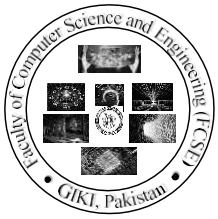
CS232L – Database Management System

LAB#08

PL/pgSQL — SQL Procedural Language

CURSORS, FUNCTIONS, PROCEDURES

Ghulam Ishaq Khan Institute of Engineering Sciences



Faculty of Computer Science & Engineering

CS232L – Database Management system Lab

Lab 08 – Postgres Cursors, Functions and Procedures

Objective

The objective of this session is to

1. Introduction of PostgreSQL Cursors
2. Functions
3. Procedures
4. Examples

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.

CURSORS

WHAT IS CURSOR?

Rather than executing a whole query at once, it is possible to set up a *cursor* that encapsulates the query, and process each individual row at a time.

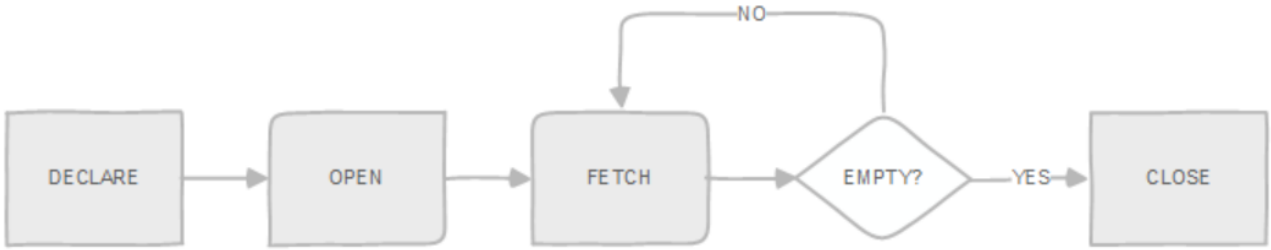
WHY WE USE CURSORS?

Typically, you use cursors when you want to divide a large result set into parts and process each part individually. Suppose if a table has 10 million or billion rows. While performing a SELECT operation on the table it will take some time to process the result and most likely give an "out of memory" error and the program will be terminated.

One reason for doing this is to avoid memory overrun when the result contains a large number of rows. (However, PL/pgSQL users do not normally need to worry about that, since FOR loops automatically use a cursor internally to avoid memory problems.)

On top of that, you can develop a function that returns a reference to a cursor. This is an effective 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.

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



1. First, declare a cursor.
2. Next, open the cursor.
3. Then, fetch rows from the result set into a 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.

1. Declaring cursors

To access to a cursor, you need to declare a cursor variable in the declaration section of a block. PostgreSQL provides you with a special type called REFCURSOR to declare a cursor variable.

```
declare my_cursor refcursor;
```

You can also declare a cursor that bounds to a query by using the following syntax:

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
   cur_films  cursor for
            select *
            from film;
   cur_films2 cursor (year integer) for
            select *
            from film
            where release_year = year;
```

The cur_films is a cursor that encapsulates all rows in the film table.

The cur_films2 is a cursor that encapsulates film with a particular release year in the film table.

2. Opening cursors

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

Details for unbounded cursors:

https://www.postgresqltutorial.com/postgresql-plpgsql/plpgsql-cursor/

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 cur_films and cur_films2 that we declared above:

open cur_films;

open cur_films2(year:=2005);

3. Using and Fetching 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 cur_films into row_film;

fetch last from row_film into title, release_year;

You can also
1. Move the cursor
2. Delete or update the row

4. 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 opened again using OPEN statement.

Example:

https://pgdocptbr.sourceforge.io/pg82/plpgsql-cursors.html

FUNCTIONS:

The create function statement allows you to define a new user-defined function.

The following illustrates the syntax of the create function statement:

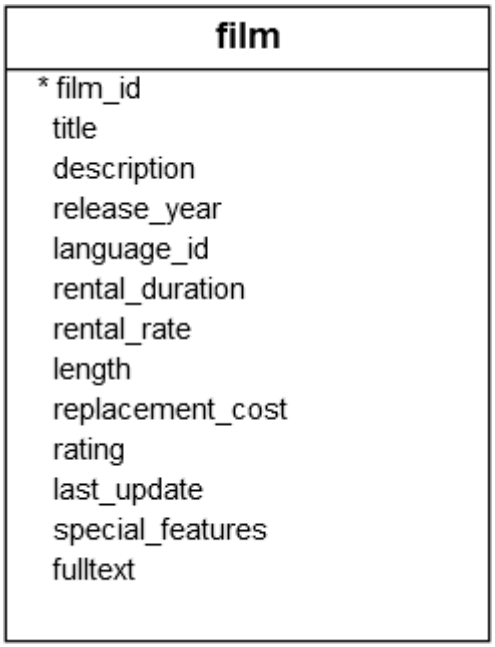
```
create [or replace] function function_name(param_list)
  returns return_type
  language plpgsql
 as
$$
declare
-- variable declaration
begin
 -- logic
end;
$$
```

In this syntax:

- First, specify the name of the function after the `create function` keywords. If you want to replace the existing function, you can use the `or replace` keywords.
- Then, specify the function parameter list surrounded by parentheses after the function name. A function can have zero or many parameters.
- Next, specify the datatype of the returned value after the `returns` keyword.
- After that, use the `language plpgsql` to specify the procedural language of the function. Note that PostgreSQL supports many procedural languages, not just `plpgsql`.
- Finally, place a block in the dollar-quoted string constant.

PostgreSQL Create Function statement examples

We'll use the film table from the dvdrental sample database.



The following statement creates a function that counts the films whose length between the `len_from` and `len_to` parameters:

```
create function get_film_count(len_from int, len_to int)
returns int
language plpgsql
as
$$
declare
  film_count integer;
```

```
begin
  select count(*)
  into film_count
  from film
  where length between len_from and len_to;

  return film_count;
end;
$$;
```

The function get_film_count has two main sections: header and body.

In the header section:

- First, the name of the function is get_film_count that follows the create function keywords.
- Second, the get_film_count() function accepts two parameters len_from and len_to with the integer datatype.
- Third, the get_film_count function returns an integer specified by the returns int clause.
- Finally, the language of the function is plpgsql indicated by the language plpgsql.

In the function body:

- Use the dollar-quoted string constant syntax that starts with $$ and ends with $$. Between these $$, you can place a block that contains the declaration and logic of the function.
- In the declaration section, declare a variable called film_count that stores the number of films selected from the film table.
- In the body of the block, use the select into statement to select the number of films whose length are between len_from and len_to and assign the result to the film_count variable. At the end of the block, use the return statement to return the film_count.

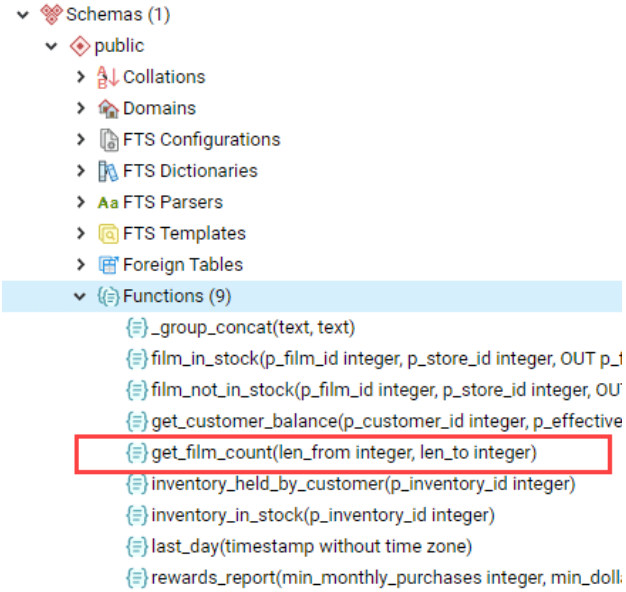
If everything is fine, you will see the following message:

```
CREATE FUNCTION

Query returned successfully in 44 msec.
```

It means that the function get_film_count is created successfully.

Finnally, you can find the function get_film_count in the **Functions** list:



Calling a user-defined function

PostgreSQL provides you with three ways to call a user-defined function:

- Using positional notation
- Using named notation
- Using the mixed notation.

1) Using positional notation

To call a function using the positional notation, you need to specify the arguments in the same order as parameters. For example:

```
select get_film_count(40,90);
```

Output:

```
 get_film_count
----------------
            325
(1 row)
```

In this example, the arguments of the get_film_count() are 40 and 90 that corresponding to the from_len and to_len parameters.

You call a function using the positional notation when the function has few parameters.

If the function has many parameters, you should call it using the named notation since it will make the function call more obvious.

2) Using named notation

The following shows how to call the get_film_count function using the positional notation:

