CS232L – Database Management System

LAB 07

PL/PG SQL BASICS – BLOCK PROCESSING, IF ELSE, CASE WHEN, LOOPS

Prepared By: Amna Arooj FCSE Computer Engineer

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS232L – Database Management system Lab

Lab 7 - PL/PG SQL BASICS – BLOCK PROCESSING, IF ELSE, CASE WHEN, LOOPS

Objective

The objective of this session is to learn how to use the PostgreSQL procedural programming features i.e. PL/PG SQL block structure, different conditional statements, Iterative structures(Loops) and case when statements.

Instructions

- Open the handout/lab manual in front of a computer in the lab during the session.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering problems, indicate the commands you entered, and the output displayed.
- Try to practice and revise all the concepts covered in all previous session before coming to the lab to avoid un-necessary ambiguities.

Overview of PostgreSQL PL/pgSQL

PL/pgSQL is a procedural programming language for the PostgreSQL database system. PL/pgSQL allows you to extend the functionality of the PostgreSQL database server by creating server objects with complex logic.
PL/pgSQL was designed to :

- Create user-defined functions, stored procedures, and triggers.
- Extend standard SQL by adding control structures such as if, case, and loop statements.
- Inherit all user-defined functions, operators, and types.

Since PostgreSQL 9.0, PL/pgSQL is installed by default.

Advantages of using PL/pgSQL

SQL is a query language that allows you to query data from the database easily. However, PostgreSQL only can execute SQL statements individually.
It means that you have multiple statements, you need to execute them one by one like this:

- First, send a query to the PostgreSQL database server.
- Next, wait for it to process.
- Then, process the result set.
- After that, do some calculations.
- Finally, send another query to the PostgreSQL database server and repeat this process.

This process incurs the interprocess communication and network overheads.
To resolve this issue, PostgreSQL uses PL/pgSQL.
PL/pgSQL wraps multiple statements in an object and store it on the PostgreSQL database server.
So instead of sending multiple statements to the server one by one, you can send one statement to execute the object stored in the server. This allows you to:

- Reduce the number of round trips between the application and the PostgreSQL database server.
- Avoid transferring the immediate results between the application and the server.

PL/pgSQL Blocks

PL/pgSQL is a block-structured language, therefore, a PL/pgSQL function or stored procedure is organized into blocks.

The following illustrates the syntax of a complete block in PL/pgSQL:

```
[ <<label>> ]
[ declare
    declarations ]
begin
    statements;
       ...
end [ label ];
```

Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

Let’s examine the block structure in more detail:

- Each block has two sections: declaration and body. The declaration section is optional while the body section is required. A block is ended with a semicolon (;) after the END keyword.
- A block may have an optional label located at the beginning and at the end. You use the block label when you want to specify it in the EXIT statement of the block body or when 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 place the code. Each statement in the body section is also terminated with a semicolon (;).

