Unit 1

Relational Database Design

1. Introduction

Relational database management systems are multiuser systems. Structured query language provides constructs using which data can be retrieved/new data inserted/existing data updated/ and unwanted data deleted from a database table.

In this chapter, we learn about procedural SQL, or also called as procedural language, Structured Query Language (PLSQL).

Since we are learning SQL using the PostgreSQL server. The procedural language extension is thus defined as PL/pgSQL. PL/pgSQL is a loadable procedural language developed for postgres, as an extension to standard SQL. It provides a mechanism to execute procedural logic on the database. It further helps to five multiple queries as a single block to the database server.

1.1 PL/PGSQL

PostgreSql is one of the most successful open source databases available. PL/pgSQL is a loadable, procedural language developed for Postgres as an extension to standard SQL to provide a way to execute procedural logic on the database. A procedural language is a programming language used to specify a sequence of steps that are followed to produce an intended result.

PL/pgSQL can be used to group sequences of SQL and programmatic statements together within a database server. This helps in reducing the overhead incurred by client applications due to constant request for data from the server and perform logic operations on that data from a remote location.

A database programmer can access all postgreSQL datatypes, operators, and functions within PL/pgSQL code.

PL/pgSQL provides a mechanism for developers to add a procedural component at the server level. So developers can have all the features of a full-fledged procedural language at server level. SQL statements can be directly used within a PL/pgSQL code block. This increases the power, flexibility and performance of the programs. When multiple SQL statements are executed from within a PL/pgSQL code block, the statements are processed at one time; instead of the normal behaviour of processing a single SQL statement at a time.

Another important feature of PL/pgSQL is its portability; its functions are compatible with all platforms that can operate the postgreSQL database system.

PL/pgSQL Data Types

SQL is a strongly typed language; implying that any piece of data represented by postgreSQL has an associated data type.

postgreSQL supports a wide variety of built-in datatypes, and it also provides an option to the users to add new data types to PostgreSQL, using the CREATE TYPE command. *Table 1.1* lists the data types officially supported by postgreSQL. Most data types supported by postgreSQL are directly derived from SQL standards.

3

Oct. 2018 – *1M*
Give any four datatypes used in PL/pgsql.

Apr. 2018 – *4M*
What are various datatypes available in PL/pgSQL.

Oct. 2017 – *3M*
Explain different data types in PL/pgSQL.

Table 1.1: PostgreSQL supported data types

Category	Data type	Description
Boolean and Binary types	boolean, bool	A single true or false value.
	bit(n)	An n-length bit string (exactly n binary bits).
	bit varying(n), varbit(n)	A variable n-length bit string (upto n binary nbits).
Character types	character(n) char(n)	A fixed n-length character string.
	character varying(n) varchar (n)	A variable length character string of upto n characters.
	text	A variable length character string of unlimited length.
Numeric types	smallint, int2	A signed 2-byte integer.
	integer, int, int4	A signed, fixed precision 4-byte number.
	bigint, int8	A signed 8-byte integer, upto 18 digits in length.
	real, float4	A 4-byte floating point number.
	double precision,float8, float	An 8-byte floating point number.
	numeric(p,s) decimal(p,s)	An exact numeric type with arbitrary precision p, and scale s.
	money	A fixed precision, U.S style currency.
	serial	An auto-incrementing 4-byte integer.
Date and time types	date	The calendar date (day, month and year).
	time	The time of day.
	time with time zone timestamp(includes time zone)	The time of day, including time zone information.
	interval	An arbitrarily specified length of time.

NULL Values

Irrespective of its data type, there is one value that any data typed column can accept: the NULL value. This value is set using the SQL keyword NULL. NULL has no data value, hence its not considered as a type. It just indicates that the field has no value. The only place where a field cannot have a NULL value, is when the NOT NULL constraint is specified on the column.

NULL is often used in places where a value is optional. NULL can be thought of as a meta value; a value that represents a lack of value which will never be equivalent to a non-null value. A NULL value represents the complete absence of value within the column, not that it is merely blank.

