PL/pgSQL

PostgreSQL allows you to extend the database functionality with user-defined functions by using various procedural languages, which often referred to as stored procedures.

The store procedures define functions for creating triggers or custom aggregate functions. In addition, stored procedures also add many procedural features e.g., control structures and complex calculation. These allow you to develop custom functions much easier and more effective.

It is possible to call a procedural code block using the DO command without defining a function.

By default, PostgreSQL supports three procedural languages: SQL, PL/pgSQL, and C. You can also load other procedural languages e.g., Perl, Python, and TCL into PostgreSQL using extensions.

Advantages of using PostgreSQL stored procedures

- Reduce the number of round trips between application and database servers. All SQL statements are wrapped inside a function stored in the PostgreSQL database server so the application only has to issue a function call to get the result back instead of sending multiple SQL statements and wait for the result between each call.
- Increase application performance because the user-defined functions are pre-compiled and stored in the PostgreSQL database server.
- Be able to reuse in many applications. Once you develop a function, you can reuse it in any applications.

Disadvantages of using PostgreSQL stored procedures

- Slow in software development because it requires specialized skills that many developers do not possess.
- Make it difficult to manage versions and hard to debug.
- May not be portable to other database management systems e.g., MySQL or Microsoft SQL Server.

Note: PL/pgSQL is similar to PL/SQL.

PL/pgSQL Block Structure

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

Let's examine the block structure in more detail:

- Each block has two sections called declaration and body. The declaration section is optional while the body section is required. The block is ended with a semicolon (;) after the END keyword.
- A block may have optional labels at the beginning and at the end. The label at the beginning and at the end must be the same. The block label is used in case you want to use the block in EXIT statement or you want to qualify the names of variables declared in the block.
- The declaration section is where you declare all variables used within the body section. Each statement in the declaration section is terminated with a semicolon (;).
- The body section is where you put the logic of the block. It contains any valid statements. Each statement in the body section is also terminated with a semicolon (;).

PL/pgSQL block structure examples:

Example:

```
DO $$
BEGIN
  RAISE NOTICE 'Hello World';
END; $$
LANGUAGE plpgsql;

NOTICE:  Hello World
```

Notice that the DO statement does not belong to the block. It is used to execute an anonymous block. PostgreSQL introduced the DO statement since version 9.0.

The "$$" signs are used for dollar quoting and are in no way specific to function definitions. It can be used to replace single quotes practically anywhere in SQL scripts.

The body of a function happens to be a string literal which has to be enclosed in single quotes. Dollar-quoting is a PostgreSQL-specific substitute for single quotes to avoid quoting issues inside the function body. You could write your function definition with single-quotes just as well. But then you'd have to escape all single-quotes in the body:

```
DO '
BEGIN
  RAISE NOTICE ''Hello World'';  --It is single quotes and not double quote
END; '
LANGUAGE plpgsql;

NOTICE:  Hello World
```

In place of "$$" you can use:

```
DO $hi$
BEGIN
  RAISE NOTICE 'Hello World';
END; $hi$
LANGUAGE plpgsql;
```

```
NOTICE:  Hello World
```

In the declaration section, we declared a variable named counter and set its value to 0. Inside the body section, we increased the counter to 1 and output its value using RAISE NOTICE statement.

Example:

```
DO $$
DECLARE
  counter integer := 0;
BEGIN
  counter := counter+1;
  RAISE NOTICE 'Value of counter %',counter;
END; $$
LANGUAGE plpgsql;
```

```
NOTICE:  Value of counter 1
```

Example - Labelling block:

```
DO $$
<<outer_block>>
DECLARE
  counter integer := 0;
BEGIN
  counter := counter+1;
  RAISE NOTICE 'Value of counter:%',counter;
END outer_block; $$
LANGUAGE plpgsql;
```

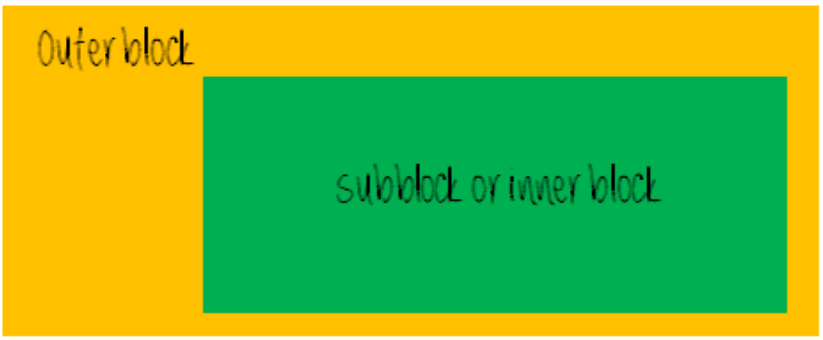
```
NOTICE:  Value of counter 1
```

The outer_block label is just for demonstration purpose. It does nothing in this example.

PL/pgSQL Subblock

You can put a block inside the body of another block. This block nested inside another is called subblock. The block that contains the subblock is referred to as an outer block.

While handling exceptions, if you have any mandatory code to be executed, you can have such code in enclosing block.



You often use subblocks for grouping statements so that a large block can be divided into smaller and more logical subblocks. The variables in the subblock can have the names as the ones in the outer block, even though it is not a good practice.

```
DO $$
DECLARE
  counter integer := 100;
BEGIN
  DECLARE
    counter integer := 200;
  BEGIN
    RAISE NOTICE 'In sub-block counter:%',counter;
  END;
  RAISE NOTICE 'In outer-block counter:%',counter;
END $$;
```

```
NOTICE:  In sub-block counter:200
NOTICE:  In outer-block counter:100
```

Example - Using label as qualifier to access variables:

When you define a variable within subblock with the same name as the one in the outer block, the variable in the outer block is hidden in the subblock. In case you want to access a variable in the outer block, you use block label to qualify its name; see the following example:

```
DO $$
<<outer_block>>
DECLARE
```

```
  counter integer := 100;
BEGIN
  <<sub_block>>
  DECLARE
    counter integer := 200;
  BEGIN
    RAISE NOTICE 'In sub-block counter:%',counter;
    RAISE NOTICE 'Outer-block counter in sub-block:%',outer_block.counter;
  END sub_block;
  RAISE NOTICE 'In outer-block counter:%',counter;
END outer_block; $$
LANGUAGE plpgsql;

NOTICE:  In sub-block counter:200
NOTICE:  Outer-block counter in sub-block:100
NOTICE:  In outer-block counter:100
```

PL/pgSQL Variables

A PL/pgSQL variable is a meaningful name for a memory location. A variable holds a value that can be changed through the block or function. A variable is always associated with a particular data type.

Before using a variable, you must declare it in the declaration section of the PL/pgSQL block. The following illustrates the syntax of declaring a variable.

variable_name data_type [:= expression];

- First, you specify the name of the variable. It is a good practice to assign a meaningful name to a variable.
- Second, you associate a specific data type with the variable. It can be any valid PostgreSQL data type such as integer, numeric, varchar, char, etc.
- Third, you can assign a default value to a variable. It is optional. If you don’t set a default value to the variable, the value of the variable is initialized to a NULL value.

Type Casts

A type cast specifies a conversion from one data type to another. PostgreSQL accepts two equivalent syntaxes for type casts:

CAST(***expression*** AS ***type***)

expression::***type***

The `CAST` syntax conforms to SQL; the syntax with `::` is historical PostgreSQL usage.

```
postgres=# select 567.96 * 234.56;
133220.6976

postgres=# select 567.96 * '234.56';  --implicit conversion
133220.6976

postgres=# select 567.96 * to_number('234.56','9999.99');
133220.6976

postgres=# select 567.96 * cast('234.56' as numeric);
133220.6976

postgres=# select 567.96 * '234.56'::numeric;
133220.6976
```

Declaring constant syntax

To declare a constant in PL/pgSQL, you use the following syntax:

```
constant_name CONSTANT data_type := expression;
```

The following example illustrates how to declare and initialize various variables:

```
DO $$
DECLARE
  counter integer := 1;
  first_name varchar(50) := 'Sachin';
  last_name varchar(50) := 'Tendulkar';
  average numeric(11,2) := 52.68;
BEGIN
  RAISE NOTICE '%. % % has an average of % runs', counter, first_name, last_name, average;
END $$;

NOTICE:  1. Sachin Tendulkar has an average of 52.68 runs
```

Assigning aliases to variables

PostgreSQL allows you to define an alias for any variable as follows:

```
new_name ALIAS FOR old_name;
```

The aliases are used mainly in a trigger procedure to assign more meaningful names for variables that have predetermined names e.g., NEW or OLD.

