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

Batch: A1 Roll No.:
16010120015_YASH
16010120006_DIKSHITA
16010120018_JINAY
Experiment / assignment / tutorial No. 6

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 :

The following creates SOLDIER procedure named Transfer that Transfer specific amount of SALARY from SOLDIERS SALARY to another.

```
CREATE OR REPLACE PROCEDURE Transfer()
LANGUAGE plpgsql
AS $$
BEGIN
   -- subtracting the amount from the SOLDIER's SALARY
   UPDATE SOLDIERS
   SET SALARY = SALARY - 10000
   WHERE SSNO = 609;

   -- adding the amount to the SOLDIER's SALARY
   UPDATE SOLDIERS
   SET SALARY = SALARY + 10000
   WHERE SSNO = 608;
   COMMIT;
END;
```

```
$$;

CALL Transfer();

SELECT *

FROM SOLDIERS
```

Query Editor Query History

```
335     UPDATE SOLDIERS
336     SET SALARY = SALARY - 10000
337     WHERE SSNO = 609;
338
339     -- adding the amount to the SOLDIER's SALARY
340     UPDATE SOLDIERS
341     SET SALARY = SALARY + 10000
342     WHERE SSNO = 608;
343
344     COMMIT;
345 END;
346 $$;
347
348
349 CALL Transfer();
350
351 SELECT *
352 FROM SOLDIERS
```

Data Output Explain Messages Notifications

	ssno [PK] integer	rno integer	name character varying (50)	dateofbirth date	rank character varying (25)	location character (25)	salary integer
1	601	14	SOMNATH SHARMA	1947-02-12	MAJOR	KASHMIR	800000
2	602	13	VIKRAM BATRA	1950-02-11	GENREAL MAJOR	GOA	300000
3	603	15	SATISH KOLI	1949-02-10	CAPTAIN	JAMMU	80000
4	604	16	SOHAM PATIL	1948-10-09	BRIGADIER	PUNJAB	140000
5	605	17	VISHESH NAIK	1947-09-08	COLONEL	DELHI	120000
6	606	12	NIRMAL SINGH	1946-08-07	GENREAL MAJOR	GOA	300000
7	607	18	YOGENDRA SINGH	1949-07-06	CAPTAIN	JAMMU	80000

Cursor

```
BEGIN;
DECLARE
   my_cursor CURSOR FOR SELECT * FROM SOLDIERS;
```

```
362  BEGIN;
363
364  DECLARE
365      my_cursor CURSOR FOR SELECT * FROM SOLDIERS;
366
```

Data Output | Explain | Messages | Notifications

```
DECLARE CURSOR

Query returned successfully in 175 msec.
```

Fetch the data

FETCH 10 FROM my_cursor;

```
362  BEGIN;
363
364  DECLARE
365      my_cursor CURSOR FOR SELECT * FROM SOLDIERS;
366
367
368  FETCH 10 FROM my_cursor;
369
370
371
```

Data Output | Explain | Messages | Notifications

	ssno [PK] integer	rno integer	name character varying (50)	dateofbirth date	rank character varying (25)	location character (25)	salary integer
1	601	14	SOMNATH SHARMA	1947-02-12	MAJOR	KASHMIR	800000
2	602	13	VIKRAM BATRA	1950-02-11	GENREAL MAJOR	GOA	300000
3	603	15	SATISH KOLI	1949-02-10	CAPTAIN	JAMMU	80000
4	604	16	SOHAM PATIL	1948-10-09	BRIGADIER	PUNJAB	140000
5	605	17	VISHESH NAIK	1947-09-08	COLONEL	DELHI	120000
6	606	12	NIRMAL SINGH	1946-08-07	GENREAL MAJOR	GOA	300000
7	607	18	YOGENDRA SINGH	1949-07-06	CAPTAIN	JAMMU	80000
8	610	21	OMKAR SINGH	1947-04-03	COLONEL	DELHI	120000
9	611	22	NIRMALJIT SINGH	1948-05-04	MAJOR	GUJARAT	90000
10	612	-22	NIRMA SINGH	1945-02-02	CAPTAIN	KASHMIR	100

