

Batch: E-2 Roll No.: 16010123325
Experiment / assignment / tutorial No. 6
Grade: AA / AB / BB / BC / CC / CD / DD
Signature of the Staff In-charge with date

Title: Queries based on Procedure, Function and Cursor

Objective: To be able to functions and procedures on the table.

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, Silberchatz, Sudarshan : "Database Systems Concept", 5th Edition , McGraw Hill
4. Elmasri and Navathe, "Fundamentals of database Systems", 4th Edition, PEARSON Education.

Resources used: Postgresql

Theory

Procedures:

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);
```

Functions

The basic syntax to create a function is as follows –

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
```

```
RETURNS return_datatype
```

```
language plpgsql
```

```
AS
```

```
$variable_name$
```

```
declare
```

```
    -- variable declaration
```

```
begin
```

```
    -- stored procedure body
```

```
end; $$
```

Explanation:

function-name specifies the name of the function.

[OR REPLACE] option allows modifying an existing function.

The function must contain a return statement.

RETURN clause specifies that data type you are going to return from the function. The return_datatype can be a base, composite, or domain type, or can reference the type of a table column.

function-body contains the executable part.

The **AS** keyword is used for creating a standalone function.

plpgsql is the name of the language that the function is implemented in. Here, we use this option for PostgreSQL, it can be SQL, C, internal, or the name of a user-defined procedural language. For backward compatibility, the name can be enclosed by single quotes.

Example

```
CREATE TABLE employee (
```

```
id SERIAL PRIMARY KEY,  
name VARCHAR(100),  
date_of_joining DATE,  
salary DECIMAL(10, 2)  
);  
INSERT INTO employee (name, dateofjoining, salary) VALUES  
('John Doe', '2022-01-01', 50000),  
('Jane Smith', '2022-02-15', 60000),  
('Michael Johnson', '2022-03-20', 70000),  
('Emily Davis', '2022-04-10', 45000),  
('Christopher Wilson', '2022-05-05', 80000);
```

create a function named `get_employee_count` to calculate the number of records in the `employee` table:

```
CREATE OR REPLACE FUNCTION get_employee_count()  
RETURNS INTEGER AS $$  
DECLARE  
    total_records INTEGER;  
BEGIN  
    -- Execute a SQL query to count the number of records in the employee table  
    SELECT COUNT(*) INTO total_records FROM employee;  
  
    -- Return the total number of records  
    RETURN total_records;  
END;  
$$ LANGUAGE plpgsql;  
  
calling the function using query:  
SELECT get_employee_count();
```

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

Syntax: **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;

Example:

```
-- Declare variables to hold fetched data
DECLARE
    emp_id INT;
    emp_first_name VARCHAR(50);
    emp_last_name VARCHAR(50);

    -- Declare an explicit cursor
    CURSOR emp_cursor IS
        SELECT employee_id, first_name, last_name
        FROM employees;

-- Open the cursor
OPEN emp_cursor;

-- Fetch and process each row
LOOP
    FETCH emp_cursor INTO emp_id, emp_first_name, emp_last_name;
    EXIT WHEN NOT FOUND; -- Exit loop when no more rows

END LOOP;

CLOSE cur_emp; -- Close Cursor

END $$;
```

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

Create a procedure to update the Patient Age

Query Editor Query History

```
470 CREATE TABLE patients (  
471     patient_id SERIAL PRIMARY KEY,  
472     name VARCHAR(100),  
473     age INT  
474 );  
475  
476 -- Insert some sample data  
477 INSERT INTO patients (name, age) VALUES  
478 ('John Doe', NULL),  
479 ('Alice Smith', NULL),  
480 ('Bob Johnson', NULL),  
481 ('Emily Brown', NULL);  
482  
483 -- Create or replace the procedure to update patient age  
484 CREATE OR REPLACE PROCEDURE update_patient_age(p_id INT, new_age INT)  
485 LANGUAGE plpgsql  
486 AS $$  
487 BEGIN  
488     UPDATE patients  
489     SET age = new_age  
490     WHERE patient_id = p_id;  
491 END;
```

