K. J. Somaiya College of Engineering, Mumbai-77
(Autonomous College Affiliated to University of Mumbai)

Batch: A3 Roll No.: 1911034

Experiment / assignment / tutorial No. 7

Grade: AA / AB / BB / BC / CC / CD /DD

Title: Implementing procedures and cursors

Objective: To be able to Implementing procedures.

Expected Outcome of Experiment:
CO 3: Use SQL for Relational database creation, maintenance and query processing

Books/ Journals/ Websites referred:

1. Dr. P.S. Deshpande, SQL and PL/SQL for Oracle 10g.Black book, Dreamtech Press
2. www.db-book.com
3. Korth, Slberchatz, Sudarshan : "Database Systems Concept", 5th Edition , McGraw Hill
4. Elmasri and Navathe,"Fundamentals of database Systems", 4th Edition,PEARSON Education.

Resources used: Postgresql

Theory

A stored procedure is a set of Structured Query Language (SQL) statements with an assigned name, which are stored in a relational database management system as a group, so it can be reused and shared by multiple programs.

Stored procedures can access or modify data in a database, but it is not tied to a specific database or object, which offers a number of advantages.

Benefits of using stored procedures

A stored procedure provides an important layer of security between the user interface and the database. It supports security through data access controls because end users may enter or change data, but do not write procedures. A stored procedure preserves data integrity because information is entered in a consistent manner. It improves productivity because statements in a stored procedure only must be written once.

Use of stored procedures can reduce network traffic between clients and servers, because the commands are executed as a single batch of code. This means only the call to execute the procedure is sent over a network, instead of every single line of code being sent individually.

Syntax:

```
CREATE [ OR REPLACE ] PROCEDURE
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = }
default_expr ] [, ...] ] )
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY
DEFINER
    | SET configuration_parameter { TO value | = value | FROM
CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
```

Parameters

Name: The name (optionally schema-qualified) of the procedure to create.

Argmode: The mode of an argument: IN, INOUT, or VARIADIC. If omitted, the default is IN. (OUT arguments are currently not supported for procedures. Use INOUT instead.)

Argname: The name of an argument.

Argtype: The data type(s) of the procedure's arguments (optionally schema-qualified), if any. The argument types can be base, composite, or domain types, or can reference the type of a table column.

Depending on the implementation language it might also be allowed to specify "pseudo-types" such as cstring. Pseudo-types indicate that the actual argument type is either incompletely specified, or outside the set of ordinary SQL data types.

The type of a column is referenced by writing table_name.column_name%TYPE. Using this feature can sometimes help make a procedure independent of changes to the definition of a table.

default_expr: An expression to be used as default value if the parameter is not specified. The expression has to be coercible to the argument type of the parameter. All input parameters following a parameter with a default value must have default values as well.

lang_name :The name of the language that the procedure is implemented in. It can be sql, c, internal, or the name of a user-defined procedural language, e.g. plpgsql. Enclosing the name in single quotes is deprecated and requires matching case.

TRANSFORM { FOR TYPE type_name } [, ...] }

Lists which transforms a call to the procedure should apply. Transforms convert between SQL types and language-specific data types; see CREATE TRANSFORM. Procedural language implementations usually have hardcoded knowledge of the built-in types, so those don't need to be listed here. If a procedural language implementation does not know how to handle a type and no transform is supplied, it will fall back to a default behavior for converting data types, but this depends on the implementation.

[EXTERNAL] SECURITY INVOKER

[EXTERNAL] SECURITY DEFINER

SECURITY INVOKER indicates that the procedure is to be executed with the privileges of the user that calls it. That is the default. SECURITY DEFINER specifies that the procedure is to be executed with the privileges of the user that owns it.

The key word EXTERNAL is allowed for SQL conformance, but it is optional since, unlike in SQL, this feature applies to all procedures not only external ones.

A SECURITY DEFINER procedure cannot execute transaction control statements (for example, COMMIT and ROLLBACK, depending on the language).

configuration_parameter

value: The SET clause causes the specified configuration parameter to be set to the specified value when the procedure is entered, and then restored to its prior value when the procedure exits. SET FROM CURRENT saves the value of the parameter that is current when CREATE PROCEDURE is executed as the value to be applied when the procedure is entered.

