PostgresPro

PL/* Programming

Ivan Panchenko

06.05.2025

PGConf.Nepal 2025

A big zoo of Procedural Languages (PLs)

Supplied together with Postgres

- PL/PgSQL
- PL/TCL
- PL/Perl
- PL/Python

Supplied independently

- PL/V8
- PL/Java
- PL/R
- PL/rust
- PL/PHP
- PL/Go
- PL/Ruby
- PL/sh

Installing PLs

UBUNTU or Debian Linux

```
apt -y install postgresql-plperl-17
apt -y install postgresql-plpython3-17
```

PLv8: build from sources

```
apt install cmake git libstdc++-12-dev
## ensure you have a non-threaded V8
## see https://github.com/plv8/plv8/issues/571 for example
git clone https://github.com/plv8/plv8
cd plv8
make PG_CONFIG=/path/to/pgconfig -j4  EXECUTION_TIMEOUT=10
## make static for PLv8 version 2
```

Build PL/python or PL/perl from sources

While building PostgreSQL:

```
PYTHON=/usr/bin/python3 ./configure --with-perl --with-python
```

- **If you need python2 (PL/python) be prepared to build the extension yourself !**
  Python2 is not supported!

Creating PLs in your database

PostgreSQL:

```
CREATE EXTENSION plperl;      -- TRUSTED language
CREATE EXTENSION plperlu;     -- UNTRUSTED language
CREATE EXTENSION plpython3u;  -- Python 3 UNTRUSTED
CREATE EXTENSION plv8;        -- V8 only TRUSTED
```

What is TRUSTED and UNTRUSTED ?

Trusted vs Untrusted languages

TRUSTED	UNTRUSTED
• CREATE FUNCTION by any user • I/O forbidden • Network forbidden	• CREATE FUNCTION by superuser only • Everything allowed
PL/PgSQL PL/Perl PL/Java PL/v8	PL/PerlU PL/Python3U PL/JavaU

Other classification

Normal imperative programming languages

`PL/Python`
`PL/Perl`
`PL/v8`
`PL/java`
`PL/rust`

Domain-specific languages (DSL)

`PL/Proxy`

PL/PgSQL

C language

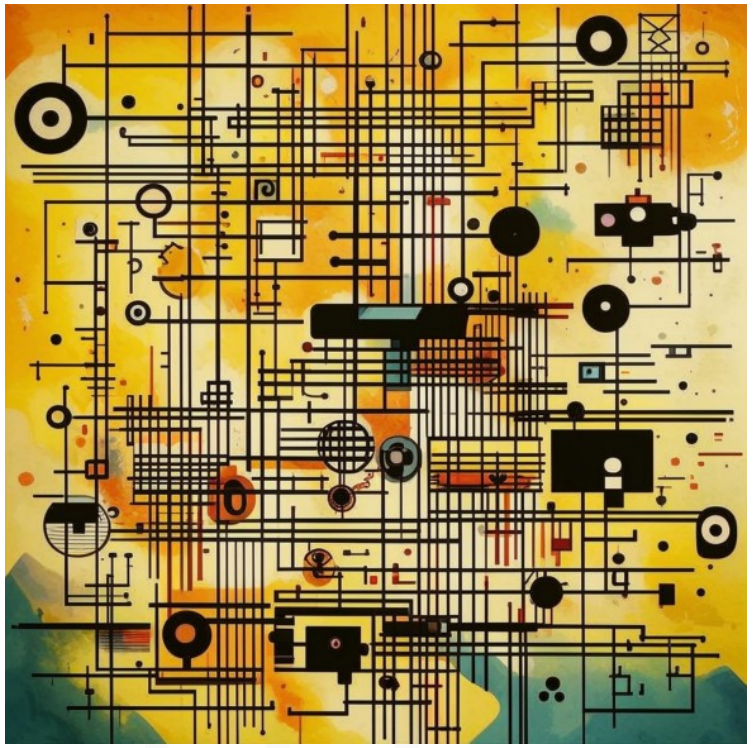
What about the standards?

