A close encounter with PostgreSQL Stored Procedures

Kaarel Moppel - PGUG.EE #4
Dec. 2018, Tallinn

Who?

Kaarel Moppel

Senior Database Consultant

km@cybertec.at

CYBERTEC
The PostgreSQL Database Company

PostgreSQL Database Services

Training

Consulting

High Availability

Development

CYBERTEC
The PostgreSQL Database Company

Cloud

Performance Tuning

24/7 Support

Setup

Replication

CYBERTEC
The PostgreSQL Database Company

What?

- General idea
- Supported languages
- PL/pgSQL features
- PL/pgSQL internals
- Use cases
- Tooling

General idea for Stored Procedures

- Input -> output
    - output can be scalars or any dataset
    - all PG data types supported
- Code stored on the server
- Interpreted during runtime
- Concepts “borrowed” from Oracle PL/SQL
    - FUNCTION vs PROCEDURE

A multitude of programming languages support procedures

- Out of the box 2 languages available
  - SQL
  - PL/PgSQL
- Additional bindings on board
  - “C”
  - PL/Python
  - PL/Perl
  - PL/Tcl
- BRING YOUR OWN

PL/pgSQL “Hello World” (1)

```
DO $$
BEGIN
  RAISE INFO ‘Hello world!’;
END;
$$;
---
INFO:  Hello world!
DO
```

PL/pgSQL “Hello World” (2)

```
CREATE FUNCTION hello_world(name text)
  RETURNS void AS $$
BEGIN
  RAISE INFO ‘Hello %!’, name;
END;
$$ LANGUAGE plpgsql;
---
SELECT hello_world(‘world’);
INFO:  Hello world!
 hello_world
-------------


(1 row)
```

PL/pgSQL “Hello World” (3)

```
CREATE FUNCTION hello_world_sql(name text)
  RETURNS text AS
$$
  SELECT ‘Hello ’  || name || ’!’;
$$ LANGUAGE sql;
---
SELECT hello_world_sql(‘world’);
 hello_world_sql
------------------------
 Hello world!
(1 row)
```

PL/pgSQL user-level features

- Anonymous functions
- Normal control / business logic
    - Nested function calls / PERFORM
- Subtransactions / capturing exceptions
- Looping over datasets / cursors / arrays
- TABLE functions
- Dynamic SQL with “EXECUTE”
- Privilege escalation
- Refcursors for returning many result-sets
- Runtime / error instrumentation available

PL/pgSQL implementation details (1)

- No packages (schemas can somewhat help)
- No package level vars / constants (consider dynamic session vars or tmp tbls*)
- Stored as plain text, thus better no secrets (*)
  - GRANT’s apply though
- Interpreted during runtime
- Cached per session

PL/pgSQL implementation details (2)

- Basically act as prepared statements
- Dollar notation $$
- Not too much validation
- Pre v11, always at least a single TX
    - new PROCEDURE / CALL syntax in v11
- Can be created in “pg_temp” for in-session re-use
    - Performs better than anonymous blocks

PL/pgSQL sample - variables

```
CREATE FUNCTION somefunc(p person_type) RETURNS void
AS $$
DECLARE
    count integer := 50;
    one_tbl_row table_name%ROWTYPE;
    my_data_object composite_type_name;
    cur refcursor;
    bounc_cur CURSOR FOR select * FROM mytbl;
    r record;
BEGIN
  RAISE INFO 'Hello %s!', p.name;
  NULL;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL sample - structure

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 50;
BEGIN
    /* 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;

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL sample - catching errors

```
CREATE FUNCTION merge(key INT, data TEXT) RETURNS VOID
AS $$
BEGIN
    BEGIN
      INSERT INTO db(a,b) VALUES (key, data);
      RETURN;
    EXCEPTION WHEN unique_violation THEN
      -- Do nothing or maybe “UPDATE”
      UPDATE db SET b = data WHERE a = key;
    END;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL sample - dynamic SQL (1)

```
CREATE OR REPLACE FUNCTION get_rowcounts()
    RETURNS void AS $$
DECLARE
  r record;
  c int;
BEGIN
  FOR r IN (select
          quote_ident(table_schema) || '.' || quote_ident(table_name) as tbl
        from information_schema.tables
        where table_type = 'BASE TABLE')
  LOOP
    EXECUTE format('select count(*) from %s', r.tbl) INTO c ;
    RAISE INFO '%: %', r.tbl, c;
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL sample - dynamic SQL (2)

