Introduction to PL/pgSQL
PostgreSQL allows user-defined functions to be written in a variety of procedural languages. The database server has no built-in knowledge about how to interpret the function’s source text. Instead, the task is passed to a handler that knows the details of that particular language.

PostgreSQL currently supports several standard procedural languages:

- PL/pgSQL
- PL/Tcl
- PL/Perl
- PL/Python
- PL/Java
- And many more
What is PL/pgSQL

- PL/pgSQL is the procedural extension to SQL with features of programming languages

- Data Manipulation and Query statements of SQL are included within procedural units of code

- Allows using general programming tools with SQL, for example: loops, conditions, functions, etc.

- This allows a lot more freedom than general SQL, and is lighter-weight than calling from a client program
How PL/pgSQL works

- PL/pgSQL is like every other “loadable, procedural language.”

- When a PL function is executed, the fmgr loads the language handler and calls it.

- The language handler then interprets the contents of the pg_proc entry for the function (proargtypes, prorettype, prosrc).
How PL/pgSQL works

• On the first call of a function in a session, the call handler will “compile” a function statement tree.

• SQL queries in the function are just kept as a string at this point.

• What might look to you like an expression is actually a SELECT query:

```sql
  my_variable := some_parameter * 100;
```
How PL/pgSQL works

- The PL/pgSQL statement tree is very similar to a PostgreSQL execution tree.
- The call handler then executes that statement tree.
- On the first execution of a statement node, that has an SQL query in it, that query is prepared via SPI.
- The prepared plan is then executed for every invocation of that statement in the current session.
The basic unit in any PL/pgSQL code is a BLOCK. All PL/pgSQL code is composed of a single block or blocks that occur either sequentially or nested within another block. There are two kinds of blocks:

- Anonymous blocks (DO)
  - Generally constructed dynamically and executed only once by the user. It is sort of a complex SQL statement
- Named blocks (Functions and Stored Procedures)
  - Have a name associated with them, are stored in the database, and can be executed repeatably, and can take in parameters
Structure of Anonymous Block

DO $$
[ <<label>> ]
DECLARE
/* Declare section (optional). */
BEGIN
/* Executable section (required). */
EXCEPTION
/* Exception handling section (optional). */
END [ label ]
$$;
Comments

- There are two types of comments in PL/pgSQL
  - `--` starts a comment that extends to the end of the line
  - `/*` multi-line comments `*/`

- Commenting is necessary to tell people what is intended and why it was done a specific way

- Err on the side of too much commenting
Variables

- Use variables for
  - Temporary storage of data
  - Manipulation of stored values
  - Re-usability
  - Ease of maintenance

- Declared in the declarative section within a block

```sql
v_last_name  VARCHAR(15);
```
Handling Variables

- Variables declared in the declarations section preceding a block are initialized to their default values every time the block is entered, not only once per function call.

- Variables in a declaration section can shadow variables of the same name in an outer block. If the outer block is named with a label, its variables are still available by specifying them as `<label>..<varname>`.
Declarations

Syntax

```
identifier [CONSTANT] datatype [NOT NULL] [:= | = | DEFAULT expr];
```

Examples

```
DECLARE
    v_birthday DATE;
    v_age INT NOT NULL = 21;
    v_name VARCHAR(15) := 'Homer';
    v_magic CONSTANT NUMERIC := 42;
    v_valid BOOLEAN DEFAULT TRUE;
```
• Declare variable according to:
  • A database column definition
  • Another previously declared variable

identifier table.column_name%TYPE;

Example

DECLARE
  v_email users.email%TYPE;
  v_my_email v_email%TYPE := 'rds-postgres-extensions-request@amazon.com';
• Declare a variable with the type of a ROW of a table

identifier table%ROWTYPE;

Example

```
DECLARE
    v_user    users%ROWTYPE;
```
• A record is a type of variable similar to ROWTYPE, but with no predefined structure

• The actual structure of the record is created when the variable is first assigned

• A record is not a true data type, only a place holder

```
DECLARE
    r record;
```
DO $$
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- 30
    quantity := 50;
    -- Create a subblock
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- 80
    END;
    RAISE NOTICE 'Quantity here is %', quantity;  -- 50
END
$$;
DO $$
<< mainblock >>
DECLARE
  quantity integer := 30;
BEGIN
  RAISE NOTICE 'Quantity here is %', quantity; --30
  quantity := 50;
  -- Create a subblock
  DECLARE
    quantity integer := 80;
  BEGIN
    RAISE NOTICE 'Quantity here is %', mainblock.quantity; --50
    RAISE NOTICE 'Quantity here is %', quantity; --80
  END;
  RAISE NOTICE 'Quantity here is %', quantity; --50
END
$$;
• Reports messages
  • Can be seen by the client if the appropriate level is used

RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
Assigning Values

- Use the assignment operator ( := or = )

```sql
DECLARE
    v_last_name VARCHAR := 'Smith';
    v_date DATE;
BEGIN
    v_last_name := lower(v_last_name);
    v_date := to_date('2000-01-01', 'YYYY-MM-DD');
```
SELECT in PL/pgSQL

- Retrieve data from the database with a SELECT statement
- Queries must return only one row
- INTO clause is required

DECLARE
    v_first_name users.first_name%TYPE;
    v_last_name users.last_name%TYPE;
BEGIN
    SELECT first_name, last_name
    INTO v_first_name, v_last_name
    FROM users
    WHERE user_id = 1;
END
DECLARE
    v_forum_name forums.name%TYPE := 'Hackers';
BEGIN
    INSERT INTO forums (name)
    VALUES (v_forum_name);

    UPDATE forums
    SET moderated = true
    WHERE name = v_forum_name;
END
PERFORM

- Evaluate an expression or query but discard the result
- Frequently used when executing maintenance commands

BEGIN
    PERFORM create_partition('moderation_log', '2016-06');
END
Structure of Named Blocks