SQL/PSM

- PL/PgPSM (dead)
- Best implementation is in IBM DB/2

SQL/JRT

- PL/Java



What if you want your own PL ?

```
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ]
    LANGUAGE name
    HANDLER      function_call_handler
    [ INLINE     inline_block_handler ]
    [ VALIDATOR validator_habdler    ]
```

There functions can be written in C only

Or any other language which you wrap in C

The reasons to use PL/* instead of PL/PgSQL

	PL/PgSQL	PL/*
Work with PostgreSQL datatypes	Native: good!	Needs conversion to the PL internal representation (may be expensive)
Global or session context	None, functions cannot remember their previous state	Global or session context to save (cache) data between function calls
Disk I/O and network	Only trusted language. No disk and network access	Untrusted languages can connect to other databases, URLs, send emails, write/read files, connect logging services etc.
Function calls Operations on complex (JSON) data structures Numeric calculations	Expensive	Fast

Common features of all PL/*

- Work with the database through the SPI interface wrapper
- Non-PostgreSQL memory management (avoid memory leaks yourself)
- Non-PostgreSQL data structures (good if more effective than PostgreSQL ones)
- Non-PostgreSQL error handling

PL/Perl features

- The interpreter is created on first usage in a session
- PL/Perl and PL/PerlU use different instances of the interpreter and cannot exchange data
- The configuration parameters:

```
plperl.on_init = 'use Data::Dumper;'
plperl.on_plperl_init = ' ... '
plperl.on_plperlu_init = ' ... '
plperl.use_strict = on
```

PL/Python features

- The interpreter is created on first usage in a session
- PL/Python2 not supported since PostgreSQL 15
- No initialization script!
- **SD** — session dictionary, **GD** — global dictionary
- Bad for one-liners
- Can define subtransactions
- Configured by environment variables PYTHON*

PL/v8 features

- Only trusted
- Initialization by a stored function:

```
plv8.start_proc=my_start_func //(initializer function name)
```

- Automatic JSON/JSONB mapping to a JavaScript data structure. Same with timestamps.
- Can define subtransactions
- Can define window functions
- Fast track calling PL/v8 functions from each other
- Execution timeout:

```
plv8.execution_timeout=300  // miliseconds
```

An interesting article on PLv8

PL/Java features

- Need to compile procedures
- **This allows to hide the source code**.
- Can use JAVA libraries
- Works with database through JDBC interface
- Do not forget to limit the JVM memory

Hello world PL/Perl

```
DO LANGUAGE plperl $$
    elog(NOTICE,"Hello World");
$$;
```

warn or **die** functions also can be used

Hello world PL/Python

```
DO LANGUAGE plpython3u $$
  plpy.notice('Hello World', hint='Hello hint', detail='Hello detail')
$$ ;
```

Also:
plpy.error(), plpy.warning(),
plpy.debug(), plpy.log(),
plpy.fatal().

Hello world PL/v8

```
DO LANGUAGE plv8 $$
  plv8.elog(NOTICE, 'Hello', ' World');
$$;
```

Possible to
`throw 'Error message';`
If an object is thrown, it will be converted to text like
`[object Object]`

Work with database: PL/Perl

```
DO LANGUAGE plperl $$
    warn Data::Dumper::Dumper(
          spi_exec_query('SELECT 57 AS x')
    )
$$;
```

```
WARNING: $VAR1 = {
  'status' => 'SPI_OK_SELECT',
  'processed' => 1,
  'rows' => [
    {'x' => '57'}
  ]
};
```

Work with database: PL/Python

```
DO LANGUAGE plpython3u $$
  plpy.notice(
    plpy.execute('SELECT 57 AS x')
  )
$$;
```

```
NOTICE:  <PLyResult status=5 nrows=1 rows=[{'x': 57}]>
```

Work with database: PL/Python

```
DO LANGUAGE plpython3u $$
  plpy.notice(
     plpy.execute('SELECT 57 AS x')[0]['x']
  )
$$;
```

```
NOTICE:  57
```

Work with database: PL/v8

```
DO LANGUAGE plv8 $$
 plv8.elog(
   NOTICE,
   JSON.stringify(
       plv8.execute('select 57 as x')
   )
 );
$$;
```

```
NOTICE:  [{"x":57}]
```

Escaping : PL/Perl

SPI functions:

`quote_literal` – takes into single quotes and duplicates quotes and backslashes

`quote_nullable` – same, but `undef` becomes `NULL`
`quote_ident` – takes identifiers into quotes, if necessary

```
DO LANGUAGE plperl $$
   warn quote_literal("macy's");
   warn quote_ident("TableName");
$$;
```

```
WARNING:  'macy''s' at line 2.
WARNING:  "TableName" at line 3.
```