```
CREATE FUNCTION generate_output(input_code)
  RETURNS TABLE(data text)
AS $$
BEGIN
  IF input_code = 'X' THEN
    RETURN QUERY SELECT data FROM tbl1;
  ELSE
    RETURN QUERY SELECT other_data FROM tbl2;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL sample - simple trigger

```
CREATE FUNCTION set_last_modified() RETURNS trigger AS
$$
    BEGIN
        NEW.last_modified_on := now();
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER last_modified BEFORE INSERT OR
UPDATE ON mytable
    FOR EACH ROW EXECUTE FUNCTION set_last_modified();
```

PL/pgSQL sample - event trigger

```
CREATE OR REPLACE FUNCTION abort_any_command()
  RETURNS event_trigger
 LANGUAGE plpgsql
  AS $$
BEGIN
  RAISE EXCEPTION 'command % is disabled', tg_tag;
END;
$$;

CREATE EVENT TRIGGER abort_ddl ON ddl_command_start
   EXECUTE FUNCTION abort_any_command();
```

PL/pgSQL sample - refcursors

```
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;
```

PL/pgSQL sample - v11 procedures

```
CREATE PROCEDURE transaction_test1()
AS $$
BEGIN
    FOR i IN 0..9 LOOP
        INSERT INTO test1 (a) VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
END
$$ LANGUAGE plpgsql ;
CALL transaction_test1();
```

PL/pgSQL sample - PL/Pythonu

```
CREATE TYPE public.load_average AS (
    load_1min real, load_5min real, load_15min real);

CREATE FUNCTION public.get_load_average() RETURNS
public.load_average AS
$$
from os import getloadavg
return getloadavg()
$$ LANGUAGE plpythonu VOLATILE SECURITY DEFINER;
```

PL/pgSQL valid use cases (1)

- Repetitive / multi-step actions
  - Deleting FK-guarded entries
- Error-prone operations
- Performance is needed
  - Automatic “prepared statements”
  - Setting planner constants
- Server maintenance with dynamic SQL

PL/pgSQL valid use cases (2)

- Triggers / Event Triggers
  - Performance? (*)
- Authoritative source of domain logic
  - Not really recommended though
- Data logic
  - Very much recommended!
- Scaling / sharding with PL/Proxy

Main benefits

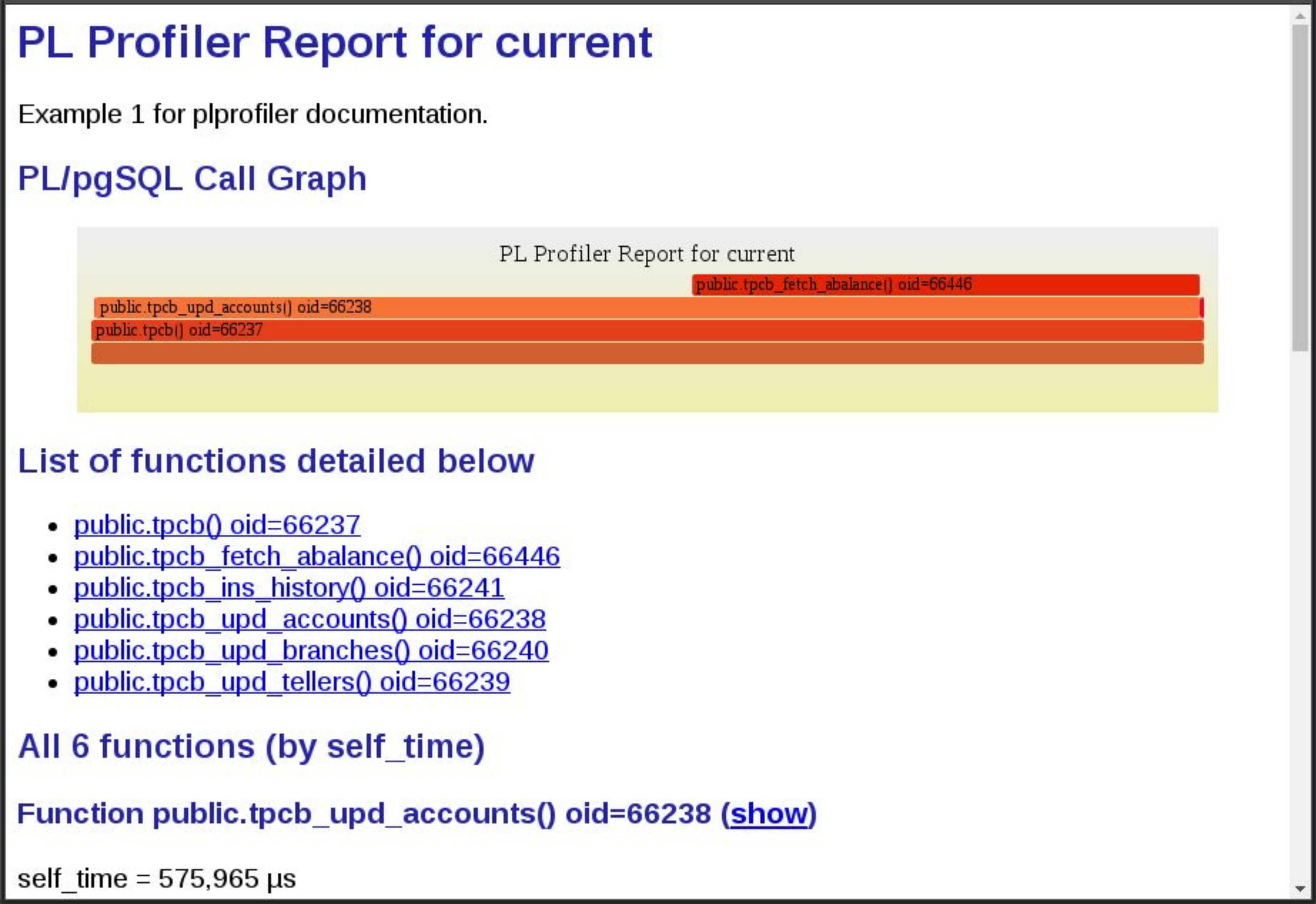
- Separation of concerns
  - i.e. another layer that can be transparently modified
- Performance
  - especially over latent/WAN networks
- Single point of “truth”
  - in case of many-many applications