CREATE FUNCTION [ function_name ] ()
RETURNS [return_type] $$
[ <<label>> ]
DECLARE
/* Declare section (optional). */
BEGIN
/* Executable section (required). */
EXCEPTION
/* Exception handling section (optional). */
END [ label ]
$$ LANGUAGE plpgsql;
CREATE FUNCTION get_user_count()
  RETURNS integer
AS $$
DECLARE
  v_count integer;
BEGIN
  SELECT count(*)
  INTO v_count
  FROM users;

  RETURN v_count;
END
$$ LANGUAGE plpgsql;
Dollar Quoting

- The tag $$ denotes the start and end of a string
- Optionally can have a non-empty tag as part of the quote
  - $$
  - $abc$
- Can be used to prevent unnecessary escape characters throughout the string

```
$function$
BEGIN
    RETURN ($1 ~ $q[\t\r\n\v\$]q$);
END;
$function$
```
Function Parameters

- One or more parameters can be used
- Parameter names are optional, but highly recommended

```sql
CREATE FUNCTION get_user_name(varchar, p_last_name varchar) RETURNS varchar AS $$
DECLARE
  v_first_name varchar;
  v_name varchar;
BEGIN
  v_first_name := $1;
  SELECT name INTO v_name FROM users
  WHERE first_name = v_first_name AND last_name = p_last_name
  LIMIT 1;

  RETURN v_name;
END$$ LANGUAGE plpgsql;
```
Default Parameters

- Parameters can have a default value
- This essentially makes them optional parameters

```
CREATE FUNCTION get_user_count(p_active boolean DEFAULT true)
  RETURNS integer AS $$
DECLARE
  v_count integer;
BEGIN
  SELECT count(*) INTO v_count
  FROM users
  WHERE active = p_active;

  RETURN v_count;
END$$
LANGUAGE plpgsql;
```
Assertions

- A convenient shorthand for inserting debugging checks
- Can be controlled by plpgsql.check_asserts variable

```sql
CREATE FUNCTION get_user_count(p_active boolean DEFAULT true)
RETURNS integer AS $$
DECLARE
  v_count integer;
BEGIN
  ASSERT p_active IS NOT NULL;

  SELECT count(*) INTO v_count
  FROM users
  WHERE active = p_active;

  RETURN v_count;
END
$$ LANGUAGE plpgsql;
```
PL/pgSQL Control Structures
Control the Flow

• The logical flow of statements can be changed using conditional IF statements and loop control structures
  • Conditional Structures
  • Loop Structures
IF Statements

**IF-THEN**

IF boolean-expression THEN
  statements
END IF;

**IF-THEN-ELSE**

IF boolean-expression THEN
  statements
ELSE
  statements
END IF;
Nested IF Statements

IF boolean-expression THEN
  IF boolean-expression THEN
    statements
  END IF;
ELSE
  statements
END IF;
ELSIF Statements

• A sequence of statements based on multiple conditions

```plaintext
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- the only other possibility is that number is null
    result := 'NULL';
END IF;
```
CASE Statements

- Used for complex conditionals
- Allows a variable to be tested for equality against a list of values

BEGIN

    CASE status
        WHEN 'Pending' THEN RAISE NOTICE 'PENDING';
        WHEN 'Accepted' THEN RAISE NOTICE 'ACCEPTED';
        WHEN 'Declined' THEN RAISE NOTICE 'DECLINED';
        WHEN 'Blocked' THEN RAISE NOTICE 'BLOCKED';
        ELSE RAISE NOTICE 'UNKNOWN';
    END CASE;

END
• Each WHEN clause sequentially evaluated until a TRUE is evaluated
• Subsequent WHEN expressions are not evaluated

BEGIN
  CASE
    WHEN x BETWEEN 0 AND 10 THEN
      RAISE NOTICE 'Value is between zero and ten';
    WHEN x BETWEEN 11 AND 20 THEN
      RAISE NOTICE 'Value is between eleven and twenty';
  END CASE;
$$;
FOUND

- FOUND, which is of type boolean, starts out false within each PL/pgSQL function call

- It is set by each of the following types of statements:
  - A SELECT INTO statement sets FOUND true if it returns a row, false if no row is returned
  - A PERFORM statement sets FOUND true if it produces (and discards) a row, false if no row is produced
  - UPDATE, INSERT, and DELETE statements set FOUND true if at least one row is affected, false if no row is affected
  - A FETCH statement sets FOUND true if it returns a row, false if no row is returned.
  - A FOR statement sets FOUND true if it iterates one or more times, else false.
DECLARE
    v_first_name  users.first_name%TYPE;
    v_last_name   users.last_name%TYPE;
BEGIN
    SELECT first_name, last_name
    INTO v_first_name, v_last_name
    FROM users
    WHERE user_id = 1;

    IF FOUND THEN
        RAISE NOTICE 'User Found';
    ELSE
        RAISE NOTICE 'User Not Found';
    END IF;
END
Loop Structures

- Unconstrained Loop
- WHILE Loop
- FOR Loop
- FOREACH Loop
Unconstrained Loops

- Allows execution of its statements at least once, even if the condition already met upon entering the loop

```
LOOP
  -- some computations
  IF count > 0 THEN
    EXIT;  -- exit loop
  END IF;
END LOOP;

LOOP
  -- some computations
  EXIT WHEN count > 0;  -- same result as previous example
END LOOP;
```
CONTINUE [ label ] [ WHEN expression ];

- If no label is given, the next iteration of the innermost loop is begun.
- If WHEN is specified, the next iteration of the loop is begun only if expression is true. Otherwise, control passes to the statement after CONTINUE.
- CONTINUE can be used with all types of loops; it is not limited to use with unconstrained loops.