Control Flow Structures:

IF statement

```
IF <condition> THEN
  <statements>;
END IF;
```

IF – ELSE statement

```
IF <condition> THEN
  <statements>;
ELSE
  <statements>;
END IF;
```

IF – ELSIF - ELSE statement

```
IF <condition> THEN
  <statements>;
ELSIF <condition> THEN
  <statements>;
  ….
ELSE
  <statements>;
END IF;
```

CASE statement

Simple case statement

```
CASE <variable>
        WHEN <value_1> THEN <statements>;
        ….
        WHEN <value_n> THEN <statements>;
        [ELSE <statements>;]
END CASE;
```

Searched case statement

```
CASE
        WHEN <condition_1> THEN <statements>;
        ....
        WHEN <condition_n> THEN <statements>;
        [ELSE <statements>;]
END CASE;
```

Iterative LOOP statements

Simple Loop

```
LOOP
  <statements>;
END LOOP;
```

WHILE LOOP

```
WHILE <condition> LOOP
  <statements>;
END LOOP;
```

FOR LOOP (INTEGER FOR LOOP)

```
FOR <variable> IN [REVERSE] lower_bound .. upper_bound LOOP
   <statements>;
END LOOP;
```

Example for Simple Loop:

```
do $$
declare
  counter integer := 1;
begin
  loop
    raise notice 'Counter:%',counter;
    counter := counter + 1;
    if counter > 5 then
      exit;
    end if;
  end loop;
end $$;
```

```
NOTICE:  Counter:1
NOTICE:  Counter:2
NOTICE:  Counter:3
NOTICE:  Counter:4
```

```
NOTICE:  Counter:5
```

EXIT WHEN statement

```
do $$
declare
  counter integer := 1;
begin
  loop
    raise notice 'Counter:%',counter;
    counter := counter + 1;
    exit when counter > 5;
  end loop;
end $$;

NOTICE:  Counter:1
NOTICE:  Counter:2
NOTICE:  Counter:3
NOTICE:  Counter:4
NOTICE:  Counter:5
```

FOR LOOP example:

```
DO $$
BEGIN
  FOR counter IN 1 .. 5 LOOP
    RAISE NOTICE 'Counter:%',counter;
  END LOOP;
END $$;
NOTICE:  Counter:1
NOTICE:  Counter:2
NOTICE:  Counter:3
NOTICE:  Counter:4
NOTICE:  Counter:5
```

REVERSE option

```
DO $$
BEGIN
  FOR counter IN REVERSE 5 .. 1 LOOP
    RAISE NOTICE 'Counter:%',counter;
  END LOOP;
END $$;
NOTICE:  Counter:5
```

```
NOTICE:  Counter:4
NOTICE:  Counter:3
NOTICE:  Counter:2
NOTICE:  Counter:1
```

Fetching data from tables:

```
DO $$
DECLARE
  v_ename varchar(12);
BEGIN
  SELECT ename INTO v_ename FROM emp
  WHERE empno = 7566;
  RAISE NOTICE 'Employee Name:%',v_ename;
END $$;
```

Declaring variables based on datatype of a column:

```
  <variable> <table>.<column>%type;
```

```
DO $$
DECLARE
  v_ename emp.ename%type;
BEGIN
  SELECT ename INTO v_ename FROM emp
  WHERE empno = 7566;
  RAISE NOTICE 'Employee Name:%',v_ename;
END $$;
```

Fetching multiple fields

```
DO $$
DECLARE
  v_ename emp.ename%type;
  v_job   emp.job%type;
  v_sal   emp.sal%type;
BEGIN
  SELECT ename,job,sal INTO v_ename,v_job,v_sal
  FROM emp
  WHERE empno = 7566;
  RAISE NOTICE 'Employee Name:% Desig:% Salary:%',v_ename, v_job, v_sal;
END $$;
```

```
NOTICE:  Employee Name:JONES Desig:MANAGER Salary:2975.00
```

Declaring variable based on ROW of a table:

```
DO $$
DECLARE
  emprec emp%rowtype;
BEGIN
  SELECT * INTO emprec
  FROM emp
  WHERE empno = 7566;
  RAISE NOTICE 'Employee Name:% DOJ:%
Salary:%',emprec.ename,emprec.hiredate,emprec.sal;
END $$;
```

```
NOTICE:  Employee Name:JONES DOJ:1981-04-02 00:00:00 Salary:2975.00
```

User RECORD types are not supported as in oracle

```
DO $$
DECLARE
  TYPE emprectype IS RECORD(
    ename emp.ename%type,
    job   emp.job%type,
    sal   numeric(11,2)
);
  emprec emprectype;
BEGIN
  SELECT ename,job,sal INTO emprec
  FROM emp
  WHERE empno = 7566;
  RAISE NOTICE 'Employee Name:% Desig:%
Salary:%',emprec.ename,emprec.job,emprec.sal;
END $$;
```

```
ERROR:  syntax error at or near "RECORD"
LINE 3:   TYPE emprectype IS RECORD(
            ^
CONTEXT:  invalid type name "emprectype IS RECORD(
    ename emp.ename%type,
    job   emp.job%type,
    sal   numeric(11,2)
)"
```

Using keyword – RECORD

```
DO $$
DECLARE
  emprec RECORD;
BEGIN
  SELECT ename,job,sal INTO emprec
  FROM emp
  WHERE empno = 7566;
  RAISE NOTICE 'Employee Name:% Desig:%
Salary:%',emprec.ename,emprec.job,emprec.sal;
END; $$
LANGUAGE plpgsql;
```

```
NOTICE:  Employee Name:JONES Desig:MANAGER Salary:2975.00
```

PostgreSQL Stored Procedures - FUNCTIONS

In PostgreSQL, procedural languages such as PL/pgSQL, C, Perl, Python, and Tcl are referred to as stored procedures. The stored procedures add many procedural elements e.g., control structures, loop, and complex calculation to extend SQL-standard. It allows you to develop complex functions in PostgreSQL that may not be possible using plain SQL statements.

The reasons for choosing PL/pgSQL are:

- PL/pgSQL is simple and easy to learn.
- PL/pgSQL comes with PostgreSQL by default. The user-defined functions developed in PL/pgSQL can be used like any built-in functions.
- PL/pgSQL has many features that allow you to develop complex user-defined functions.

Developing User-defined Functions Using PostgreSQL CREATE FUNCTION Statement

```
CREATE FUNCTION function_name(p1 type, p2 type)
RETURNS type AS
BEGIN
-- executable code
END;
LANGUAGE language_name;
```

Let's examine the `CREATE FUNCTION` statement in more detail.

- First, specify the name of the function followed by the `CREATE FUNCTION` clause.
- Then, put a comma-separated list of parameters inside the parentheses following the function name.
- Next, specify the return type of the function after the `RETURNS` keyword.
- After that, place the code inside the `BEGIN` and `END` block. The function always ends with a semicolon (;) followed by the `END` keyword.
- Finally, indicate the procedural language of the function e.g., `plpgsql` in case PL/pgSQL is used.