1.2 Language Structure

PL/pgSQL follows a simple language structure, where in any piece of code is designed to exist as a function. Its language structure is similar to C programming language structure. Hence in PL/pgSQL too, each portion of code acts as a function, all variables must be declared before being used and code segments accept arguments when called and return arguments at their end.

PL/pgSQL functions are case insensitive (you may use mixed upper, or lower case for keywords and identifiers). We use a pair of apostrophes, whenever we want to escape an apostrophe within a function code. Hence we use a pair of apostrophes within the function definition, since a function definition is actually a large string constant within a CREATE FUNCTION statement.

We discuss the block organization of PL/pgSQL code, how to use comments, the organization of PL/pgSQL expressions and the usage of statements.

Code Blocks

PL/pgSQL uses block structured type of organization to organize its code. Code blocks are entered within an SQL CREATE FUNCTION call that creates a PL/pgSQL function in the PostgreSQL database.

The CREATE FUNCTION is used to name a define function, its argument types and its return type. The function's main block code then starts with a declaration section.

All variables are declared and optionally initialized to a default value in the declaration section of a code block. A variable declaration specifies the variable's name and its data type. The declaration section is denoted by the DECLARE keyword. Each declaration is ended with a semicolon.

After the declaration section, the main body of code starts with a BEGIN keyword. The code's block statements appear after the BEGIN keyword.

The END keyword is used to denote the end of a code block. The main block of a PL/pgSQL function should return a value of its specified return type and end any sub-code blocks before its END is reached.

Example 1: Shows the structure of a PL/pgSQL code block:

```
CREATE FUNCTION identifier (arguments) RETURNS type AS
'DECLARE
    Variable-declarations;
     [……]
BEGIN
     Statement;
     [……..]
END ;
'LANGUAGE 'plpgsql';
```

In the above, variable–declaration refer to any variables that is to be defined and is used later in the program–code.

The declaration section is optional. The Begin and End are the keywords that delimit the procedural portion of the block. The semicolon at the end is the PL/pgSQL statement terminator and it signifies the end of the block. PL/pgSQL blocks can be nested. One block can contain another block as given below:

```
CREATE FUNCTION identifier (arguments) RETURNS type AS
'Declare
  Variable-declarations
Begin
    Some Program-code
Declare
      Variable declarations;
Begin
  Code in a nested block
End;
  More program-code
End;
'LANGUAGE 'plpgsql';
```

Nested blocks are useful for organization of code within a large PL/pgSQL function. All sub blocks follow normal block structure, that is they must start with the DECLARE keyword, followed by the BEGIN keyword and a body of statements, then end with the END keyword.

Comments

There are two ways of commenting in PL/pgSQL, similar to the commenting structure of other programming languages. They are the single line comments and the block comments.

The single line comments begin with two dashes (--) and have no end character. The language parser interprets all characters on the same line after the dashes as part of the comments.

Example 2: Shows the use of single line comments.

```
-- This is a single-line comment.
```

The second type of commenting is the block comments or the multiple line comments. Block comments begin with a forward slash and asterisk (/*) and end with an asterisk and forward slash character (*/). Block comments can span multiple lines, and any text between the opening /* and closing */ is considered as a comment.

Example 3: Demonstrates a block comment.

```
/* this is
an example of a
block or multi line comment */
```

Block comments cannot be nested within other block comments; but single line comments can be nested within a block comment.

Statements and Expressions

PL/pgSQL code is composed of statements and expressions. Most of the code is made of statements and expressions, since they are essential to certain types of data manipulations.

A statement performs an action within a PL/pgSQL code, like assigning a value to a variable or execution of an SQL query. The execution order within a PL/pgSQL code block depends on the order of statements in that code block. The bulk of statements are placed within the main operation section of a code block, which is after the BEGIN keyword and before the END keyword. The declarative statements are placed in the DECLARE section after the DECLARE keyword. The declarative statements are used to declare and/or initialize variables that will be referenced within the code block. Every statement should end with a semicolon (;) .

Expressions

Expressions are calculations or operations that return their results as one of the PostgreSQL's base data types. An *example* of an expression is a:= b + c where the result of addition of variables c, b is assigned to another variable a.