LOOP
  -- some computations
  EXIT WHEN count > 100;
  CONTINUE WHEN count < 50;
  -- some computations for count IN [50 .. 100]
END LOOP;
WHILE Loops

WHILE condition LOOP
    statement1..;
END LOOP;

• Repeats a sequence of statements until the controlling condition is no longer TRUE
• Condition is evaluated at the beginning of each iteration

WHILE NOT done LOOP
    -- some computations here
END LOOP;
FOR Loops

FOR <loop_counter> IN [REVERSE] <low bound>..<high bound> LOOP
  -- some computations here
END LOOP;

- Use a FOR loop to shortcut the test for the number of iterations.
- Do not declare the counter; it is declared implicitly

DO $$
BEGIN
  FOR i IN 1..10 LOOP
    RAISE NOTICE 'value: %', i;
  END LOOP;
END $$;
Looping Over Results

- For loops can directly use a query result

DECLARE
    r record;
BEGIN
    FOR r IN SELECT email FROM users LOOP
        RAISE NOTICE 'Email: %', r.email;
    END LOOP;
END
Looping Over Results

- The last row is still accessible after exiting the loop

```sql
DECLARE
    r record;
BEGIN
    FOR r IN SELECT email FROM users LOOP
        RAISE NOTICE 'Email: %', r.email;
    END LOOP;
    RAISE NOTICE 'Email: %', r.email;
END
```
Looping Over Results

- Looping over dynamic SQL
- Re-planned each time it is executed

```sql
DECLARE
    rec RECORD;
    sql text;
BEGIN
    sql := 'SELECT email FROM users';
    FOR rec IN EXECUTE sql LOOP
        RAISE NOTICE 'Email: %', rec.email;
    END LOOP;
END
```
Looping Over Arrays

- Uses the FOREACH statement

```sql
DECLARE
    users     varchar[] := ARRAY['Mickey', 'Donald', 'Minnie'];
    v_user    varchar;
BEGIN
    FOREACH v_user IN ARRAY users LOOP
        RAISE NOTICE 'User: %', v_user;
    END LOOP;
END
```
Looping Over Arrays

- Use the SLICE syntax to iterate over multiple dimensions

```sql
DECLARE
    users   varchar[];
    v_dim   varchar[];
BEGIN
    users := ARRAY[ARRAY['Mickey', 'Donald'], ARRAY['Mouse', 'Duck']];
    FOREACH v_dim SLICE 1 IN ARRAY users LOOP
        RAISE NOTICE 'Dimension: %', v_dim;
    END LOOP;
END
```
Nested Loops

- Nest loops to multiple levels
- Use labels to distinguish between blocks
- Exit the outer loop with the EXIT statement that references the label

BEGIN

<<Outer_loop>>
LOOP
  v_counter := v_counter + 1;
  EXIT WHEN v_counter > 10; -- leaves both loops
<<Inner_loop>>
  LOOP
    EXIT Outer_loop WHEN total_done = 'YES';
    -- leaves both loops
    EXIT WHEN inner_done = 'YES';
    -- leaves inner loop only
  END LOOP Inner_loop;
END LOOP Outer_loop;
END

© 2019, Amazon Web Services, Inc. or its Affiliates. All rights reserved.
Dynamic SQL
Dynamic SQL

• A programming methodology for generating and running SQL statements at run time

• Useful for:
  • Ad-hoc query systems
  • DDL and database maintenance

EXECUTE command-string [ INTO target ] [ USING expression [, ... ] ];
Dynamic SQL - CAUTION

- There is no plan caching for commands executed via EXECUTE
  - The command is planned each time it is run

- Open to SQL injection attacks
  - All incoming parameters need to be validated
  - Bind the parameters to the command instead of generating the string
CREATE FUNCTION grant_select(p_table varchar, p_role varchar)
  RETURNS void AS
$$
DECLARE
  sql     varchar;
BEGIN
  sql := 'GRANT SELECT ON TABLE ' || p_table || ' TO ' || p_role;
  EXECUTE sql;
END
$$ LANGUAGE plpgsql;

Note: Do not do this. Validate the parameters first.
CREATE FUNCTION get_connection_count(p_role varchar)
    RETURNS integer
AS $$
DECLARE
    v_count integer;
    sql varchar;
BEGIN
    sql := 'SELECT count(*) FROM pg_stat_activity
            WHERE usename = ''' || p_role || '''';
    EXECUTE sql INTO v_count;
    RETURN v_count;
END
$$ LANGUAGE plpgsql;

Note: Do not do this. Validate the parameters first.
CREATE FUNCTION get_connection_count(p_role varchar)
  RETURNS integer
AS $$
DECLARE
  v_count integer;
  sql varchar;
BEGIN
  sql := 'SELECT count(*) FROM pg_stat_activity
  WHERE usename = $1';
  EXECUTE sql INTO v_count USING p_role;

  RETURN v_count;
END
$$ LANGUAGE plpgsql;
PL/pgSQL Cursors
Cursors

• Every SQL statement executed by PostgreSQL has an individual cursor associated with it
  • Implicit cursors: Declared for all DML and PL/pgSQL SELECT statements
  • Explicit cursors: Declared and named by the programmer

• Use CURSOR to individually process each row returned by a multiple-row SELECT Statement
Cursor Flow

1. DECLARE: Create a named SQL statement
2. OPEN: Identify the active result set
3. FETCH: Load the row into variables
4. EMPTY: Check if the result set is exhausted
   - Yes: CLOSE: Release the resources
   - No: Return to OPEN
Declaring Cursors

- A cursor must be declared as a variable
- Use the SCROLL keyword to move backwards through a cursor

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) FOR SELECT *
        FROM tenk1
        WHERE unique1 = key;
```
• The OPEN method to use is dependent on the way it was declared

```sql
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
OPEN cur2;
OPEN curs3(42);
OPEN curs3 (key := 42);
```
Fetching Data

- FETCH returns the next row
  ```sql
  FETCH curs2 INTO foo, bar, baz;
  ```

- FETCH can also move around the cursor
  ```sql
  FETCH LAST FROM curs3 INTO x, y;
  ```
CREATE FUNCTION grant_select(p_role varchar)
  RETURNS void AS $$
DECLARE
    sql      varchar;
    r        record;
    tbl_cursor CURSOR FOR SELECT schemaname, relname
                  FROM pg_stat_user_tables;
BEGIN
  OPEN tbl_cursor;
  LOOP
    FETCH tbl_cursor INTO r;
    EXIT WHEN NOT FOUND;
    sql := 'GRANT SELECT ON TABLE ' || r.schemaname || '
          . ' || r.relname || ' TO ' || p_role;
    EXECUTE sql;
  END LOOP;
  CLOSE tbl_cursor;
END$$ LANGUAGE plpgsql;
PL/pgSQL Returning Data
Returning Scalars

- Simplest return type