Things to be aware of

- Set correct optimizer behaviour
  - VOLATILE is default
- Non-parallel by default
- Try not to use them in the WHERE part
- Enable monitoring with “track_functions=on”
- EXECUTE can be used to force re-planning
- Works great with custom TYPE-s for complex stuff

Tools

- Built-in config params for extra checks
- “audit-trigger” extension to help with triggers
- plpgsql_check - extra validations for your sprocs
- plprofiler - per line code coverage and performance info
- IDE-like debugging possible via pgadmin4/OmniDB (requires an extension)
- pgTAP - unit testing framework for Postgres
- “orafce” extension for less painful Oracle-migrations



PL Profiler Report for current

Example 1 for plprofiler documentation.

PL/pgSQL Call Graph

PL Profiler Report for current
public.tpcb_fetch_abalance() oid=66446
public.tpcb_upd_accounts() oid=66238
public.tpcb() oid=66237

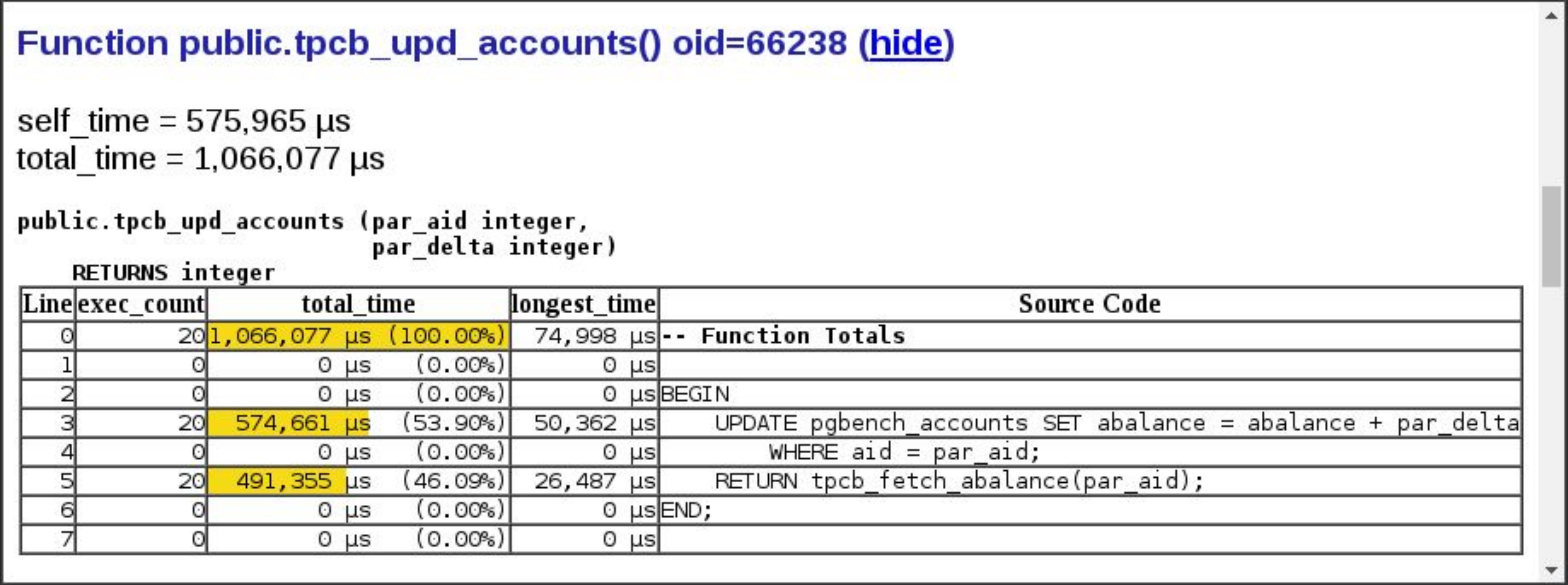
List of functions detailed below

- public.tpcb() oid=66237
- public.tpcb_fetch_abalance() oid=66446
- public.tpcb_ins_history() oid=66241
- public.tpcb_upd_accounts() oid=66238
- public.tpcb_upd_branches() oid=66240
- public.tpcb_upd_tellers() oid=66239

All 6 functions (by self_time)

Function public.tpcb_upd_accounts() oid=66238 (show)

self_time = 575,965 µs