PL/pgSQL block structure example

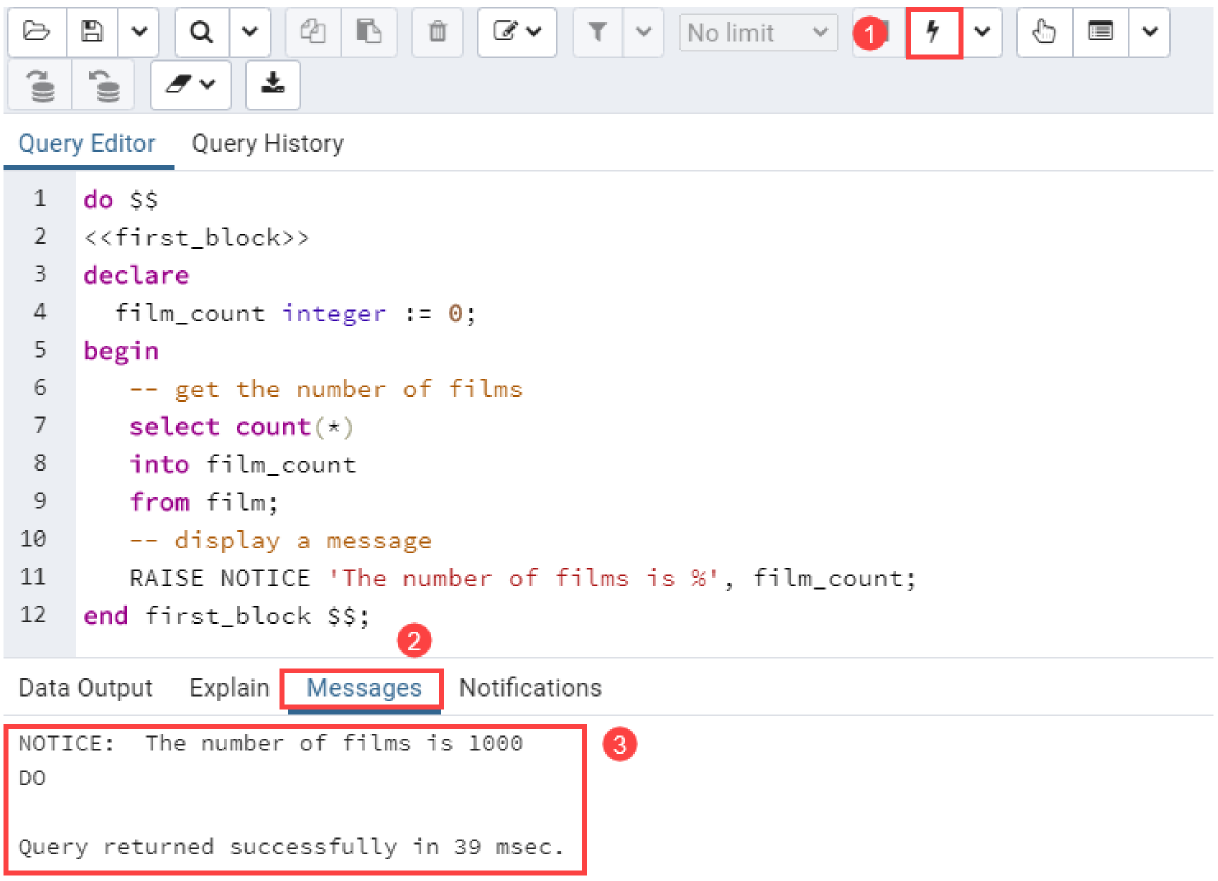
The following example illustrates a very simple block. It is called an anonymous block.

```
do $$
<<first_block>>
declare
  film_count integer := 0;
begin
   -- get the number of films
   select count(*)
   into film_count
   from film;
   -- display a message
   raise notice 'The number of films is %', film_count;
end first_block $$;
```

Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

```
NOTICE:  The current value of counter is 1
```

To execute a block from pgAdmin, you click the Execute button as shown in the following picture:



```
do $$
<<first_block>>
declare
  film_count integer := 0;
begin
   -- get the number of films
   select count(*)
   into film_count
   from film;
   -- display a message
   RAISE NOTICE 'The number of films is %', film_count;
end first_block $$;
```

```
NOTICE:  The number of films is 1000
DO

Query returned successfully in 39 msec.
```

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.

However, we used the dollar-quoted string constant syntax to make it more readable.

In the declaration section, we declared a variable film_count and set its value to zero.

```
film_count integer := 0;
```
Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

Inside the body section, we used a select into statement with the count() function to get the number of films from the film table and assign the result to the film_count variable.

```
select count(*)
into film_count
from film;
```
Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

After that, we showed a message using raise notice statement:

```
raise notice 'The number of films is %', film_count;
```
Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

The % is a placeholder that is replaced by the content of the film_count variable.

Conditional Statements:

PL/pgSQL provides you with three forms of the if statements.

- if then
- if then else
- if then elsif

1) PL/pgSQL if-then statement

The following illustrates the simplest form of the if statement:

```
if condition then
   statements;
end if;
```

The if statement executes statements if a condition is true. If the condition evaluates to false, the control is passed to the next statement after the END if part. The condition is a boolean expression that evaluates to true or false. The statements can be one or more statements that will be executed if the condition is true. It can be any valid statement, even another if statement. When an if statement is placed inside another if statement, it is called a nested-if statement.

See the following example:

```
do $$
declare
  selected_film film%rowtype;
  input_film_id film.film_id%type := 0;
begin

  select * from film
  into selected_film
  where film_id = input_film_id;

  if not found then
     raise notice'The film % could not be found',
                    input_film_id;
  end if;
end $$
```

In this example, we selected a film by a specific film id (0).

The found is a global variable that is available in PL/pgSQL procedure language. If the select into statement sets the found variable if a row is assigned or false if no row is returned.

