Content

- Function management
- Language handlers
- PL/pgSQL function life cycle
- PL/pgSQL architecture
  - expression parsing
  - expression evaluating
Function management

- described in `pg_proc`
- indirect call by `FunctionCallInvoke(fcinfo)`

typedef struct FmgrInfo
{
    PGFunction    fn_addr;    /* pointer to function or handler to be called */
    Oid           fn_oid;     /* OID of function (NOT of handler, if any) */
    short         fn_nargs;   /* 0..FUNC_MAX_ARGS, or -1 if variable arg */
    bool          fn_strict;  /* function is "strict" (NULL in => NULL out) */
    bool          fn_retset;  /* function returns a set */
    unsigned char fn_stats;  /* collect stats if track_functions > this */
    void          *fn_extra;  /* extra space for use by handler */
    MemoryContext fn_mcxt;    /* memory context to store fn_extra in */
    fmNodePtr     fn_expr;    /* expression parse tree for call, or NULL */
} FmgrInfo;

#define FunctionCallInvoke(fcinfo)      ((* (fcinfo)->flinfo->fn_addr) (fcinfo))
Language handlers

- **Call function**
  - execute code, translate arguments and result (from/to) PG types
- **Validator function**
  - validate record in pg_proc
- **Inline function**
  - execute string

```
postgres=# \h CREATE LANGUAGE
Command:    CREATE LANGUAGE
Description: define a new procedural language
Syntax:
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE name
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ INLINE inline_handler ] [ VALIDATOR valfunction ]
```
Predefined language handlers

postgres=# select * from pg_language ;

<table>
<thead>
<tr>
<th>lanname</th>
<th>lanowner</th>
<th>lanispl</th>
<th>lanpltrusted</th>
<th>lanplcallfoid</th>
<th>laninline</th>
<th>lanvalidator</th>
<th>lanacl</th>
</tr>
</thead>
<tbody>
<tr>
<td>internal</td>
<td>10</td>
<td>f</td>
<td>f</td>
<td>0</td>
<td>0</td>
<td>2246</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>10</td>
<td>f</td>
<td>f</td>
<td>0</td>
<td>0</td>
<td>2247</td>
<td></td>
</tr>
<tr>
<td>sql</td>
<td>10</td>
<td>f</td>
<td>t</td>
<td>0</td>
<td>0</td>
<td>2248</td>
<td></td>
</tr>
<tr>
<td>plpgsql</td>
<td>10</td>
<td>t</td>
<td>t</td>
<td>12654</td>
<td>12655</td>
<td>12656</td>
<td></td>
</tr>
</tbody>
</table>

(4 rows)

postgres=# \sf plpgsql_call_handler
CREATE OR REPLACE FUNCTION pg_catalog.plpgsql_call_handler()
RETURNS language_handler
LANGUAGE c
AS '$libdir/plpgsql', $function$plpgsql_call_handler$function$

postgres=# \sf plpgsql_validator
CREATE OR REPLACE FUNCTION pg_catalog.plpgsql_validator(oid)
RETURNS void
LANGUAGE c
STRICT
AS '$libdir/plpgsql', $function$plpgsql_validator$function$
Lifecycle registration

CREATE FUNCTION

pg_proc

VALIDATOR

COMPILATION

DEV/NULL
Lifecycle invocation

EXEC

CALL HANDLER

found in cache

EXECUTION

COMPILATION

PLPGSQL SESSION CACHE (code)
Notes about PL/pgSQL

- has no own expression unit
  - expressions are not cheap
- shares data types with PostgreSQL
- run in PostgreSQL session process
  - processing query result is not expensive
    - no interprocess communication
    - no data types conversions
- uses late I/O casting
  - expensive - when result is not of same type as target, then CAST based on IOfunc is invoked
PL/pgSQL architecture

- **PL/pgSQL parser/executor**
  - PL/pgSQL statements

- **SQL parser/executor**
  - Parsing embedded queries
  - Expressions

- **SPI interface**
  - Query execution
  - Cursors
CREATE OR REPLACE FUNCTION Hello(msg text)
RETURNS text AS $$
BEGIN
  RETURN 'Hello,' || msg;
END;
$$ LANGUAGE plpgsql IMMUTABLE;