PL/pgSQL Functions

User defined type:

```
CREATE TYPE product_summary as(
prod_id integer,
title varchar,
actor varchar,
price numeric(10,2));

CREATE OR REPLACE FUNCTION get_product_summary(p_id INT)
```

```
RETURNS product_summary AS
$$
SELECT prod_id,title,actor,price
FROM products
WHERE prod_id=p_id;
$$ LANGUAGE SQL;
```

Note: When you have SELECT statement only without BEGIN .. END language is “SQL”.

```
SELECT get_product_summary(1);
(1,"ACADEMY ACADEMY","PENELOPE GUINESS",25.99)
```

```
edbstore=> \dT+
                                    List of data types
 Schema |      Name       | Internal name | Size | Elements |  Owner  | Access privileges |
Description
---------+-----------------+-----------------+-------+----------+---------+-------------------+-------------
 edbuser | product_summary | product_summary | tuple |          | edbuser |                   |
```

Positional Parameters: $1, $2, …, $n

```
create or replace function add2nos(a integer,b integer)  --a & b are just decorators
returns integer as $$
begin
  raise notice '%,%',$2,$1 ;
  return $1 + $2;
end; $$
language plpgsql;

select add2nos(4,5);
NOTICE:  5,4
     9
```

-Parameter names are optional in PostgreSQL

```
create or replace function add2nos(integer,integer)
returns integer as $$
begin
  raise notice '%,%',$2,$1 ;
  return $1 + $2;
end; $$
language plpgsql;

select add2nos(10,20);
NOTICE:  20,10
    30
```

Returning a record
In PL/pgSQL, you can also define **set-returning functions** (**SRF**). These functions can return either a type defined by an existing table or a generic record type.

```
CREATE OR REPLACE FUNCTION getdepts()
RETURNS SETOF dept
AS 'SELECT * FROM dept;'
LANGUAGE sql SECURITY DEFINER;
```

`SECURITY DEFINER` specifies that the function is to be executed with the privileges of the user that created it.
`SECURITY INVOKER` indicates that the function is to be executed with the privileges of the user that calls it.

PL/pgSQL Function Parameters

- PL/pgSQL function parameters: IN, OUT, INOUT and VARIADIC.
- Default parameter mode is IN

Function to compute income tax

```
CREATE OR REPLACE FUNCTION find_tax(annual_income numeric(12,2))
RETURNS numeric AS $$
DECLARE
  edu_cess numeric(8,2) := 0;
  tax_on_income numeric(10,2) := 0;
  total_tax numeric(10,2) := 0;
BEGIN
  CASE
    WHEN annual_income > 1000000 THEN
      tax_on_income := annual_income * 0.3;
      edu_cess := tax_on_income * 0.03;
      total_tax := tax_on_income + edu_cess;
    WHEN annual_income BETWEEN 500000 AND 1000000 THEN
      tax_on_income := annual_income * 0.2;
      edu_cess := tax_on_income * 0.03;
      total_tax := tax_on_income + edu_cess;
    WHEN annual_income BETWEEN 250000 AND 499999 THEN
      tax_on_income := annual_income * 0.1;
      edu_cess := tax_on_income * 0.03;
      total_tax := tax_on_income + edu_cess;
    ELSE
      total_tax := 0;
    END CASE;
    RETURN total_tax;
```

```
END $$ LANGUAGE plpgsql;


SELECT find_tax(689450);
142026.70

edbstore=> SELECT find_tax(240000);
    0.00

edbstore=> SELECT ename,find_tax(sal*12*71+coalesce(comm,0)*71) from emp;
SMITH  |  140409.60
 ALLEN  |  427810.50
 WARD   |  340054.50
 JONES  |  783222.30
 MARTIN |  359799.60
 BLAKE  |  750313.80
 CLARK  |  645006.60
```

To find bonus with "OUT" parameter

```
CREATE OR REPLACE FUNCTION find_tax(IN annual_income NUMERIC,bonus OUT NUMERIC)
AS $$
BEGIN
  CASE
    WHEN annual_income > 36000 THEN
      bonus := annual_income * 0.3;
    WHEN annual_income BETWEEN 24000 AND 36000 THEN
      bonus := annual_income * 0.2;
    ELSE
      bonus := annual_income * 0.1;
    END CASE;
END $$ LANGUAGE plpgsql;

edbstore=> SELECT find_tax(50000);
 find_tax
----------
  15000.0

edbstore=> SELECT ename,find_tax(sal*12) from emp;
 ename  | find_tax
--------+-----------
 SMITH  |   960.000
 ALLEN  |  1920.000
 WARD   |  1500.000
```

Invoking above function from anonymous block

```
do $$
declare
  tax numeric := 0;
begin
  tax := find_tax(50000);
  raise notice '%',tax;
end;$$
language plpgsql;
NOTICE:  15000.0
```

You can use SELECT statement within anonymous block

```
do $$
declare
  tax numeric := 0;
begin
  SELECT find_tax(50000) INTO tax;
  raise notice '%',tax;
end;$$
language plpgsql;
NOTICE:  15000.0
```

INOUT or IN OUT parameter

```
CREATE OR REPLACE FUNCTION find_tax(annual_income IN OUT NUMERIC)
AS $$
BEGIN
  CASE
    WHEN annual_income > 36000 THEN
      annual_income := annual_income * 0.3;
    WHEN annual_income BETWEEN 24000 AND 36000 THEN
      annual_income := annual_income * 0.2;
    ELSE
      annual_income := annual_income * 0.1;
    END CASE;
END $$ LANGUAGE plpgsql;

edbstore=> SELECT find_tax(30000);
 find_tax
----------
   6000.0
(1 row)

edbstore=> SELECT ename,sal,find_tax(sal*12) FROM emp
edbstore-> WHERE deptno=10;
```

```
 ename  |   sal   | find_tax
--------+---------+-----------
 CLARK  | 2450.00 |  5880.000
 KING   | 5000.00 | 18000.000
 MILLER | 1300.00 |  1560.000
```

VARIADIC parameter:

```
CREATE OR REPLACE FUNCTION agg_func(
 VARIADIC list NUMERIC[],
 OUT minimum NUMERIC,
 maximum OUT NUMERIC,
 average OUT NUMERIC,
 total OUT  NUMERIC)
AS $$
BEGIN
  SELECT MIN(list[i]) into minimum
  FROM generate_subscripts(list,1) g(i);

  SELECT MAX(list[i]) into maximum
  FROM generate_subscripts(list,1) g(i);

  SELECT into average AVG(list[i])
  FROM generate_subscripts(list,1) g(i);

  SELECT into total SUM(list[i])
  FROM generate_subscripts(list,1) g(i);
END $$
LANGUAGE plpgsql;

edbstore=> SELECT agg_func(10,20,30,40,50,60);
(10,60,35.0000000000000000,210)
```

generate_subscripts is a "set-returning function", which will return multiple rows when you call it. That's why it's most often put in the FROM clause.

By default, the results from generate_subscripts, which comes built-in with Postgres is anonymous and does automatically have any name to use as a handle in order to refer to it in the rest of the query. This is what the g(i) is; it's an alias for the table (g) and the column (i) returned by generate_subscripts. So this expression:

```
FROM generate_subscripts($1, 1) g(i)
```

means:

execute the function generate_subscripts and assign its results to a table called "g" with a single column called "i". “1” means single dimensional array. In place of “g” you can use any name i.e.,”tab” also.

PL/pgSQL Function Overloading

PostgreSQL allows more than one function to have the same name, so long as the arguments are different. If more than one function has the same name, we say those functions are overloaded. When a function is called, PostgreSQL determines the exact function is being called based on the input arguments.

Function for adding row into department table

```
CREATE SEQUENCE dept_seq
START WITH 10
INCREMENT BY 5
NO MAXVALUE;

CREATE TABLE department (like dept);
```

First function add_dept()

```
CREATE OR REPLACE FUNCTION add_dept(IN p_deptno INT,IN p_dname VARCHAR,p_loc VARCHAR)
RETURNS VOID AS $$
BEGIN
  INSERT INTO department VALUES(p_deptno,p_dname,p_loc);
  RAISE NOTICE 'Inserted 1 row';
END $$ LANGUAGE plpgsql;
```

Second function add_dept() with same name

```
CREATE OR REPLACE FUNCTION add_dept(IN p_dname VARCHAR,p_loc VARCHAR)
RETURNS VOID AS $$
BEGIN
  INSERT INTO department VALUES(nextval('dept_seq'),p_dname,p_loc);
  RAISE NOTICE 'Inserted 1 row';
END $$ LANGUAGE plpgsql;
```

```
edbstore=> \df
                                   List of functions
 Schema  |     Name     | Result data type |                    Argument data types                     | Type
---------+--------------+------------------+------------------------------------------------------------+--------
 edbuser | add_dept     | void             | p_deptno integer, p_dname character varying, p_loc character varying
edbuser | add_dept     | void             | p_dname character varying, p_loc character varying

edbstore=> SELECT add_dept('ACCOUNTING','NEW YORK');
NOTICE:  Inserted 1 row

edbstore=> SELECT add_dept(15,'RESEARCH','DALLAS');
NOTICE:  Inserted 1 row

edbstore=> SELECT * FROM department;
```

```
 deptno |   dname    |   loc
--------+------------+----------
    10 | ACCOUNTING | NEW YORK
    15 | RESEARCH   | DALLAS
(2 rows)

edbstore=> SELECT currval('dept_seq');
 currval
---------
      10
```