```sql
CREATE FUNCTION get_connection_count()
  RETURNS integer AS $$
DECLARE
  v_count integer;
BEGIN
  SELECT count(*) INTO v_count
  FROM pg_stat_activity;

  RETURN v_count;
END
$$
LANGUAGE plpgsql;

SELECT get_connection_count();
```

```
get_connection_count
----------------------
  
11
(1 row)
```
Some functions do not need a return value
- This is usually a maintenance function of some sort such as creating partitions or data purging
- Starting in PostgreSQL 11, Stored Procedures can be used in these cases

Return VOID

```
CREATE FUNCTION purge_log()
RETURNS void AS
$$
BEGIN
    DELETE FROM moderation_log
    WHERE log_date < now() - '90 days'::interval;
END
$$ LANGUAGE plpgsql;
```
Returning Sets

- Functions can return a result set
- Use SETOF
- Use RETURN NEXT
  - RETURN NEXT does not actually return from the function
  - Successive RETURN NEXT commands build a result set
- A final RETURN exits the function
CREATE FUNCTION fibonacci(num integer) RETURNS SETOF integer AS $$
DECLARE
  a int := 0;
  b int := 1;
BEGIN
  IF (num <= 0) THEN RETURN; END IF;

  RETURN NEXT a;
  LOOP
    EXIT WHEN num <= 1;
    RETURN NEXT b;
    num = num - 1;
    SELECT b, a + b INTO a, b;
  END LOOP;
END;
$$ language plpgsql;
Returning Records

- More complex structures can be returned

```sql
CREATE FUNCTION get_oldest_session()
    RETURNS record AS
$$
DECLARE
    r record;
BEGIN
    SELECT *
    INTO r
    FROM pg_stat_activity
    WHERE usename = SESSION_USER
    ORDER BY backend_start DESC
    LIMIT 1;

    RETURN r;
END
$$ LANGUAGE plpgsql;
```
• Using a generic record type requires the structure to be defined at run time

```sql
# SELECT * FROM get_oldest_session();
ERROR:  a column definition list is required for functions ...
LINE 1: SELECT * FROM get_oldest_session();

SELECT * FROM get_oldest_session()
AS (a oid, b name, c integer, d oid, e name, f text, g inet,
    h text, i integer, j timestamp_tz, k timestamp_tz,
    l timestamp_tz, m timestamp_tz, n boolean, o text, p xid,
    q xid, r text);
```
Returning Records

- All tables and views automatically have corresponding type definitions so they can be used as return types

```sql
CREATE FUNCTION get_oldest_session()
RETURNS pg_stat_activity AS $$
DECLARE
    r record;
BEGIN
    SELECT * INTO r
    FROM pg_stat_activity
    WHERE usename = SESSION_USER
    ORDER BY backend_start DESC
    LIMIT 1;

    RETURN r;
END
$$ LANGUAGE plpgsql;
```
Returning Sets of Records

- Many times, a subset of the table data is needed
- A view can be used to define the necessary structure

```sql
CREATE VIEW running_queries AS
SELECT CURRENT_TIMESTAMP - query_start as runtime, pid,
  usename, waiting, query
FROM pg_stat_activity
ORDER BY 1 DESC
LIMIT 10;
```
RETURN QUERY can be used to simplify the function

```sql
CREATE FUNCTION running_queries(p_rows int, p_len int DEFAULT 50) 
RETURNS SETOF running_queries AS $$
BEGIN
    RETURN QUERY SELECT runtime, pid, usename, waiting, 
       substring(query,1,p_len) as query 
       FROM running_queries 
       ORDER BY 1 DESC 
       LIMIT p_rows;
END $$ LANGUAGE plpgsql;
```
OUT Parameters

- Used to return structured information
- RETURNS is optional, but must be record if included

```sql
CREATE FUNCTION active_locks(OUT p_exclusive int, OUT p_share int)
```
CREATE FUNCTION active_locks(OUT p_exclusive int, OUT p_share int) AS $$
DECLARE
    r    record;
BEGIN
    p_exclusive := 0;
p_share := 0;
    FOR r IN SELECT l.mode
        FROM pg_locks l, pg_stat_activity a
        WHERE a.pid = l.pid
        AND a.usename = SESSION_USER
    LOOP
        IF r.mode = 'ExclusiveLock' THEN
            p_exclusive := p_exclusive + 1;
        ELSIF r.mode = 'ShareLock' THEN
            p_share := p_share + 1;
        END IF;
    END LOOP;
END
$$ LANGUAGE plpgsql;
OUT Parameters

• TIP: Think in sets not loops when writing functions for better performance
• NOTE: Use “OR REPLACE” when updating functions

CREATE OR REPLACE FUNCTION active_locks(OUT p_exclusive int, OUT p_share int)
AS $$
BEGIN

SELECT sum(CASE l.mode WHEN 'ExclusiveLock' THEN 1 ELSE 0 END),
       sum(CASE l.mode WHEN 'ShareLock' THEN 1 ELSE 0 END)
INTO p_exclusive, p_share
FROM pg_locks l, pg_stat_activity a
WHERE a.pid = l.pid
  AND a.usename = SESSION_USER;

END
$$ LANGUAGE plpgsql;
Structured Record Sets

- Use OUT parameters and SETOF record

```sql
CREATE FUNCTION all_active_locks(
    OUT p_lock_mode varchar,
    OUT p_count int
) RETURNS SETOF record AS $$
DECLARE
    r record;
BEGIN
    FOR r IN SELECT l.mode, count(*) as k
    FROM pg_locks l, pg_stat_activity a
    WHERE a.pid = l.pid
    AND a.usename = SESSION_USER
    GROUP BY 1
    LOOP
        p_lock_mode := r.mode;
        p_count := r.k;
        RETURN NEXT;
    END LOOP;
    RETURN;
$$;
```
Structured Record Sets

• Can return a TABLE