Function public.tpcb_upd_accounts() oid=66238 (hide)

self_time = 575,965 µs
total_time = 1,066,077 µs

```
public.tpcb_upd_accounts (par_aid integer,
                          par_delta integer)
    RETURNS integer
```

Line	exec_count	total_time	longest_time	Source Code
0	20	1,066,077 µs (100.00%)	74,998 µs	-- Function Totals
1	0	0 µs (0.00%)	0 µs	
2	0	0 µs (0.00%)	0 µs	BEGIN
3	20	574,661 µs (53.90%)	50,362 µs	UPDATE pgbench_accounts SET abalance = abalance + par_delta
4	0	0 µs (0.00%)	0 µs	WHERE aid = par_aid;
5	20	491,355 µs (46.09%)	26,487 µs	RETURN tpcb_fetch_abalance(par_aid);
6	0	0 µs (0.00%)	0 µs	END;
7	0	0 µs (0.00%)	0 µs	

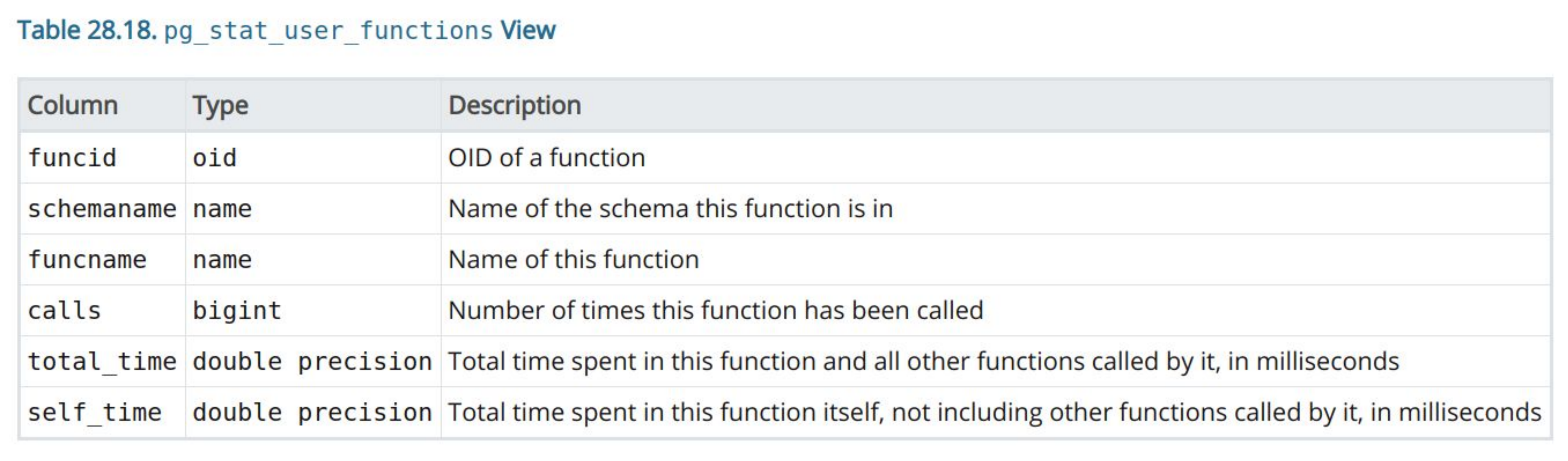


Table 28.18. pg_stat_user_functions View

Column	Type	Description
funcid	oid	OID of a function
schemaname	name	Name of the schema this function is in
funcname	name	Name of this function
calls	bigint	Number of times this function has been called
total_time	double precision	Total time spent in this function and all other functions called by it, in milliseconds
self_time	double precision	Total time spent in this function itself, not including other functions called by it, in milliseconds

Not to forget *pg_stat_statements* also!

kthxbye

Don’t be a stranger:

https://www.cybertec-postgresql.com/en/blog/