PL/pgSQL Function That Returns A Table

```
CREATE OR REPLACE FUNCTION get_emps(p_deptno IN INT)
RETURNS TABLE(empname varchar,salary numeric)
AS $$
BEGIN
  RETURN QUERY SELECT
  ename,sal
  FROM emp
  WHERE deptno=p_deptno;
END; $$
LANGUAGE plpgsql;
```

RETURNS QUERY appends the results of executing a query to the function's result set

```
edbstore=> select * from get_emps(10);
 empname | salary
---------+---------
 CLARK   | 2450.00
 KING    | 5000.00
 MILLER  | 1300.00

edbstore=> select get_emps(10);
    get_emps
-----------------
 (CLARK,2450.00)
 (KING,5000.00)
 (MILLER,1300.00)

CREATE OR REPLACE FUNCTION get_emps(IN p_deptno INT)
RETURNS TABLE(empname varchar,
                      salary numeric)
AS $$
DECLARE
  rec RECORD;
```

```
BEGIN
  FOR rec IN (SELECT ename,sal FROM emp WHERE deptno=p_deptno) LOOP
    empname=rec.ename;
    salary=rec.sal;
    RETURN NEXT;
  END LOOP;
END; $$
LANGUAGE plpgsql;
```

`RETURN NEXT` and `RETURN QUERY` do not actually return from the function — they simply append zero or more rows to the function's result set. Execution then continues with the next statement in the PL/pgSQL function. As successive `RETURN NEXT` or `RETURN QUERY` commands are executed, the result set is built up. A final `RETURN`, which should have no argument, causes control to exit the function (or you can just let control reach the end of the function).

```
edbstore=> \df get_emps
                                List of functions
 Schema |   Name   |              Result data type              | Argument data types |  Type
--------+----------+--------------------------------------------+---------------------+--------
 edbuser | get_emps | TABLE(empname character varying, salary numeric) | p_deptno
integer    | normal
(1 row)

edbstore=> SELECT * FROM get_emps(10);
 empname | salary
---------+---------
 CLARK   | 2450.00
 KING    | 5000.00
 MILLER  | 1300.00
(3 rows)


edbstore=> SELECT get_emps(20);
   get_emps
-----------------
 (SMITH,800.00)
 (JONES,2975.00)
 (SCOTT,3000.00)
 (ADAMS,1100.00)
 (FORD,3000.00)
```

Special variables: FOUND & NOT FOUND

```
CREATE TABLE ACCOUNTS(
name varchar(12),
balance numeric(10,2));
```

```
INSERT INTO accounts VALUES('KRISHNA',5000);

INSERT INTO accounts VALUES('MURTHY',6700);

CREATE OR REPLACE FUNCTION transfer_amount(
  p_payer varchar(12),
  p_recipient varchar(12),
  p_amount numeric(10,2))
RETURNS text AS
$$
DECLARE
  payer_balance numeric(10,2);
BEGIN
  SELECT balance INTO payer_balance
  FROM accounts
  WHERE name = p_payer FOR UPDATE;
  IF NOT FOUND THEN
    RETURN 'Payer account not found';
  END IF;
  IF payer_balance < p_amount THEN
    RETURN 'Insufficient Balance';
  END IF;
  UPDATE accounts
    SET balance = balance + p_amount
    WHERE name = p_recipient;
  IF NOT FOUND THEN
    RETURN 'Recipient account not found';
  END IF;
  UPDATE accounts
    SET balance = balance - p_amount
    WHERE name = p_payer;
  RETURN 'Transaction successful.';
END; $$
LANGUAGE plpgsql;

SELECT * FROM accounts;
KRISHNA  | 5000.00
 MURTHY  | 6700.00

SELECT transfer_amount('KRISHNA','MURTHY',1000);
Transaction successful.

SELECT transfer_amount('VEKATESH','MURTHY',1000);
```

Payer account not found

SELECT transfer_amount('KRISHNA','SRINIVAS',1000);
Recipient account not found

SELECT * FROM accounts;
MURTHY | 7700.00
KRISHNA | 4000.00

SELECT transfer_amount('KRISHNA','SRINIVAS',4500);
Insufficient Balance

PL/pgSQL Errors and Messages

We will discuss how to report messages and raise errors using RAISE statement.

PL/pgSQL reporting messages

To raise a message, you use the RAISE statement as follows:

RAISE level format;

Let's examine the components of the RAISE statement in more detail.

Followed the RAISE statement is the level option that specifies the error severity. There are following levels in PostgreSQL:

- DEBUG
- LOG
- NOTICE
- INFO
- WARNING
- EXCEPTION

If you don't specify the ***level***, by default, the RAISE statement will use EXCEPTION level that raises an error and stops the current transaction. We will discuss the RAISE EXCEPTION later.

The ***format*** is a string that specifies the message. The format uses percentage (%) placeholders that will be substituted by the next arguments. The number of placeholders must match the number of arguments, otherwise, PostgreSQL will report the following error message:

[Err] ERROR: too many parameters specified for RAISE

The following example illustrates the RAISE statement that reports different messages at the current time.

```
edbstore=> \c edbstore postgres
You are now connected to database "edbstore" as user "postgres".

edbstore=# set search_path=edbuser,postgres,"$user";

DO $$
BEGIN
  RAISE INFO 'Information Message:%',now();
  RAISE LOG  'Log Message:%',now();
```

```
  RAISE DEBUG 'Debug Message:%',now();
  RAISE WARNING 'Warning Message:%',now();
  RAISE NOTICE 'Notice Message:%',now();
END $$;

INFO:  Information Message:2018-11-20 13:06:17.507501-05
WARNING:  Warning Message:2018-11-20 13:06:17.507501-05
NOTICE:  Notice Message:2018-11-20 13:06:17.507501-05
```

Notice that not all messages are reported back to client, only INFO, WARNING, and NOTICE level messages are reported to the client. This is controlled by the client_min_messages and log_min_messages configuration parameters.

```
edbstore=> show client_min_messages;
 notice

edbstore=> show log_min_messages;
 warning

edbstore=# set client_min_messages='debug';

edbstore=# DO $$
BEGIN
  RAISE INFO 'Information Message:%',now();
  RAISE LOG  'Log Message:%',now();
  RAISE DEBUG 'Debug Message:%',now();
  RAISE WARNING 'Warning Message:%',now();
  RAISE NOTICE 'Notice Message:%',now();
END $$;

INFO:  Information Message:2018-11-20 13:18:36.383492-05
LOG:  Log Message:2018-11-20 13:18:36.383492-05
DEBUG:  Debug Message:2018-11-20 13:18:36.383492-05
WARNING:  Warning Message:2018-11-20 13:18:36.383492-05
NOTICE:  Notice Message:2018-11-20 13:18:36.383492-05
```

Exception Handling:

What happens when SELECT statement fails to fetch a row?

```
CREATE OR REPLACE FUNCTION disp_ename(p_deptno IN integer)
RETURNS VOID AS $$
DECLARE
  v_ename varchar(12);
```

```
BEGIN
  SELECT ename INTO v_ename FROM emp
  WHERE deptno = p_deptno;
  RAISE NOTICE 'Employee Name:%',v_ename;
  RAISE NOTICE 'Fetch Successful.';
END; $$
LANGUAGE plpgsql;

SELECT disp_ename(50);

NOTICE:  Employee Name:<NULL>
NOTICE:  Fetch Successful.
```

Note: Exception is not raised when SELECT INTO fails to FETCH row.

What happens when SELECT statement returns more than 1 row?

```
SELECT disp_ename(10);

NOTICE:  Employee Name:CLARK
NOTICE:  Fetch Successful.
```

Note: First record in that dept is FETCHED. Actually, SELECT INTO returns more than 1 row.

