PL/pgSQL

PL/pgSQL (Procedural Language/PostgreSQL) is a loadable procedural programming language supported by the PostgreSQL. PL/pgSQL, as a fully featured programming language, allows much more procedural control than SQL, including the ability to use loops and other control structures. Functions created in the PL/pgSQL language can be called from an SQL statement, or as the action that a trigger performs.

Following lists show the features of PL/pgSQL:

- It is easy to use.
- Can be used to create functions and trigger procedures.
- Adds control structures to the SQL language.
- Can perform complex computations.
- Inherits all user-defined types, functions, and operators.
- Can be defined to be trusted by the server.

Advantages of Using PL/pgSQL

PostgreSQL and most other relational databases use SQL as a query language. In database server, every SQL statement executes individually, therefore after sending a query wait for it to be processed, receive and process the result, then send further queries to the server. With PL/pgSQL you can group a block of computation and a series of queries inside the database server, thus having the power of a procedural language and the ease of use of SQL, but with considerable savings of client/server communication overhead.

- Extra round trips between client and server are eliminated.
- Intermediate results that the client does not need do not have to be marshaled or transferred between server and client.
- Multiple rounds of query parsing can be avoided.

Structure of PL/pgSQL

PL/pgSQL is a block-structured language and each statement within a block is terminated by a semicolon. A block that appears within another block must have a semicolon after END, however, the final END that concludes a function body does not require a semicolon. All keywords are case-insensitive and identifiers are implicitly converted to lower case unless double-quoted, just as they are in ordinary SQL commands.See the following syntax:

Syntax:

```
[ <<label>> ]
[ DECLARE
    declarations ]
 BEGIN
    statements
 END;
```

A label (optional) is only needed if you want to identify the block for use in an EXIT statement, or to qualify the names of the variables declared in the block. If a label is given after END, it must match the label at the block's beginning. Comments work the same way in PL/pgSQL code as in ordinary SQL.

All variables must be declared in the declarations section of the block (marked in red color).

Version: 9.3

Syntax:

[<<label>>]
[DECLARE
declarations]
BEGIN
statements
END [label];

Note: The only exceptions are that the loop variable of a FOR loop iterating over a range of integer values is automatically declared as an integer variable.

The types of PL/pgSQL variables are similar to SQL data types, such as integer, varchar, and char.

Examples:

Examples:

roll_no integer;
qty numeric(5);
description varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;

Here is the general syntax of a variable declaration:

Syntax:

```
name [ CONSTANT ] type [ COLLATE collation_name ] [ NOT NULL ] [ { DEFAULT | := } expression ];
```

The DEFAULT clause, if given, specifies the initial value assigned to the variable when the block is entered. If the DEFAULT clause is not given then the variable is initialized to the SQL null value. A variable's default value is evaluated and assigned to the variable each time the block is entered

Variable Types

%TYPE is used to get the data type of a variable or table column. In the following example roll_no is a column in student table. To declare a variable with the same data type as student.roll_no you should write:

Syntax:

```
variable_name table_name.column_name%TYPE
```

Examples:

Code:

```
DECLARE
roll_no student.roll_no%TYPE;
```

Row Types

A variable of a composite type is called a row variable which can hold a whole row of a SELECT or FOR query result, so long as that query's column set matches the declared type of the variable.

name table_name%ROWTYPE;
name composite_type_name;
The individual fields of the row value are accessed using the usual dot notation, for example, rowvar.table_field. The fields of the row type inherit the table's field size or precision for data types such as char(n). See the following example:

24.2.2. Comments

There are two types of comments in PL/pgSQL. A double dash -- starts a comment that extends to the end of the line. A /* starts a block comment that extends to the next occurrence of */. Block comments cannot be nested, but double dash comments can be enclosed into a block comment and a double dash can hide the block comment delimiters /* and */.

24.2.3. Variables and Constants

All variables, rows and records used in a block or its sub-blocks must be declared in the declarations section of a block. The exception being the loop variable of a FOR loop iterating over a range of integer values.

PL/pgSQL variables can have any SQL datatype, such as INTEGER, VARCHAR and CHAR. All variables have as default value the SQL NULL value.

Here are some examples of variable declarations:

user_id INTEGER;
quantity NUMBER(5);

url VARCHAR;

24.2.3.1. Constants and Variables With Default Values

The declarations have the following syntax:

name [CONSTANT] ***type*** [NOT NULL] [{ DEFAULT | := } ***value***];
The value of variables declared as CONSTANT cannot be changed. If NOT NULL is specified, an assignment of a NULL value results in a runtime error. Since the default value of all variables is the SQL NULL value, all variables declared as NOT NULL must also have a default value specified.

