Batch: B2	Roll No.: 110 and 109
Experiment / assignment / tutorial No. 7	

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):

```
insert into pokemontype(typeid,typename) values (1,'rock');
create procedure addMove(_movename Varchar(10),_moveID int,_accuracy int,_mtype int)
language plpgsql
as $$
begin
insert into pokemonMove(movename,moveID,accuracy,mtype) values (_movename,_moveID,_accuracy,_mtype);
commit;
end
$$
call addMove('stone edge',2,20,1);
select * from pokemonMove
```

Data Output | Explain | Messages | Notifications

	movename character varying (12)	moveid [PK] integer	accuracy integer	mtype integer
1	stone edge	2	20	1

Conclusion:

In conclusion, stored procedures in SQL are a powerful tool for managing and processing data within a database. By encapsulating logic and functionality within a procedure, we can improve performance, reduce network traffic, and performance. With careful planning and implementation, stored procedures can greatly enhance the capabilities and efficiency of a database system.

Post Lab Questions:

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

Storing data in stored procedures does not necessarily increase the access time. In fact, it can improve performance in certain cases.

When data is stored in a stored procedure, it can be compiled and optimized by the database engine, resulting in faster execution times. Additionally, by storing the logic for retrieving and processing data in a stored procedure, you can reduce the amount of network traffic and improve overall system performance.

However, there are some cases where storing data in stored procedures can negatively impact performance. For example, if the stored procedure is poorly written or contains inefficient queries, it can increase the access time.

Ultimately, the impact on performance will depend on the specific implementation and usage scenario. It is important to carefully design and test stored procedures to ensure they are optimized for the specific use case.

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