Escaping : PL/Perl

Also defines functions:

```
encode_bytea()                  decode_bytea()
encode_array_literal()          encode_typed_literal()
encode_array_constructor()

DO LANGUAGE plperl $$
  warn encode_typed_literal(
     ["one", "twenty one"],
     "text[]"
  )
$$;

WARNING:  {one,"twenty one"} at line 2.
```

Escaping : PL/Python

SPI functions:

```
plpy.quote_literal()
plpy.quote_nullable()
plpy.quote_ident()
```

```
DO LANGUAGE plpython3u $$
  plpy.notice(
    plpy.quote_literal("Macy's")
  )
$$;

NOTICE:  'Macy''s'
```

Escaping : PL/v8

SPI functions:

```
plv8.quote_literal()
plv8.quote_nullable()
plv8.quote_ident()
```

```
DO LANGUAGE plv8 $$
  plv8.elog(NOTICE,
     plv8.quote_nullable("Macy's")
  );
$$;
```

```
NOTICE:  'Macy''s'
```

Performance 1.1 Single SQL query wrapped in PL

Perform a simple performance test

```
pgbench -c 1 -t 1000 -f test.sql
```

```
SELECT count(*) FROM pg_class;
```

11k tps

```
DO LANGUAGE plpgsql $$
  DECLARE a int;
  BEGIN
    SELECT count(*) INTO a FROM pg_class;
  END;
$$;
```

9.5k tps

```
DO LANGUAGE plperl $$
   my $x = spi_exec_query(q!SELECT count(*) FROM pg_class!);
$$;
```

6.4k tps

Performance 1.2

```
DO LANGUAGE plpython3u $$
 x = plpy.execute('SELECT count(*) FROM pg_class');
$$;
```

7k tps

```
DO LANGUAGE plv8 $$
  var x = plv8.execute('SELECT count(*) FROM pg_class');
$$;
```

7.6k tps

There languages seem slower…

But these all are variants of zero.

Performance 2.1 Repeat many times to avoid function call overhead

```
DO LANGUAGE plpgsql $$
  DECLARE a int;
          i int;
  BEGIN
    FOR i IN 0..999 LOOP
       SELECT count(*) INTO a FROM pg_class;
    END LOOP;
  END;
$$;
```

24 tps

```
DO LANGUAGE plperl $$
   for (0..999) {
       spi_exec_query('SELECT count(*) FROM pg_class');
   }
$$;
```

21 tps

Performance 2.2

```
DO LANGUAGE plpython3u $$
  for i in range (0,1000) :
      plpy.execute('SELECT count(*) FROM pg_class')
$$;
```

26 tps

```
DO LANGUAGE plv8 $$
  for(var i=0;i<1000;i++)
    plv8.execute('SELECT count(*) FROM pg_class');
$$;
```

25 tps

Performance 3.1 (same with prepared SQL)

```
DO LANGUAGE plperl $$
  my $h = spi_prepare( 'SELECT count(*) FROM pg_class');
  for (0..999) { spi_exec_prepared($h); }
  spi_freeplan($h);
$$;
```

34 tps

```
DO LANGUAGE plpython3u $$
  h = plpy.prepare('SELECT count(*) FROM pg_class')
  for i in range (0,1000) :
      plpy.execute(h)
$$;
```

34 tps

```
DO LANGUAGE plv8 $$
  var h=plv8.prepare('SELECT count(*) FROM pg_class');
  for(var i=0;i<1000;i++) h.execute();
$$;
```

35 ms

Here all PL's are faster than PL/PgSQL (see 2.1)

Performance 4.1 (calculation)

```
DO LANGUAGE plpgsql $$
  DECLARE i int;
          a bigint DEFAULT 0;
  BEGIN
    FOR i IN 0..1000000 LOOP
      a=a+i*i::bigint;
    END LOOP;
    RAISE NOTICE '%', a;
  END;
$$;
```

17 tps

Performance 4.2 (calculation)

```
DO LANGUAGE plperl $$
  my $a=0;
  for my $i (0..1000000) { $a+=$i*$i; };
  warn $a;
$$;
```

41 tps

```
DO LANGUAGE plpython3u $$
  a=0
  for i in range(0,1000001):
      a=a+i*i
  plpy.notice(a)
$$;
```

17 tps

Performance 4.3 (calculation)

```
DO LANGUAGE plv8 $$
  var a=0;
  for(var i=0;i<=1000000;i++)
     a+=i*i;
  plv8.elog(NOTICE, a);
$$;
```