We used the if statement to check if the film with id (0) exists and raise a notice if it does not.

```
ERROR:  The film 0 could not be found
```
Code language: Shell Session (shell)

If you change the value of the input_film_id variable to some value that exists in the film table like 100, you will not see any message.

2) PL/pgSQL if-then-else statement

The following illustrates the syntax of the if-then-else statement:

```
if condition then
  statements;
else
  alternative-statements;
END if;
```

The if then else statement executes the statements in the if branch if the condition evaluates to true; otherwise, it executes the statements in the else branch.

See the following example:

```
do $$
declare
  selected_film film%rowtype;
  input_film_id film.film_id%type := 100;
begin

  select * from film
  into selected_film
  where film_id = input_film_id;

  if not found then
     raise notice 'The film % could not be found',
           input_film_id;
  else
     raise notice 'The film title is %', selected_film.title;
  end if;
end $$
```

In this example, the film id 100 exists in the film table so that the FOUND variable was set to true. Therefore, the statement in the else branch executed.

Here is the output:

```
NOTICE:  The film title is Brooklyn Desert
```

3) PL/pgSQL if-then-elsif Statement

The following illustrates the syntax of the if then elsif statement:

```
if condition_1 then
  statement_1;
elsif condition_2 then
  statement_2
...
```

```
elsif condition_n then
  statement_n;
else
  else-statement;
end if;
```
Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

The if and ifthen else statements evaluate one condition. However, the if then elsif statement evaluates multiple conditions.

If a condition is true, the corresponding statement in that branch is executed.

For example, if the condition_1 is true then the if then ELSif executes the statement_1 and stops evaluating the other conditions.

If all conditions evaluate to false, the if then elsif executes the statements in the else branch.

Let’s look at the following example:

```
do $$
declare
   v_film film%rowtype;
   len_description varchar(100);
begin

  select * from film
  into v_film
  where film_id = 100;

  if not found then
     raise notice 'Film not found';
  else
      if v_film.length >0 and v_film.length <= 50 then
             len_description := 'Short';
        elsif v_film.length > 50 and v_film.length < 120 then
             len_description := 'Medium';
        elsif v_film.length > 120 then
             len_description := 'Long';
        else
             len_description := 'N/A';
        end if;

        raise notice 'The % film is %.',
           v_film.title,
           len_description;
  end if;
end $$
```

How it works:

- First, select the film with id 100. If the film does not exist, raise a notice that the film is not found.
- Second, use the if then elsif statement to assign the film a description based on the length of the film.

PL/pgSQL CASE Statement

Besides the if statement, PostgreSQL provides you with case statements that allow you to execute a block of code based on a condition.
The case statement selects a when section to execute from a list of when sections based on a condition. The case statement has two forms:

- Simple case statement
- Searched case statement

Notice that you should not confuse about the case statement and case expression.
The case expression evaluates to a value while the case statement selects a section to execute based on condition.

1) Simple case statement
Let’s start with the syntax of the simple case statement:

```
case search-expression
  when expression_1 [, expression_2, ...] then
    when-statements
 [ ... ]
 [else
    else-statements ]
END case;
```

The search-expression is an expression that evaluates to a result. The case statement compares the result of the search-expression with the expression in each when branch using equal operator (=) from top to bottom. If the case statement finds a match, it will execute the corresponding when section. Also, it stops comparing the result of the search-expression with the remaining expressions. If the case statement cannot find any match, it will execute the else section.
The else section is optional. If the result of the search-expression does not match expression in the when sections and the else section does not exist, the case statement will raise a case_not_found exception. The following is an example of the simple case statement.

```
do $$
declare
        rate   film.rental_rate%type;
        price_segment varchar(50);
begin
   -- get the rental rate
   select rental_rate into rate
   from film
   where film_id = 100;

        -- assign the price segment
        if found then
                case rate
```

```
                        when 0.99 then
            price_segment =  'Mass';
                        when 2.99 then
            price_segment = 'Mainstream';
                        when 4.99 then
            price_segment = 'High End';
                        else
                       price_segment = 'Unspecified';
                        end case;
                     raise notice '%', price_segment;
   end if;
end; $$
```

Output:

```
NOTICE:  High End
```

This example first selects the film with id 100. Based on the rental rate, it assigns a price segment to the film that can be mass, mainstream, or high end. In case the price is not 0.99, 2.99 or 4.99, the case statement assigns the film the price segment as unspecified.

