Lab 9

Objectives:
At the end of this lab, you should be able to

- Create Procedures and Functions.
- Create, and test triggers.

PL/SQL - Procedures

A **subprogram** is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

PL/SQL provides two kinds of subprograms:

- **Functions** − These subprograms return a single value; mainly used to compute and return a value.
- **Procedures** − These subprograms do not return a value directly; mainly used to perform an action.

Each PL/SQL subprogram has a name, and may also have a parameter list. Like anonymous PL/SQL blocks, the named blocks will also have the following three parts :

- **Declarative (optional):** contains all variables, constants, coursers that are used in the executable section.
- **Executable (required):** contains SQL statements to manipulate data in the database and PL/SQL statements to manipulate data in the block.
- **Exception handling (optional):** specifies the action to perform when errors and abnormal conditions arise in the executable section.

```
CREATE [OR REPLACE] PROCEDURE
procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
< procedure_body >
END procedure_name;
```

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters.

IN represents the value that will be passed from outside

OUT represents the parameter that will be used to return a value outside of the procedure.

- *procedure-body* contains the executable part.

Example

--Stored Procedure for Department Wide Salary Raise

```
CREATE OR REPLACE PROCEDURE emp_sal( dep_id NUMBER, sal_raise
NUMBER)
IS
BEGIN
  UPDATE EMP SET SAL = SAL+(SAL * sal_raise) WHERE DEPTNO = dep_id;
  DBMS_OUTPUT.PUT_LINE ('salary updated successfully');
END;
```

```
EXEC emp_sal(20,0.5);
SELECT * FROM EMP;
```

PL/SQL - Functions

A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

Creating a Function

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows –

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
   < function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.

Example

The following example illustrates how to create and call a standalone function. This function returns the total number of EMPLOYEES in the emp table.

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
  total number(2) := 0;
BEGIN
  SELECT count(*) into total
  FROM EMP;
  RETURN total;
END;
```

Calling a Function

```
DECLARE
  c number(2);
BEGIN
  c := totalCustomers();
  dbms_output.put_line('Total no. of Customers: ' || c);
END;
```

Oracle Triggers:

Oracle allows you to define procedures that are implicitly executed when an INSERT, UPDATE, or DELETE statement is issued against the associated table. These procedures are called database triggers. Triggers are similar to stored procedures, discussed above. A trigger can include SQL and PL/SQL statements to execute as a unit and can invoke stored procedures. However, procedures and triggers differ in the way that they are invoked. While a procedure is explicitly executed by a user, application, or trigger, one or more triggers are implicitly fired (executed) by Oracle when a triggering INSERT, UPDATE, or DELETE statement is issued, no matter which user is connected or which application is being used.

Triggers Types:

The trigger type determines whether the code in the trigger executes for each row or only once for the triggering statement.

- A statement trigger:

Executes once for the triggering event. It is the default type of trigger
Fires once even if no rows are affected at all

- A row trigger:

Executes once for each row affected by the triggering event. It is not executed if the triggering event does not affect any rows. Is indicated by specifying the FOR EACH ROW clause

---

Triggers Timing:

- BEFORE: Execute the trigger body before the triggering DML event on a table.
- AFTER: Execute the trigger body after the triggering DML event on a table.

Note: If multiple triggers are defined for the same object, then the order of firing triggers is arbitrary.

Example # 1

```
CREATE OR REPLACE TRIGGER CHANGE
AFTER INSERT ON EMP
FOR EACH ROW
BEGIN
        IF SYSDATE- :NEW.HIREDATE >30 THEN
                RAISE_APPLICATION_ERROR(-20001,'HIREDATE CAN NOT BE BEFORE 30 DAYS');
        END IF;
END;
```

Now try to insert a new employee whose hire date a year ago.

To log insert operations on table EMP, we will create table AUDITEMP that saves data about user who insert new records, date of insertion, and the number of the new employee.

```
CREATE TABLE AUDITEMP (
        USERNAME VARCHAR2(20),
        OPDATE DATE,
        EMPNO NUMBER(4));
```

The trigger is :

```
        CREATE OR REPLACE TRIGGER LOGINSERT
        AFTER INSERT ON EMP
        FOR EACH ROW
        BEGIN
                INSERT INTO AUDITEMP VALUES(USER,SYSDATE,:NEW.EMPNO);
        END;
```

Now try to insert a new employee

```
        Insert into EMP( EMPNO, ENAME) VALUES (333,'AHMED');
```

Now query the table AUDITEMP

Example # 2

```
        CREATE OR REPLACE TRIGGER INS_REC
                AFTER INSERT ON EMP
                FOR EACH ROW
        BEGIN
                DBMS_OUTPUT.PUT_LINE('NEW RECORD INSERTED');
        END;
```

Now try to insert a new record to table EMP. (Note: to get the output of DBMS_OUTPUT, you should use the command SET SERVEROUTPUT ON.

Example # 3

```
CREATE OR REPLACE TRIGGER Print_salary_changes
  BEFORE UPDATE ON Emp
  FOR EACH ROW
WHEN (new.SAL > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff  := :new.sal  - :old.sal;
   dbms_output.put('Old salary:  ' || :old.sal);
   dbms_output.put(' New salary:  ' || :new.sal);
   dbms_output.put_line(' Difference:  ' || sal_diff);
END;
```

Now Test the above trigger by updating JAMES's salary to 4000.

Disable and Enable Triggers

To disable a trigger

ALTER TRIGGER T1 DISABLE;

To enable a trigger

ALTER TRIGGER T1 ENABLE;

To remove a trigger

DROP TRIGGER T1;

EXERCISE:

1) Write a stored Procedure Called Display_Emp that will Query the following: DeptNo , DName , Ename.
   DeptNo is either 10 or 20.
   Call the Procedure using **EXEC DISPLAY_EMP(10,20);**

2) Write a Function called FindName that will find and display the name of employee with the highest salary.
3) Write a trigger to display the system date and username of the person who did an update or delete to the department located in DALLAS. In addition to that, your trigger must also display the entire modified record.
   Now try to update LOC from DALLAS to DENVER.