/* from postgresql log */
Execution tree of successfully compiled PL/pgSQL function hello(text):

Function's data area:
  entry 0: VAR $1  type text (typoid 25) atttypmod -1
  entry 1: VAR found  type bool (typoid 16) atttypmod -1

Function's statements:
  3:BLOCK <<*unnamed*>>
    4: RETURN 'SELECT 'Hello,' || msg'
      END -- *unnamed*

End of execution tree of function hello(text)
CREATE OR REPLACE FUNCTION foo(a int)
RETURNS int AS $$
#option dump
DECLARE
  s int := 0;
  i int := 1;
BEGIN
  WHILE i <= a
  LOOP
    s := s + i;
    i := i + 1;
  END LOOP;
RETURN i;
END;
$$ LANGUAGE plpgsql IMMUTABLE;
"WHILE LOOP" dump

Execution tree of successfully compiled PL/pgSQL function foo(integer):

Function's data area:
  entry 0: VAR $1               type int4 (typoid 23) atttypmod -1
  entry 1: VAR found            type bool (typoid 16) atttypmod -1
  entry 2: VAR s                type int4 (typoid 23) atttypmod -1
      DEFAULT 'SELECT 0'
  entry 3: VAR i                type int4 (typoid 23) atttypmod -1
      DEFAULT 'SELECT 1'

Function's statements:
  6: BLOCK <<*unnamed*>>
  7:   WHILE 'SELECT i <= a'
  9:     ASSIGN var 2 := 'SELECT s + i'
 10:    ASSIGN var 3 := 'SELECT i + 1'
       ENDWHILE
12:   RETURN 'SELECT i'
     END -- *unnamed*

End of execution tree of function foo(integer)
PLpgSQL architecture

- it is glue for SQL statements
  - basic control structures
    - IF, WHILE, FOR, BEGIN
  - nested variables stack
    - assign statement, references to variables
- it is very simple interpret of abstract syntax tree
  - PL/pgSQL parser skips expressions
  - every node type has exec handler
PLpgsql architecture

- it is glue for SQL statements
  - basic control structures
    - IF, WHILE, FOR, BEGIN
  - nested variables stack
    - assign statement, references to variables
- it is very simple interpret of abstract syntax tree
  - every node type has exec handler
  - Execution ~ iteration over nodes (via handler invocations)
Good to know

- There is no compilation to byte code
- There is no JIT
- There is no any optimization
- Function is compiled to AST when it is first called in session – source code is in readable form in pg_proc (compilation is very simple and then relative fast)
- It is just relative simple, relative fast glue of SQL statements binary compatible with PostgreSQL (speed is comparable with other simple interprets)
  - Reduce network, protocol, ... overheads
  - Possible use faster interprets for different task (Perl)
WHILE statement

Syntax

<<label>> WHILE expression
LOOP
{statements}
END LOOP

Basic structure

typedef struct
{
    int cmd_type;
    int lineno;
    char *label;
    PLpgSQL_expr *cond;
    List *body;
} PLpgSQL_stmt_while;

/* WHILE cond LOOP statement */
/* List of statements */
WHILE statement

parser

class proc_stmt:
    pl_block (';')
    stmt_assign
    stmt_while

stmt_while:
    opt_block_label K_WHILE expr_until_loop loop_body
    PLpgSQL_stmt_while *new;
    new = palloc0(sizeof(PLpgSQL_stmt_while));
    new->cmd_type = PLPGSQL_STMT_WHILE;
    new->lineno = plpgsql_location_to_lineno(@2);
    new->label = $1;
    new->cond = $3;
    new->body = $4.stmts;
    check_labels($1, $4.end_label, $4.end_label_location);
    plpgsql_ns_pop();
    $$ = (PLpgSQL_stmt *)new;