Using keyword "STRICT"

If the `STRICT` option is specified, the query must return exactly one row or a run-time error will be reported

```
CREATE OR REPLACE FUNCTION disp_ename1(p_deptno IN integer)
RETURNS VOID AS $$
DECLARE
  v_ename varchar(12);
BEGIN
  SELECT ename INTO STRICT v_ename FROM emp
  WHERE deptno = p_deptno;
  RAISE NOTICE 'Employee Name:%',v_ename;
END; $$
LANGUAGE plpgsql;

SELECT disp_ename1(50);

ERROR:  query returned no rows
CONTEXT:  PL/pgSQL function disp_ename1(integer) line 5 at SQL statement
```

```
SELECT disp_ename1(10);

ERROR:  query returned more than one row
CONTEXT:  PL/pgSQL function disp_ename1(integer) line 5 at SQL statement
```

Pre-defined exception handlers:

Either `NO_DATA_FOUND` (no rows) or `TOO_MANY_ROWS` (more than one row). You can use an exception block if you wish to catch the error, for example:

```
CREATE OR REPLACE FUNCTION disp_ename2(p_deptno IN integer)
RETURNS VOID AS $$
DECLARE
  v_ename varchar(12);
BEGIN
  SELECT ename INTO STRICT v_ename FROM emp
  WHERE deptno = p_deptno;
  RAISE NOTICE 'Employee Name:%',v_ename;
EXCEPTION
  WHEN no_data_found THEN
    RAISE NOTICE 'No employee in this dept.';
  WHEN too_many_rows THEN
    RAISE NOTICE 'More than one employee in this dept';
END; $$
LANGUAGE plpgsql;
```

Successful execution of a command with `STRICT` always sets `FOUND` variable to true.

When OTHERS exception handler – SQLSTATE & SQLERRM

```
CREATE OR REPLACE FUNCTION add_dept(IN p_deptno INT,IN p_dname VARCHAR,p_loc
VARCHAR)
RETURNS VOID AS $$
BEGIN
  INSERT INTO department VALUES(p_deptno,p_dname,p_loc);
  RAISE NOTICE 'Inserted 1 row';
EXCEPTION
  WHEN unique_violation THEN
    RAISE NOTICE 'Unique constraint violated!!!';
  WHEN others THEN
    RAISE NOTICE 'Errorcode:% Message:%',sqlstate,sqlerrm;
END $$ LANGUAGE plpgsql;

select add_dept(10,'ACCOUNTING','NEW YORK');
NOTICE:  Unique constraint violated!!!

select add_dept(10,'abcdabcdabcdabcd ACCOUNTING','NEW YORK');
NOTICE:  Errorcode:22001 Message:value too long for type character varying(14)
```

Exception with CASE statement

```
DO $$
DECLARE
  grade char := 'a';
BEGIN
  CASE grade
    WHEN 'A' THEN
      RAISE NOTICE 'Excellent';
    WHEN 'B' THEN
      RAISE NOTICE 'Very Good';
  END CASE;
  RAISE NOTICE 'Done.';
END $$;
```

```
ERROR:  case not found
HINT:  CASE statement is missing ELSE part.
CONTEXT:  PL/pgSQL function inline_code_block line 5 at CASE
```

Handling Exception:

```
DO $$
DECLARE
  grade char := 'a';
BEGIN
  CASE grade
    WHEN 'A' THEN
      RAISE NOTICE 'Excellent';
    WHEN 'B' THEN
      RAISE NOTICE 'Very Good';
  END CASE;
  RAISE NOTICE 'Done.';
EXCEPTION
  WHEN others THEN
    RAISE NOTICE 'Exception:(%)',SQLSTATE;
END $$;
```

CASE_NOT_FOUND exception handler

```
DO $$
DECLARE
  grade char := 'a';
BEGIN
  CASE grade
    WHEN 'A' THEN
```

```
    RAISE NOTICE 'Excellent';
  WHEN 'B' THEN
    RAISE NOTICE 'Very Good';
 END CASE;
 RAISE NOTICE 'Done.';
EXCEPTION
 WHEN case_not_found THEN
  RAISE NOTICE 'Exception:(%)',SQLSTATE;
END $$;
```

```
NOTICE:  Exception:(20000)
```

PL/pgSQL raising errors

To raise errors, you use the EXCEPTION level after the RAISE statement. Note that RAISE statement uses EXCEPTION level by default.

Besides raising an error, you can add more detailed information using the following clause with the RAISE statement:

USING option = expression

The option can be:

- MESSAGE: to set error message text
- HINT: to provide the hint message so that the root cause of the error is easier to be discovered.
- DETAIL: to give detailed information about the error.
- ERRCODE: to identify the error code, which can be either by condition name or directly five-character SQLSTATE code. Please refer to the table of error codes and condition names.

The expression is a string-valued expression.

The following example raises a duplicate email error message:

Demo:

```
create table emps(
empid serial,
ename varchar(12),
mailid varchar(20));
```

```
insert into emps(ename,mailid) values('smith','smith@123');
```

```
insert into emps(ename,mailid) values('jones','jones@456');

select * from emps;
 empid | ename |  mailid
-------+-------+-----------
     1 | smith | smith@123
     2 | jones | jones@456
```

Initially without exception

```
DO $$
DECLARE
  email varchar(20) := 'allen@123';
  v_mailid varchar(20);
BEGIN
  SELECT mailid INTO v_mailid
  FROM emps;
  IF v_mailid = email THEN
    RAISE EXCEPTION 'Duplicate email: %', email
    USING HINT = 'Check email again';
  ELSE
    RAISE NOTICE 'Update successful...';
  END IF;
END $$;
NOTICE:  Update successful...
```

Now let us raise exception

```
DO $$
DECLARE
  email varchar(20) := 'smith@123';    --already existing
  v_mailid varchar(20);
BEGIN
  SELECT mailid INTO v_mailid
  FROM emps;
  IF v_mailid = email THEN
    RAISE EXCEPTION 'Duplicate email: %', email
    USING HINT = 'Check email again';
  ELSE
    RAISE NOTICE 'Update successful...';
  END IF;
END $$;

ERROR:  Duplicate email: smith@123
```

```
HINT:  Check email again
CONTEXT:  PL/pgSQL function inline_code_block line 9 at RAISE
```

The following examples illustrate how to raise an SQLSTATE and its corresponding condition:

```
DO $$
BEGIN
  INSERT INTO emps(ename,mailid)
  VALUES('allen','allen@321');
END $$;
```

Run the block once again:

```
DO $$
BEGIN
  INSERT INTO emps(ename,mailid)
  VALUES('allen','allen@321');
END $$;

ERROR:  duplicate key value violates unique constraint "emps_mailid_unq"
DETAIL:  Key (mailid)=(allen@321) already exists.
CONTEXT:  SQL statement "INSERT INTO emps(ename,mailid)
  VALUES('allen','allen@321')"
PL/pgSQL function inline_code_block line 3 at SQL statement
```

Reading SQLSTATE

```
DO $$
BEGIN
  INSERT INTO emps(ename,mailid)
  VALUES('allen','allen@321');
EXCEPTION
  WHEN others THEN
    RAISE NOTICE 'SQLSTATE:%',sqlstate;
END $$;
NOTICE:  SQLSTATE:23505
```

Using SQLSTATE in EXCEPTION block

```
DO $$
BEGIN
  INSERT INTO emps(ename,mailid)
  VALUES('allen','allen@321');
```

```
EXCEPTION
WHEN SQLSTATE '23505' THEN
  RAISE NOTICE 'Mailid is already existing';
END $$;
```

```
NOTICE:  Mailid is already existing
```

Using CONDITION NAME in EXCEPTION block

```
DO $$
BEGIN
  INSERT INTO emps(ename,mailid)
  VALUES('allen','allen@321');
EXCEPTION
WHEN unique_violation THEN
  RAISE NOTICE 'Mailid is already existing';
END $$;
```

```
NOTICE:  Mailid is already existing
```

Raising EXCEPTION explicitly using CONDITION NAME

```
DO $$
DECLARE
  email varchar(20) := 'smith@123';
  v_mailid varchar(20);
BEGIN
  SELECT mailid INTO v_mailid
  FROM emps;
  IF v_mailid = email THEN
    RAISE unique_violation;
  ELSE
    RAISE NOTICE 'Update successful...';
  END IF;
EXCEPTION
  WHEN others THEN
    RAISE NOTICE 'Error:%',SQLSTATE;
END $$;
```

```
NOTICE:  Error:23505
```

Raising EXCEPTION explicitly using SQLSTATE

```
DO $$
```

```
DECLARE
  email varchar(20) := 'smith@123';
  v_mailid varchar(20);
BEGIN
  SELECT mailid INTO v_mailid
  FROM emps;
  IF v_mailid = email THEN
    RAISE SQLSTATE '23505';
  ELSE
    RAISE NOTICE 'Update successful...';
  END IF;
EXCEPTION
  WHEN others THEN
    RAISE NOTICE 'Error:%',SQLSTATE;
END $$;
```

```
NOTICE:  Error:23505
```

PL/pgSQL putting debugging checks using ASSERT statement

PostgreSQL introduced the ASSERT statement since version 9.5.

