PostgreSQL function parameters

Function authorization related parameters

- The first parameters are related to security; when functions are called, they are executed within a security context that determines their privileges.
- The following options control the function privileges context:
  - `SECURITY DEFINER`
  - `SECURITY INVOKER`

- The default value for this option is `SECURITY INVOKER`, which indicates that the function will be executed with the privileges of the user that calls it.
- The `SECURITY DEFINER` functions will be executed using the privileges of the user that created it.
- For the `SECURITY INVOKER` functions, the user must have the permissions to execute the `CRUD` operations that the function implements; otherwise, the function will raise an error.
- The `SECURITY DEFINER` functions are very useful in defining triggers or for temporarily promoting the user to perform tasks only supported by the function.

- To test these security parameters, let's create two dummy functions using the `postgres` user and execute them in different sessions, as follows:

```
(base) joshua@joshua-VirtualBox:~$ cd /etc/postgresql/11/main
(base) joshua@joshua-VirtualBox:/etc/postgresql/11/main$ ls
conf.d  environment  pg_ctl.conf  pg_hba.conf  pg_ident.conf  postgresql.conf  start.conf
(base) joshua@joshua-VirtualBox:/etc/postgresql/11/main$ sudo vim pg_hba.conf
```

```
# DO NOT DISABLE!
# If you change this first entry you will need to make sure that the
# database superuser can access the database using some other method.
# Noninteractive access to all databases is required during automatic
# maintenance (custom daily cronjobs, replication, and similar tasks).
#
# Database administrative login by Unix domain socket
local   all             postgres                                peer

# TYPE  DATABASE        USER            ADDRESS                 METHOD

# "local" is for Unix domain socket connections only
local   all             all                                     peer
# IPv4 local connections:
host    all             all             127.0.0.1/32            md5
# IPv6 local connections:
host    all             all             ::1/128                 md5
```

```
# DO NOT DISABLE!
# If you change this first entry you will need to make sure that the
# database superuser can access the database using some other method.
# Noninteractive access to all databases is required during automatic
# maintenance (custom daily cronjobs, replication, and similar tasks).
#
# Database administrative login by Unix domain socket
local   all             all                                     trust

# TYPE  DATABASE        USER            ADDRESS                 METHOD

# "local" is for Unix domain socket connections only
local   all             all                                     peer
# IPv4 local connections:
host    all             all             127.0.0.1/32            md5
# IPv6 local connections:
host    all             all             ::1/128                 md5
```

```
(base) joshua@joshua-VirtualBox:/etc/postgresql/11/main$ sudo pg_ctlcluster 11 main reload
(base) joshua@joshua-VirtualBox:/etc/postgresql/11/main$
```

```
(base) joshua@joshua-VirtualBox:/etc/postgresql/11/main$ psql -U postgres car_portal
psql (11.5 (Ubuntu 11.5-3.pgdg18.04+1))
Type "help" for help.

car_portal=# CREATE FUNCTION test_security_definer () RETURNS TEXT AS $$
car_portal$#         SELECT format ('current_user:%s   session_user:%s', current_user, session_user);
car_portal$# $$ LANGUAGE SQL SECURITY DEFINER;
CREATE FUNCTION
car_portal=# CREATE FUNCTION test_security_invoker () RETURNS TEXT AS $$
car_portal$#         SELECT format ('current_user:%s   session_user:%s', current_user, session_user);
car_portal$# $$ LANGUAGE SQL SECURITY INVOKER;
CREATE FUNCTION
car_portal=# \q
(base) joshua@joshua-VirtualBox:/etc/postgresql/11/main$
```

- To test the functions, let's execute them using a session created by the `postgres` user, as follows:

```
(base) joshua@joshua-VirtualBox:/etc/postgresql/11/main$ psql -U postgres car_portal
psql (11.5 (Ubuntu 11.5-3.pgdg18.04+1))
Type "help" for help.

car_portal=# SELECT test_security_definer() , test_security_invoker();
             test_security_definer              |             test_security_invoker
------------------------------------------------+------------------------------------------------
 current_user:postgres   session_user:postgres | current_user:postgres   session_user:postgres
(1 row)

car_portal=# \q
(base) joshua@joshua-VirtualBox:/etc/postgresql/11/main$
```