67 tps

The fastest language for calculation? Mind the result! 333333833333127550

Exact result is $$\sum k^2 = \frac{n*(n+1)*(2n+1)}{6}$$ 333333833333500000

Javascript uses floats for numbers!

```
DO LANGUAGE plv8 $$
  var a=BigInt(0);
  for(var i=BigInt(0);i<=1000000;i++)
     a+=i*i;
  plv8.elog(NOTICE, a);
$$;
```

21 tps

No magic

Performance 5.1 (JSON operations)

```
DO LANGUAGE plpgsql $$
  DECLARE i int;
          a jsonb DEFAULT '{}';
  BEGIN
    FOR i IN 0..10000 LOOP
      a=jsonb_set(a, ARRAY['x' || i::text], to_jsonb(i));
    END LOOP;
  END;
$$;
```

0.29 tps
grows ~ n^2

```
DO LANGUAGE plperl $$
  my $a={};
  for my $i (0..10000) { $a->{"x$i"} =$i; }
$$;
```

400 tps
grows ~n

```
DO LANGUAGE plpython3u $$
  a={}
  for i in range(0,10001):
      a['x' + str(i)]=i
$$;
```

450 tps
grows ~n

Performance 5.2 (JSON operations)

```
DO LANGUAGE plv8 $$
  var a={};
  for(var i=0;i<=10000;i++) a['x'+ i] =i;
$$;
```

420 tps
grows ~n

V8 is fast, but consumes significantly more memory than PL/Perl or PL/Python!

Memory: PL/Perl

Good :

```
CREATE OR REPLACE FUNCTION cr()
  RETURNS int
  LANGUAGE plperl AS $$
     return spi_exec_query('SELECT count(*) FROM pg_class')
          ->{rows}->[0]->{count};
$$;
```

No memory leak

Bad:

```
CREATE OR REPLACE FUNCTION cr()
  RETURNS int
  LANGUAGE plperl AS $$
    my $h = spi_prepare('SELECT count(*) FROM pg_class');
    return spi_exec_prepared($h)->{rows}->[0]->{count};
$$;
```

Leaks memory

Fix:
 Use `spi_freeplan($h)`

No memory leak

Memory: PL/Python

Good :

```
CREATE OR REPLACE FUNCTION cr3()
  RETURNS int
  LANGUAGE plpython3u AS $$
     return plpy.execute('SELECT count(*) FROM pg_class')[0]['count']
$$;
```

No memory leak

```
CREATE OR REPLACE FUNCTION cr4()
  RETURNS int
  LANGUAGE plpython3u AS $$
  h = plpy.prepare('SELECT count(*) FROM pg_class')
  return plpy.execute (h)[0]['count']
$$;
```

No memory leak

Memory: PL/v8

Good :

```
CREATE OR REPLACE FUNCTION crq()
  RETURNS int
  LANGUAGE plv8 AS $$
    return plv8.execute('SELECT count(*) FROM pg_class')[0].count;
$$;
```

No memory leak

Bad :

```
CREATE OR REPLACE FUNCTION crq()
  RETURNS int
  LANGUAGE plv8 AS $$
    var h = plv8.prepare('SELECT count(*) FROM pg_class');
    return h.execute()[0].count;
$$;
```

Leaks memory

Fix:

Use `h.free();`

No memory leak

Function parameters: PL/Perl

How to work with parameters in PL/Perl function?

```
CREATE OR REPLACE FUNCTION crq(a int, b bytea, c int[], d jsonb )
   RETURNS void LANGUAGE plperl AS
$$ warn Data::Dumper::Dumper(@_) $$;

SELECT crq(1,'abcd', ARRAY[1,2,3],'{"a":2,"b":3}');
```

```
WARNING:
$VAR1 = '1';
$VAR2 = '\\x61626364';
$VAR3 = bless( {
                 'typeoid' => 1007,
                 'array' => [
                              '1',
                              '2',
                              '3'
                            ]
               }, 'PostgreSQL::InServer::ARRAY' );
$VAR4 = '{"a": 2, "b": 3}';
```