Example 4: A simple PL/pgSQL function that assigns the result of an addition of two integers to result and returns it.

```
Create function example_function() returns integer as
'Declare
     result integer;
 begin
    result :=  5 + 4;
    return result;
end;
'language 'plpgsql';
```

Example 5: Shows the output of selecting the function in plsql.

```
Sydb2# select example_function()as output;
Output
---------
    9
(1 row)
```

Using Variables

Variables are used within a PL/pgSQL code block to store modifiable data of an explicitly stated type. All variables that are to be used within a code block needs to be declared under the DECLARE keyword. Variables may be initialized during its declaration; if not, then its initialized to the SQL NULL type.

Note As you will see later, the FOR loop's iterator variable does not have to be pre-declared in the DECLARE section for the block the loop is located within.

User defined functions can be written anywhere,

Variables in PL/pgSQL can be represented by any of SQL's standard types, as given below in the list:

i. Boolean
ii. Text
iii. Char
iv. Integer
v. Double precision
vi. Date
vii. Time

Declaration

In order to make the variables available to a code block, it must be declared within the declaration section of the block, denoted by the DECLARE keyword. A variable declare in a block will be available to all the sub-blocks within it. The scope of a declared variable ends when the block ends; hence variables declared within a sub-block are destroyed when that sub-block ends, and are not available to their parent blocks.

The format for declaring a variable is

```
variable_name data_type [ := value ];
```

Example 6: Shows the declaration of variables of different types in a code block.

```
Create function variable_example() returns integer as
 'Declare
        Roll_no integer;
        Name varchar(30);
        Marks float;
Begin
     Statements;
End;
'language 'plpgsql';
```

Additional options can also be specified during variable declaration. A CONSTANT keyword with declaration shows that the variable created is a constant. The NOT NULL keyword indicates that the variable cannot be set as NULL; hence a default value must be provided for any variable that is declared as NOT NULL.

The DEFAULT keyword is used to provide a default value for a variable. We can also use the := operator and assign a value during declaration, instead of using the DEFAULT keyword. The following illustrates the use of these options within a variable declaration:

```
Variable_name [ CONSTANT ] data_type [ NOT NULL ] [{ DEFAULT|:=}value];
```

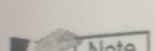
Assignment

PL/pgSQL provides the assignment operator (:=) for assigning a value to a variable. Default values can be assigned to variables during its declaration, using the assignment operator.

Example 7: Demonstrates the use of assignment operator for assigning default values to variables.

```
Create function variable_assignment() returns integer as
'Declare
      Roll_no integer := 1;
      Name varchar(30);
      Marks float;
Begin
     Statements;
End;
'language 'plpgsql' ;
```

The SELECT INTO clause can be used, along with the assignment operator, to assign the result of a query to a variable.



The above SELECT INTO is different from the SQL command SELECT INTO which assigns the results of a query into a new table. To assign the results of a query into a new table within PL/pgSQL, we can use the SQL syntax CREATE TABLE AS SELECT.

SELECT INTO is used to assign a row or a record information to variables, that are declared as %RECORD types. In order to use SELECT INTO with a normal variable, the variable must be declared as the same type as the column we referred in the SQL query statement provided.

Example 8: A function that demonstrates the use of a SELECT INTO statement.

```
Create function variable_select (text, text) returns integer as
'Declare
   -- declare aliases for user input
       Surname  alias for $1;
       Name  alias for $2;
        Customer_no integer;
Begin
  /* select customer id of the customer whose name and surname match
the inputs provided as arguments by the user */
Select into Customer_no  cust_id from customer where fname = Name and
sname = Surname;
-- return the id number.
 return  Customer_no;
End;
'language 'plpgsql' ;
```

The above function will be executed or called from the plsql as follows:

```
Sydb2# select variable_select('Aparna', 'Joshi');
```