- Now, let's use another session created by the `car_portal_app` user, as follows:

```
(base) joshua@joshua-VirtualBox:/etc/postgresql/11/main$ psql -U car_portal_app  car_portal
psql (11.5 (Ubuntu 11.5-3.pgdg18.04+1))
Type "help" for help.

car_portal=> SELECT test_security_definer() , test_security_invoker();
                test_security_definer                 |                   test_security_invoker
------------------------------------------------------+----------------------------------------------------------
 current_user:postgres   session_user:car_portal_app | current_user:car_portal_app   session_user:car_portal_app
(1 row)

car_portal=> \q
```

Function planner related parameters

- Function planner related parameters give the planner information about a function's execution cost.
- This helps the planner to generate a good execution plan.
- The following three parameters are used by the planner to determine the cost of executing the function, the number of rows that are expected to be returned, and whether the function pushes down when evaluating predicates:
  - `LEAKPROOF`: `LEAKPROOF` means that the function has no side effects. It does not reveal any information about its argument. For example, it does not throw error messages about its argument. This parameter affects views with the `security_barrier` parameter.
  - `COST`: This declares the execution cost per row; the default value for the C language function is `1`, and for PL/pgSQL, it is `100`. The cost is used by the planner to determine the best execution plan.
  - `ROWS`: The estimated number of rows returned by the function, if the function is set-returning. The default value is `1000`.

- To understand the effect of the `rows` configuration parameter, let's consider the following example:

```
car_portal=# CREATE OR REPLACE FUNCTION a() RETURNS SETOF INTEGER AS $$
car_portal$#         SELECT 1;
car_portal$# $$ LANGUAGE SQL;
CREATE FUNCTION
```

- Now, let's execute the following query:

```
car_portal=# EXPLAIN SELECT * FROM a() CROSS JOIN (Values(1),(2),(3)) as foo;
                              QUERY PLAN
-------------------------------------------------------------------------
 Nested Loop  (cost=0.25..47.80 rows=3000 width=8)
   ->  Function Scan on a  (cost=0.25..10.25 rows=1000 width=4)
   ->  Materialize  (cost=0.00..0.05 rows=3 width=4)
         ->  Values Scan on "*VALUES*"  (cost=0.00..0.04 rows=3 width=4)
(4 rows)
```

- The SQL function return type is `SETOF INTEGER`, which means that the planner expected more than one row to be returned from the function.
- Since the `ROWS` parameter is not specified, the planner uses the default value, which is `1000`.
- Finally, due to `CROSS JOIN`, the total estimated number of rows is `3000`, which is calculated as `3 x 1,000`.

- In the preceding example, an incorrect estimation is not critical.
- However, in a real-life example, where we might have several joins, the error of row estimation will be propagated and amplified, leading to poor execution plans.

- The `COST` function parameter determines when the function will be executed, as follows:
  - It determines the function execution order
  - It determines whether the function call can be pushed down

