Exercise-7

Program development using WHILE LOOPS, numeric FOR LOOPS, nested loops using ERROR Handling, BUILT –IN Exceptions, USER defined Exceptions, RAISE- APPLICATION ERROR.

1. **WHILE LOOP**: A **WHILE LOOP** statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

Syntax:

```
WHILE condition LOOP
    sequence_of_statements
END LOOP;
```

a) A PL/SQL Program to find sum of ODD number upto given number using While loop

Program:

```
set serveroutput on;
DECLARE
  num NUMBER(3) := 1;
  sumvar NUMBER(4) := 0;

BEGIN
  dbms_output.put_line('The odd numbers are : ');
    WHILE num <= 7 LOOP
      dbms_output.put_line(num);
      sumvar := sumvar+num;
      num := num + 2;
    END LOOP;
dbms_output.put_line('Sum of odd numbers is '|| sumvar);
END;
/
```

Output:

```
SQL> @E:GSK\odd.sql
The odd numbers are :
1
3
5
7
Sum of odd numbers is 16

PL/SQL procedure successfully completed.
```

2) FOR Loop: A **FOR LOOP** is a repetition control structure that allows us to efficiently write a loop that needs to execute a specific number of times.

Syntax

```
FOR counter IN initial_value .. final_value LOOP
sequence_of_statements;
END LOOP;
```

b) A PL/SQL code to print multiplication table using for loop

Program:

```
declare
i number;
n number;
begin
n:=&n;
for i in 1..10
loop
 dbms_output.put_line( n || ' * ' || i || ' = ' || n*i);
end loop;
end;
/
```

Output:

```
SQL> @E:GSK\mul.sql
Enter value for n: 5
old   5: n:=&n;
new   5: n:=5;
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
```

5 * 5 = 25

5 * 6 = 30

5 * 7 = 35

5 * 8 = 40

5 * 9 = 45

5 * 10 = 50

PL/SQL procedure successfully completed.

3) **NESTED LOOP:** PL/SQL allows using one loop inside another loop. It may be either basic, while or for loop.

Syntax for a nested FOR LOOP

```
FOR counter1 IN initial_value1 .. final_value1 LOOP
  sequence_of_statements1
  FOR counter2 IN initial_value2 .. final_value2 LOOP
    sequence_of_statements2
  END LOOP;
END LOOP;
```

Syntax for nested WHILE LOOP:

```
WHILE condition1 LOOP sequence_of_statements1
      WHILE condition2 LOOP sequence_of_statements2
END LOOP;
END LOOP;
```

c) A PL/SQL program to print n prime number using nested loop.

Program:

```
DECLARE
  i number(3);
  j number(3);
BEGIN
  i := 2;
  LOOP
    j:= 2;
    LOOP
      exit WHEN ((mod(i, j) = 0) or (j = i));
      j := j +1;
    END LOOP;
  IF (j = i ) THEN
```

```
      dbms_output.put_line(i || ' is prime');
    END IF;
    i := i + 1;
    exit WHEN i = 50;
    END LOOP;
END;
/
```

Output:

```
SQL> @E:GSK\prime.sql
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime

PL/SQL procedure successfully completed.
```

4) Exception Handling

In exception is an error condition during a program execution. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions −

- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling

The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can handle. The default exception will be handled using ***WHEN others THEN*** –

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling goes here >

WHEN exception1 THEN

exception1-handling-statements

WHEN exception2 THEN

exception2-handling-statements

.....

WHEN others THEN

exception3-handling-statements

END;

d) Write a PL/SQL program to implement BUILT –IN Exceptions Program.

```
SET SERVEROUTPUT ON;
DECLARE
 EMPID EMP1.ENO%TYPE;
BEGIN
 SELECT ENO INTO EMPID FROM EMP1 WHERE ENO=105;
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('NO RECORD FOUND IN THE EMP1
TABLE');
END;
/
```

Output:

```
SQL> @E:GSK\NO.sql
NO RECORD FOUND IN THE EMP1 TABLE

PL/SQL procedure successfully completed.
```

5) User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure **DBMS_STANDARD.RAISE_APPLICATION_ERROR**.