If a SET clause is attached to a procedure, then the effects of a SET LOCAL command executed inside the procedure for the same variable are restricted to the procedure: the configuration parameter's prior value is still restored at procedure exit. However, an ordinary SET command (without LOCAL) overrides the SET clause, much as it would do for a previous SET LOCAL command: the effects of such a command will persist after procedure exit, unless the current transaction is rolled back.

If a SET clause is attached to a procedure, then that procedure cannot execute transaction control statements (for example, COMMIT and ROLLBACK, depending on the language).

Definition

A string constant defining the procedure; the meaning depends on the language. It can be an internal procedure name, the path to an object file, an SQL command, or text in a procedural language.

It is often helpful to use dollar quoting to write the procedure definition string, rather than the normal single quote syntax. Without dollar quoting, any single quotes or backslashes in the procedure definition must be escaped by doubling them.

obj_file, link_symbol

This form of the AS clause is used for dynamically loadable C language procedures when the procedure name in the C language source code is not the same as the name of the SQL procedure. The string obj_file is the name of the shared library file containing the compiled C procedure, and is interpreted as for the LOAD command. The string link_symbol is the procedure's link symbol, that is, the name of the procedure in the C language source code. If the link symbol is omitted, it is assumed to be the same as the name of the SQL procedure being defined.

When repeated CREATE PROCEDURE calls refer to the same object file, the file is only loaded once per session. To unload and reload the file (perhaps during development), start a new session.

Example:

We will use the following accounts table for the demonstration:

```
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);
```

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

```
CREATE OR REPLACE PROCEDURE transfer(INT, INT, DEC)
LANGUAGE plpgsql
AS $$
BEGIN
   -- subtracting the amount from the sender's account
   UPDATE accounts
   SET balance = balance - $3
   WHERE id = $1;

   -- adding the amount to the receiver's account
   UPDATE accounts
   SET balance = balance + $3
   WHERE id = $2;

   COMMIT;
END;
$$;


CALL stored_procedure_name(parameter_list);
CALL transfer(1,2,1000);
```

Cursors

Rather than executing a whole query at once, it is possible to set up a cursor that encapsulates the query, and then read the query result a few rows at a time. 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.) A more interesting usage is to return a reference to a cursor that a function has created, allowing the caller to read the rows. This provides an efficient way to return large row sets from functions.

Before a cursor can be used to retrieve rows, it must be opened. (This is the equivalent action to the SQL command DECLARE CURSOR.) PL/pgSQL has three forms of the OPEN statement, two of which use unbound cursor variables while the third uses a bound cursor variable.

OPEN FOR query

Syntax: OPEN unbound_cursorvar [[NO] SCROLL] FOR query;

example:

OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;

OPEN FOR EXECUTE

Syntax: OPEN unbound_cursorvar [[NO] SCROLL] FOR EXECUTE query_string
[USING expression [, ...]];

example:

OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident(tabname)
|| ' WHERE col1 = $1' USING keyvalue;

Opening a Bound Cursor

Syntax: OPEN bound_cursorvar [([argument_name :=] argument_value [, ...])];

Examples (these use the cursor declaration examples above):

OPEN curs2;

OPEN curs3(42);

OPEN curs3(key := 42);

Because variable substitution is done on a bound cursor's query, there are really two ways to pass values into the cursor: either with an explicit argument to OPEN, or implicitly by referencing a PL/pgSQL variable in the query. However, only variables declared before the bound cursor was declared will be substituted into it. In either case the value to be passed is determined at the time of the OPEN. For example, another way to get the same effect as the curs3 example above is

```
DECLARE
   key integer;
   curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
   key := 42;
```

OPEN curs4;

Using Cursors

FETCH

Synatx: FETCH [direction { FROM | IN }] cursor INTO target;

Examples:

FETCH curs1 INTO rowvar;

FETCH curs2 INTO foo, bar, baz;

FETCH LAST FROM curs3 INTO x, y;

FETCH RELATIVE -2 FROM curs4 INTO x;

MOVE

MOVE [direction { FROM | IN }] cursor;

MOVE repositions a cursor without retrieving any data. MOVE works exactly like the FETCH command, except it only repositions the cursor and does not return the row moved to. As with SELECT INTO, the special variable FOUND can be checked to see whether there was a next row to move to.