- The following example shows how the execution order for functions is affected by the `COST` function.
- Let's suppose that we have two functions, as follows:

```
car_portal=# CREATE OR REPLACE FUNCTION slow_function (anyelement) RETURNS BOOLEAN AS $$
car_portal$# BEGIN
car_portal$#         RAISE NOTICE 'Slow function %', $1;
car_portal$#         RETURN TRUE;
car_portal$# END; $$ LANGUAGE PLPGSQL COST 10000;
CREATE FUNCTION
car_portal=# CREATE OR REPLACE FUNCTION fast_function (anyelement) RETURNS BOOLEAN AS $$
car_portal$# BEGIN
car_portal$#         RAISE NOTICE 'Fast function %', $1;
car_portal$#         RETURN TRUE;
car_portal$# END; $$ LANGUAGE PLPGSQL COST 0.0001;
CREATE FUNCTION
```

```
car_portal=# EXPLAIN SELECT * FROM pg_language WHERE fast_function(lanname) AND slow_function(lanname) AND lanname ILIKE '%
sql%';
                                      QUERY PLAN
-------------------------------------------------------------------------------------------------
 Seq Scan on pg_language  (cost=0.00..101.05 rows=1 width=114)
   Filter: (fast_function(lanname) AND (lanname ~~* '%sql%'::text) AND slow_function(lanname))
(2 rows)

car_portal=# EXPLAIN SELECT * FROM pg_language WHERE slow_function(lanname) AND fast_function(lanname) AND lanname ILIKE '%
sql%';
                                      QUERY PLAN
-------------------------------------------------------------------------------------------------
 Seq Scan on pg_language  (cost=0.00..101.05 rows=1 width=114)
   Filter: (fast_function(lanname) AND (lanname ~~* '%sql%'::text) AND slow_function(lanname))
(2 rows)
```

- The preceding two SQL statements are identical, but the predicates are shuffled.
- Both of the statements give the same execution plan.
- Notice how the predicates are arranged in the filter execution plane node.
- `fast_function` is evaluated first, followed by the `ILIKE` operator, and finally, `slow_function` is pushed.

- When you execute one of the preceding statements, you will get the following results:

```
car_portal=# SELECT lanname FROM pg_language WHERE lanname ILIKE '%sql%' AND slow_function(lanname)AND fast_function(lannam
e);
NOTICE:  Fast function internal
NOTICE:  Fast function c
NOTICE:  Fast function sql
NOTICE:  Slow function sql
NOTICE:  Fast function plpgsql
NOTICE:  Slow function plpgsql
 lanname
---------
 sql
 plpgsql
(2 rows)
```

- Notice that `fast_function` was executed four times and `slow_function` was only executed twice.
- This behavior is known as **short circuit** evaluation.
- `slow_function` is only executed when `fast_function` and the `ILIKE` operator have returned true.

- Views can be used to implement authorization, and they can be used to hide data from some users.
- The function cost parameter could be exploited in earlier versions of PostgreSQL to crack views; however, this has been improved by the introduction of the `LEAKPROOF` and `SECURITY_BARRIER` flags.

- To be able to exploit the function `cost` parameter to get data from a view, several conditions should be met, some of which are as follows:
  - The function `cost` should be very low.
  - The function should be marked as `LEAKPROOF`. Note that only superusers are allowed to mark functions as `LEAKPROOF`.
  - The `VIEW security_barrier` flag shouldn't be set.
  - The function should be executed and not ignored due to short-circuit evaluation.

- Meeting all of these conditions is very difficult.
- The following code shows a hypothetical example of exploiting views.
- First, let's create a view, alter the `fast_function` function, and set it as `LEAKPROOF`:

```
car_portal=# CREATE OR REPLACE VIEW pg_sql_pl AS SELECT lanname FROM pg_language WHERE lanname ILIKE '%sql%';
CREATE VIEW
car_portal=# ALTER FUNCTION fast_function(anyelement) LEAKPROOF;
ALTER FUNCTION
car_portal=#
```

- To exploit the function, let's execute the following query:

```
car_portal=# SELECT * FROM pg_sql_pl WHERE fast_function(lanname);
NOTICE:  Fast function internal
NOTICE:  Fast function c
NOTICE:  Fast function sql
NOTICE:  Fast function plpgsql
 lanname
---------
 sql
 plpgsql
(2 rows)
```

- In the preceding example, the view itself should not show `c` and `internal`.
- By exploiting the function `cost`, the function was executed before executing the `lanname ILIKE '%sql%'` filter , exposing information that will never be shown by the view.

- Since only superusers are allowed to mark a function as `LEAKPROOF`, exploiting the function `cost` is not possible in newer versions of PostgreSQL.