The default value is evaluated every time the function is called. So assigning 'now' to a variable of type timestamp causes the variable to have the time of the actual function call, not when the function was precompiled into its bytecode.

Examples:

```
quantity INTEGER := 32;
url varchar := "http://mysite.com";
user_id CONSTANT INTEGER := 10;
```

Attributes

Using the %TYPE and %ROWTYPE attributes, you can declare variables with the same datatype or structure of another database item (e.g: a table field).

%TYPE

> %TYPE provides the datatype of a variable or database column. You can use this to declare variables that will hold database values. For example, let's say you have a column named user_id in your users table. To declare a variable with the same datatype as users you do:
>
> ```
> user_id users.user_id%TYPE;
> ```
>
> By using %TYPE you don't need to know the datatype of the structure you are referencing, and most important, if the datatype of the referenced item changes in the future (e.g: you change your table definition of user_id to become a REAL), you won't need to change your function definition.

name table%ROWTYPE;

> Declares a row with the structure of the given table. ***table*** must be an existing table or view name of the database. The fields of the row are accessed in the dot notation. Parameters to a function can be composite types (complete table rows). In that case, the

corresponding identifier $n will be a rowtype, but it must be aliased using the ALIAS command described above.

Only the user attributes of a table row are accessible in the row, no OID or other system attributes (because the row could be from a view). The fields of the rowtype inherit the table's field sizes or precision for char() etc. data types.

```
DECLARE
    users_rec users%ROWTYPE;
  user_id users%TYPE;
BEGIN
    user_id := users_rec.user_id;
    ...
```

Statements

Anything not understood by the PL/pgSQL parser as specified below will be put into a query and sent down to the database engine to execute. The resulting query should not return any data.

24.2.5.1. Assignment

An assignment of a value to a variable or row/record field is written as:

identifier := ***expression***;
If the expressions result data type doesn't match the variables data type, or the variable has a size/precision that is known (as for char(20)), the result value will be implicitly casted by the PL/pgSQL bytecode interpreter using the result types output- and the variables type input-functions. Note that this could potentially result in runtime errors generated by the types input functions.

```
user_id := 20;
tax := subtotal * 0.06;
```

24.2.5.2. Calling another function

All functions defined in a Postgres database return a value. Thus, the normal way to call a function is to execute a SELECT query or doing an assignment (resulting in a PL/pgSQL internal SELECT).

But there are cases where someone is not interested in the function's result. In these cases, use the PERFORM statement.

PERFORM ***query***
This executes a SELECT ***query*** over the SPI manager and discards the result. Identifiers like local variables are still substituted into parameters.

```
PERFORM create_mv("cs_session_page_requests_mv","
```

```
select   session_id, page_id, count(*) as n_hits,
        sum(dwell_time) as dwell_time, count(dwell_time) as dwell_count
from     cs_fact_table
group by session_id, page_id ");
```

Control Structures

Control structures are probably the most useful (and important) part of PL/SQL. With PL/pgSQL's control structures, you can manipulate PostgreSQL data in a very flexible and powerful way.

24.2.6.1. Conditional Control: IF statements

IF statements let you take action according to certain conditions. PL/pgSQL has three forms of IF: IF-THEN, IF-THEN-ELSE, IF-THEN-ELSE IF. NOTE: All PL/pgSQL IF statements need a corresponding END IF statement. In ELSE-IF statements you need two: one for the first IF and one for the second (ELSE IF).

IF-THEN

IF-THEN statements is the simplest form of an IF. The statements between THEN and END IF will be executed if the condition is true. Otherwise, the statements following END IF will be executed.

```
IF v_user_id <> 0 THEN
   UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

IF-THEN-ELSE

IF-THEN-ELSE statements adds to IF-THEN by letting you specify the statements that should be executed if the condition evaluates to FALSE.