```
select get_film_count(
    len_from => 40,
     len_to => 90
);
```

Output:

```
 get_film_count
----------------
            325
(1 row)
```

In the named notation, you use the => to separate the argument's name and its value.

For backward compatibility, PostgreSQL supports the older syntax based on := as follows:

```
select get_film_count(
    len_from := 40,
    len_to := 90
);
```

3) Using mixed notation

The mixed notation is the combination of positional and named notations. For example:

```
select get_film_count(40, len_to => 90);
```

Note that you cannot use the named arguments before positional arguments like this:

```
select get_film_count(len_from => 40, 90);
```

Error:

```
ERROR:  positional argument cannot follow named argument
LINE 1: select get_film_count(len_from => 40, 90);
```

PROCEDURES

So far, you have learned how to define user-defined functions using the create function statement.

A drawback of user-defined functions is that they cannot execute transactions. In other words, inside a user-defined function, you cannot start a transaction, and commit or rollback it.

PostgreSQL 11 introduced stored procedures that support transactions.

What are Stored Procedures in PostgreSQL?

Stored procedures in PostgreSQL are a collection of SQL commands manipulated to achieve a particular operation.

The benefits of using PostgreSQL Stored Procedures are immense. Just imagine deploying a new function every time a new use case arises. Instead, creating a stored procedure to later use in an app makes sense. Likewise, it is also an essential step while creating user-defined functions.

WHY TO USE PROCEDURES?

PostgreSQL Stored Procedures support procedural operations, which are helpful while building powerful database apps — it increases their **performance, productivity, and scalability**.

In short, developing custom functions becomes easier using the PostgreSQL Stored Procedures. Moreover, once created, you can deploy stored procedures in any app based on your requirements.

SYNTAX:

To define a new stored procedure, you use the create procedure statement.

The following illustrates the basic syntax of the create procedure statement:

```
create [or replace] procedure procedure_name(parameter_list)
language plpgsql
as $$
declare
-- variable declaration
begin
-- stored procedure body
end; $$Code language: SQL (Structured Query Language) (sql)
```

In this syntax:

- First, specify the name of the stored procedure after the create procedure keywords.

- Second, define parameters for the stored procedure. A stored procedure can accept zero or more parameters.
- Third, specify plpgsql as the procedural language for the stored procedure. Note that you can use other procedural languages for the stored procedure such as SQL, C, etc.
- Finally, use the dollar-quoted string constant syntax to define the body of the stored procedure.

Parameters in stored procedures can have the in and inout modes. They cannot have the out mode.

A stored procedure does not return a value. You cannot use the return statement with a value inside a store procedure like this:

```
return expression;Code language: JavaScript (javascript)
```

However, you can use the return statement without the expression to stop the stored procedure immediately:

```
return;Code language: SQL (Structured Query Language) (sql)
```

If you want to return a value from a stored procedure, you can use parameters with the inout mode.

EXAMPLES:

We will use the following accounts table for the demonstration:

```
drop table if exists accounts;

create table accounts (
    id int generated by default as identity,
    name varchar(100) not null,
    balance dec(15,2) not null,
    primary key(id)
);

insert into accounts(name,balance)
values('Bob',10000);

insert into accounts(name,balance)
values('Alice',10000);Code language: SQL (Structured Query Language) (sql)
```

The following statement shows the data from the accounts table:

```
select * from accounts;Code language: SQL (Structured Query Language) (sql)
```

	id integer	name character varying (100)	balance numeric (15,2)
1	1	Bob	10000.00
2	2	Alice	10000.00

The following example creates a stored procedure named transfer that transfers a specified amount of money from one account to another.

```
create or replace procedure transfer(
   sender int,
   receiver int,
   amount dec
)
language plpgsql
as $$
begin
    -- subtracting the amount from the sender's account
    update accounts
    set balance = balance - amount
    where id = sender;

    -- adding the amount to the receiver's account
    update accounts
    set balance = balance + amount
    where id = receiver;

    commit;
end;$$
```

Calling a stored procedure

o call a stored procedure, you use the CALL statement as follows:

```
call stored_procedure_name(argument_list);
```

More Details about procedures:

https://www.sqlshack.com/psql-stored-procedures-overview-and-examples/