loop_body:
    proc_sect K_END K_LOOP opt_label (';')
    $$.stmts = $1;
    $$.end_label = $4;
    $$.end_label_location = @4;
    $$;
WHILE statement parser

```
proc_stmt : pl_block ';'               { $$ = $1; }
           | stmt_assign   { $$ = $1; }
           | stmt_while    { $$ = $1; }
           ....

stmt_while : opt_block_label K_WHILE expr_until_loop loop_body
            { 
              PLpgSQL_stmt_while *new;
              new = palloc0(sizeof(PLpgSQL_stmt_while));
              new->cmd_type = PLPGSQL_STMT_WHILE;
              new->lineno   = plpgsql_location_to_lineno(@2);
              new->label        = $1;
              new->cond         = $3;
              new->body         = $4.stmts;
              check_labels($1, $4.end_label, $4.end_label_location);
              plpgsql_ns_pop();
              $$ = (PLpgSQL_stmt *)new;
            }

loop_body : proc_sect K_END K_LOOP opt_label ';'         { 
            $$.stmts = $1;
            $$.end_label = $4;
            $$.end_label_location = @4;
          }
```

WHILE statement
executor/main switch

```c
static int
exec_stmt(PLpgsql_execstate *estate, PLpgsql_stmt *stmt)
{
    PLpgsql_stmt *save_estmt;
    int rc = -1;

    save_estmt = estate->err_stmt;
    estate->err_stmt = stmt;

    /* Let the plugin know that we are about to execute this statement */
    if (*plugin_ptr && (*plugin_ptr)->stmt_beg)
        ((*plugin_ptr)->stmt_beg) (estate, stmt);

    CHECK_FOR_INTERRUPTS();

    switch (((enum PLpgsql_stmt_types) stmt->cmd_type))
    {
    case PLPGSQL_STMT_BLOCK:
        rc = exec_stmt_block(estate, (PLpgsql_stmt_block *) stmt);
        break;

    case PLPGSQL_STMT_ASSIGN:
        rc = exec_stmt_assign(estate, (PLpgsql_stmt_assign *) stmt);
        break;

    case PLPGSQL_STMT_PERFORM:
        rc = exec_stmt_perform(estate, (PLpgsql_stmt_perform *) stmt);
        break;

    case PLPGSQL_STMT_GETDIAG:
        rc = exec_stmt_getdiag(estate, (PLpgsql_stmt_getdiag *) stmt);
        break;
    }
```
WHILE statement node handler

```c
static int
exe_stmt_while(PLpgSQL_execstate *estate, PLpgSQL_stmt_while *stmt)
{
    for (;;)
    {
        int                     rc;
        bool            value;
        bool            isnull;

        value = exec_eval_boolean(estate, stmt->cond, &isnull);
        exec_eval_cleanup(estate);

        if (isnull || !value)
            break;

        rc = exec_stmts(estate, stmt->body);

        switch (rc)
        {
            case PLPGSQL_RC_OK:
                break;
            case PLPGSQL_RC_EXIT:
                return PLPGSQL_RC_OK;
            case PLPGSQL_RC_CONTINUE:
                break;
            case PLPGSQL_RC_RETURN:
                return rc;
            default:
                elog(ERROR, "unrecognized rc: %d", rc);
        }
    }
    return PLPGSQL_RC_OK;
}
```
Known issues

• Collisions of PL/pgSQL and SQL identifiers
  – solved in 9.0 (smart parameter placeholder positioning)

• Suboptimal plan for some queries
  – solved in 9.2 (prepared statements optimization)
  – in older version - using DYNAMIC SQL

• Late complete check of expressions
  – it is feature
    • + don't need to solve dependencies
    • - some errors are detected only in run-time
      – missing columns, wrong identifiers
Identifiers collisions

- dumb algorithm (8.4 and older)
  - use placeholder $n$ everywhere, where is varname in query string
    - collisions cannot be detected (strange errors)
    - positioning placeholder on wrong position (strange run-time errors)

```sql
CREATE OR REPLACE FUNCTION foo(a integer) 
RETURNS int AS $$
DECLARE x int;
BEGIN
  SELECT a FROM mytab WHERE mytab.a = a INTO x;
$$
```
Identifiers collisions

