

## **PRACTICAL-11**

- **Working with Cursor**

PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

- **Implicit cursors**
- **Explicit cursors**

- **Implicit Cursors:**

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND**, **%ISOPEN**, **%NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK\_ROWCOUNT** and **%BULK\_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes:

S.No	Attribute & Description
1	<b>%FOUND</b> Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	<b>%NOTFOUND</b> The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

3	<b>%ISOPEN</b> Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	<b>%ROWCOUNT</b> Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute\_name** as shown below in the example.

✓ **Example:**

- We will be using the EMPLOYEE table we had created earlier.

```
SQL> select * from EMPLOYEE;
```

E_NO	E_NAME	E_SALARY	E_CONTACT_NO	D_NO	J_ID
10001	Juned	100000	9876543210	101	1003
10002	Mitesh	50000	8967452301	101	1003
10003	Hitesh	40000	6543219870	102	1001
10004	Pooja	30000	9870654321	104	1002

- The following program will update the table and increase the salary of each employee by 5000 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected:

```
set serveroutput on;
declare
total_rows number(2);
begin
update EMPLOYEE set e_salary = e_salary + 5000;
if sql%notfound then
dbms_output.put_line('No employees selected');
elsif sql%found then
total_rows := sql%rowcount;
dbms_output.put_line(total_rows || ' employees selected');
end if;
end;
/
```

- When the above code is executed at the SQL prompt, it produces the following result:

```
SQL> set serveroutput on;
SQL> declare
  2 total_rows number(2);
  3 begin
  4 update EMPLOYEE set e_salary = e_salary + 5000;
  5 if sql%notfound then
  6 dbms_output.put_line('No employees selected');
  7 elsif sql%found then
  8 total_rows := sql%rowcount;
  9 dbms_output.put_line(total_rows || ' employees selected');
 10 end if;
 11 end;
 12 /
4 employees selected

PL/SQL procedure successfully completed.
```

- If you check the records in EMPLOYEE table, you will find that the rows have been updated:

```
SQL> select * from EMPLOYEE;
```

E_NO	E_NAME	E_SALARY	E_CONTACT_NO	D_NO	J_ID
10001	Juned	105000	9876543210	101	1003
10002	Mitesh	55000	8967452301	101	1003
10003	Hitesh	45000	6543219870	102	1001
10004	Pooja	35000	9870654321	104	1002

### ○ **Explicit Cursors:**

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is:

**CURSOR cursor\_name IS select\_statement;**

Working with an explicit cursor includes the following steps:

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

#### ➤ **Declaring the Cursor:**

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example:

**CURSOR emp\_details IS SELECT e\_no, e\_name, e\_salary from EMPLOYEE;**

#### ➤ **Opening the Cursor:**

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows:

**OPEN emp\_details;**

#### ➤ **Fetching the Cursor:**

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows:

**FETCH emp\_details into emp\_no, emp\_name, emp\_salary;**

#### ➤ **Closing the Cursor:**

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows:

**CLOSE emp\_details;**

✓ Example:

- Following is a complete example to illustrate the concepts of explicit cursors:

```
set serveroutput on;
declare
emp_no EMPLOYEE.e_no%type;
emp_name EMPLOYEE.e_name%type;
emp_salary EMPLOYEE.e_salary%type;
CURSOR emp_details IS SELECT e_no, e_name, e_salary from EMPLOYEE;
begin
OPEN emp_details;
loop
FETCH emp_details into emp_no, emp_name, emp_salary;
exit when emp_details%notfound;
dbms_output.put_line(emp_no || ' ' || emp_name || ' ' || emp_salary);
end loop;
CLOSE emp_details;
end;
/
```

- When the above code is executed at the SQL prompt, it produces the following result:

```
SQL> set serveroutput on;
SQL> declare
  2 emp_no EMPLOYEE.e_no%type;
  3 emp_name EMPLOYEE.e_name%type;
  4 emp_salary EMPLOYEE.e_salary%type;
  5 CURSOR emp_details IS SELECT e_no, e_name, e_salary from EMPLOYEE;
  6 begin
  7 OPEN emp_details;
  8 loop
  9 FETCH emp_details into emp_no, emp_name, emp_salary;
 10 exit when emp_details%notfound;
 11 dbms_output.put_line(emp_no || ' ' || emp_name || ' ' || emp_salary);
 12 end loop;
 13 CLOSE emp_details;
 14 end;
 15 /
10001 Juned 105000
10002 Mitesh 55000
10003 Hitesh 45000
10004 Pooja 35000

PL/SQL procedure successfully completed.
```

## **PRACTICAL-12**

### • **Creating Procedures and Functions in PL/SQL**

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the **calling program**.