```
IF parentid IS NULL or parentid = ""
THEN
   return fullname;
ELSE
   return hp_true_filename(parentid) || "/" || fullname;
END IF;
```

```
IF v_count > 0 THEN
   INSERT INTO users_count(count) VALUES(v_count);
   return "t";
ELSE
   return "f";
END IF;
```

IF statements can be nested and in the following example:

```
IF demo_row.sex = "m" THEN
  pretty_sex := "man";
ELSE
  IF demo_row.sex = "f" THEN
    pretty_sex := "woman";
  END IF;
END IF;
```

IF-THEN-ELSE IF

When you use the "ELSE IF" statement, you are actually nesting an IF statement inside the ELSE statement. Thus you need one END IF statement for each nested IF and one for the parent IF-ELSE.

For example:

```
IF demo_row.sex = "m" THEN
   pretty_sex := "man";
ELSE IF demo_row.sex = "f" THEN
       pretty_sex := "woman";
    END IF;
END IF;
```

24.2.6.2. Iterative Control: LOOP, WHILE, FOR and EXIT

With the LOOP, WHILE, FOR and EXIT statements, you can control the flow of execution of your PL/pgSQL program iteratively.

LOOP

[<<label>>]
LOOP
  statements
END LOOP;

An unconditional loop that must be terminated explicitly by an EXIT statement. The optional label can be used by EXIT statements of nested loops to specify which level of nesting should be terminated.

EXIT

EXIT [***label***] [WHEN ***expression***];

If no ***label*** is given, the innermost loop is terminated and the statement following END LOOP is executed next. If ***label*** is given, it must be the label of the current or an upper level of nested loop blocks. Then the named loop or block is terminated and control continues with the statement after the loops/blocks corresponding END.

Examples:

```
LOOP
```

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

LOOP
    -- some computations
    EXIT WHEN count > 0;
END LOOP;

BEGIN
    -- some computations
    IF stocks > 100000 THEN
       EXIT;  -- illegal. Can't use EXIT outside of a LOOP
    END IF;
END;
```

WHILE

With the WHILE statement, you can loop through a sequence of statements as long as the evaluation of the condition expression is true.

```
[<<label>>]
WHILE expression LOOP
   statements
END LOOP;
```

For example:

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
   -- some computations here
END LOOP;

WHILE NOT boolean_expression LOOP
   -- some computations here
END LOOP;
```

FOR

```
[<<label>>]
FOR name IN [ REVERSE ] expression .. expression LOOP
   statements
END LOOP;
```

A loop that iterates over a range of integer values. The variable ***name*** is automatically created as type integer and exists only inside the loop. The two expressions giving the lower and upper bound of the range are evaluated only when entering the loop. The iteration step is always 1.

Some examples of FOR loops (see **Section 24.2.7** for iterating over records in FOR loops):

```
FOR i IN 1..10 LOOP
  -- some expressions here

    RAISE NOTICE 'i is %',i;
END LOOP;

FOR i IN REVERSE 1..10 LOOP
    -- some expressions here
END LOOP;
```

24.2.7. Working with RECORDs

Records are similar to rowtypes, but they have no predefined structure. They are used in selections and FOR loops to hold one actual database row from a SELECT operation.

24.2.7.1. Declaration

One variables of type RECORD can be used for different selections. Accessing a record or an attempt to assign a value to a record field when there is no actual row in it results in a runtime error. They can be declared like this:

name RECORD;

What is the PostgreSQL Function?

A PostgreSQL function or a **stored procedure** is a set of SQL and procedural commands such as **declarations, assignments, loops, flow-of-control** etc. stored on the database server and can be involved using the **SQL interface**. And it is also known as **PostgreSQL stored procedures**.

We can create PostgreSQL functions in serval languages, for example, SQL, PL/pgSQL, C, Python etc.

It enables us to perform operations, which would generally take various commands and round trips in a function within the database.

What is the PostgreSQL CREATE Function command?

In PostgreSQL, if we want to specify a new user-defined function, we can use the **CREATE FUNCTION** command.

Syntax of PostgreSQL CREATE Function command

The Syntax for **PostgreSQL CREATE Function command** is as follows:

