Authors: Egor Rogov, Pavel Luzanov
Translated by Liudmila Mantrova

Contact Us
Please send your feedback to: edu@postgrespro.ru

Agenda

Motivation

Executing dynamic queries

Constructing dynamic queries

Dynamic SQL

The text of an SQL command is constructed at run time

Motivation

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

Cost

- statements cannot be 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 when the application is running.

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 specified parameters 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 available in PL/pgSQL. 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 find all executable commands in source code using full-text search.

Executing Dynamic Queries

`EXECUTE` operator

- 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` clause

To run dynamic commands, PL/pgSQL uses the `EXECUTE` operator that launches the SQL operator 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 lecture "Architecture. A General Overview of PostgreSQL"). 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` operator can be used on its own (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` command: in all these cases, `EXECUTE` replaces the SQL operator.

https://postgrespro.com/docs/postgresql/12/plpgsql-statements#PLPGSQL-STATEMENTS-EXECUTING-DYN

Note the following fact: a procedure cannot perform transaction control if it is called by the `EXECUTE` operator.

Executing Dynamic Queries

The EXECUTE operator allows running SQL commands provided as text strings.

```
=> 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 operator enables saving a single row of the result (the first returned row) into a variable of a composite type or into several scalar variables.

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

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

```
NOTICE:  ("The Shard","Renzo Piano",2009)
NOTICE:  Added rows: 3
DO
```

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

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

```
NOTICE:  ("London City Hall","Norman Foster",2000)
NOTICE:  ("30 St Mary Axe","Norman Foster",2001)
NOTICE:  ("The Shard","Renzo Piano",2009)
DO
```

Here is the same example using a cursor:

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

```
NOTICE:  ("London City Hall","Norman Foster",2000)
NOTICE:  ("30 St Mary Axe","Norman Foster",2001)
NOTICE:  ("The Shard","Renzo Piano",2009)
DO
```

The RETURN QUERY operator can also use dynamic queries to return rows from functions. Let's create a function that retrieves all buildings constructed by a particular architect, possibly in 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)
```

Constructing Commands

Substituting parameter values

- `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 in the case of correct use

Regular string functions

- concatenation, etc.
- there is a risk of SQL injection!

Using the EXECUTE operator 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. But it should be done with great care as there is a risk of SQL injection.

If the values are passes 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/12/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 usename FROM pg_user
        UNION ALL
        SELECT name FROM city_london', '');
```

```
NOTICE:
        SELECT name FROM city_london WHERE false
        UNION ALL
        SELECT usename FROM pg_user
        UNION ALL
        SELECT name FROM city_london
        WHERE architect = $1 AND ($2 IS NULL OR founded = $2)
 sel_city
----------
 student
 postgres
 employee
 buyer
(4 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 have to perform two extra type casting operations: first, the integer variable is converted into a string, and then it is cast back to integer at run time:

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

 sel_city
-----------
 The Shard
(1 row)
```

An attempt to pass an invalid value will not succeed:

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

NOTICE:  identifier "city_london WHERE false
        UNION ALL
        SELECT usename 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 usename 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
```

Summary

Dynamic commands provide additional flexibility

Constructing separate queries for different parameter values can improve performance

Dynamic commands are not suitable for short, frequently used queries

Maintenance gets more complicated

Practice

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.

Task 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 real life unless you have a solid reason to use this technique. To learn more about query performance tuning, check out the QPT course.

Task 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 $$
DECLARE
    title_cond text := '';
    author_cond text := '';
    qty_cond text := '';
BEGIN
    IF book_title != '' THEN
        title_cond := format(
            ' AND cv.title ILIKE %L', '%'||book_title||'%'
        );
    END IF;
    IF author_name != '' THEN
        author_cond := format(
            ' AND cv.authors ILIKE %L', '%'||author_name||'%'
        );
    END IF;
    IF in_stock THEN
        qty_cond := ' AND cv.onhand_qty > 0';
    END IF;
    RETURN QUERY EXECUTE '
        SELECT cv.book_id,
               cv.display_name,
               cv.onhand_qty
        FROM   catalog_v cv
        WHERE  true'
        || title_cond || author_cond || qty_cond || '
        ORDER BY display_name';
END;
$$ STABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Practice

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

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

   What statement should you write to invoke this function?

Task 1. Here is a possible output:

```
       schema        total  postgres  student ...
 information_schema     12        12        0
 pg_catalog           2811      2811        0
 public                  3         0        3
 ...
```

The number of columns returned by a query is unknown in advance. So it is required 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`.

Task 1. Getting a Matrix Report

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

```
CREATE DATABASE
```

```
=> \c plpgsql_dynamic
```

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

An auxiliary function for constructing the text of dynamic queries:

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

    -- A dynamic part of the query
    -- Getting the list of function owners, each in a separate column
    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 final 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 let's create a function for generating a matrix report:

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

```
CREATE FUNCTION
```

If we simply execute this query, we will get an error as the structure of the returned rows is not specified:

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

```
ERROR:  a column definition list is required for functions returning "record"
LINE 1: SELECT * FROM matrix();
                      ^
```

It is an important limitation of using functions that return an arbitrary result. We must know the structure of the record to be returned and specify it at the time of the function call.

In general, the structure of the returned record can be unknown. But in the case of our matrix report, we can run one more query and see how to call the matrix function correctly.

Let's prepare a query text:

```
=> CREATE FUNCTION matrix_call() RETURNS text
AS $$
DECLARE
    cmd text;
    r record;
BEGIN
    cmd := 'SELECT * FROM matrix() AS m(
        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 get the structure of the matrix report in the first query, and then construct this report in the second query (and it will be done in a single psql command):

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

```
BEGIN
```

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

```
NOTICE:  SELECT * FROM matrix() AS m(
        schema text, total bigint, postgres bigint, student bigint
)
       schema       | total | postgres | student
--------------------+-------+----------+---------
 information_schema |    12 |       12 |       0
 pg_catalog         |  2948 |     2948 |       0
 public             |     3 |        0 |       3
(3 rows)


=> COMMIT;
```

```
COMMIT
```

The matrix report is built correctly.

- The Repeatable Read isolation level ensures that the report is built even if a function with a new owner is added between the two queries.
- The query returned by the form_query function could also be run directly. But we still have to get the list of the returned columns in the client application. The matrix_call function shows how to do it using an additional query.

There is one more possible solution: instead of returning a set of records of arbitrary structure, we could return a set of rows of a semistructured type (such as JSON or XML). These types are covered in the DEV2 course.