Examples:

MOVE curs1;

MOVE LAST FROM curs3;

MOVE RELATIVE -2 FROM curs4;

MOVE FORWARD 2 FROM curs4;

UPDATE/DELETE WHERE CURRENT OF

UPDATE table SET ... WHERE CURRENT OF cursor;

DELETE FROM table WHERE CURRENT OF cursor;

When a cursor is positioned on a table row, that row can be updated or deleted using the cursor to identify the row. There are restrictions on what the cursor's query can be (in particular, no grouping) and it's best to use FOR UPDATE in the cursor. For more information see the DECLARE reference page.

An example:

UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;

CLOSE

CLOSE cursor;

CLOSE closes the portal underlying an open cursor. This can be used to release resources earlier than end of transaction, or to free up the cursor variable to be opened again.

An example:

CLOSE curs1;

Implementation Screenshots (Problem Statement, Query and Screenshots of Results):

1. IMPLEMENTATION OF PROCEDURES

Creating a procedure with IN parameter:

Original customer table:

	name_c	age_c	id_no	budget	type_p	no_of_emi	asc_bank
▶	Ashwini	48	1122	5000	ownership	12	HDFC Bank
	Aditi	19	1210	9000	rental	7	ICPC Bank
	Dhruvi	19	1998	10000	rental	9	HDFC Bank
	Samiksha	19	2133	4500	ownership	8	Canara
	Madhuri	25	9021	3000	rental	9	HDFC Bank
	Pinky	21	9987	2300	rental	9	Baroda
	Siddhi	22	9989	3000	ownership	10	Canara
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL

This procedure will select and display the details of all customers whose budget is greater than 4000 and whose associated bank is “HDFC Bank”

```
delimiter &&
create procedure project3 (IN v1 int)
BEGIN
select * from customer where budget > v1 AND asc_bank ="HDFC Bank";
end&&
delimiter ;
```

Result:

	name_c	age_c	id_no	budget	type_p	no_of_emi	asc_bank
▶	Ashwini	48	1122	5000	ownership	12	HDFC Bank
	Dhruvi	19	1998	10000	rental	9	HDFC Bank

2. Creating procedures with an OUT parameter

Original customer table:

	name_c	age_c	id_no	budget	type_p	no_of_emi	asc_bank
▶	Ashwini	48	1122	5000	ownership	12	HDFC Bank
	Aditi	19	1210	9000	rental	7	ICPC Bank
	Dhruvi	19	1998	10000	rental	9	HDFC Bank
	Samiksha	19	2133	4500	ownership	8	Canara
	Madhuri	25	9021	3000	rental	9	HDFC Bank
	Pinky	21	9987	2300	rental	9	Baroda
	Siddhi	22	9989	3000	ownership	10	Canara
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL

This procedure has an in and an out parameter , the IN parameter specifies the ID number of the customer and the out parameter displays the name of the customer with the associated ID

```
use propsys ;
select * from customer;
delimiter &&
create procedure project5 (IN v1 int, OUT v2 varchar(50))
BEGIN
 Select name_c
 from customer
 where id_no= v1  ;
 set v2= name_c;
end&&
delimiter ;

call project5 (2133, @name) ;
```

Output:

	name_c
▶	Samiksha

Using INOUT parameter :

Original Property Table:

	area	location	cost	no_of_rooms	type_p	p_id
▶	200	mumbai	2355.16	4	rental	1111
	350	pune	1297.66	3	ownership	2110
	410	thane	3500.01	2	rental	2159
	300	churchgate	3500.77	3	rental	2211
	250	andheri	2200.22	2	rental	3221
	500	ghatkopar	1799.66	5	ownsership	7651
*	NULL	NULL	NULL	NULL	NULL	NULL

This procedure accepts a value as the number of rooms as the INOUT parameter and displays the associated property ID's and locations of the properties with the given number of rooms

Code:

```
delimiter &&
create procedure project6 (INOUT v1 int, OUT v2 varchar(50), OUT v3 int)
BEGIN
 Select no_of_rooms, location, p_id
 from property
 where no_of_rooms= v1  ;
 set v1= no_of_rooms ;
 set v2 = location ;
 set v3=p_id;
end&&
delimiter ;
```

```
set @c = 2;
call project6 (@c, @loc, @id) ;
```

Output:

	no_of_rooms	location	p_id
▶	2	thane	2159
	2	andheri	3221

Creating simple procedures without parameters:

```
delimiter &&
create procedure project7 ()
BEGIN
 Select area, location , cost , type_p
 from property
 where type_p="rental";
end&&
delimiter ;
call project7()
```

This code selects all properties of “rental” type and displays their area, location, cost and type.