```
1. CREATE [OR REPLACE] FUNCTION function_name (arguments)
2. RETURNS return_datatype
3. LANGUAGE plpgsql
4. AS $variable_name$
5. DECLARE
6. declaration;
7. [...] -- variable declaration
8.  BEGIN
9. < function_body >
10. [...]  -- logic
11. RETURN { variable_name | value }
12. END;
```

Parameters	Description
function_name	○ The **function name** parameter is used to define the function name. ○ The function name is written after the **CREATE FUNCTION** keyword.
[OR REPLACE]	○ We can use the **OR REPLACE** keyword if we want to change the current function. ○ And it is an optional parameter.
Function	○ After using the **OR REPLACE** keyword, we can define the **function** parameter list that are covered in the parentheses after the **Function Name.** ○ And a function can contain zero or several parameters.
RETURN	○ We can define the data type after the **RETURN** keyword, which we are going to return from the function. ○ It can be a base, composite, or domain type or reference of the type of a table column.

Language plpgsql	○ It is used to define the name of the Procedural language in which the function is executed. ○ And not just **plpgsql,** the PostgreSQL supports various procedural languages.
Function_body	○ The **function_body** contains the executable parts of the logics.

Trigger Procedures

PL/pgSQL can be used to define trigger procedures. They are created with the usual **CREATE FUNCTION** command as a function with no arguments and a return type of OPAQUE.

There are some Postgres specific details in functions used as trigger procedures.

First they have some special variables created automatically in the top-level blocks declaration section. They are

NEW

Data type RECORD; variable holding the new database row on INSERT/UPDATE operations on ROW level triggers.

OLD

Data type RECORD; variable holding the old database row on UPDATE/DELETE operations on ROW level triggers.

TG_NAME

Data type name; variable that contains the name of the trigger actually fired.

TG_WHEN

Data type text; a string of either BEFORE or AFTER depending on the triggers definition.

TG_LEVEL

Data type text; a string of either ROW or STATEMENT depending on the triggers definition.

TG_OP

Data type text; a string of INSERT, UPDATE or DELETE telling for which operation the trigger is actually fired.

TG_RELID

Data type oid; the object ID of the table that caused the trigger invocation.

TG_RELNAME

Data type name; the name of the table that caused the trigger invocation.

TG_NARGS

Data type integer; the number of arguments given to the trigger procedure in the **CREATE TRIGGER** statement.

TG_ARGV[]

Data type array of text; the arguments from the **CREATE TRIGGER** statement. The index counts from 0 and can be given as an expression. Invalid indices (< 0 or >= tg_nargs) result in a NULL value.

Second they must return either NULL or a record/row containing exactly the structure of the table the trigger was fired for. Triggers fired AFTER might always return a NULL value with no effect. Triggers fired BEFORE signal the trigger manager to skip the operation for this actual row when returning NULL. Otherwise, the returned record/row replaces the inserted/updated row in the operation. It is possible to replace single values directly in NEW and return that or to build a complete new record/row to return.

Example 24-1. A PL/pgSQL Trigger Procedure Example

This trigger ensures, that any time a row is inserted or updated in the table, the current user name and time are stamped into the row. And it ensures that an employees name is given and that the salary is a positive value.

CREATE TABLE emp (

```
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp () RETURNS OPAQUE AS '
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname ISNULL THEN
            RAISE EXCEPTION "empname cannot be NULL value";
        END IF;
        IF NEW.salary ISNULL THEN
            RAISE EXCEPTION "% cannot have NULL salary", NEW.empname;
        END IF;

        -- Who works for us when she must pay for?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION "% cannot have a negative salary", NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := "now";
        NEW.last_user := current_user;
        RETURN NEW;
    END;
' LANGUAGE 'plpgsql';

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

PL/pgSQL Cursor

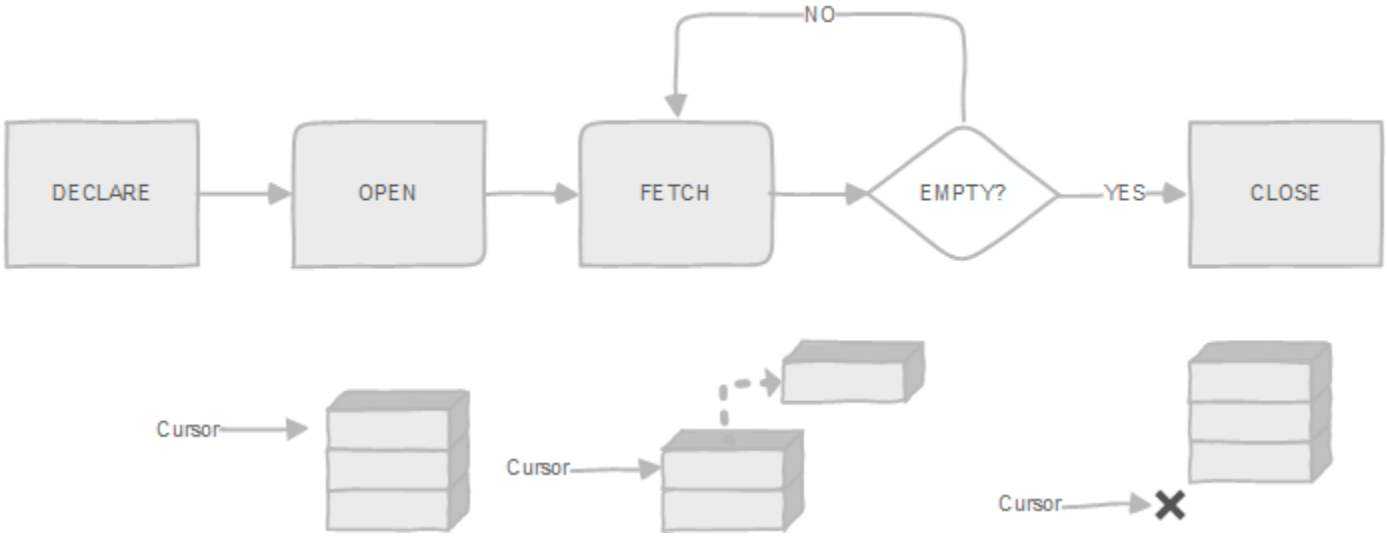
Summary: in this tutorial, you will learn about the PL/pgSQL Cursors and how to use them to process a result set, row by row.