Function parameters: PL/Perl : Transform helps

```
CREATE TRANSFORM FOR jsonb LANGUAGE plperl (
  FROM SQL WITH FUNCTION jsonb_to_plperl(internal),
  TO   SQL WITH FUNCTION plperl_to_jsonb(internal)
);

CREATE OR REPLACE FUNCTION crq(a int, b bytea, c int[], d jsonb )
   RETURNS void LANGUAGE plperl
   TRANSFORM FOR TYPE jsonb
   AS
$$ warn Data::Dumper::Dumper(@_) $$;

SELECT crq(1,'abcd', ARRAY[1,2,3],'{"a":2,"b":3}');
```

```
WARNING: ....
$VAR4 = {
          'a' => '2',
          'b' => '3'
        };
```

So JSON is converted to Perl internal data structure

There is also a transform for `bool`

Function parameters: PL/Python

```
CREATE OR REPLACE FUNCTION pdump(a int, b bytea, c int[], d jsonb )
  RETURNS void
  LANGUAGE plpython3u
  AS
$$
  plpy.warning(a,b,c,d)
$$;

SELECT pdump(1,'abcd', ARRAY[1,2,3],'{"a":2,"b":3}');

WARNING:  (1, b'abcd', [1, 2, 3], '{"a": 2, "b": 3}')
```

Note:

The best for bytea and arrays

JSONB also needs a TRANSFORM

Function parameters: PL/v8

```
CREATE OR REPLACE FUNCTION jdump(a int, b bytea, c int[], d jsonb, e timestamptz )
  RETURNS void
  LANGUAGE plv8
  AS
$$
  plv8.elog(WARNING,a,b,c,d,e)
$$;

SELECT jdump(1,'abcd', ARRAY[1,2,3],'{"a":2,"b":3}', now());

WARNING:  1 97,98,99,100 1,2,3 [object Object], Sun May 04 2025 00:24:47 GMT+0300
(MSK)
```

Note:

- Bytea — character code array as array of numbers
- JSONB converts to JS object natively, TRANSFORM not needed.
- Timestamptz converts to JS Date object

PL/v8 initialization

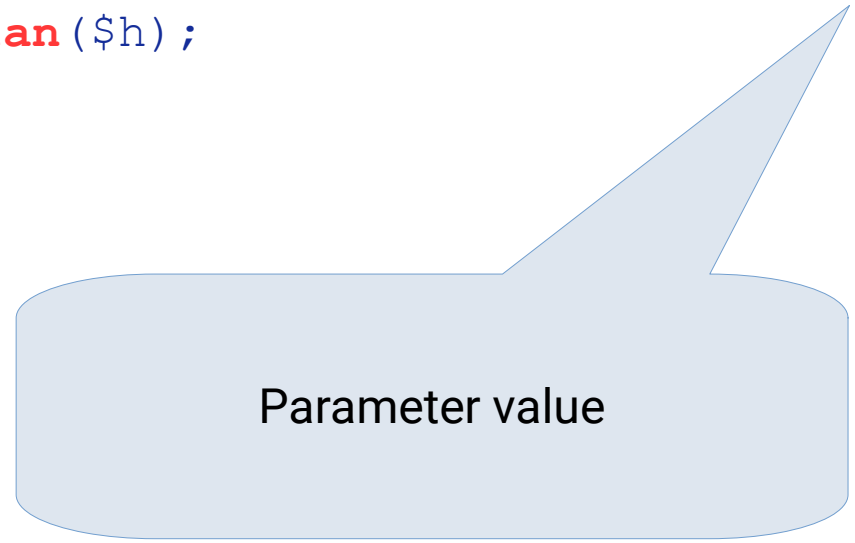
- A startup function is defined in a GUC plv8.start_proc