Example 9: Demonstrates the select of multiple values into multiple variables, and returning the values, from a function.

```
Create function multiple_variable_select (integer) returns text as
'Declare
    -- declare alias for user input
        Customer_no alias for $1;
        Name  text;
         Surname text;
Begin
   /* select name and surname of the customer whose id matches the
input provided as argument by the user */
Select into Name, Surname cust_name, cust_surname from customer where
cust_id = Customer_no ;
-- return the name and surname.
 return  Name ||"    "|| Surname;
End;
'language 'plpgsql';
```

Note You may use the special FOUND Boolean variable directly after the SELECT INTO statement to check whether the variable/s have been populated with the retrieved values. Alternatively we can also use the ISNULL or IS NULL to find out if the specified variable is NULL after being selected into. Of course the ISNULL, FOUND and the IS NULL should be used within a conditional (IF/THEN) statement.

Example 10: Demonstrates the use of the FOUND Boolean.

```
Create function multiple_variable_select (integer) returns text as
'Declare
    -- declare alias for user input
        Customer_no alias for $1;
        Name text;
        Surname text;
Begin
   /* select name and surname of the customer whose id matches the
input provided as argument by the user */
Select into Name, Surname  cust_name, cust_surname from customer where
cust_id = Customer_no ;
-- if a match could not be found then return -1
If not found then
  return -1;
end if;
-- return the name and surname.
 return  Name ||"    "|| Surname;
End;
'language 'plpgsql';
```

Argument Variables

PL/pgSQL functions can accept argument variables of different types. Function arguments allow a user to pass information into a function that the function may need. The usage of PL/pgSQL functions become more effective and meaningful, through the use of arguments. Users pass arguments to functions, during the function call, by including them within parenthesis, separated by commas.

The order in which the arguments are passed during function call, must be the same as the order of arguments provided during the function definition.

Each function argument that is received by a function is incrementally assigned to an identifier that begins with the dollar ($) sign and is labelled with the argument number. Thus the identifier $1 is used for the first argument, $2 is used for the second argument and so forth. The maximum number of arguments that can be processed is sixteen, hence the argument identifiers can range from $1 to $16.

Example 11: Demonstrates the concept of function arguments. It accepts an integer argument and returns the double of it.

```
Create function func_arguments (integer) returns integer as
'Declare
 Begin
   -- returns the double of the argument passed
   Return  $1 * $1;
End;
'language 'plpgsql';
```

When a large number of arguments is to be passed to a function, it becomes difficult to remember or access the arguments using the $ concept, within the function code.

PL/pgSQL provides an easy mechanism.

Thus, to help in functions where the ability to better distinguish argument variables from one another is needed, PL/pgSQL allows us to create variable aliases.

Aliases are created using the ALIAS keyword and it gives us the ability to designate an alternate variable to use when referencing argument variables. All aliases should be declared within the DECLARE section of a block before they are used.

Example 12: Shows the syntax of the ALIAS keyword.

```
Create function alias_demo (text, integer) returns integer as
'Declare
   Arg1 alias for $1;
   Arg2 alias for $2;
Begin
   Statements;
End;
  'language 'plpgsql';
```

Returning Variables

PL/pgSQL functions must return values that matches the data type specified as the return type during the function definition. Values are returned from a function using the RETURN statement.

A RETURN statement is typically located at the end of a function, but may not be always.

Example 13: Demonstrates the syntax of the RETURN statement.

```
Create function function_identifier (arguments) return type
as
'Declare
    Declarations;
Begin
    Statements;
   Return { variable_name | value}
End;
'language 'plpgsql';
```

Attributes

PL/pgSQL provides variable attributes that basically assists or helps the database programmer, when working with database objects. These attributes are %TYPE and %ROWTYPE. These attributes are used to declare variables to match the type of a database object (using the %TYPE attribute) or to match the row structure of a row (with the %ROWTYPE attribute). A variable is declared using an attribute when it will be used within the code block to hold values taken from a database object. If a database object's type changes in the future, the variable's type will automatically change to that data type, without the need for any extra code.