2) Searched case statement

The following syntax shows syntax of the searched case statement:

```
case
   when boolean-expression-1 then
     statements
  [ when boolean-expression-2 then
     statements
   ... ]
  [ else
     statements ]
end case;
```

In this syntax, the case statement evaluates the boolean expressions sequentially from top to bottom until it finds an expression that evaluates to true
Once it finds an expression that evaluates to true, the case statement executes the corresponding when section and immediately stops searching for the remaining expressions.
In case no expression evaluates to true, the case statement will execute the the else section.
The else section is optional. If you omit the else section and there is no expression evaluates to true, the case statement will raise the case_not_found exception.
The following example illustrates how to use a simple case statement:

```
do $$
declare
   total_payment numeric;
   service_level varchar(25) ;
begin
    select sum(amount) into total_payment
    from Payment
    where customer_id = 100;
```

```
          if found then
            case
                    when total_payment > 200 then
                          service_level = 'Platinum' ;
                   when total_payment > 100 then
                          service_level = 'Gold' ;
           else
                  service_level = 'Silver' ;
                  end case;
                  raise notice 'Service Level: %', service_level;
    else
            raise notice 'Customer not found';
          end if;
end; $$
```

How it works:
First, select the total payment paid by the customer id 100 from the payment table.
Then, assign the service level to the customer based on the total payment.

PL/pgSQL While Loop

The while loop statement executes a block of code until a condition evaluates to false.

```
[ <<label>> ]
while condition loop
   statements;
end loop;
```
Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

In this syntax, PostgreSQL evaluates the condition before executing the statements. If the condition is true, it executes the. After each iteration, the while loop evaluates the condition again. Inside the body of the while loop, you need to change the values of some variables to make the condition false or null at some points. Otherwise, you will have an indefinite loop. Because the while loop tests the condition before executing the statements, the while loop is sometimes referred to as a pretest loop.

PL/pgSQL while loop example

The following example uses the `while` loop statement to display the value of a counter:

```
do $$
declare
  counter integer := 0;
begin
  while counter < 5 loop
    raise notice 'Counter %', counter;
            counter := counter + 1;
  end loop;
end$$;
```
Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

Output:

```
NOTICE:  Counter 0
NOTICE:  Counter 1
NOTICE:  Counter 2
NOTICE:  Counter 3
NOTICE:  Counter 4
```

Code language: Shell Session (shell)

How it works.

- First, declare the counter variable and initialize its value to 0.
- Second, use the while loop statement to show the current value of the counter as long as it is less than 5. In each iteration, increase the value of counter by one. After 5 iterations, the counter is 5 therefore the while loop is terminated.

PL/pgSQL Loop Statements

The loop defines an unconditional loop that executes a block of code repeatedly until terminated by an exit or return statement.

The following illustrates the syntax of the loop statement:

```
<<label>>
loop
   statements;
end loop;
```

Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

Typically, you use an if statement inside the loop to terminate it based on a condition like this:

```
<<label>>
loop
   statements;
   if condition then
      exit;
   end if;
end loop;
```

Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

It’s possible to place a loop statement inside another loop statement. When a loop statement is placed inside another loop statement, it is called a nested loop:

```
<<outer>>
loop
   statements;
   <<inner>>
   loop
     /* ... */
     exit <<inner>>
   end loop;
end loop;
```

Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

When you have nested loops, you need to use the loop label so that you can specify it in the exit and continue statement to indicate which loop these statements refer to.

PL/pgSQL loop statement example

The following example shows how to use the loop statement to calculate the Fibonacci sequence number.

```
do $$
declare
   n integer:= 10;
   fib integer := 0;
   counter integer := 0 ;
   i integer := 0 ;
   j integer := 1 ;
begin
	if (n < 1) then
		fib := 0 ;
	end if;
	loop
		exit when counter = n ;
		counter := counter + 1 ;
		select j, i + j into i,  j ;
	end loop;
	fib := i;
    raise notice '%', fib;
end; $$
```

Code language: PostgreSQL SQL dialect and PL/pgSQL (pgsql)

Output:

```
NOTICE:  55
```

The block calculates the nth Fibonacci number of an integer (n).

By definition, Fibonacci numbers are a sequence of integers starting with 0 and 1, and each subsequent number is the sum of the two previous numbers, for example, 1, 1, 2 (1+1), 3 (2+1), 5 (3 +2), 8 (5+3), …

In the declaration section, the counter variable is initialized to zero (0). The loop is terminated when counter equals n. The following select statement swaps values of two variables i and j :

```
SELECT j, i + j INTO i, j ;
```