```
CREATE FUNCTION all_active_locks()
  RETURNS TABLE (p_lock_mode varchar, p_count int) AS $$
DECLARE
    r record;
BEGIN
  FOR r IN SELECT l.mode, count(*) as k
       FROM pg_locks l, pg_stat_activity a
       WHERE a.pid = l.pid
          AND a.usename = SESSION_USER
       GROUP BY 1
  LOOP
    p_lock_mode := r.mode;
    p_count := r.k;
    RETURN NEXT;
  END LOOP;
  RETURN;
END$$ LANGUAGE plpgsql;
```
Refcursors

- A cursor can be returned for large result sets
- The only way to return multiple result sets from a function

```
CREATE FUNCTION active_info(
    OUT p_queries refcursor,
    OUT p_locks refcursor)
```
CREATE FUNCTION active_info(OUT p_queries refcursor,
                           OUT p_locks refcursor)
   AS $$
BEGIN

   OPEN p_queries FOR SELECT runtime, pid, usename, waiting,
                        substring(query, 1, 50) as query
       FROM running_queries
       ORDER BY 1 DESC;

   OPEN p_locks FOR SELECT l.mode, count(*) as k
       FROM pg_locks l, pg_stat_activity a
       WHERE a.pid = l.pid
       AND a.usename = SESSION_USER
       GROUP BY 1;

END
$$ LANGUAGE plpgsql;
Handling Meta Information and Exceptions
Meta Information

- Information about the last command run inside of a function
- Several available values
  - ROW_COUNT
  - RESULT_OID
  - PG_CONTEXT

GET DIAGNOSTICS variable { = | := } item [ , ... ];
CREATE OR REPLACE FUNCTION purge_log()
    RETURNS void AS
$$
DECLARE
    l_rows int;
BEGIN
    DELETE FROM moderation_log
    WHERE log_date < now() - '90 days'::interval;

    GET DIAGNOSTICS l_rows = ROW_COUNT;
    RAISE NOTICE 'Deleted % rows from the log', l_rows;
END
$$ LANGUAGE plpgsql;
Exceptions

- An exception is an identifier in PL/pgSQL that is raised during execution.
- It is raised when an error occurs or explicitly by the function.
- It is either handled in the EXCEPTION block or propagated to the calling environment.

[DECLARE]
BEGIN
    Exception/Error is Raised
EXCEPTION
    Error is Trapped
END
Exceptions

- Use the WHEN block inside of the EXCEPTION block to catch specific cases

- Can use the error name or error code in the EXCEPTION block

```sql
WHEN division_by_zero THEN ... 
WHEN SQLSTATE '22012' THEN ...
```

- Use the special conditions OTHERS as a catch all

```sql
WHEN OTHERS THEN ...
```
### Sample Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>22000</td>
<td>data_exception</td>
</tr>
<tr>
<td>22012</td>
<td>division_by_zero</td>
</tr>
<tr>
<td>2200B</td>
<td>escape_character_conflict</td>
</tr>
<tr>
<td>22007</td>
<td>invalid_datetime_format</td>
</tr>
<tr>
<td>22023</td>
<td>invalid_parameter_value</td>
</tr>
<tr>
<td>2200M</td>
<td>invalid_xml_document</td>
</tr>
<tr>
<td>2200S</td>
<td>invalid_xml_comment</td>
</tr>
<tr>
<td>23P01</td>
<td>exclusion_violation</td>
</tr>
</tbody>
</table>
CREATE OR REPLACE FUNCTION get_connection_count()
    RETURNS integer AS $$
DECLARE
    v_count integer;
BEGIN
    SELECT count(*)
    INTO STRICT v_count
    FROM pg_stat_activity;
    RETURN v_count;
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        RAISE NOTICE 'More than 1 row returned';
        RETURN -1;
    WHEN OTHERS THEN
        RAISE NOTICE 'Unknown Error';
        RETURN -1;
END $$
LANGUAGE plpgsql;
Exception Information

- SQLSTATE Returns the numeric value for the error code.

- SQLERRM Returns the message associated with the error number.

```sql
DECLARE
    v_count integer;
    err_num integer;
    err_msg varchar;
BEGIN
    ...
EXCEPTION
    WHEN OTHERS THEN
        err_num := SQLSTATE;
        err_msg := SUBSTR(SQLERRM,1,100);
        RAISE NOTICE 'Trapped Error: %', err_msg;
        RETURN -1;
END
```
Exception Information

- The details of an error are usually required when handling
- Use GET STACKED DIAGNOSTICS to return the details

GET STACKED DIAGNOSTICS variable { = | := } item [ , ... ];
## Exception Information

<table>
<thead>
<tr>
<th>Diagnostic Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETURNED_SQLSTATE</td>
</tr>
<tr>
<td>COLUMN_NAME</td>
</tr>
<tr>
<td>CONSTRAINT_NAME</td>
</tr>
<tr>
<td>PG_DATATYPE_NAME</td>
</tr>
<tr>
<td>MESSAGE_TEXT</td>
</tr>
<tr>
<td>TABLE_NAME</td>
</tr>
<tr>
<td>SCHEMA_NAME</td>
</tr>
<tr>
<td>PG_EXCEPTION_DETAIL</td>
</tr>
<tr>
<td>PG_EXCEPTION_HINT</td>
</tr>
<tr>
<td>PG_EXCEPTION_CONTEXT</td>
</tr>
</tbody>
</table>
Exceptions can be raised explicitly by the function

```sql
CREATE OR REPLACE FUNCTION grant_select(p_role varchar)
RETURNS void AS
$$
DECLARE
    sql    varchar;
    r      record;
    tbl_cursor CURSOR FOR SELECT schemaname, relname
        FROM pg_stat_user_tables;
BEGIN
    IF NOT EXISTS (SELECT 1 FROM pg_roles
        WHERE rolname = p_role) THEN
        RAISE EXCEPTION 'Invalid Role: %', p_role;
    END IF;
    ...
```
Exceptions

- TIP: Use exceptions only when necessary, there is a large performance impact
- Sub transactions are created to handle the exceptions

```sql
CREATE FUNCTION t1()
    RETURNS void AS $$
DECLARE
    i integer;
BEGIN
    i := 1;
END
$$ LANGUAGE plpgsql;
```

Avg Time: 0.0017ms