- smart algorithm (9.0 and higher)
  - callback functions from PostgreSQL parser
    - p_pre_columnref_hook
    - **p_post_columnref_hook**
      - called when PostgreSQL parser process column references - raise error or return placeholder node
    - p_paramref_hook

```c
/*
 * plpgsql_parser_setup         set up parser hooks for dynamic parameters
 *
 * Note: this routine, and the hook functions it prepares for, are logically
 * part of plpgsql parsing. But they actually run during function execution,
 * when we are ready to evaluate a SQL query or expression that has not
 * previously been parsed and planned.
 */
void
plpgsql_parser_setup(struct ParseState *pstate, PLpgSQL_expr *expr)
{
    pstate->p_pre_columnref_hook = plpgsql_pre_column_ref;
    pstate->p_post_columnref_hook = plpgsql_post_column_ref;
    pstate->p_paramref_hook = plpgsql_param_ref;
    /* no need to use p_coerce_param_hook */
    pstate->p_ref_hook_state = (void *) expr;
}
```
Dumb positioning

DECLARE
  a int;
  r record;
BEGIN
  SELECT x AS a, y AS b FROM tab INTO rec;

/* RESULT */
  SELECT x AS $1, y AS b FROM tab INTO rec; --- SYNTAX ERROR
Expression evaluation lifecycle

1. **EVALUATE EXPRESSION**
   - has plan?

2. **PREPARE PLAN**
   - TRY RUN AS SIMPLE EXPRESSION
     - returns result

3. **EVAL FULL QUERY**
   - has result?

4. **PLPGSQL SESSION CACHE (plan)**
   - simple expressions are evaluated significantly faster
     - result is scalar

5. **RETURNS RESULT**
static bool
exec_eval_boolean(PLpgSQL_execstate *estate, PLpgSQL_expr *expr, bool *isNull)
{
    Datum exprdatum; Oid exprtypeid;

    exprdatum = exec_eval_expr(estate, expr, isNull, &exprtypeid);
    exprdatum = exec_simple_cast_value(estate, exprdatum, exprtypeid, BOOLOID, -1, *isNull);
    return DatumGetBool(exprdatum);
}

static Datum
exec_cast_value(PLpgSQL_execstate *estate, Datum value, Oid valtype, Oid reqtype,
                FmgrInfo *reqinput, Oid reqtypioparam, int32 reqtypmod, bool isnull)
{
    /* If the type of the given value isn't what's requested, convert it. */
    if (valtype != reqtype || reqtypmod != -1)
    {
        MemoryContext oldcontext;

        oldcontext = MemoryContextSwitchTo(estate->eval_econtext->ecxt_per_tuple_memory);
        if (!isNull)
        {
            char *extval;

            extval = convert_value_to_string(estate, value, valtype);
            value = InputFunctionCall(reqinput, extval, reqtypioparam, reqtypmod);
        }
        else
        {
            value = InputFunctionCall(reqinput, NULL, reqtypioparam, reqtypmod);
        }
        MemoryContextSwitchTo(oldcontext);
    }
    return value;
}
PL/pgSQL expressions

- Reuse PostgreSQL parser/executor
- No redundant code
- Absolutely compatible with PostgreSQL
- Some operations are slow – array update
- Usually fast enough – bottle neck is in query processing – it is little bit slower than Python (1M iterations ~ 370 ms, Python ~ 256ms)
Late (IO) casting issue

- IO cast can be slow
- Possible lost of precision
- Different behave than SQL Casting
- It is not solved yet

```sql
postgres=# \sf test_assign
CREATE OR REPLACE FUNCTION public.test_assign()
    RETURNS void
    LANGUAGE plpgsql
AS $function$
declare x int;
BEGIN
    x := 9E3/2;
END
.firebaseapp$

postgres=# select test_assign();
ERROR:  invalid input syntax for integer: "4500.0000000000000000"
CONTEXT:  PL/pgSQL function test_assign() line 3 at assignment
```
Late (IO) casting issue