Sometimes, a PL/pgSQL function is so big that makes it more difficult to detect the bugs. To facilitate this, PostgreSQL provides you with the ASSERT statement for adding debugging checks into a PL/pgSQL function.

The following illustrates syntax of the ASSERT statement:

```
ASSERT condition [, message];
```

- The condition is a boolean expression.
- If the condition evaluates to TRUE, ASSERT statement does nothing.
- If the condition evaluates to FALSE or NULL, the ASSERT_FAILURE is raised.
- If you don’t provide the message, PL/pgSQL uses “assertion failed” message by default.
- If the message is provided, the ASSERT statement will use it to replace the default message.

```
CREATE OR REPLACE FUNCTION assert1(NUMERIC)
RETURNS NUMERIC
AS $$
BEGIN
ASSERT $1 < 20,'Must be less than 20' ;
ASSERT $1 > 10,'Must be greater than 10' ;
RETURN $1 * 2 ;
END;
$$ LANGUAGE plpgsql ;

edbstore=# select assert1(30);
ERROR:  Must be less than 20
CONTEXT:  PL/pgSQL function assert1(numeric) line 3 at ASSERT

edbstore=# select assert1(5);
ERROR:  Must be greater than 10
CONTEXT:  PL/pgSQL function assert1(numeric) line 4 at ASSERT

edbstore=# select assert1(15);
 assert1
---------
     30
(1 row)
```

It is important to note that the ASSERT statement is used for debugging only.

Now you can use RAISE statement to either raise a message or report an error.

Disabling/Enabling ASSERT:

Parameter for controlling the operation of the ASSERT statement is plpgsql.check_asserts. The default value is "on", and ASSERT statement will work. If this parameter is set to "off" (false), ASSERT statement will no longer work.

```
edbstore=# SET plpgsql.check_asserts = false ;
SET
edbstore=# select assert1(5);
 assert1
---------
     10
(1 row)
```

PL/pgSQL Cursor

A PL/pgSQL cursor allows us to encapsulate a query and process each individual row at a time. We use cursors when we want to divide a large result set into parts and process each part individually. If we process it at once, we may have a memory overflow error.

In addition, we can develop a function that returns a reference to a cursor. This is an efficient way to return a large result set from a function. The caller of the function can process the result set based on the cursor reference.

Managing Cursors:

Cursors can be managed using following steps:

1. First, declare a cursor.
2. Next, open the cursor.
3. Then, fetch rows from the result set into a target.
4. After that, check if there is more row left to fetch. If yes, go to step 3, otherwise, go to step 5.
5. Finally, close the cursor.

Declaring cursors

To access to a cursor, you need to declare a cursor variable in the declaration section of a block.

```
cursor_name [ [NO] SCROLL ] CURSOR [( name datatype, name data type, ...)] FOR query;
```

First, you specify a variable name for the cursor.

Next, you specify whether the cursor can be scrolled backward using the `SCROLL`. If you use `NO SCROLL`, the cursor cannot be scrolled backward.

Then, you put the `CURSOR` keyword followed by a list of comma-separated arguments (`name` `datatype`) that defines parameters for the query. These arguments will be substituted by values when the cursor is opened.

After that, you specify a query following the `FOR` keyword. You can use any valid SELECT statement here.

The following example illustrates how to declare cursors:

```
DECLARE
  empcur1 CURSOR FOR SELECT * FROM emp;
  empcur2 CURSOR (p_deptno INT) FOR SELECT * FROM EMP WHERE deptno = p_deptno;
```

PostgreSQL also provides us with a special type called `REFCURSOR` to declare a cursor variable.

```
DECLARE
  my_cursor REFCURSOR;
```

Opening cursors

Cursors must be opened before they can be used to query rows. PostgreSQL provides syntax for opening an unbound and bound cursors.

We open an unbound cursor using the following syntax:

```
OPEN unbound_cursor_variable [ [ NO ] SCROLL ] FOR query;
```

Because unbound cursor variable is not bounded to any query when we declared it, we have to specify the query when we open it. See the following example:

```
OPEN my_cursor FOR SELECT * FROM emp WHERE deptno = p_deptno;
```

PostgreSQL allows us to open a cursor and bound it to a dynamic query. Here is the syntax:

```
OPEN unbound_cursor_variable[ [ NO ] SCROLL ]
FOR EXECUTE query_string [USING expression [, ... ] ];
```

In the following example, we build a dynamic query that sorts rows based on a `sort_field` parameter and open the cursor that executes the dynamic query.

```
query := 'SELECT * FROM emp ORDER BY $1';

OPEN emp_cur FOR EXECUTE query USING sort_field;
```

Opening bound cursors

Because a bound cursor already bounds to a query when we declared it, so when we open it, we just need to pass the arguments to the query if necessary.

```
OPEN cursor_variable[ (name:=value,name:=value,...)];
```

In the following example, we open bound cursors `empcur1` and `empcur2` that we declared above:

```
OPEN empcur1;
OPEN empcur2(p_deptno := 20);
```

Using cursors

After opening a cursor, we can manipulate it using `FETCH`, `MOVE`, UPDATE, or DELETE statement.

Fetching the next row

```
FETCH [ direction { FROM | IN } ] cursor_variable INTO target_variable;
```

The `FETCH` statement gets the next row from the cursor and assigns it a `target_variable`, which could be a record, a row variable, or a comma-separated list of variables. If no more row found, the `target_variable` is set to `NULL`(s).

By default, a cursor gets the next row if you don't specify the direction explicitly. The following is valid for the cursor:

- NEXT
- LAST
- PRIOR
- FIRST
- ABSOLUTE count
- RELATIVE count
- FORWARD
- BACKWARD

Note that `FORWARD` and `BACKWARD` directions are only for cursors declared with `SCROLL` option.

See the following examples of fetching cursors.

```
FETCH empcur1 INTO emprow;
FETCH LAST FROM emprow INTO ename, sal;
```

Moving the cursor

```
MOVE [ direction { FROM | IN } ] cursor_variable;
```

If you want to move the cursor only without retrieving any row, you use the `MOVE` statement. The direction accepts the same value as the `FETCH` statement.

```
MOVE empcur2;
MOVE LAST FROM empcur2;
MOVE RELATIVE -1 FROM empcur2;
MOVE FORWARD 3 FROM empcur2;
```

Deleting or updating the row

Once a cursor is positioned, we can delete or update row identifying by the cursor using `DELETE WHERE CURRENT OF` or `UPDATE WHERE CURRENT OF` statement as follows:

```
UPDATE table_name
SET column = value, ...
WHERE CURRENT OF cursor_variable;

DELETE FROM table_name
WHERE CURRENT OF cursor_variable;
```

See the following example.

```
UPDATE emp SET sal = p_sal
WHERE CURRENT OF empcur1;
```

Closing cursors

To close an opening cursor, we use `CLOSE` statement as follows:

```
CLOSE cursor_variable;
```

The `CLOSE` statement releases resources or frees up cursor variable to allow it to be opene d again using `OPEN` statement.