The %TYPE attribute

The %TYPE attribute is used to declare a variable with the data type of a referenced database object (a table column). The format for declaring a %TYPE variable is shown below:

```
Variable_name table_name.column_name%TYPE ;
```

Example 14: Demonstrates a function that uses %TYPE to store the name of a customer.

```
Create function type_example (integer) returns text as
'Declare
  -- declare an alias for the function argument.
      Cust_no alias for $1;
 /* declare a variable that has the same data type as the cust_name
field in the customer table. */
     Name customer.cust_name%type;

Begin
   Select into Name cust_name from customer where custno = Cust_no;
   Return Name;

End;
'language 'plpgsql';
```

The %ROWTYPE attribute

%ROWTYPE attribute is used to declare a PL/pgSQL record variable to have the same structure as the rows in a table, that we specify in the function block. This is similar to a RECORD data type, but with %ROWTYPE it will have the exact structure of a table's row, whereas a RECORD variable is not structured and will accept a row from any table.

Example 15: Demonstrates the %ROWTYPE attribute usage.

```
Create function rowtype_demo (integer) returns text as
'Declare
    Cust_no   alias for $1;
/* declare a variable to have the same row structure as a record from
the customer  table */
    Crec customer%rowtype;
Begin
/* We use * in select into statement to denote selection of the entire
matching row into our Crec variable */
Select into Crec * from customer where custno = Cust_no;
/* We use the dot ( . ) within the Crec variable  to reference a named
field value in Crec . We assume the customer schema to be
customer(fname: varchar(20),lname : varchar(20) ,address : text )*/
Return Crec.fname || "   " || Crec.lname || "   " || Crec.address;
End ;
'language 'plpgsql';
```

Concatenation

Concatenation is the process of combining two or more strings together into one string. It is a standard built-in operator in PL/pgSQL. Concatenation can be used only with character strings, by placing the concatenation operator (||) between them. Multiple strings can be combined in this manner to form complex character strings.

Concatenation is used only when we need to return a string or when we need to assign a new value to a string variable, as shown in *example 16*.

Example 16: Demonstration of returning multiple strings from a function.

```
Create function concat_demo() returns text as
'Declare
        Word1 varchar default:= 'abcd';
        Word2 varchar default:= 'efgh';
     Begin
 /* we concatenate word1 and word2 with spaces between them, followed
by a string constant 'example'. So the output that we get will be
abcd  efgh  example */
        Return word1 || "   " || word2 || "  " || 'example';
End;
'language 'plpgsql';
```

2. Controlling Program Flow

Most programming languages provide different ways of controlling the flow of execution, within a program. PL/pgSQL also provides various statements using which a programmer can control the way actions will be executed within a PL/pgSQL code block. PL/pgSQL provides conditional statements and control loops for the same.

2.1 Conditional Statements

A conditional statement specifies an action / actions that should be executed instead of continuing execution of the function, based on the result of logical condition specified within the statement.

The IF/THEN Statement

The IF/THEN statement allows a user to specify a statement (or block of statements) that should be executed if a given condition evaluates true.

The Syntax of IF/THEN is as follows:

```
If<Condition>Then
   Statement;
      [..]
Else
      Statement;
   [..]
End if;
```

Example

```
Declare
X Number(2) :=50;
Y Number(2);
Begin
If X > 30 Then
   Y := X + 2;
Else
   Y :=0;
End if;
End;
```

Nested if Statement (If – Else if)

Syntax

```
If <Condition1> Then
Statement;
[..]
Else if <Condition 2> Then
   Statement;
   [….]
Else if <Condition 3> Then
   Statement;
   [……]
   Else
 Statement;
      [….]
 End if;
  End if;
End if;
```

2.2 Loops

Loops are another way of controlling the execution flow within a function. Loops use iteration in different ways and hence through this iteration, the functionality of a PL/pgSQL block can be greatly increased.

PL/pgSQL implements three iterative loops. The basic loop. The WHILE loop and the FOR loop.

The Basic Loop

We use the LOOP keyword to indicate the starting of a basic, unconditional loop within a function. An unconditional loop will execute the statements within its body, until an exit condition is met. The EXIT keyword can be used to form an exit statement. The EXIT keyword is accompanied by WHEN, followed by an expression to specify when the loop should exit. The expression is a Boolean expression.