A subprogram can be created:

- **At the schema level**
- **Inside a package**
- **Inside a PL/SQL block**

At the schema level, 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.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. 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.

### ❖ **Parts of a PL/SQL Subprogram:**

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:

S.No	Parts & Description
1	<b><u>Declarative Part:</u></b> It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.
2	<b><u>Executable Part:</u></b> This is a mandatory part and contains statements that perform the designated action.
3	<b><u>Exception-handling:</u></b> This is again an optional part. It contains the code that handles run-time errors.

### ❖ Creating a Procedure:

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

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

Where,

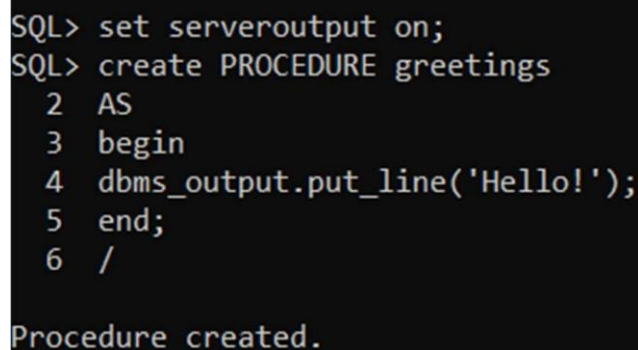
- **procedure-name** specifies the name of the procedure.
- **[OR REPLACE]** option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and **OUT** represents the parameter that will be used to return a value outside of the procedure.
- **procedure-body** contains the executable part.
- The **AS** keyword is used instead of the **IS** keyword for creating a standalone procedure.

### ✓ Example:

- The following example creates a simple procedure that displays the string **'Hello!'** on the screen when executed.

```
set serveroutput on;
create PROCEDURE greetings
AS
begin
dbms_output.put_line('Hello!');
end;
/
```

- When the above code is executed using the SQL prompt, it will produce the following result:



```
SQL> set serveroutput on;
SQL> create PROCEDURE greetings
2 AS
3 begin
4 dbms_output.put_line('Hello!');
5 end;
6 /

Procedure created.
```

### ❖ Executing a Standalone Procedure:

- A standalone procedure can be called in two ways:
  - Using the **EXECUTE** keyword
  - Calling the name of the procedure from a PL/SQL block
- The above procedure named '**greetings**' can be called with the EXECUTE keyword as:  
**EXECUTE greetings;**
- The above call will display:

```
SQL> EXECUTE greetings;
Hello!

PL/SQL procedure successfully completed.
```

- The procedure can also be called from another PL/SQL block:  
**begin**  
**greetings;**  
**end;**  
**/**
- The above call will display:

```
SQL> begin
  2  greetings;
  3  end;
  4  /
Hello!

PL/SQL procedure successfully completed.
```

### ❖ Deleting a Standalone Procedure:

A standalone procedure is deleted with the **DROP PROCEDURE** statement.

- Syntax for deleting a procedure is:  
**DROP PROCEDURE procedure-name;**
- You can drop the greetings procedure by using the following statement:  
**drop PROCEDURE greetings;**

```
SQL> drop PROCEDURE greetings;

Procedure dropped.
```



### ❖ Parameter Modes in PL/SQL Subprograms:

The following table lists out the parameter modes in PL/SQL subprograms:

S.No	Parameter Mode & Description
1	<p><b><u>IN:</u></b></p> <p>An IN parameter lets you pass a value to the subprogram. <b>It is a read-only parameter.</b> Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. <b>It is the default mode of parameter passing. Parameters are passed by reference.</b></p>
2	<p><b><u>OUT:</u></b></p> <p>An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. <b>The actual parameter must be variable and it is passed by value.</b></p>
3	<p><b><u>IN OUT:</u></b></p> <p>An <b>IN OUT</b> parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read.</p> <p>The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. <b>Actual parameter is passed by value.</b></p>

### ✓ IN & OUT Mode Example 1:

- This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

```
declare
a number;
b number;
c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
Begin
if x < y then
z := x;
else
z := y;
end if;
end;
begin
a := 10;
b := 20;
findMin(a, b, c);
dbms_output.put_line('Minimum of (' || a || ', ' || b || ') : ' || c);
end;
/
```

- When the above code is executed at the SQL prompt, it produces the following result:

```
SQL> set serveroutput on;
SQL> declare
  2 a number;
  3 b number;
  4 c number;
  5 PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
  6 begin
  7 if x < y then
  8 z := x;
  9 else
 10 z := y;
 11 end if;
 12 end;
 13 begin
 14 a := 10;
 15 b := 20;
 16 findMin(a, b, c);
 17 dbms_output.put_line('Minimum of (' || a || ', ' || b || ') : ' || c);
 18 end;
 19 /
Minimum of (10, 20) : 10

PL/SQL procedure successfully completed.
```