Introduction to PL/pgSQL Cursor

In PostgreSQL, a cursor is a database object that allows you to traverse the result set of a query one row at a time.

Cursors can be useful when you deal with large result sets or when you need to process rows sequentially.

The following diagram illustrates how to use a cursor in PostgreSQL:



1. First, declare a cursor.
2. Next, open the cursor.
3. Then, fetch rows from the result set into a record or a variable list.
4. After that, process the fetched row and exit the loop if there is no more row to fetch.
5. Finally, close the cursor.

We will examine each step in more detail in the following sections.

Step 1. Declaring a cursor

To declare a cursor, you use the DECLARE statement. Here's the syntax for declaring a cursor:

```
DECLARE cursor_name CURSOR FOR query;
```
Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

In this syntax:

- First, specify the name of the cursor (cursor_name) after the DECLARE keyword.

- Second, provide a query that defines the result set of the cursor.

Step 2. Opening the cursor

After declaring a cursor, you need to open it using the OPEN statement:

```
OPEN cursor_name;Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)
```

Step 3. Fetching rows from the cursor

Once the cursor is open, you can fetch rows from it using the FETCH statement. PostgreSQL offers different ways to fetch rows:

- FETCH NEXT: fetches the next row from the cursor.
- FETCH PRIOR: fetches the previous row from the cursor.
- FETCH FIRST: fetches the first row from the cursor.
- FETCH LAST: fetches the last row from the cursor.
- FETCH ALL: fetches all rows from the cursor.

In practice, you often use the FETCH NEXT that fetches the next row from a cursor:

```
FETCH NEXT FROM cursor_name INTO variable_list;Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)
```

In this syntax:

- cursor_name specifies the name of the cursor.
- variable_list: is a comma-separated list of variables that store the values fetched from the cursor. It also can be a record.

Step 4. Processing rows

After fetching a row, you can process it. Typically, you use a LOOP statement to process the rows fetched from the cursor:

```
LOOP
   -- Fetch the next row
   FETCH NEXT FROM cursor_name INTO variable_list;
```

```
    -- exit if not found
    EXIT WHEN NOT FOUND;

    -- Process the fetched row
    ...

END LOOP;
```
Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

Step 5. Closing the cursor

Once completing fetching rows, you need to close the cursor using the CLOSE statement:

```
CLOSE cursor_name;
```
Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

The CLOSE statement releases the resources and frees up the cursor variable, allowing it to be opened again using the OPEN statement.

PL/pgSQL cursor example

The following example illustrates how to use a cursor to traverse the rows from the film table in the sample database:

```
CREATE OR REPLACE FUNCTION fetch_film_titles_and_years(
  OUT p_title VARCHAR(255),
  OUT p_release_year INTEGER
)
RETURNS SETOF RECORD AS
$$
DECLARE
    film_cursor CURSOR FOR
        SELECT title, release_year
        FROM film;
    film_record RECORD;
BEGIN
    -- Open cursor
    OPEN film_cursor;

    -- Fetch rows and return
    LOOP
        FETCH NEXT FROM film_cursor INTO film_record;
        EXIT WHEN NOT FOUND;

        p_title = film_record.title;
        p_release_year = film_record.release_year;
        RETURN NEXT;
    END LOOP;

    -- Close cursor
```

```
    CLOSE film_cursor;
END;
$$
LANGUAGE PLPGSQL;
```