Data Output Explain Messages

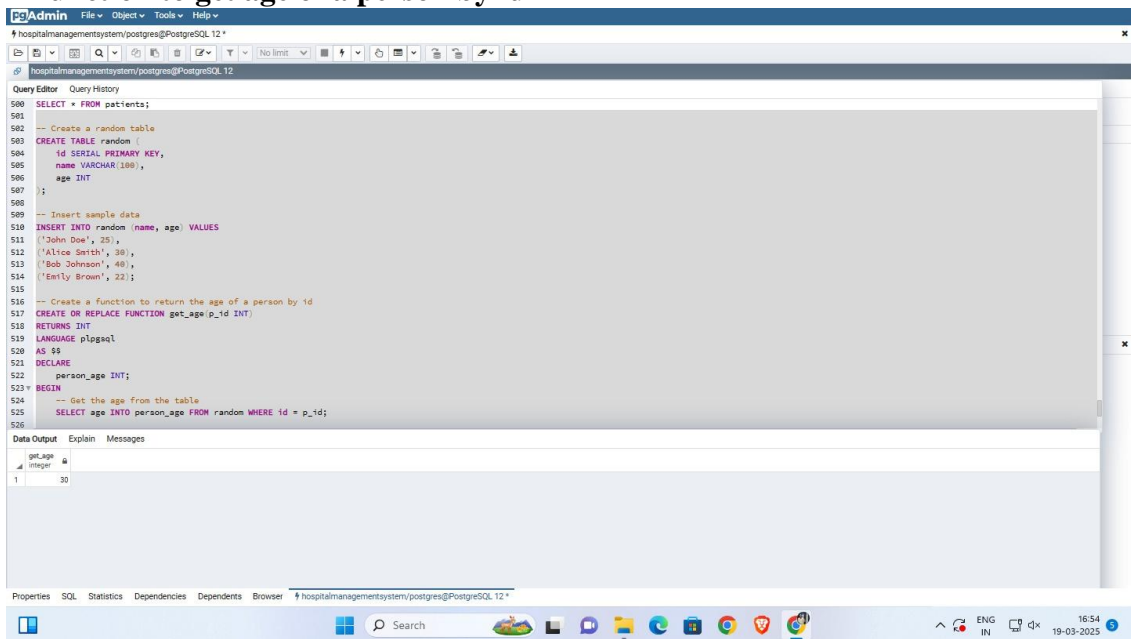
	patient_id (PK) integer	name character varying (100)	age integer
1	2	Alice Smith	[null]
2	3	Bob Johnson	[null]
3	4	Emily Brown	[null]
4	1	John Doe	45

hospitalmanagementsystem/postgres@PostgreSQL 12 *

Search

ENG IN 16:37 19-03-2025

Function to get age of a person by id



The screenshot shows the PgAdmin interface with the following SQL queries in the Query Editor:

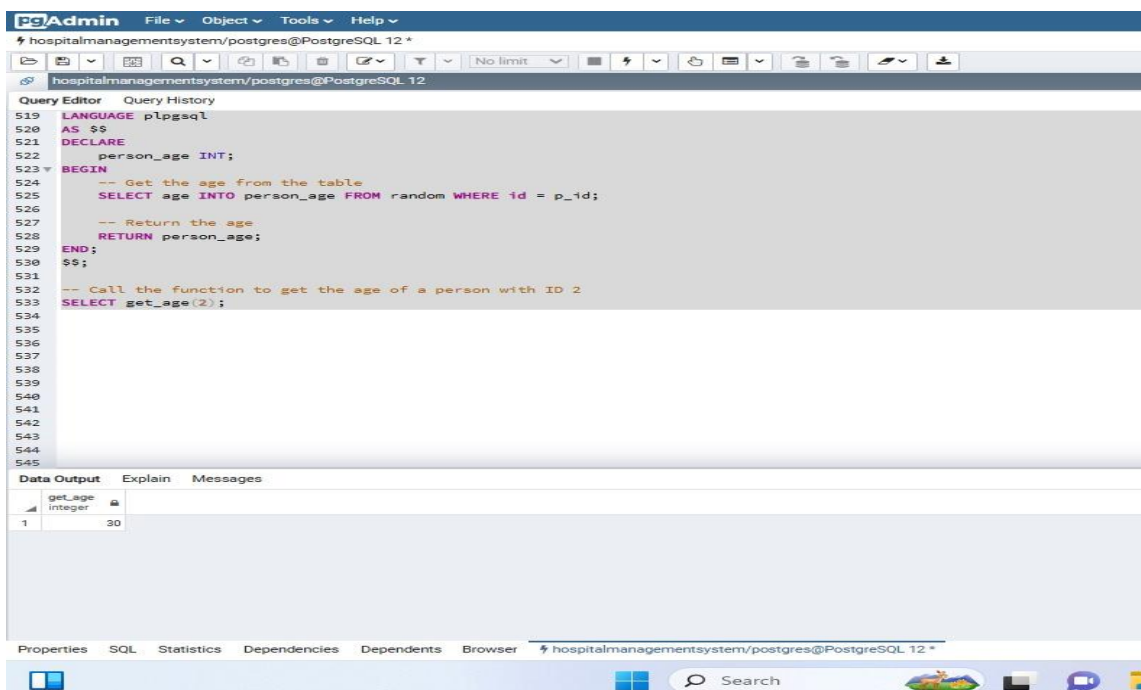
```

500 SELECT * FROM patients;
501
502 -- Create a random table
503 CREATE TABLE random (
504     id SERIAL PRIMARY KEY,
505     name VARCHAR(100),
506     age INT
507 );
508
509 -- Insert sample data
510 INSERT INTO random (name, age) VALUES
511     ('John Doe', 25),
512     ('Alice Smith', 30),
513     ('Bob Johnson', 40),
514     ('Emily Brown', 22);
515
516 -- Create a function to return the age of a person by id
517 CREATE OR REPLACE FUNCTION get_age(p_id INT)
518 RETURNS INT
519 LANGUAGE plpgsql
520 AS $$
521 DECLARE
522     person_age INT;
523 BEGIN
524     -- Get the age from the table
525     SELECT age INTO person_age FROM random WHERE id = p_id;
526
527     -- Return the age
528     RETURN person_age;
529 END;
530 $$;