### ✓ IN & OUT Mode Example 2:

- This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

```
declare
a number;
PROCEDURE squareNum(x IN OUT number) IS
begin
x := x * x;
end;
begin
a := 9;
squareNum(a);
dbms_output.put_line('Square of (9) : ' || a);
end;
/
```

- When the above code is executed at the SQL prompt, it produces the following result:

```
SQL> set serveroutput on;
SQL> declare
  2  a number;
  3  PROCEDURE squareNum(x IN OUT number) IS
  4  begin
  5  x := x * x;
  6  end;
  7  begin
  8  a := 9;
  9  squareNum(a);
 10  dbms_output.put_line('Square of (9) : ' || a);
 11  end;
 12  /
Square of (9) : 81

PL/SQL procedure successfully completed.
```

### ❖ Creating a Function:

A function is same as a procedure except that it returns a value.

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.
- The **AS** keyword is used instead of the **IS** keyword for creating a standalone function.

### ✓ Example:

- We will use the EMPLOYEE table, which we had created earlier:

```
SQL> select * from EMPLOYEE;
```

E_NO	E_NAME	E_SALARY	E_CONTACT_NO	D_NO	J_ID
10001	Juned	105000	9876543210	101	1003
10002	Mitesh	55000	8967452301	101	1003
10003	Hitesh	45000	6543219870	102	1001
10004	Pooja	35000	9870654321	104	1002

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

```
set serveroutput on;
create FUNCTION total_employees
RETURN number IS
total number(2) := 0;
begin
select count(*) into total from EMPLOYEE;
RETURN total;
end;
/
```

- When the above code is executed using the SQL prompt, it will produce the following result:

```
SQL> set serveroutput on;
SQL> create FUNCTION total_employees
  2 RETURN number IS
  3 total number(2) := 0;
  4 begin
  5 select count(*) into total from EMPLOYEE;
  6 RETURN total;
  7 end;
  8 /

Function created.
```

### ❖ Calling a Function:

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value.

- Following program calls the function **total\_employees** from an anonymous block:

```
declare
c number(2);
begin
c := total_employees();
dbms_output.put_line('Total no. of employees : ' || c);
end;
/
```

- When the above code is executed at the SQL prompt, it produces the following result:

```
SQL> declare
  2 c number(2);
  3 begin
  4 c := total_employees();
  5 dbms_output.put_line('Total no. of employees : ' || c);
  6 end;
  7 /
Total no. of employees : 4
```

✓ **Example:**

- The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values:

```
set serveroutput on;
declare
a number;
b number;
c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number IS
z number;
begin
if x > y then
z := x;
else
z := y;
end if;
RETURN z;
end;
begin
a := 10;
b := 20;
c := findMax(a, b);
dbms_output.put_line('Maximum of (' || a || ', ' || b || ') : ' || c);
end;
/
```

- When the above code is executed at the SQL prompt, it produces the following result:

```
SQL> set serveroutput on;
SQL> declare
  2 a number;
  3 b number;
  4 c number;
  5 FUNCTION findMax(x IN number, y IN number)
  6 RETURN number IS
  7 z number;
  8 begin
  9 if x > y then
10 z := x;
11 else
12 z := y;
13 end if;
14 RETURN z;
15 end;
16 begin
17 a := 10;
18 b := 20;
19 c := findMax(a, b);
20 dbms_output.put_line('Maximum of (' || a || ', ' || b || ') : ' || c);
21 end;
22 /
Maximum of (10, 20) : 20

PL/SQL procedure successfully completed.
```

### ❖ PL/SQL Recursive Functions:

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as **recursion**.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number  $n$  is defined as:

$$\begin{aligned} n! &= n*(n-1)! \\ &= n*(n-1)*(n-2)! \\ &\dots \\ &= n*(n-1)*(n-2)*(n-3)\dots 1 \end{aligned}$$

- The following program calculates the factorial of a given number by calling itself recursively:

```
set serveroutput on;
declare
num number;
factorial number;
FUNCTION fact(x number)
RETURN number IS
f number;
begin
if x = 0 then
f := 1;
else
f := x * fact(x - 1);
end if;
RETURN f;
end;
begin
num := 9;
factorial := fact(num);
dbms_output.put_line('Factorial of ' || num || ' is ' || factorial);
end;
/
```

- When the above code is executed at the SQL prompt, it produces the following result:

```
SQL> set serveroutput on;
SQL> declare
  2 num number;
  3 factorial number;
  4 FUNCTION fact(x number)
  5 RETURN number IS
  6 f number;
  7 begin
  8 if x = 0 then
  9 f := 1;
 10 else
 11 f := x * fact(x - 1);
 12 end if;
 13 RETURN f;
 14 end;
 15 begin
 16 num := 9;
 17 factorial := fact(num);
 18 dbms_output.put_line('Factorial of ' || num || ' is ' || factorial);
 19 end;
 20 /
Factorial of 9 is 362880

PL/SQL procedure successfully completed.
```