Cursor Example:

```
CREATE OR REPLACE FUNCTION fetch_emps(p_deptno integer)
RETURNS TEXT
AS $$
DECLARE
  empcursor CURSOR(c_deptno integer)
  FOR SELECT * FROM emp WHERE deptno = c_deptno;
  emprec RECORD;
  emps TEXT DEFAULT '';
BEGIN
  OPEN empcursor(p_deptno);
  LOOP
    FETCH empcursor INTO emprec;
    EXIT WHEN NOT FOUND;
    emps := emprec.ename||','||emprec.sal||','||emps;
  END LOOP;
  CLOSE empcursor;
  RETURN emps;
END; $$
```

```
LANGUAGE plpgsql;

edbstore=> SELECT fetch_emps(10);
MILLER,1300.00,KING,5000.00,CLARK,2450.00,
(1 row)

edbstore=> SELECT fetch_emps(20);
FORD,3000.00,ADAMS,1100.00,SCOTT,3000.00,JONES,2975.00,SMITH,800.00,
(1 row)

edbstore=> SELECT fetch_emps(200);
```

REFCURSOR example:

```
CREATE OR REPLACE FUNCTION show_emps()
RETURNS refcursor AS $$
DECLARE
  ref refcursor;
BEGIN
  OPEN ref FOR SELECT empno,ename FROM emp;
  RETURN ref;
END;
$$ LANGUAGE plpgsql;
```

Processing the result sets and designing the procedures returning result sets may depend on the caller.

Let's assume you need to call a procedure and output the result set in PSQL tool, pgAdmin Query tool or another function:

```
SELECT show_emps();

edbstore=> select * from show_emps();
<unnamed portal 1>   --name of the cursor. It might be portal 2 or portal 3 …..
```

The query returns the *name* of the cursor, it does *not* output the rows of the result set. To get the rows you need to use FETCH statement and specify the cursor name:

```
edbstore=> FETCH ALL IN "<unnamed portal 1>";
ERROR:  cursor "<unnamed portal 1>" does not exist
```

The problem is that the cursor already closed, as we did not use a transaction. Let's start a transaction, execute the procedure, and fetch rows again:

```
BEGIN TRANSACTION;
```

```
SELECT show_emps();

FETCH ALL IN "<unnamed portal 1>";

Output:
empno | ename
-------+--------
  7369 | SMITH
  7499 | ALLEN
  7521 | WARD

COMMIT;
```

Cursor Name Problem

As you may have noticed, the name of the cursor may change, and it is quite inconvenient to fetch the cursor name first, and then use it in the FETCH statement.

As an option you can slightly redesign a procedure and pass the cursor name as a parameter, so the caller always knows which cursor to fetch:

```
CREATE OR REPLACE FUNCTION show_emps1(p_ref REFCURSOR)
RETURNS refcursor AS $$
BEGIN
  OPEN p_ref FOR SELECT empno,ename FROM emp;
  RETURN p_ref;
END;
$$ LANGUAGE plpgsql;

edbstore=> BEGIN TRANSACTION;

edbstore=> SELECT show_emps1('emp_refcur');
emp_refcur

edbstore=> FETCH ALL IN "emp_refcur";
  7369 | SMITH
  7499 | ALLEN
  7521 | WARD
  …….
edbstore=> COMMIT;
```

PostgreSQL Arrays

Integer Array example

```
DO $$
DECLARE
  X INTEGER[];              -- Declaration
BEGIN
  X[1]:=10;
  X[2]:=20;
  X[3]:=30;
  FOR i IN 1..3 LOOP
    RAISE NOTICE '%',X[i];
  END LOOP;
END; $$
LANGUAGE plpgsql;
```

```
NOTICE:  10
NOTICE:  20
NOTICE:  30
```

Character Array example:

```
DO $$
DECLARE
  X varchar[];
BEGIN
  X[1]:='A';
  X[2]:='B';
  X[3]:='C';
  FOR i IN 1..3 LOOP
    RAISE NOTICE '%',X[i];
  END LOOP;
END; $$
LANGUAGE plpgsql;
```

```
NOTICE:  A
NOTICE:  B
NOTICE:  C
```

String Array Example:

```
DO $$
DECLARE
  X varchar[];
BEGIN
  X[1]:='welcome';
```

```
  X[2]:='to';
  X[3]:='plpgsql';
  FOR i IN array_lower(x,1).. array_upper(x,1)  LOOP
    RAISE NOTICE '%',X[i];
  END LOOP;
END; $$
LANGUAGE plpgsql;
```

```
NOTICE:  welcome
NOTICE:  to
NOTICE:  plpgsql
```

Initializing Arrays at the time of declaration:

```
DO $$
DECLARE
  X int[] := ARRAY[10,20,30];
BEGIN
  FOR i IN 1.. array_upper(x,1)  LOOP
    RAISE NOTICE '%',X[i];
  END LOOP;
END; $$
LANGUAGE plpgsql;
```

```
NOTICE:  10
NOTICE:  20
NOTICE:  30
```

```
DO $$
DECLARE
  X varchar[] := ARRAY['A','B','C'];
BEGIN
  FOR i IN 1.. array_upper(x,1)  LOOP
    RAISE NOTICE '%',X[i];
  END LOOP;
END; $$
LANGUAGE plpgsql;
```

```
NOTICE:  A
NOTICE:  B
NOTICE:  C
```

```
DO $$
DECLARE
  X varchar[] := ARRAY['Welcome','to','PostgreSQL'];
BEGIN
  FOR i IN 1..array_upper(x,1) LOOP
```

```
   RAISE NOTICE '%',X[i];
  END LOOP;
END; $$
LANGUAGE plpgsql;

NOTICE:  Welcome
NOTICE:  to
NOTICE:  PostgreSQL
```

Storing all rows into an Array:

```
CREATE OR REPLACE FUNCTION disp_dept()
RETURNS SETOF dept AS $$
DECLARE
  arr dept[] = (select array(select dept from dept));
BEGIN
  raise notice'%  % %',arr[1].deptno,arr[1].dname,arr[1].loc;
  return query SELECT * FROM unnest(arr);
END; $$
LANGUAGE plpgsql;

edbstore=> select disp_dept();
NOTICE:  10  ACCOUNTING NEW YORK
       disp_dept
---------------------------
 (10,ACCOUNTING,"NEW YORK")
 (20,RESEARCH,DALLAS)
 (30,SALES,CHICAGO)
 (40,OPERATIONS,BOSTON)
```

Observe:

```
select dept from dept;
         dept
---------------------------
 (10,ACCOUNTING,"NEW YORK")
 (20,RESEARCH,DALLAS)
 (30,SALES,CHICAGO)
 (40,OPERATIONS,BOSTON)
```

-In following example array is populated in executable section

```
CREATE OR REPLACE FUNCTION disp_dept()
RETURNS SETOF dept AS $$
DECLARE
  arr dept[];
BEGIN
```

```
  arr = (select array(select dept from dept));
  return query SELECT * FROM unnest(arr);
END; $$
LANGUAGE plpgsql;
```

Array columns

Array plays an important role in PostgreSQL. Every data type has its own companion array type e.g., integer has an integer[] array type, character has character[] array type, etc. In case you define your own data type, PostgreSQL creates a corresponding array type in the background for you.

PostgreSQL allows you to define a column to be an array of any valid data type including built-in type, user-defined type or enumerated type.

The following CREATE TABLE statement creates the *contacts* table with the *phones* column is defined as an array of text.

```
CREATE TABLE contacts(
 id serial primary key,
 name varchar(12),
 phones text []
);

insert into contacts(name,phones)
values('krishna',array['+919880280040','+919886771323']);
```

We used the ARRAY constructor to construct an array and insert it into the contacts table.