The syntax for a basic loop is given as

```
LOOP
Statement;
   [.....]
END LOOP;
```

An unconditional loop will continue to execute in loop until it reaches an exit statement. EXIT statements explicitly terminate unconditional loops. An optional label and/or a condition can be specified, on which the loop should exit from.

A label is an arbitrary identifier, prefixed with a pair of less than symbols (<<) and suffixed with a pair of greater than symbols (>>). In case of a loop, the label can be placed directly before the loop block begins to identify that loop block with a chosen label, as shown below.

```
<<label_name>>
LOOP
   [.....]
END LOOP;
```

The advantage of using a label is that we can specify which loop to exit, in case of nested loops. We provide a condition in an exit statement , which specifies that the loop should be terminated when the condition is true.

Following is the syntax for an EXIT statement, within a LOOP:

```
[<< label >>]
LOOP
  Statement;
    [.....]
  EXIT [ label ] [WHEN condition];
END LOOP;
```

Example 17: Demonstrates a basic loop, with exit.

```
Create function loop_demo() returns integer as
'Declare
  Cube-num integer:= 2;
Begin
  Loop
    Cube-num := Cube-num + 2;
  Exit When Cube-num > 20;
End loop;
  Return Cube-num;
End; 'language 'plpgsql';
```

The WHILE Loop

The WHILE loop is used to loop through a block of statements until a specified condition becomes false. Each time a while loop is entered, its condition will be evaluated before the statement block is executed. If the condition evaluates to true, the statements will be executed. The syntax of a while loop is given as

Syntax

```
[ <<label>>  ]
While Condition
loop
  Statement;
  [.....]
End loop;
```

The While...Loop requires the keywords 'Loop' and 'End Loop' in order to designate the statements to execute. The While... Loop enables to evaluate a condition before a sequence of statements would be executed.

Example 18: Demonstrates the WHILE loop.

```
Create function while_demo() returns integer as
'Declare
    V-cal Number:=0;
Begin
 While V-cal <= 10
  loop
     V-cal:= V-cal + 1;
  End loop;
 Return V-cal;
End; 'language 'plpgsql';
```

The FOR loop

The FOR loop is one of the most important loop implemented by PL/pgSQL. The for loop is used to iterate over a block of statements, over a range of integers that is specified. The structure of a FOR loop is similar to the FOR loops in other procedural languages like C.

In PL/pgSQL FOR loop, an integer variable is stated first(the iterator), to track the iteration of the loop, then the iterator range is given, and finally a statement block is provided. The iterator variable is destroyed once the control exits from the loop. The iterator need not be declared within the declaration section of the block.

The syntax of the FOR loop is as follows

```
[ <<label>> ]
For Loop-Index In {Reverse} expression1 ..... expression2
Loop
Statement;
[.....]
End loop;
```

The Loop–Index is a local variable of type Integer. Reverse allows us to execute the loop in Reverse order.

The expression1 and expression2 can be constants or they can be variables.

Note The identifier used to track iteration (iterator) does not need to be declared outside the FOR block, unless we need to access the value of the iterator after the loop has finished.

The FOR loop can also be used to cycle through the results of a query. The syntax of the FOR loop that iterates through RECORD and %ROWTYPE variables is shown below:

```
[<<label>>]
For {record_variable | %rowtype_variable } IN
select_statement
LOOP
   Statement;
   [.....]
END LOOP;
```

Example 19: Demonstrates the use of FOR loops to iterate through the results of an SQL query.