## **PRACTICAL-13**

### • **Creating Database Triggers**

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

- A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
- A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
- A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

#### ❖ **Benefits of Triggers:**

Triggers can be written for the following purposes:

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

#### ❖ **Creating Triggers:**

The syntax for creating a trigger is:

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
    Declaration-statements
BEGIN
    Executable-statements
EXCEPTION
    Exception-handling-statements
END;
```

Where,

- **CREATE [OR REPLACE] TRIGGER trigger\_name:** Creates or replaces an existing trigger with the *trigger\_name*.

- **{BEFORE | AFTER | INSTEAD OF}**: This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- **{INSERT [OR] | UPDATE [OR] | DELETE}**: This specifies the DML operation.
- **[OF col\_name]**: This specifies the column name that will be updated.
- **[ON table\_name]**: This specifies the name of the table associated with the trigger.
- **[REFERENCING OLD AS o NEW AS n]**: This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- **[FOR EACH ROW]**: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise, the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- **WHEN (condition)**: This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

### ✓ Example:

- To start with, we will be using the EMPLOYEE table we had created and used in the previous chapters:

```
SQL> select * from EMPLOYEE;
```

E_NO	E_NAME	E_SALARY	E_CONTACT_NO	D_NO	J_ID
10001	Juned	105000	9876543210	101	1003
10002	Mitesh	55000	8967452301	101	1003
10003	Hitesh	45000	6543219870	102	1001
10004	Pooja	35000	9870654321	104	1002

- The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the EMPLOYEE table. This trigger will display the salary difference between the old values and new values:

```
set serveroutput on;
create TRIGGER display_salary_changes
BEFORE delete OR insert OR update on EMPLOYEE
FOR EACH ROW
when (new.e_no > 0)
declare
sal_diff number;
begin
sal_diff := :new.e_salary - :old.e_salary;
dbms_output.put_line('Old salary: ' || :old.e_salary);
dbms_output.put_line('New salary: ' || :new.e_salary);
dbms_output.put_line('Salary difference: ' || sal_diff);
end;
/
```

- When the above code is executed at the SQL prompt, it produces the following result:

```
SQL> set serveroutput on;
SQL> create TRIGGER display_salary_changes
  2 BEFORE delete OR insert OR update on EMPLOYEE
  3 FOR EACH ROW
  4 when (new.e_no > 0)
  5 declare
  6 sal_diff number;
  7 begin
  8 sal_diff := :new.e_salary - :old.e_salary;
  9 dbms_output.put_line('Old salary: ' || :old.e_salary);
10 dbms_output.put_line('New salary: ' || :new.e_salary);
11 dbms_output.put_line('Salary difference: ' || sal_diff);
12 end;
13 /

Trigger created.
```

- The following points need to be considered here:
  - OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
  - If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
  - The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table

### ❖ Triggering a Trigger:

- Let us perform some DML operations on the EMPLOYEE table. Here is one INSERT statement, which will create a new record in the table:

```
insert into EMPLOYEE values('&e_no', '&e_name', '&e_salary', '&e_contact_no',  
'&d_no', '&j_id');
```

- When a record is created in the EMPLOYEE table, the above created trigger, **display\_salary\_changes** will be fired and it will display the following result:

```
SQL> insert into EMPLOYEE values('&e_no', '&e_name', '&e_salary', '&e_contact_no', '&d_no', '&j_id');  
Enter value for e_no: 10005  
Enter value for e_name: Jainil  
Enter value for e_salary: 70000  
Enter value for e_contact_no: 9999988888  
Enter value for d_no: 105  
Enter value for j_id: 1005  
old 1: insert into EMPLOYEE values('&e_no', '&e_name', '&e_salary', '&e_contact_no', '&d_no', '&j_id')  
new 1: insert into EMPLOYEE values('10005', 'Jainil', '70000', '9999988888', '105', '1005')  
Old salary:  
New salary: 70000  
Salary difference:  
  
1 row created.
```

- Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the EMPLOYEE table. The UPDATE statement will update an existing record in the table:

```
update EMPLOYEE set e_salary = e_salary + 5000 where e_no = 10005;
```

- When a record is updated in the EMPLOYEE table, the above created trigger, **display\_salary\_changes** will be fired and it will display the following result:

```
SQL> update EMPLOYEE set e_salary = e_salary + 5000 where e_no = 10005;  
Old salary: 70000  
New salary: 75000  
Salary difference: 5000  
  
1 row updated.
```