Output:

	area	location	cost	type_p
▶	200	mumbai	2355.16	rental
	410	thane	3500.01	rental
	300	churchgate	3500.77	rental
	250	andheri	2200.22	rental

Procedure with IN and OUT parameter:

```
use propsys ;
select * from property;
delimiter &&
create procedure project9 (IN loc varchar(50), OUT type_pa varchar(50))
BEGIN
 Select type_p
 from property
 where location=@loc;
 set type_pa = type_p;
end&&
delimiter ;
set @loc = "churchgate";

call project9(@loc,@type_p)
```

	type_p
▶	rental

2. IMPLEMENTATION OF CURSORS

Original Property table

	area	location	cost	no_of_rooms	type_p	p_id
▶	200	mumbai	2355.16	4	rental	1111
	350	pune	1297.66	3	ownership	2110
	410	thane	3500.01	2	rental	2159
	300	churchgate	3500.77	3	rental	2211
	250	andheri	2200.22	2	rental	3221
	500	ghatkopar	1799.66	5	ownsership	7651
*	NULL	NULL	NULL	NULL	NULL	NULL

Original Customer Table:

name_c	age_c	id_no	budget	type_p	no_of_emi	asc_bank
Ashwini	48	1122	5000	ownership	12	HDFC Bank
Aditi	19	1210	9000	rental	7	ICPC Bank
Dhruvi	19	1998	10000	rental	9	HDFC Bank
Samiksha	19	2133	4500	ownership	8	Canara
Madhuri	25	9021	3000	rental	9	HDFC Bank
Pinky	21	9987	2300	rental	9	Baroda
Siddhi	22	9989	3000	ownership	10	Canara
NULL	NULL	NULL	NULL	NULL	NULL	NULL

1. **Creating a cursor to show the location and price of each property from property table**

```
use propsys ;

# creating a cursor to show the location and cost of all properties
DELIMITER //
create procedure loc_cost()
BEGIN
DECLARE locat varchar(50); #declaring variables to be used in the cursor
DECLARE c float;
DECLARE check_c int default 0;
DECLARE cur1 cursor for select location , cost from property;
DECLARE continue HANDLER for not found set check_c=1; #when all the rows have been iterated through
open cur1;
label1 : loop
fetch cur1 into locat,c;
if check_c=1 then
leave label1;
end if;
select concat(locat , concat(" ",c)); #the variables from the cursor are fed into locat,c and they are concatenated to show them together. Then we display them individually using the select statement
end loop label1;
close cur1;
end//
DELIMITER ;
```

call loc_cost()

Result

	concat(locat , concat(" ",c))
▶	mumbai 2355.16

	concat(locat , concat(" ",c))
▶	pune 1297.66

	concat(locat , concat(" ",c))
▶	thane 3500.01

	concat(locat , concat(" ",c))
▶	churchgate 3500.77

	concat(locat , concat(" ",c))
▶	andheri 2200.22

	concat(locat , concat(" ",c))
▶	ghatkopar 1799.66

2. **Declaring cursor for showing all customers whose budget > Rs 4000:**

```
DELIMITER //
create procedure name_budget1()
BEGIN
DECLARE name_c1 varchar(50); #declaring variables to be used in the cursor
```

```
DECLARE b int;
DECLARE check_c int default 0;
DECLARE cur2 cursor for select name_c , budget from customer where budget>4000;
DECLARE continue HANDLER for not found set check_c=1; #when all the rows have been iterated through
open cur2;
label1 : loop
fetch cur2 into name_c1,b;
if check_c=1 then
leave label1;
end if;
select concat(name_c1 , concat(" ",b)); #the variables from the cursor are fed into locat,c and they are concatenated to show them together. Then we display them individually using the select statement
end loop label1;
close cur2;
end//
DELIMITER ;

call name_budget1()
```