```sql
CREATE FUNCTION t2()
    RETURNS void AS $$
DECLARE
    i integer;
BEGIN
    i := 1;
EXCEPTION
    WHEN OTHERS THEN
        RETURN;
END
$$ LANGUAGE plpgsql;
```

Avg Time: 0.0032ms
PL/pgSQL Triggers
Triggers

• Code that gets executed when an event happens in the database
  • INSERT, UPDATE, DELETE

• Event Triggers fire on DDL
  • CREATE, DROP, ALTER
Use Cases

- Table Partitioning before PostgreSQL 10
- Automatically generate derived column values
- Enforce complex constraints
- Enforce referential integrity across nodes in a distributed database
- Provide transparent event logging
- Provide auditing
  - Invalidate cache entries
Structure

- Unlike other databases, a trigger is broken into two pieces
  - Trigger
  - Trigger Function

```sql
CREATE TRIGGER name
{ BEFORE | AFTER | INSTEAD OF }
{ event [ OR ... ] }
ON table_name
[ FOR [ EACH ] { ROW | STATEMENT } ]
[ WHEN ( condition ) ]
EXECUTE PROCEDURE function_name ( arguments )
```
Trigger Function

A function with no parameters that returns TRIGGER

```
CREATE FUNCTION trg() RETURNS trigger AS $$
BEGIN
    RETURN NEW;
END;
$$
LANGUAGE plpgsql;
```
Trigger Events

- Insert
- Update
- Delete
- Truncate
Timing

• **Before**
  - The trigger is fired before the change is made to the table
  - Trigger can modify NEW values
  - Trigger can suppress the change altogether

• **After**
  - The trigger is fired after the change is made to the table
  - Trigger sees final result of row
Frequency

- **For Each Row**
  - The trigger is fired once each time a row is affected

- **For Each Statement**
  - The trigger is fired once each time a statement is executed
A firing trigger adds overhead to the calling transaction.

The percentage overhead can be found with a simple pgbench test:

```
INSERT INTO trigger_test (value) VALUES ('hello');
\set keys :scale
\setrandom key 1 :keys
UPDATE trigger_test SET value = 'HELLO' WHERE key = :key;
```
Trigger Overhead

```
pgbench -n -t 100000  
    -f INSERTS.pgbench postgres

pgbench -n -s 100000 -t 10000  
    -f UPDATES.pgbench postgres

Inserts: 4510 tps
Updates: 4349 tps
```
CREATE FUNCTION empty_trigger()
  RETURNS trigger AS $$
BEGIN
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER empty_trigger
  BEFORE INSERT OR UPDATE ON trigger_test
  FOR EACH ROW EXECUTE PROCEDURE empty_trigger();
Trigger Overhead

```
pgbench -n -t 100000 -f INSERTS.pgbench postgres
pgbench -n -s 100000 -t 10000 -f UPDATES.pgbench postgres

Inserts: 4296 tps (4.8% overhead)
Updates: 3988 tps (8.3% overhead)
```
Arguments

- **NEW**
  - Variable holding the new row for INSERT/UPDATE operations in row-level triggers

- **OLD**
  - Variable holding the old row for UPDATE/DELETE operations in row-level triggers
CREATE TABLE audit (  
    event_time timestamp NOT NULL,  
    user_name varchar NOT NULL,  
    old_row json,  
    new_row json  
);
CREATE OR REPLACE FUNCTION audit_trigger()
RETURNS TRIGGER AS $$
BEGIN

    INSERT INTO audit
    VALUES (CURRENT_TIMESTAMP,
            CURRENT_USER,
            row_to_json(OLD),
            row_to_json(NEW));

    RETURN NEW;
END;
$$
LANGUAGE plpgsql;
Arguments

- TG_OP
  - A string of INSERT, UPDATE, DELETE, or TRUNCATE telling for which operation the trigger was fired

- TG_NAME
  - Variable that contains the name of the trigger actually fired

- TG_WHEN
  - A string of BEFORE, AFTER, or INSTEAD OF, depending on the trigger’s definition

- TG_LEVEL
  - A string of either ROW or STATEMENT depending on the trigger’s definition
```
CREATE TABLE audit (
    event_time  timestamp NOT NULL,
    user_name   varchar NOT NULL,
    operation   varchar NOT NULL,
    old_row     json,
    new_row     json
);
```
CREATE OR REPLACE FUNCTION audit_trigger() RETURNS TRIGGER AS $$
BEGIN
  IF (TG_OP = 'DELETE') THEN
    INSERT INTO audit VALUES
    (CURRENT_TIMESTAMP, CURRENT_USER, TG_OP, row_to_json(OLD), null);
    RETURN OLD;
  ELSIF (TG_OP = 'UPDATE') THEN
    INSERT INTO audit VALUES
    (CURRENT_TIMESTAMP, CURRENT_USER, TG_OP, row_to_json(OLD), row_to_json(NEW));
    RETURN NEW;
  ELSIF (TG_OP = 'INSERT') THEN
    INSERT INTO audit VALUES
    (CURRENT_TIMESTAMP, CURRENT_USER, TG_OP, null, row_to_json(NEW));
    RETURN NEW;
  END IF;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
Arguments

- **TG_TABLE_NAME**
  - The name of the table that caused the trigger invocation.

- **TG_RELNAME**
  - The name of the table that caused the trigger invocation

- **TG_RELID**
  - The object ID of the table that caused the trigger invocation

- **TG_TABLE_SCHEMA**
  - The name of the schema of the table that caused the trigger invocation
CREATE TABLE audit (  
event_time timestamp NOT NULL,  
user_name varchar NOT NULL,  
operation varchar NOT NULL,  
table_name varchar NOT NULL,  
old_row json,  
new_row json
);
CREATE OR REPLACE FUNCTION audit_trigger() RETURNS TRIGGER AS $$
BEGIN
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO audit
        VALUES (CURRENT_TIMESTAMP, CURRENT_USER, TG_OP, TG_TABLE_NAME, row_to_json(OLD), null);
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO audit
        VALUES (CURRENT_TIMESTAMP, CURRENT_USER, TG_OP, TG_TABLE_NAME, row_to_json(OLD), row_to_json(NEW));
        RETURN NEW;
    ...
Arguments

- TG_NARGS
  - The number of arguments given to the trigger procedure in the CREATE TRIGGER statement

- TG_ARGV[]
  - The arguments from the CREATE TRIGGER statement
Trigger Use Cases

- **Table Partitioning**
  - Splitting what is logically one large table into smaller physical pieces

- **Used to:**
  - Increase performance
  - Archive data
  - Storage tiering
• Create child tables for each partition

