL
    SELECT name FROM city_london', '');

```

```

NOTICE: identifier "city_london WHERE false
    UNION ALL
    SELECT username FROM pg_user
    UNION ALL
    SELECT name FROM city_london" will be truncated to "city_london WHERE false
    UNION ALL
    SELECT usenam"

```

```

ERROR: relation "city_london WHERE false
    UNION ALL
    SELECT usenam" does not exist
LINE 2:     SELECT name FROM "city_london WHERE false
                        ^

```

```

QUERY:
    SELECT name FROM "city_london WHERE false
    UNION ALL
    SELECT username FROM pg_user
    UNION ALL
    SELECT name FROM city_london"
    WHERE architect = '' AND (NULL IS NULL OR founded < NULL::integer)
CONTEXT: PL/pgSQL function sel_city(text,text,integer) line 7 at RETURN QUERY

```

Dynamic commands provide additional flexibility

Constructing separate queries for different arguments can improve performance

Dynamic commands are not suitable for short, frequent queries

Maintenance gets more complicated

1. Modify the `get_catalog` function so that the query to the `catalog_v` view is constructed dynamically and takes into account only those fields that are filled out in the search form of the Store tab.
Make sure that your implementation is protected against SQL injection.
Check your function in the application.

1. Suppose we have to generate the following query if these conditions are met: the “In stock” option is selected in the search form, but “Book Title” and “Author” fields are empty.

```
SELECT ... FROM catalog_v WHERE onhand_qty > 0;
```

You should keep in mind that this implementation will not necessarily speed up search, but it will certainly be harder to maintain. Avoid such solutions in production environments unless you have a solid reason to use this technique. To learn more about query performance tuning, check out the QPT course.

1. The get_catalog Function

```
=> CREATE OR REPLACE FUNCTION get_catalog(  
    author_name text,  
    book_title text,  
    in_stock boolean  
)  
RETURNS TABLE(book_id integer, display_name text, onhand_qty integer)  
AS $$  
DECLARE  
    title_cond text := '';  
    author_cond text := '';  
    qty_cond text := '';  
    cmd text;  
BEGIN  
    IF book_title != '' THEN  
        title_cond := format(  
            ' AND cv.title ILIKE %L', '%' || book_title || '%'  
        );  
    END IF;  
    IF author_name != '' THEN  
        author_cond := format(  
            ' AND cv.authors ILIKE %L', '%' || author_name || '%'  
        );  
    END IF;  
    IF in_stock THEN  
        qty_cond := ' AND cv.onhand_qty > 0';  
    END IF;  
    cmd := 'SELECT cv.book_id,  
                cv.display_name,  
                cv.onhand_qty  
            FROM   catalog_v cv  
            WHERE  true'  
            || title_cond || author_cond || qty_cond || '  
            ORDER BY display_name';  
    RAISE NOTICE '%', cmd;  
    RETURN QUERY EXECUTE cmd;  
END  
$$ STABLE LANGUAGE plpgsql;  
  
CREATE FUNCTION
```

1. Create a function that returns a matrix report on functions available in the database.

The columns must contain names of function owners, the rows must provide schema names, while the cells must display the number of functions that belong to a particular owner in a particular schema.

What statement should you write to invoke this function?

10

1. Here is a possible output:

<i>schema</i>	<i>total</i>	<i>postgres</i>	<i>student</i>	<i>...</i>
information_schema	12	12	0	
pg_catalog	2811	2811	0	
public	3	0	3	
...				

The number of columns returned by a query is unknown beforehand. You need to construct a query and then execute it dynamically. The query text can be as follows:

```
SELECT pronamespace::regnamespace::text AS schema,
       COUNT(*) AS total
       ,SUM(CASE WHEN proowner = 10 THEN 1 ELSE 0 END) postgres
       ,SUM(CASE WHEN proowner = 16384 THEN 1 ELSE 0 END) student
FROM pg_proc
GROUP BY pronamespace::regnamespace
ORDER BY schema
```

The highlighted lines are a dynamic part that has to be constructed by a separate query. The start and the end of the query are static.

The *proowner* column has the *oid* type. To get the name of the owner, you can use the following construct: `proowner::regrole::text`.

Building a Matrix Report