You can also use curly braces as follows:

```
insert into contacts(name,phones)
values('prakash','{"+919845801302","+919862567897"}');
```

Notice that when you use curly braces, you use single quotes ' to wrap the array and double quotes " to wrap text array items.

```
select * from contacts;

  1 | krishna | {+919880280040,+919886771323}
  2 | prakash | {+919845801302,+919862567897}
```

We access array elements using the subscript within square brackets []. By default, PostgreSQL uses one-based numbering for array elements. It means the first array element starts with number 1. Suppose, we want to get the contact's name and the first phone number, we use the following query:

```
select name,phones[1]
from contacts;

 krishna | +919880280040
 prakash | +919845801302
```

We can use array element in the WHERE clause as the condition to filter the rows. For example, to find out who has the phone number +919886771323 as the second phone number, we use the following query.

```
select * from contacts
where phones[2]='+919886771323';

  1 | krishna | {+919880280040,+919886771323}
```

Modifying PostgreSQL array

PostgreSQL allows you to update each element of an array or the whole array. The following statement updates the second phone number of

```
update contacts
set phones[2]='+911234567890'
where name='krishna';
```

-if you don't use subscript both numbers would be overwritten.

```
select name,phones from contacts
where name='krishna';

krishna | {+919880280040,+911234567890}
```

Suppose, we want to know who has the phone number +919880280040 regardless of position of the phone number in the phones array, we use ANY() function as follows:

```
select name,phones
from contacts
where '+919880280040'=ANY(phones);

krishna | {+919880280040,+911234567890}
```

Expand Arrays : unnest()

PostgreSQL provides the unnest() function to expand an array to a list of rows. For example, the following query expands all phone numbers of the phones array.

```
select name,unnest(phones) phones
from contacts;
```

prakash | +919845801302
prakash | +919862567897
krishna | +919880280040
krishna | +911234567890

Adding an array element: ||, array_cat(), array_append()

```
update contacts
set phones = array_cat(phones,'{+919876543210}')
where name='prakash';

update contacts
set phones = array_append(phones,'+919876541111') --don't use {} braces
where name='krishna';

update contacts
set phones = phones || '{+91987600000}'
where name='krishna';
```

Removing array element: array_remove()

```
update contacts
set phones = array_remove(phones,'+919876541111');
```

--don't use {}. Removes matching elements in all rows

Updating a particular array element: array_replace()

```
update contacts
set phones = array_replace(phones,'+919880280040','12345')
where name='krishna';
```

To replace one element with multiple elements:

Let us try with array_replace() as we did earlier

```
update contacts
set phones = array_replace(phones,'12345','{"1111","2222"}')
where name='krishna';

select name,phones from contacts where name='krishna';
```

krishna | {"{\"1111\",\"2222\"}",+911234567890}

```
select name,unnest(phones) from contacts where name='krishna';
```

```
 krishna | {"1111","2222"}  --invalid entry
 krishna | +911234567890
```
-Let us remove the element

```
update contacts
set phones=array_remove(phones,'{"1111","2222"}')
where name='krishna';
```

-To replace with multiple elements, use following method:

```
update contacts
set phones=array_remove(phones,'+911234567890')||'{1111,2222}'
where name='krishna';

select name,unnest(phones) from contacts
where name='krishna';
  name   | unnest
---------+--------
 krishna | 1111
 krishna | 2222
```

PostgreSQL Triggers

A PostgreSQL trigger is a function invoked automatically whenever an event e.g., insert, update, or delete occurred.

A trigger is a special user-defined function that binds to a table. To create a new trigger, you must define a trigger function first, and then bind this trigger function to a table. The difference between a trigger and a user-defined function is that a trigger is automatically invoked when an event occurs.

PostgreSQL provides two main types of triggers: row and statement level triggers. The differences between the two are how many times the trigger is invoked and at what time. For example, if you issue an UPDATE statement that affects 20 rows, the row level trigger will be invoked 20 times, while the statement level trigger will be invoked 1 time.

You can specify whether the trigger is invoked before or after an event. If the trigger is invoked before an event, it can skip the operation for the current row or even change the row being updated or inserted. In case the trigger is invoked after the event, all changes are available to the trigger.

Triggers are useful in case the database is accessed by various applications, and you want to keep the cross-functionality within database that runs automatically whenever the data of the table is modified. For example, if you want to keep history of data without requiring application to have logic to check for every event such as INSERT or UDPATE.

You can also use triggers to maintain complex data integrity rules which you cannot implement elsewhere except at the database level. For example, when a new row is added into the customer table, other rows must be also created in tables of banks and credits. The main drawback of using trigger is that you must know the trigger exists and understand its logic in order to figure it out the effects when data changes.

Even though PostgreSQL implements SQL standard, triggers in PostgreSQL has some specific features as follows:

- PostgreSQL fires trigger for the TRUNCATE event.
- PostgreSQL allows you to define statement-level trigger on views.
- PostgreSQL requires you to define a user-defined function as the action of the trigger, while the SQL standard allows you to use any number of SQL commands.

To create a new trigger in PostgreSQL, you need to:

- Create a trigger function using CREATE FUNCTION statement.
- Bind this trigger function to a table using CREATE TRIGGER statement.

Creating the trigger function

A trigger function is similar to an ordinary function, except that it does not take any arguments and has return value type `trigger` as follows:

```
CREATE FUNCTION trigger_function() RETURN trigger AS
```

The trigger function receives data about their calling environment through a special structure called *TriggerData*, which contains a set of local variables. For example, `OLD` and `NEW` represent the states of row in the table before or after the triggering event. PostgreSQL provides other local variables starting with `TG_` as the prefix such as `TG_WHEN`, `TG_TABLE_NAME`, etc.

Once the trigger function is defined, you can bind it to specific actions on a table.

Creating the trigger

To create a new trigger, you use the `CREATE TRIGGER` statement. The complete syntax of the `CREATE TRIGGER` is complex with many options. However, for the sake of demonstration, we will use the simple form of the `CREATE TRIGGER` syntax as follows:

```
CREATE TRIGGER trigger_name {BEFORE | AFTER | INSTEAD OF} {event [OR ...]}
  ON table_name
  [FOR [EACH] {ROW | STATEMENT}]
  EXECUTE PROCEDURE trigger_function
```

The event could be `INSERT, UPDATE, DELETE` or `TRUNCATE`. You can define trigger that fires before (`BEFORE`) or after (`AFTER`) event. The `INSTEAD OF` is used only for `INSERT, UPDATE,` or `DELETE` on the views.

PostgreSQL provides two kinds of triggers: row level trigger and statement level trigger, which can be specified by `FOR EACH ROW` (row level trigger) or `FOR EACH STATEMENT` (statement level trigger).

PostgreSQL Trigger example

-To maintain CASE consistency

```
CREATE TABLE emps(
  empid integer primary key,
  ename varchar(12) check(ename=upper(ename))
);

postgres=# insert into emps values(101,'krishna');
ERROR:  new row for relation "emps" violates check constraint "emps_ename_check"
DETAIL:  Failing row contains (101, krishna).
```

```
CREATE OR REPLACE function upp_trig_func()
RETURNS trigger AS
$$
BEGIN
  NEW.ename := upper(NEW.ename);
  RETURN NEW;
END; $$
LANGUAGE plpgsql;

CREATE TRIGGER upp_trig
BEFORE INSERT OR UPDATE on EMPS
FOR EACH ROW
EXECUTE PROCEDURE upp_trig_func();

insert into emps values(101,'krishNA');

select * from emps;

  101 | KRISHNA
```

Managing PostgreSQL Trigger

Modifying the trigger

To modify the trigger, you use `ALTER TRIGGER` statement. This statement is a PostgreSQL extension of the SQL standard. The syntax of the `ALTER TRIGGER` statement is as follows:

```
ALTER TRIGGER trigger_name ON table_name
RENAME TO new_name;
```

First, you specify the name of trigger associated with a particular table that you want to change.

Second, you put the new trigger name in the `RENAME TO` clause.

Disabling the trigger

PostgreSQL does not provide any specific statement such as `DISABLE TRIGGER` for disabling an existing trigger. However, you can disable a trigger using ALTER TABLE statement as follows:

```
ALTER TABLE table_name
DISABLE TRIGGER trigger_name | ALL
```

You specify the trigger name after the `DISABLE TRIGGER` clause to disable a particular trigger. To disable all triggers associated with a table, you use `ALL` instead of a particular trigger name.

Notice that a disabled trigger is still available in the database. However, it is not fired when its triggering event occurs.

Removing the trigger

To remove an existing trigger definition, you use `DROP TRIGGER` statement as follows:

```
DROP TRIGGER [IF EXISTS] trigger_name ON table_name;
```