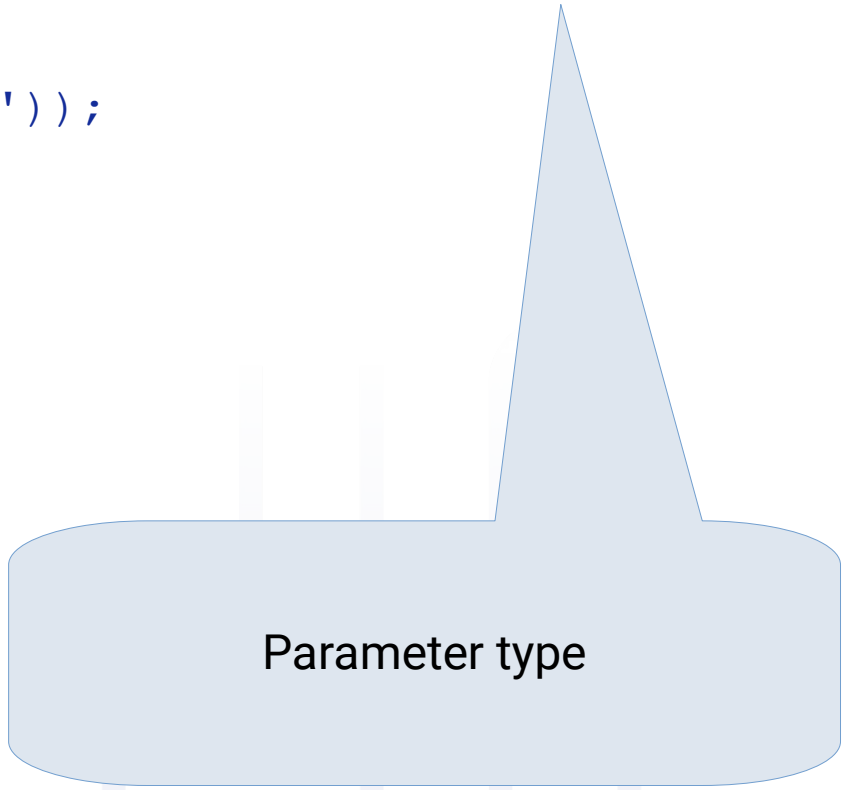
```
CREATE OR REPLACE FUNCTION my_init()
    RETURNS void
    LANGUAGE plv8 AS $$
      this.xxx = function() { return 'Something'; };
      this.qqq = 157;
$$;

SET plv8.start_proc = 'my_init';
```

SQL queries with parameters: PL/Perl

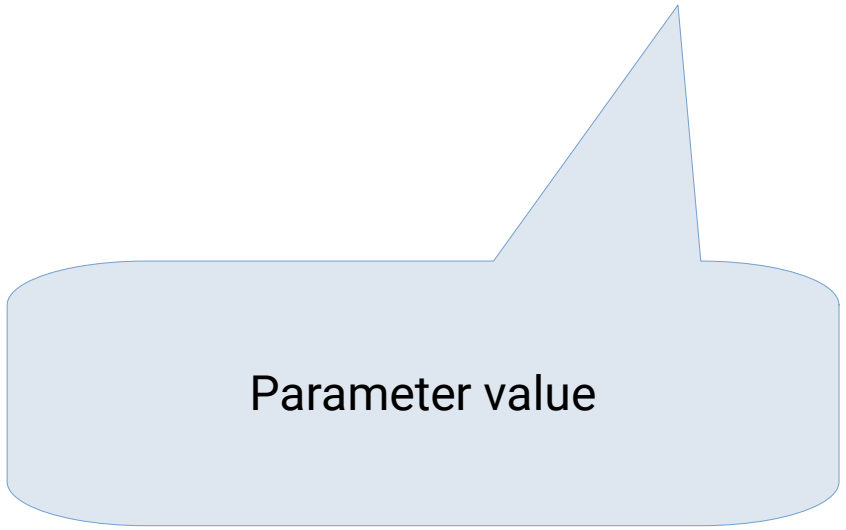
```
DO LANGUAGE plperl $$
  my $h= spi_prepare('SELECT * FROM pg_class WHERE relname ~ $1', 'text' );
  warn Dumper(spi_query_prepared($h, 'pg_class'));
  warn Dumper(spi_query_prepared($h, 'pg_napespace'));
  warn Dumper(spi_query_prepared($h, 'pg_proc'));
  spi_freeplan($h);
$$;
```