In each iteration of the for loop, it places the contents of a result row from a query against a student(stud_id, name,address) table into the row_data variable, and then inserts the value of the row's title field into the stud_info variable.

```
Create function for_demo(varchar) returns text as
'Declare
studaddr  alias for $1;
      /* declare a variable to hold student names and set its default
value to a new line. */
    stud_info   text := ' '\n'';
  --declare a variable to hold rows from the student table.
     row_data student%Rowtype;
Begin
  --iterate through the results of a query.
  For row_data in select * from student
      where address = studaddr  order by name
    Loop
/*insert the name of the matching student into the stud_info variable*/
     stud_info :=  stud_info || row_data.name || ''\n'';
     end loop;
      -- return the list of students.
     Return stud_info;
    End; 'language 'plpgsql';
```

3. Views

While working with SQL, we often come across situations where we may want our SQL statements to be re-usable. This is especially the case when working with large or intricate queries. Further it can be highly inefficient to pass large queries over a network to the postgresql server for commonly executed routines.

1

Oct. 2018 – *3M*
What is a view? Explain with example how views can be used.

In such cases, views come handy. Views can be thought of as stored queries, which allows us to create a database object that functions very similarly to a table, but whose contents are dynamically and directly reflective only of the rows which it is defined to select. Views are very flexible; you may use a view to address common simple queries to a single table, as well as for extraordinarily complicated ones which may span across several tables.

Creating a View

```
CREATE [OR REPLACE] [TEMP | TEMPORARY] VIEW name [(column_name[, ...])]
```

AS *query*

CREATE VIEW defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

CREATE OR REPLACE VIEW is similar, but if a view of the same name already exists, it is replaced. The new query must generate the same columns that were generated by the existing view query (that is, the same column names in the same order and with the same data types), but it may add additional columns to the end of the list. The calculations giving rise to the output columns may be completely different.

1

Oct. 2017 – *5M*
How to create a view? Explain with an example.

If a schema name is given (for example, CREATE VIEW myschema.myview ...) then the view is created in the specified schema. Otherwise it is created in the current schema. Temporary views exist in a special schema, so a schema name cannot be given when creating a temporary view. The name of the view must be distinct from the name of any other view, table, sequence, index or foreign table in the same schema parameters.

Temporary or Temp

If specified, the view is created as a temporary view. Temporary views are automatically dropped at the end of the current session. Existing permanent relations with the same name are not visible to the current session while the temporary view exists, unless they are referenced with schema-qualified names.

If any of the tables referenced by the view are temporary, the view is created as a temporary view (whether TEMPORARY is specified or not).

name

The name (optionally schema-qualified) of a view to be created.

column_name

An optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.

query

A SELECT command which will provide the columns and rows of the view.

Note Currently, views are read only: the system will not allow an insert, update, or delete on a view. We can get the effect of an updatable view by creating INSTEAD triggers on the view, which must convert attempted inserts, etc. on the view into appropriate actions on other tables.

The DROP VIEW statement is used to drop views.

Examples

Create a view consisting of all comedy films:

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE type = 'Comedy';
```

This will create a view containing the columns that are in the film table at the time of view creation. Though * was used to create the view, columns added later to the table will not be part of the view.

Alter view

Alter view statement is used to change the definition of a view.

Syntax

```
ALTER VIEW name ALTER [ COLUMN ] column SET DEFAULT expression
ALTER VIEW name ALTER [ COLUMN ] column DROP DEFAULT
```

Description

ALTER VIEW changes various auxiliary properties of a view. (If you want to modify the view's defining query, use CREATE OR REPLACE VIEW.)

Parameters

name

The name (optionally schema-qualified) of an existing view.

SET/DROP DEFAULT

These forms set or remove the default value for a column. A default value associated with a view column is inserted into INSERT statements on the view before the view's ON INSERT rule is applied, if the INSERT does not specify a value for the column.

`new_owner`

The user name of the new owner of the view.

`new_name`

The new name for the view.

`new_schema`

The new schema for the view.

Examples

To rename the view foo to bar:

```
ALTER VIEW foo RENAME TO bar;
DROP VIEW
DROP VIEW -- remove a view
```

Syntax

```
DROP VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP VIEW drops an existing view. To execute this command you must be the owner of the view.

Parameters

`IF EXISTS`

Do not throw an error if the view does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the view to remove.

CASCADE

Automatically drop objects that depend on the view (such as other views).

RESTRICT

Refuse to drop the view if any objects depend on it. This is the default.