```sql
CREATE TABLE audit_2014 (  
    CHECK ( event_time >= DATE '2014-01-01'  
            AND event_time < DATE '2015-01-01')  
) INHERITS (audit);

CREATE TABLE audit_2015 (  
    CHECK ( event_time >= DATE '2015-01-01'  
            AND event_time < DATE '2016-01-01')  
) INHERITS (audit);
```
The trigger function will move the row to the correct child table

```sql
CREATE OR REPLACE FUNCTION partition_audit_trigger()
    RETURNS TRIGGER AS $$
BEGIN
    EXECUTE 'INSERT INTO audit_ ' ||
              to_char(NEW.event_time, 'YYYY') ||
              ' VALUES ($1, $2, $3, $4, $5, $6)'
    USING NEW.event_time, NEW.user_name, NEW.operation,
         NEW.table_name, NEW.old_row, NEW.new_row;

    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```
A trigger needs to be added to the parent table

```
CREATE TRIGGER partition_audit_trigger
    BEFORE INSERT ON audit
    FOR EACH ROW
    EXECUTE PROCEDURE
        partition_audit_trigger();
```
Execution Performance

- Performance is much better if dynamic SQL is not used

```sql
CREATE OR REPLACE FUNCTION partition_audit_trigger()
RETURNS TRIGGER AS $$
BEGIN

    IF ( NEW.event_time >= DATE '2015-01-01' AND NEW.event_time < DATE '2016-01-01' ) THEN
        INSERT INTO audit_2015 VALUES (NEW.*);
    ELSIF ( NEW.event_time >= DATE '2014-01-01' AND NEW.event_time < DATE '2015-01-01' ) THEN
        INSERT INTO audit_2014 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date out of range. Fix partition_audit_trigger() function!';
    END IF;

    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```
If the column used for the partition key changes, the row may need to be moved to a different partition.

```sql
CREATE TRIGGER move_partition_audit_trigger
    BEFORE UPDATE
    ON audit_2014
    FOR EACH ROW EXECUTE PROCEDURE
    move_partition_audit_trigger('2014-01-01', '2015-01-01');

CREATE TRIGGER move_partition_audit_trigger
    BEFORE UPDATE
    ON audit_2015
    FOR EACH ROW EXECUTE PROCEDURE
    move_partition_audit_trigger('2015-01-01', '2016-01-01');
```
CREATE FUNCTION move_partition_audit_trigger() RETURNS TRIGGER AS $$
DECLARE
    start_date DATE;
    end_date DATE;
BEGIN
    start_date := TG_ARGV[0];
    end_date := TG_ARGV[1];

    IF ( NEW.event_time IS DISTINCT FROM OLD.event_time ) THEN
        IF (NEW.event_time < start_date OR NEW.event_time >= end_date) THEN
            EXECUTE 'DELETE FROM ' || TG_TABLE_SCHEMA || '.' || TG_TABLE_NAME || ' WHERE ctid = $1' USING OLD.ctid;
            INSERT INTO audit VALUES (NEW.*);
            RETURN null;
        END IF;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
Moving Partitions

- Only fire the trigger if the partition key changes

```
CREATE TRIGGER move_partition_audit_trigger
    BEFORE UPDATE
    ON audit_2014
    FOR EACH ROW
    WHEN (NEW.event_time IS DISTINCT FROM OLD.event_time)
    EXECUTE PROCEDURE move_partition_audit_trigger('2014-01-01', '2015-01-01');

CREATE TRIGGER move_partition_audit_trigger
    BEFORE UPDATE
    ON audit_2015
    FOR EACH ROW
    WHEN (NEW.event_time IS DISTINCT FROM OLD.event_time)
    EXECUTE PROCEDURE move_partition_audit_trigger('2015-01-01', '2016-01-01');
```
Trigger Use Cases

- Calculate columns
  - Calculate complex values
  - Extract values from complex structures
  - Enforce derived values when using denormalization
- Used to:
  - Increase performance
  - Simplify queries
Extract JSON

```
$ head -n 5 zips.json
{
  "_id" : "01001", "city" : "AGAWAM",
  "loc" : [ -72.622739, 42.070206 ], "pop" : 15338, "state" : "MA" }
{
  "_id" : "01002", "city" : "CUSHMAN",
  "loc" : [ -72.51564999999999, 42.377017 ], "pop" : 36963, "state" : "MA" }
{
  "_id" : "01005", "city" : "BARRE",
  "loc" : [ -72.10835400000001, 42.409698 ], "pop" : 4546, "state" : "MA" }
{
  "_id" : "01007", "city" : "BELCHERTOWN",
  "loc" : [ -72.41095300000001, 42.275103 ], "pop" : 10579, "state" : "MA" }
{
  "_id" : "01008", "city" : "BLANDFORD",
  "loc" : [ -72.936114, 42.182949 ], "pop" : 1240, "state" : "MA" }

CREATE TABLE zips (zip_code varchar PRIMARY KEY,
  state varchar,
  data json
);
```
CREATE OR REPLACE FUNCTION extract_data_trigger()
RETURNS TRIGGER AS $$
BEGIN
  NEW.zip_code := NEW.data->> '_id';
  NEW.state := NEW.data->> 'state';

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER extract_data_trigger
  BEFORE UPDATE OR INSERT ON zips
  FOR EACH ROW EXECUTE PROCEDURE extract_data_trigger();
Trigger Use Cases

- Cache invalidation
  - Remove stale entries from a cache
  - The database tracks all data so is the single source of truth
  - Used to:
    - Simplify cache management
    - Remove application complexity

Note: Foreign Data Wrappers simplify this process significantly

Note: ON (action) CASCADE contraints can simplify this too.
CREATE FUNCTION remove_cache_trigger()
  RETURNS TRIGGER AS $$
BEGIN
  DELETE from myredis_cache
  WHERE key = OLD.id::varchar;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER remove_cache_trigger
  AFTER UPDATE OR DELETE ON users
  FOR EACH ROW EXECUTE PROCEDURE remove_cache_trigger();
Cache Invalidation - Async

- The latency of updating the cache may not be an acceptable as part of the main transaction

