PL/pgSQL functions in PostgreSQL

SQL functions lack any logic control.

To write more complex function we need a procedural language.

PostgreSQL supports 4 procedural languages: pgSQL, pgPython, pgPerl , pgTcl.

PL/pgSQL is installed by default while the other need specific extension setup.

> We're working on PL/pgSQL but do check out pgPython if your data pipeline is based on python. pgPython integrates very well with Pandas.

PL/pgSQL was designed with the goals:

- can be used to create functions, procedures, and triggers,
- adds control structures to the SQL language,
- can perform complex computations,

SQL vs PL/pgSQL : reduce overhead

SQL is the core PostgreSQL language.

Every SQL statement must be executed individually by the database server.

The client application must send each query to the database server, wait for it to be processed, receive and process the results, do some computation, then send further queries to the server.

All this incurs interprocess communication and will also incur network overhead if your client is on a different machine than the database server.

With PL/pgSQL functions, the code is executed **inside** the database server which greatly reduces client/server communication overhead. This might make a significant difference in execution speed if the server is remote.

Basic Syntax

The generic syntax to create a PL/pgSQL function is

```SQL
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS $$
    function body text
$$ LANGUAGE plpgsql;
```

Note `LANGUAGE plpgsql` instead of `LANGUAGE sql`.

also note that the `$$` are used for quoting the `function body text` instead of single quotes.

The `body` of the function is composed of BLOCKS

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

PL/pgSQL's `BEGIN/END` are only for grouping; they do not start or end a transaction.

Blocks and variable scope

The following function demonstrates the use of blocks by printing out the value of the variable `quantity`.

Blocks in PL/pgSQL are similar to indentations in Python.

```SQL
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
    quantity := 50;
    --
    -- Create a subblock
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;
    END;

    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50

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

Function Parameters

You pass parameters to the function by specifying their names and type

```SQL
CREATE FUNCTION sales_tax(subtotal real)
RETURNS real
AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Note an alternate style uses the position of the parameter with $1, $2 and then aliases the $n to a variable name.

see https://www.postgresql.org/docs/current/plpgsql-declarations.html#PLPGSQL-DECLARATION-PARAMETERS

RETURN values

Consider for instance:

```SQL
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY
            SELECT s.quantity, s.quantity * s.price FROM sales AS s
            WHERE s.itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

This will return the results of the query

LOOPING, VARIABLES and Conditional logic

You can use if then statements as such

```SQL
IF expression THEN ...
```

Variable assignment uses the `:=` sign

```SQL
my_record.user_id := 20;
```

You can also execute SQL queries exactly as you do in SQL.

```SQL
INSERT INTO mytable VALUES (1,'one'), (2,'two');
```

Testing number of rows returned

For a given SELECT query, you can test if no rows were returned with INTO

```SQL
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

or even test if more than one row was returned with `INTO STRICT`

```SQL
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'employee % not found', myname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'employee % not unique', myname;
END;
```

PL/pgSQl basics

Hello World

A function named hello_word that takes no argument and returns a string output. The type of the returned value is explicitly declared.

```SQL
CREATE FUNCTION hello_world()
RETURNS VARCHAR AS $$
BEGIN
    RETURN 'Hello, World!';
END;
$$ LANGUAGE plpgsql;
```

Then we can execute the function with

```SQL
SELECT hello_world();
```

Function with Argument

A function with argument

```SQL
CREATE OR REPLACE FUNCTION hello_world(name VARCHAR)
RETURNS VARCHAR AS $$
BEGIN
    RETURN FORMAT('Hello, %s', name);
END;
$$ LANGUAGE plpgsql;
```

returns

```sql
select hello_world('Alexis');
```

Function Overloading

You can define functions that have the same name but different number or types of arguments.

```sql
-- Function-1: (integer, integer) => integer
CREATE FUNCTION add(a INTEGER, b INTEGER)
RETURNS INTEGER AS $$
BEGIN
    RETURN a + b;
END;
$$ LANGUAGE plpgsql;
```

and the same `add` function with 3 parameters

```sql
-- Function-2: (integer, integer, integer) => integer
CREATE FUNCTION add(a INTEGER, b INTEGER, c INTEGER)
RETURNS INTEGER AS $$
BEGIN
    RETURN a + b + c;
END;
$$ LANGUAGE plpgsql;
```

which you would execute with

```sql
-- Invocation with Arguments
SELECT add(1, 2) AS "Two Sum", add(1, 2, 3) AS "Three Sum";
```

Variable Declaration

Let's calculate the radius of a circle given its circumference

The function returns the radius as NUMERIC

```sql
-- Function to calculate the radius of a circle given its circumference
CREATE OR REPLACE FUNCTION calculate_radius(
  circumference NUMERIC,
  round_digits INTEGER DEFAULT 2)
RETURNS NUMERIC AS $$
DECLARE
    -- define PI as constant
    pi CONSTANT NUMERIC := 3.1415;
    radius NUMERIC;
BEGIN
    radius := circumference / (2 * pi);
    radius := ROUND(radius, round_digits);
    RETURN radius;
END;
$$ LANGUAGE plpgsql;
```

Handling Exception

Let's handle the case of division by zero using the EXCEPTION and RAISE NOTICE statements

```sql
CREATE OR REPLACE FUNCTION safe_divide(a INTEGER, b INTEGER)
RETURNS NUMERIC AS $$
DECLARE
    result NUMERIC;
BEGIN
    BEGIN
        result := a / b;
    EXCEPTION
        WHEN division_by_zero THEN
            RAISE NOTICE 'Handling division_by_zero exception';
            -- result is not overwritten so it remains null
    END;

    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

what happens :

```sql
select safe_divide(10,0); -- null
```

RAISE NOTICE is used to generate a debugging message during function execution. It’s

primarily used for logging or debugging purposes to provide information about the function's state or execution flow.

The **BEGIN … EXCEPTION … END** block in PostgreSQL's PL/pgSQL language allows you to enclose code where exceptions may occur, providing a structured way to handle specific exceptions (WHEN clauses) and execute custom error-handling logic within database functions or procedures.