- IO cast can be slow
- Possible lost of precision
- Different behave than SQL Casting
Cached query plans

- Every query, every expression has an execution plan.
- Plans are stored in a session cache, created when the query is evaluated for the first time.
- Plans are dropped when the related relations are dropped.
Cached query plans

- Every query, every expression has a execution plan
- Plans are stored in session cache, created when query is evaluated first time
- Plans are dropped when related relations are dropped
- Plans are dropped when cost is significantly different for current parameters (9.2)
Cached plan issue (solved in 9.2)

- Index are used when should not be used
- Index are not used, but should be used

```sql
postgres=# \d omega
    Table "public.omega"
   Column | Type   | Modifiers
----------+---------+-----------
a       | integer |
Indexes:
   "omega_a_idx" btree (a)

postgres=# insert into omega select 1 from generate_series(1,1000000);
INSERT 0 10000
postgres=# insert into omega select 2 from generate_series(1,1000);
INSERT 0 10
```
Optimization based on heuristic (blind optimization)

postgres=# prepare x(int) as select count(*) from omega where a = $1;
PREPARE
postgres=# explain execute x(1);

QUERY PLAN

Aggregate (cost=17808.36..17808.37 rows=1 width=0)
  -> Index Scan using omega_a_idx on omega (cost=0.00..16545.86 rows=505000 width=0)
      Index Cond: (a = $1)

(3 rows)

postgres=# explain execute x(2);

QUERY PLAN

Aggregate (cost=17808.36..17808.37 rows=1 width=0)
  -> Index Scan using omega_a_idx on omega (cost=0.00..16545.86 rows=505000 width=0)
      Index Cond: (a = $1)

(3 rows)
postgres=# prepare x(int) as select count(*) from omega where a = $1;
PREPARE

postgres=# explain execute x(1);

QUERY PLAN

Aggregate (cost=19085.83..19085.84 rows=1 width=0)
  ->  Seq Scan on omega (cost=0.00..16586.00 rows=999934 width=0)
       Filter: (a = 1)
(3 rows)

postgres=# explain execute x(2);

QUERY PLAN

Aggregate (cost=318.73..318.74 rows=1 width=0)
  ->  Index Only Scan using omega_a_idx on omega (cost=0.00..293.57 rows=10066 width=0)
       Index Cond: (a = 2)
(3 rows)
Performance tips

• In 99% SQL and built-in function and functionality will be faster than your code

• FOR statement will be faster than WHILE
  – FOR IN int
  – FOR IN SELECT

• Minimalist code is usually faster

• PL/pgSQL is perfect language for data operations (based on SQL), and worst language for intensive mathematic op
Performance tips - examples

--bad
DECLARE v varchar;
BEGIN
  v := 'a';
  v := v || 'b';
  v := v || 'c';
  RETURN v;
END;

--good
BEGIN
  RETURN 'a' || 'b' || 'c';
END;

--bad
DECLARE s varchar := '';
BEGIN
  s := s || 'NULL,';
  IF x1 IS NULL THEN
      s := s || 'NULL,';
  ELSE
      s := s || x1;
  END IF;
  IF x2 IS NULL THEN
      s := s || 'NULL,';
  ELSE
      s := s || x2;
  END IF;
  ...

-- good
DECLARE s varchar;
BEGIN
  s := COALESCE(x1 || ',', 'NULL,')
    || COALESCE(x2 || ',', 'NULL,');
END;
Pavel Stěhule

- PostgreSQL lector, consultant
- Now in GoodData performance team
- Some patches to PostgreSQL (PL/pgSQL)
  - CONTINUE statement
  - EXECUTE **USING**
  - RETURN QUERY
  - CASE statement in PL/pgSQL
  - RAISE **USING** ..
  - VARIADIC FUNCTIONS, DEFAULT parameters
  - FOREACH IN ARRAY
  - GET STACKED DIAGNOSTICS