```sql
CREATE FUNCTION remove_cache_trigger()
RETURNS TRIGGER AS $$
BEGIN
    PERFORM pg_notify(TG_TABLE_NAME, OLD.id::varchar);

    RETURN NEW;
END;
$$
LANGUAGE plpgsql;
```
Event Triggers

- Event triggers fire for DML commands (CREATE, ALTER, DROP, etc)

- They are not tied to a single table

- They are global to a database
CREATE OR REPLACE FUNCTION notice_ddl() RETURNS event_trigger AS $$
BEGIN
  RAISE NOTICE 'DDL Fired: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER notice_ddl
  ON ddl_command_start
  EXECUTE FUNCTION notice_ddl();
Event Trigger Events

- `ddl_command_start`
- `ddl_command_end`
- `table_rewrite`
- `sql_drop`
### ddl_command_start

- Fired just before the command starts
  - This is before any information is known about the command
- Fires for all event trigger command tags
- Does not fire for shared objects such as databases and roles
- Does not fire for commands involving event triggers
• Fired after the command ends

• Fires for all event trigger command tags

• The objects have been affected so the details of the command can be obtained
sql_drop

- Fired just before `ddl_command_end` fires
  - The objects have already been removed so they are not accessible

- Only fired for commands that drop an object

- The objects have been affected so the details of the command can be obtained
• Fired just before the table is rewritten by the command

• Only fired for commands that rewrites an object

• CLUSTER and VACUUM FULL do not fire the event
Event Tags

- ALTER POLICY
- ALTER SCHEMA
- ALTER SEQUENCE
- ALTER TABLE
- CREATE EXTENSION
- CREATE FUNCTION
- CREATE INDEX
- CREATE SEQUENCE

- CREATE TABLE
- CREATE TABLE AS
- CREATE VIEW
- DROP INDEX
- DROP TABLE
- DROP VIEW
- GRANT
- REVOKE

The full list is available in the documentation

https://www.postgresql.org/docs/current/event-trigger-matrix.html
Event Trigger Functions

A set of functions to help retrieve information from event triggers

- `pg_event_trigger_ddl_commands`
- `pg_event_trigger_dropped_objects`
- `pg_event_trigger_table_rewrite_oid`
- `pg_event_trigger_table_rewrite_reason`
Understanding `pg_event_trigger_ddl_commands`

- Returns a line for each DDL command executed
- Only valid inside a `ddl_command_end` trigger

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>classid</td>
<td>oid</td>
</tr>
<tr>
<td>objid</td>
<td>oid</td>
</tr>
<tr>
<td>objsubid</td>
<td>integer</td>
</tr>
<tr>
<td>command_tag</td>
<td>text</td>
</tr>
<tr>
<td>object_type</td>
<td>text</td>
</tr>
<tr>
<td>schema_name</td>
<td>text</td>
</tr>
<tr>
<td>object_identity</td>
<td>text</td>
</tr>
<tr>
<td>in_extension</td>
<td>bool</td>
</tr>
<tr>
<td>command</td>
<td><code>pg_ddl_command</code></td>
</tr>
</tbody>
</table>
**Understanding pg_event_trigger_dropped_objects**

- Returns a line for each object dropped by the DDL executed
- Only valid inside a sql_drop trigger

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Column</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>classid</td>
<td>oid</td>
<td>object_type</td>
<td>text</td>
</tr>
<tr>
<td>objid</td>
<td>oid</td>
<td>schema_name</td>
<td>text</td>
</tr>
<tr>
<td>objsubid</td>
<td>integer</td>
<td>object_name</td>
<td>text</td>
</tr>
<tr>
<td>original</td>
<td>bool</td>
<td>object_identity</td>
<td>text</td>
</tr>
<tr>
<td>normal</td>
<td>bool</td>
<td>address_names</td>
<td>text[]</td>
</tr>
<tr>
<td>is_temporary</td>
<td>bool</td>
<td>address_args</td>
<td>text[]</td>
</tr>
</tbody>
</table>
Understanding rewrite functions

- Only valid inside a `table_rewrite` trigger
  - `pg_event_trigger_table_rewrite_oid`
    - Returns the OID of the table about to be rewritten
  - `pg_event_trigger_table_rewrite_reason`
    - Returns the reason code of why the table was rewritten
CREATE OR REPLACE FUNCTION stop_drops()
    RETURNS event_trigger AS
$$
DECLARE
    l_tables varchar[] := '{sales, inventory}';
BEGIN
    IF EXISTS(SELECT 1
        FROM pg_event_trigger_dropped_objects()
        WHERE object_name = ANY (l_tables)) THEN
        RAISE EXCEPTION 'Drops of critical tables are not permitted';
    END IF;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER stop_drops
    ON sql_drop
    EXECUTE FUNCTION stop_drops();
Things to Remember

• Triggers are part of the parent transaction
  • The trigger fails, the main transaction fails
  • The main transaction rolls back, the trigger call never happened
  • If the trigger takes a long time, the whole transaction timing is affected

• Triggers can be difficult to debug
  • Especially cascaded triggers
PL/pgSQL Best Practices
Follow good programming practices
  • Indent code consistently
  • Comment code liberly
  • Code reuse/modularity practices are different than other programming languages
  • Deep call stacks in PL/pgSQL can be performance intensive
Create and follow a consistent naming convention for objects

- PostgreSQL is case insensitive so init cap does not work, use "_" to separate words in names
- Prefix all parameter names with something like "p_"
- Prefix all variable names with something like "v_"
Performance

- Avoid expensive constructs unless necessary
  - Dynamic SQL
  - EXCEPTION blocks