```
=> CREATE DATABASE plpgsql_dynamic;
```

```
CREATE DATABASE
```

```
=> \c plpgsql_dynamic
```

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

A function for generating dynamic query text:

```
=> CREATE FUNCTION form_query() RETURNS text
AS $$
DECLARE
    query_text text;
    columns text := '';
    r record;
BEGIN
    -- Static part of the query.
    -- First two columns: schema name and number of functions
    query_text :=
$query$
SELECT pronamespace::regnamespace::text AS schema
      , count(*) AS total{{columns}}
FROM pg_proc
GROUP BY pronamespace::regnamespace
ORDER BY schema
$query$;

    -- Dynamic part of the query.
    -- Get a list of function owners, a separate column for each
    FOR r IN SELECT DISTINCT proowner AS owner FROM pg_proc ORDER BY 1
    LOOP
        columns := columns || format(
            E'\n      , sum(CASE WHEN proowner = %s THEN 1 ELSE 0 END) AS %I',
            r.owner,
            r.owner::regrole
        );
    END LOOP;

    RETURN replace(query_text, '{{columns}}', columns);
END
$$ STABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

The resulting query text:

```
=> SELECT form_query();
```

```
----- form_query -----
SELECT pronamespace::regnamespace::text AS schema
      , count(*) AS total
      , sum(CASE WHEN proowner = 10 THEN 1 ELSE 0 END) AS postgres
      , sum(CASE WHEN proowner = 16384 THEN 1 ELSE 0 END) AS student
FROM pg_proc
GROUP BY pronamespace::regnamespace
ORDER BY schema

(1 row)
```

Now create a matrix report function:

```
=> CREATE FUNCTION matrix() RETURNS SETOF record
AS $$
BEGIN
    RETURN QUERY EXECUTE form_query();
END
$$ STABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Simply executing the query will result in an error, as the structure of the returned records is not specified:

```
=> SELECT * FROM matrix();
```

ERROR: a column definition list is required for functions returning "record"

```
LINE 1: SELECT * FROM matrix();
      ^
```

This is an important limitation on using functions that return an arbitrary set of values. You need to know and specify the structure of the record to be returned as the function is being called.

In general, the returned record structure may be unknown, but for our matrix report, we can run another query that will show you how to call the matrix function correctly.

Prepare the query text:

```
=> CREATE FUNCTION matrix_call() RETURNS text
AS $$
DECLARE
    cmd text;
    r record;
BEGIN
    cmd := 'SELECT * FROM matrix() AS (
        schema text, total bigint';

    FOR r IN SELECT DISTINCT proowner AS owner FROM pg_proc ORDER BY 1
    LOOP
        cmd := cmd || format(', %I bigint', r.owner::regrole::text);
    END LOOP;
    cmd := cmd || E'\n)';

    RAISE NOTICE '%', cmd;
    RETURN cmd;
END
$$ STABLE LANGUAGE plpgsql;
```

CREATE FUNCTION

Now we can use matrix_call to get a query that reflects the structure of the matrix report, and then execute this query and get the report (psql allows you to do all this with single \gexec command):

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

BEGIN

```
=> SELECT matrix_call() \gexec
```

```
NOTICE: SELECT * FROM matrix() AS (
        schema text, total bigint, postgres bigint, student bigint
    )
```

schema	total	postgres	student
information_schema	11	11	0
pg_catalog	3286	3286	0
public	3	0	3

(3 rows)

```
=> COMMIT;
```

COMMIT

The matrix report is generated correctly.

- Repeatable Read isolation level guarantees that the report will be generated, even if a function with a new owner will come up between two queries.
- The query returned by form_query could have been run manually. However, you still need to return the list of columns to the client. The matrix_call function demonstrates how to solve this with an extra query.

Another solution is to return a set of strings of a semi-structured type (such as JSON or XML) instead of an arbitrary structured set. These types are covered in the DEV2 course.

```
=> \c postgres
```

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

```
=> DROP DATABASE plpgsql_dynamic;
```

DROP DATABASE