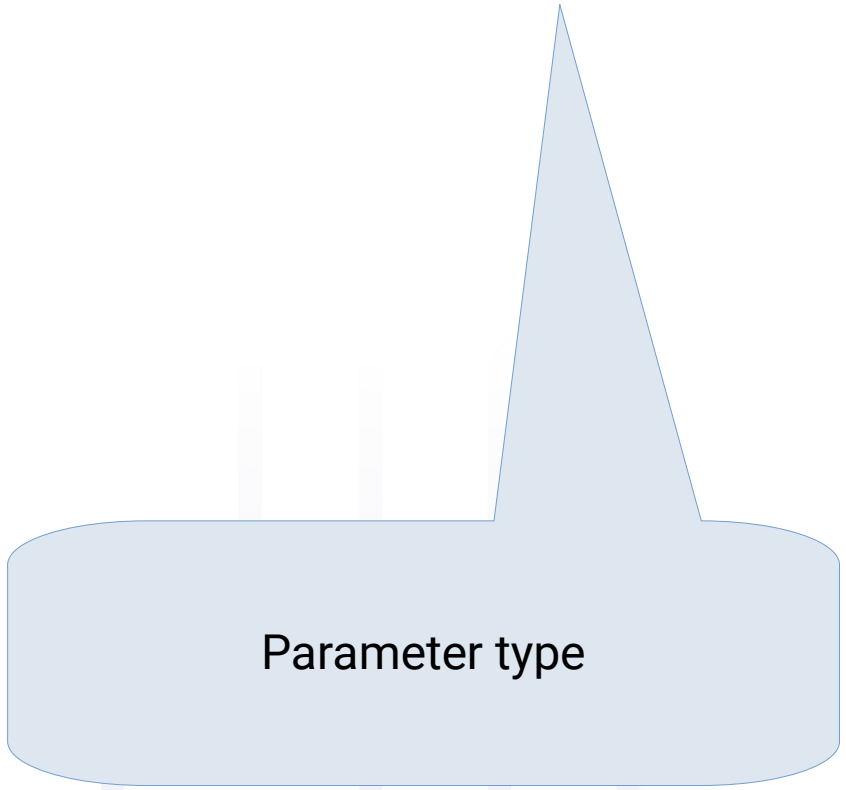




SQL queries with parameters: PL/Python

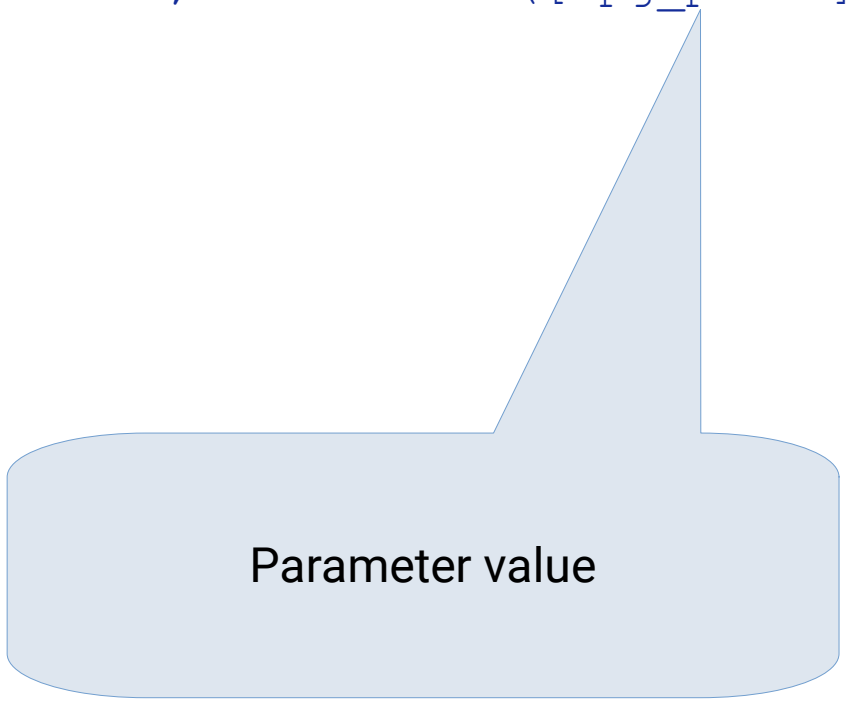
```
DO LANGUAGE plpython3u $$
  h = plpy.prepare('SELECT * FROM pg_class WHERE relname ~ $1', ['text'] )
  plpy.notice(plpy.execute (h, ['pg_class']))
  plpy.notice(plpy.execute (h, ['pg_namespace']))
  plpy.notice(plpy.execute (h, ['pg_function']))
$$;
```