Examples

This command will remove the view called kinds:

```
DROP VIEW kinds;
```

4. Stored Functions in PL/pgSQL

Stored functions are user defined functions that are created using the CREATE FUNCTION statement. The functions, thus created, are called stored functions because they are stored as database objects within the PostgreSQL database.

The CREATE FUNCTION command names the new function, states its arguments and return type. Thus the create function calls creates a new function and stores it as an object in the Postgresql database.

Since they are stored as database objects, you can always delete them from your database, using the DROP FUNCTION statement, provided by pgsql.

We have seen several examples of these stored functions.

5. Handling Errors and Exceptions

The RAISE statements raise errors and exceptions during a PL/pgSQL function's execution. PL/pgSQL provides an error logging facility, the elog mechanism, which typically logs data to /var/log/message or to $PGDATA/serverlog, in addition to displaying to stderr.

2

Oct. 2018 – *3M*
What are the different levels used with the RAISE statement?

Oct. 2017 – *4M*
Explain the use of RAISE statement with an example.

A Raise is also given the level of error it should raise and the string error message it should send to postgreSQL. The string can also be embedded with variables and expressions, which one needs to list along with the error message.

The percent (%) sign is used as the place holder for the variables that are inserted into the string.

The syntax of the RAISE statement is as follows:

```
RAISE level ''message string '' [, identifier [….]];
```

The *table 1.2* lists the three possible values for the RAISE statement's level, along with their meanings:

Table 1.2: Error levels in Raise statement

Value	Explanation
DEBUG	Debug level statements send the specified text as a debug message to the PostgreSQL log.
NOTICE	Notice level statements send the specified text as a Notice.
EXCEPTION	Exception level statements send the specified text as an ERROR. The exception level also causes the current transaction to be aborted.

In below *example 20,* the first raise statement gives a debug level message. The second and third raise statements send a notice to the user. The fourth raise statement displays an error and throws an exception, causing the function to end and the transaction to be aborted.

Example 20: Demonstrates the RAISE statement usage.

```
Create function raise_demo() returns integer as
'Declare
     Int_var integer:=1;
Begin
      -- raise a debug level message
       Raise debug ''the raise_demo function began'';
 Int_var:= int_var+1;
--raise a notice stating the change in value of variable
Raise notice '' variable int_var's value is now % .'',int_var;
--raise an exception
Raise exception ''variable % changed. Transaction aborted.'',int_var;
Return 1;
End;
'language 'plpgsql';
```

Below is the result of executing the function raise_demo() given in *example 20.*

```
Sydb2 # select raise_demo();
```

NOTICE: variable int_var was changed.

NOTICE: variable int_var's value is now 2.

ERROR: variable 2 is changed. Transaction aborted.

Calling Functions

The general syntax to call another PL/pgSQL function from within PL/pgSQL is to either reference the function in an SQL statement, or during the assignment of a variable, as given below.

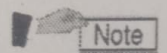
```
Select function_identifier(arguments);
Variable_id := function_identifier(arguments);
```

All functions in PL/pgSQL must return a value of some type. Hence it can be called in the above manner.

Other than above, we can also use the PERFORM keyword to call a function and ignore its return data. The syntax of the PERFORM keyword is as follows:

```
PERFORM function_identifier(arguments);
```

In the *example 21*, the function perform_demo(), we call another function get_stud_id(varchar) which takes a student name as input and returns the corresponding student id. Then we use the Perform keyword and call another function dump_id(integer) which will insert this id into a backup student table.



Students are asked to write both the referenced functions.

Example 21: Demonstrates the use of the PERFORM keyword to call a PL/pgSQL function, and shows how to call another PL/pgSQL function through assignment (using a select into statement).

```
Create function perform_demo(varchar) returns integer as
'Declare
      Studname alias for $1;
      Studentid integer;
  Begin
       Select into studentid get_stud_id(studname);
        /* if student doesn't exist, the function returns -1 */
    If Studentid = -1 then
       Return -1;
    End if;
/*add the Studentid to backuptable through the function dump_id() */
Perform dump_id(Studentid);
Return 1;
End; 'language 'plpgsql';
```