Result:

	concat(name_c1 , concat(" ",b))
▶	Ashwini 5000

	concat(name_c1 , concat(" ",b))
▶	Aditi 9000

	concat(name_c1 , concat(" ",b))
▶	Dhruvi 10000

	concat(name_c1 , concat(" ",b))
▶	Samiksha 4500

3. **Declaring cursor for procedure with one INT parameter. This procedure accepts the number of queries and prints the name and budget of those many customers , ordered by their ID.**

```
DELIMITER //
create procedure countandprint(IN v1 INT)
BEGIN
DECLARE name_c1 varchar(50);
DECLARE b  int ;
DECLARE counter int ; #declaring variables to be used in the cursor
DECLARE check_c int default 0;
DECLARE cur3 cursor for select name_c , budget from customer;
DECLARE continue HANDLER for not found set check_c=1; #when all the rows have been iterated through
SET counter = 0;
open cur3;
label1 : loop
fetch cur3 into name_c1,b;
set counter = counter+1;
if check_c=1 or counter>v1 then #if the value provided is greater than number of customers itself , it terminates where the table ends.
leave label1;
end if;
select concat(name_c1 , concat(" ",b)); #the variables from the cursor are fed into locat,c and they are concatenated to show them together. Then we display them individually using the select statement
end loop label1;
close cur3;
end//
DELIMITER ;

call countandprint(5);
```

here we will be calling the function to print the names of first 5 customers ordered by their ID's from the customer table

Output:

	concat(name_c1 , concat(" ",b))
▶	Ashwini 5000

Result 21 × | Result 22 | Result 23 | Result 24 | Result 25

	concat(name_c1 , concat(" ",b))
▶	Aditi 9000

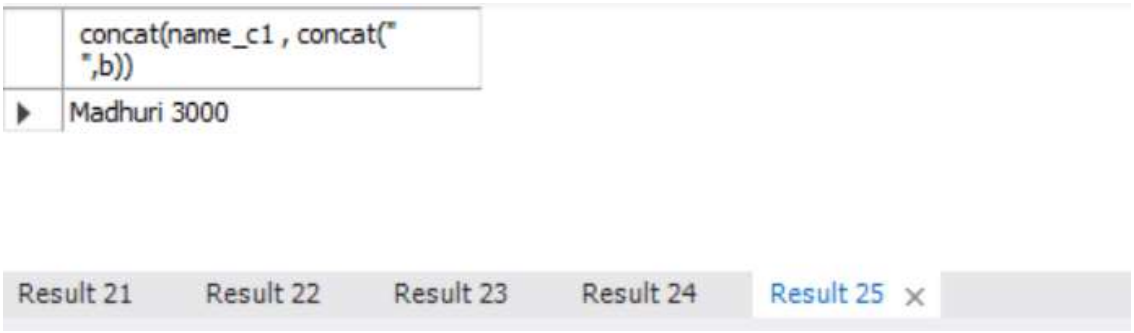
Result 21 | Result 22 × | Result 23 | Result 24 | Result 25

	concat(name_c1 , concat(" ",b))
▶	Dhruvi 10000

Result 21 | Result 22 | Result 23 × | Result 24 | Result 25

	concat(name_c1 , concat(" ",b))
▶	Samiksha 4500

Result 21 | Result 22 | Result 23 | Result 24 × | Result 25



	concat(name_c1 , concat(" ",b))
▶	Madhuri 3000

Result 21 | Result 22 | Result 23 | Result 24 | Result 25 ×

Conclusion: In this experiment , the concepts of procedures and cursors was well understood and implemented for various applications in MySQL.

Post Lab Questions:

1. **Does Storing Of Data In Stored Procedures Increase The Access Time? Explain?**

   No , storing of data does not increase , rather it reduces the access time. This is because Data stored in stored procedures can be retrieved much faster than the data stored in SQL database. Data can be precompiled and stored in Stored procedures. This reduces the time gap between query and compiling as the data has been precompiled and stored in the procedure. To avoid repetitive nature of the data base statement caches are used.

2. **Point out the wrong statement.**

   a) We should use cursor in all cases
   b) A static cursor can move forward and backward direction
   c) A forward only cursor is the fastest cursor
   d) All of the mentioned
   Ans : option a) , since we should use cursor only in those cases where there is no option other than using the cursor.