FETCH PRIOR FROM my_cursor;

```
370
371  FETCH PRIOR FROM my_cursor;
```

Data Output | Explain | Messages | Notifications

	ssno [PK] integer	rno integer	name character varying (50)	dateofbirth date	rank character varying (25)	location character (25)	salary integer
1	611	22	NIRMALJIT SINGH	1948-05-04	MAJOR	GUJARAT ...	90000

```
368
369  FETCH PRIOR FROM my_cursor;
370
371  COMMIT;
372
```

Data Output | Explain | Messages | Notifications

```
COMMIT

Query returned successfully in 33 msec.
```

Move:

MOVE my_cursor;

```
374  MOVE my_cursor;
375
```

Data Output | Explain | Messages | Notifications

```
MOVE 1

Query returned successfully in 37 msec.
```

UPDATE:

UPDATE SOLDIERS SET SSNO = 608 WHERE CURRENT OF my_cursor;

```
372  UPDATE  SOLDIERS SET SSNO = 608 WHERE CURRENT OF my_cursor;
373
```

Data Output | Explain | Messages | Notifications

```
UPDATE 1

Query returned successfully in 26 msec.
```

Data Output | Explain | Messages | Notifications

	ssno [PK] integer	rno integer	name character varying (50)	dateofbirth date	rank character varying (25)	location character (25)	salary integer
1	601	14	SOMNATH SHARMA	1947-02-12	MAJOR	KASHMIR ...	800000
2	602	13	VIKRAM BATRA	1950-02-11	GENREAL MAJOR	GOA ...	300000
3	603	15	SATISH KOLI	1949-02-10	CAPTAIN	JAMMU ...	80000
4	604	16	SOHAM PATIL	1948-10-09	BRIGADIER	PUNJAB ...	140000
5	605	17	VISHESH NAIK	1947-09-08	COLONEL	DELHI ...	120000
6	606	12	NIRMAL SINGH	1946-08-07	GENREAL MAJOR	GOA ...	300000
7	607	18	YOGENDRA SINGH	1949-07-06	CAPTAIN	JAMMU ...	80000
8	610	21	OMKAR SINGH	1947-04-03	COLONEL	DELHI ...	120000
9	611	22	NIRMALJIT SINGH	1948-05-04	MAJOR	GUJARAT ...	90000
10	612	-22	NIRMA SINGH	1945-02-02	CAPTAIN	KASHMIR ...	100000
11	609	20	VISHAL NAIK	1947-05-04	COLONEL	DELHI ...	130000
12	608	19	DYANCHAND THAPA	1948-06-05	BRIGADIER	PUNJAB ...	130000

DELETE

DELETE FROM SOLDIERS WHERE CURRENT OF my_cursor;

```
368  DELETE FROM SOLDIERS WHERE CURRENT OF my_cursor;
369
```

Data Output | Explain | Messages | Notifications

```
DELETE 1

Query returned successfully in 27 msec.
```

CLOSE:

CLOSE my_cursor;

```
373  CLOSE my_cursor;
374
375
```

Data Output | Explain | Messages | Notifications

```
CLOSE CURSOR

Query returned successfully in 30 msec.
```

Conclusion:

we implemented procedure and cursor and we understood that

PROCEDURE

- A stored procedure provides an important layer of security between the user interface and the database.
- It supports security through data access controls
- A stored procedure preserves data integrity because information is entered in a consistent manner.
- It improves productivity .

CURSOR

- It is used to saves memory on the server.
- If we are reading data from the cursor, other sessions can do their operations; there is no impact on other connections.
- Using cursor, better response time is achieved

Post Lab Questions:

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

Ans.

Data stored in stored procedures can be retrieved much faster than the data stored in SQL database. Data is precompiled and stored in Stored procedures. As compared to SQL database the time gap between query and compiling as the data has been precompiled and stored in the procedure. Stored procedures help in reducing the network traffic and also improve the performance of the data. The integrity of data can be ensured while making use of stored procedures. It improves the performance of the database and Faster execution takes place.

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. a) we use cursor only in some cases , when there is no option except cursor.