Function configuration related parameters

- PostgreSQL can be configured per session, as well as within a function scope.
- This is quite useful in particular cases, when we want to override the session settings in the function.
- The settings parameters can be used to determine resources (such as the amount of memory required to perform an operation such as `work_mem`), or they can be used to determine execution behavior, such as disabling a sequential scan or nested loop joins.
- Only parameters that have the context of the user can be used (referring to the settings parameters that can be assigned to the user session).

- The `SET` clause causes the specified setting parameter to be set with a specified value when the function is entered; the same setting parameter value is reset back to its default value when the function exits.
- The parameter configuration setting can be set explicitly for the whole function or overwritten locally inside of the function, and it can inherit the value from the session setting using the `CURRENT` clause.

- Let's suppose that a developer would like to quickly fix the following statement, which uses an external merge sort method, without altering the `work_mem` session:

```
car_portal=# EXPLAIN (analyze, buffers) SELECT md5(random()::text) FROM generate_series(1, 1000000) order by 1;
```

```
                                                       QUERY PLAN

------------------------------------------------------------------------------------------------------------
---------
 Sort  (cost=69.83..72.33 rows=1000 width=32) (actual time=3895.917..4763.415 rows=1000000 loops=1)
   Sort Key: (md5((random())::text))
   Sort Method: external merge  Disk: 42096kB
   Buffers: shared hit=3, temp read=9422 written=9447
   ->  Function Scan on generate_series  (cost=0.00..20.00 rows=1000 width=32) (actual time=227.126..1317.507 rows=1000000
loops=1)
         Buffers: temp read=1709 written=1709
 Planning Time: 0.136 ms
 Execution Time: 4844.019 ms
(8 rows)
```

- The `SELECT` statement in the preceding example can be wrapped in a function, and the function can be assigned a specific `work_mem`, as follows:

```
car_portal=# CREATE OR REPLACE FUNCTION configuration_test () RETURNS TABLE(md5 text) AS $$
car_portal$#     SELECT md5(random()::text) FROM generate_series(1, 1000000) order by 1;
car_portal$# $$ LANGUAGE SQL
car_portal-# SET enable_seqscan FROM current
car_portal-# SET work_mem = '100MB';
CREATE FUNCTION
```

- Now, let's run the function to see the results of the `work_mem` setting effect:

```
car_portal=# EXPLAIN (ANALYZE ,BUFFERS) SELECT * FROM configuration_test();
```

```
                                                            QUERY PLAN

--------------------------------------------------------------------------------------------------------------------
-------
 Function Scan on configuration_test  (cost=0.25..10.25 rows=1000 width=32) (actual time=4625.267..4709.364 rows=1000000 lo
ops=1)
 Planning Time: 0.022 ms
 Execution Time: 4803.256 ms
(3 rows)
```

- When the function is entered, `work_mem` is assigned a value of `100MB`; this affects the execution plan, and the sorting is now done in the memory.
- The function's execution time is faster than the query.

- To confirm this result, let's change `work_mem` for the session, in order to compare the results, as follows:

```
car_portal=> set work_mem to '100MB';
SET
car_portal=>
```

```
car_portal=# EXPLAIN (analyze, buffers) SELECT md5(random()::text) FROM generate_series(1, 1000000) order by 1;
```

```
                                                          QUERY PLAN

--------------------------------------------------------------------------------------------------------------------------
---------
 Sort  (cost=69.83..72.33 rows=1000 width=32) (actual time=4406.384..4612.767 rows=1000000 loops=1)
   Sort Key: (md5((random())::text))
   Sort Method: quicksort  Memory: 101756kB
   ->  Function Scan on generate_series  (cost=0.00..20.00 rows=1000 width=32) (actual time=146.018..1121.472 rows=1000000
loops=1)
 Planning Time: 0.038 ms
 Execution Time: 4711.441 ms
(6 rows)
```