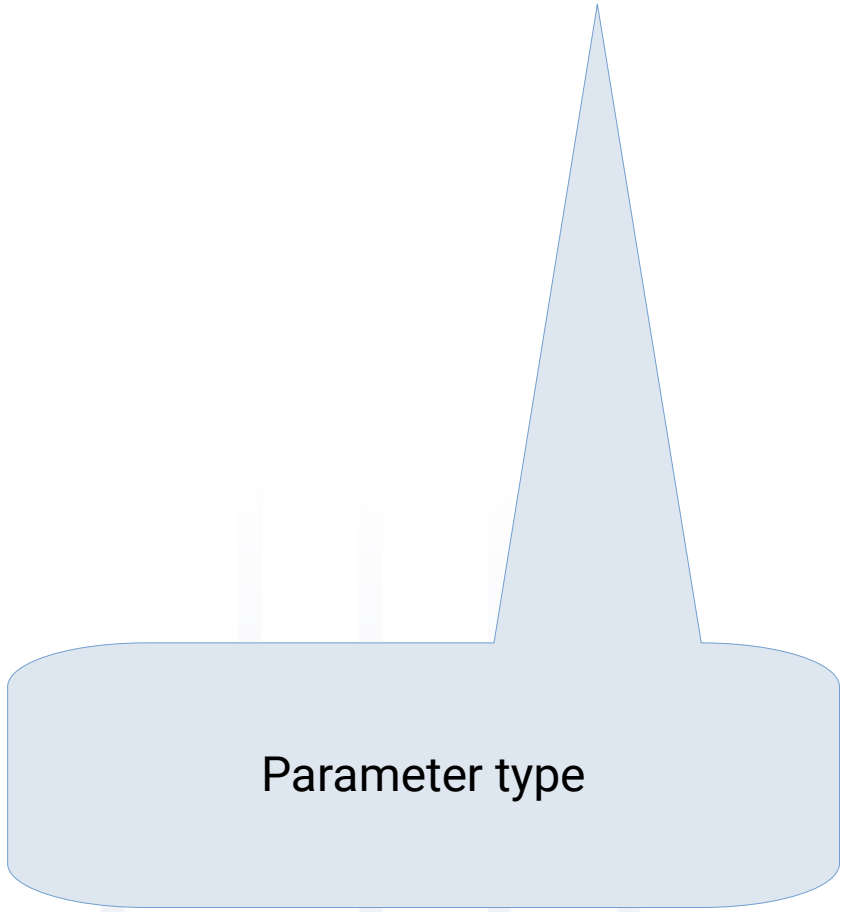




SQL queries with parameters: PL/v8

```
DO LANGUAGE plv8 $$
  var h= plv8.prepare('SELECT * FROM pg_class WHERE relname ~ $1', ['text'] );
  plv8.elog(NOTICE, h.execute (['pg_class']));
  plv8.elog(NOTICE, h.execute (['pg_namespace']));
  plv8.elog(NOTICE, h.execute (['pg_proc']));
  h.free();
$$;
```





PL/Perl work with SQL cursor

- A simple wrapper over SPI

```
DO LANGUAGE plperl $$
  my $cursor = spi_query('SELECT * FROM pg_class');
  my $row;
  while(defined($row = spi_fetchrow($cursor))) {
      warn $row->{relname};
  }
$$;
```

PL/Python work with SQL cursor

- A smart wrapper over SPI

```
DO LANGUAGE plpython3u $$
  h = plpy.prepare('SELECT * FROM pg_class');
  cursor = plpy.cursor(h);
  for row in cursor:
     plpy.warning(row)
$$;
```

PL/v8 work with SQL cursor

- The same simple wrapper over SPI

```
DO LANGUAGE plv8 $$
  var h = plv8.prepare('SELECT * FROM pg_class ');
   var cursor = h.cursor();
   var row;
   while(row = cursor.fetch()) {
       plv8.elog(NOTICE, row);
   }
   cursor.close();
$$;
```

PL/Perl: transaction control

- Transactions can be controlled from procedures, not functions

```
CREATE PROCEDURE ...
...
spi_commit();
...
spi_rollback();
...
```

PL/Python: transaction control

- Same SPI wrapper

```
CREATE PROCEDURE ...
...
plpy.commit();
...
plpy.rollback();
...
```

PL/Python: subtransactions

```
try:
    with plpy.subtransaction():
        plpy.execute("...")
        plpy.execute("...")
except plpy.SPIError, e:
...
else:
...
```

PL/v8: subtransactions

```
try {
    plv8.subtransaction(function() {
        plv8.execute('UPDATE...');
        plv8.execute('UPDATE...');
    });
}
catch(e) {
   ...
}
```

Conslusion. Where to use PL/* ?

- Where imperative programming style is more clear than declarative SQL.
- Work with complex data structures and algorithms.
- Dynamic formation of complex SQL queries (ORM, reporting)
- Using functions from the reach set of libraries of Perl, Python etc.
- Working with external data (file I/O, network I/O, external databases)
- Code prototyping before starting to develop in C

PostgresPro

The End.
Thanks for your attention