```

The Data Output tab shows the result of the first query:

get_age	integer
1	30



The screenshot shows the PgAdmin interface with the following SQL queries in the Query Editor:

```

519 LANGUAGE plpgsql
520 AS $$
521 DECLARE
522     person_age INT;
523 BEGIN
524     -- Get the age from the table
525     SELECT age INTO person_age FROM random WHERE id = p_id;
526
527     -- Return the age
528     RETURN person_age;
529 END;
530 $$;
531
532 -- Call the function to get the age of a person with ID 2
533 SELECT get_age(2);
534
535
536
537
538
539
540
541
542
543
544
545

```

The Data Output tab shows the result of the function call:

get_age	integer
1	30

Cursor for fetching all doctor's name

```
Query Editor Query History
537 -- Create a function that fetches all doctor names using a cursor
538 CREATE OR REPLACE FUNCTION fetch_doctor_names()
539 RETURNS SETOF TEXT -- Returning a list of full names
540 LANGUAGE plpgsql
541 AS $$
542 DECLARE
543     doctor_cursor CURSOR FOR SELECT fname, lname FROM doctor;
544     first_name VARCHAR(100);
545     last_name VARCHAR(100);
546     full_name TEXT;
547 BEGIN
548     -- Open the cursor
549     OPEN doctor_cursor;
550
551     LOOP
552         -- Fetch each row from the cursor
553         FETCH doctor_cursor INTO first_name, last_name;
554
555         -- Exit the loop if no more rows
556         EXIT WHEN NOT FOUND;
557
558         -- Concatenate first name and last name
559         full_name := first_name || ' ' || last_name;
560
561         -- Print the fetched full name
562         RAISE NOTICE 'Doctor: %', full_name;
563
564         -- Return the fetched name
565         RETURN NEXT full_name;
566     END LOOP;
567
568     -- Close the cursor
569     CLOSE doctor_cursor;
570 END;
571 $$;
572
573 -- Call the function to fetch doctor names using the cursor
574 SELECT * FROM fetch_doctor_names();
575
576
577
578
579
580
581
582
```

Data Output Explain Messages

NOTICE: Doctor: John Smith
NOTICE: Doctor: Sarah Johnson
NOTICE: Doctor: Michael Brown
NOTICE: Doctor: Emily Davis
NOTICE: Doctor: Robert Wilson
NOTICE: Doctor: Lisa Anderson
NOTICE: Doctor: David Martinez
NOTICE: Doctor: Jennifer Taylor
NOTICE: Doctor: William Thomas
NOTICE: Doctor: Mary Garcia

Successfully run. Total query runtime: 181 msec.

Properties SQL Statistics Dependencies Dependents Browser hospitalmanagementsystem/postgres@PostgreSQL 12*

```
563
564 -- Return the fetched name
565 RETURN NEXT full_name;
566 END LOOP;
567
568 -- Close the cursor
569 CLOSE doctor_cursor;
570 END;
571 $$;
572
573 -- Call the function to fetch doctor names using the cursor
574 SELECT * FROM fetch_doctor_names();
575
576
577
578
579
580
581
582
```

Data Output Explain Messages

NOTICE: Doctor: John Smith
NOTICE: Doctor: Sarah Johnson
NOTICE: Doctor: Michael Brown
NOTICE: Doctor: Emily Davis
NOTICE: Doctor: Robert Wilson
NOTICE: Doctor: Lisa Anderson
NOTICE: Doctor: David Martinez
NOTICE: Doctor: Jennifer Taylor
NOTICE: Doctor: William Thomas
NOTICE: Doctor: Mary Garcia

Successfully run. Total query runtime: 181 msec.

Properties SQL Statistics Dependencies Dependents Browser hospitalmanagementsystem/postgres@PostgreSQL 12*

Conclusion:

Functions, procedures, and cursors play a vital role in PL/pgSQL by enabling efficient database operations. In this experiment the following have been implemented for the airline database.

Post Lab Questions:

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

No, stored procedures do not store data themselves; they contain SQL statements that process data. They can improve performance by reducing network latency and pre-compiling execution plans, but excessive logic inside procedures may slightly impact execution time.

2. Explain the FETCH statement in SQL cursors.

The FETCH statement retrieves the next row from a cursor into a variable or record. It is used in loops to process query results one row at a time. It stops when no more rows are available.

```
FETCH cur_name INTO var1, var2;
```

3. What is the difference between a function and a stored procedure in PostgreSQL?

- **Function** returns a value and is used in queries. Example: `SELECT my_function();`
- **Procedure** does not return a value and is used for performing actions. Example: `CALL my_procedure();`
Functions can be used in `SELECT` statements, while procedures cannot.