The syntax for declaring an exception is −

```
DECLARE
   my-exception EXCEPTION;
```

e) Write a PL/SQL program to implement USER defined Exceptions

Program

```
Set serveroutput on;
declare
 zero_price exception;
   eid emp1.eno%type;
  begin
     select eno into eid from emp1 where eno=101;
 if eid=101 then
   raise zero_price;
  end if;
  exception
  when zero_price then
  dbms_output.put_line('RAISED ZERO-PRICE USER DEFINED
EXCEPTION');
  end;
/
```

Output:

```
SQL> @E:GSK\USER.sql
RAISED ZERO-PRICE USER DEFINED EXCEPTION

PL/SQL procedure successfully completed.
```

6) Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax for raising an exception –

```
DECLARE

exception_name EXCEPTION;

BEGIN

IF condition THEN

        RAISE exception_name;

END IF;

EXCEPTION

WHEN exception_name THEN

         statement;

END;
```

f) Write a PL/SQL program to implement RAISE- APPLICATION ERROR.

Program:

```
DECLARE
   mynumber EXCEPTION;
   n NUMBER :=10;

BEGIN
   FOR i IN 1..n LOOP
   dbms_output.put_line(i*i);
      IF i*i=36 THEN
      RAISE mynumber;
      END IF;
   END LOOP;
```

```
EXCEPTION
   WHEN mynumber THEN
      RAISE_APPLICATION_ERROR(-20015, 'I can raise my own number
exception');

END;
/
```

Output:

```
SQL> @E:GSK\raise.sql
1
4
9
16
25
36
DECLARE
*
ERROR at line 1:
ORA-20015: I can raise my own number exception
ORA-06512: at line 15
```

EXERCISE -8

A **stored procedure** or in simple a **proc** is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages.

A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block.

A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.

General Syntax to create a procedure is:

```
CREATE [OR REPLACE] PROCEDURE proc_name [list of parameters]
IS
   Declaration section
BEGIN
   Execution section
EXCEPTION
  Exception section
END;
```

IS - marks the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

The syntax within the brackets [ ] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

Procedures: Passing Parameters

In PL/SQL, we can pass parameters to procedures and functions in three ways.

1) **IN type parameter:** These types of parameters are used to send values to stored procedures.

General syntax to pass a IN parameter is

CREATE [OR REPLACE] PROCEDURE procedure_name (
param_name1 IN datatype, param_name12 IN datatype ...)

- param_name1, param_name2... are unique parameter names.
- datatype - defines the datatype of the variable.
- IN - is optional, by default it is a IN type parameter.

2) OUT type parameter: These types of parameters are used to get values from stored procedures. This is similar to a return type in functions.

The General syntax to create an OUT parameter is

CREATE [OR REPLACE] PROCEDURE proc2 (param_name OUT datatype)

The parameter should be explicity declared as OUT parameter.

3) IN OUT parameter: These types of parameters are used to send values and get values from stored procedures.A procedure may or may not return any value.

The General syntax to create an IN OUT parameter is

CREATE [OR REPLACE] PROCEDURE proc3 (param_name IN OUT datatype)

Deleting a Standalone Procedure

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

For dropping/deleting Procedure

SYNTAX:

DROP PROCEDURE Pro_Name;

1. **PL/SQL program for the creation of procedures**

```
SQL>  create or replace procedure high(a number,b number) is
   begin
   if a>b then
   dbms_output.put_line('max value iS:='||a);
   else
   dbms_output.put_line('max value iS:='||b);
   end if;
   end;
   /
```

OUTPUT:

```
Procedure created.
SQL> exec high(20,10);
max value iS:=20
PL/SQL procedure successfully completed.
```

2. **PL/SQL Program to illustrate Procedure for passing parameters with IN mode**

```
SQL> create or replace procedure fact(n in number) is
   fact number:=1;
   i number;
    begin
   for i in 1..n loop
    fact:=fact * i;
   end loop;
   dbms_output.put_line('the factorial value is'||fact);
   end;
    /
```

OUTPUT:

```
Procedure created.
SQL> exec fact (10);
the factorial value is3628800
PL/SQL procedure successfully completed.
```

3. **PL/SQL Program to illustrate Procedure for passing parameters with IN and IN OUT of Procedures**

```
DECLARE
        a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
        x := x * x;
END;
BEGIN
        a:= 23;
        squareNum(a);
        dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

Output:

```
Square of (23): 529

PL/SQL procedure successfully completed.
```

4. **PL/SQL Program to illustrate Procedure for passing parameters with IN and OUT of Procedures**

```
SQL> create or replace procedure fact (n in number,f out number) is
   f1 number:=1;
   i number;
   begin
   for i in 1..n loop
   f1:=f1 * i;
   end loop;
   f:=f1;
  end;
  /
```

OUTPUT:

Procedure created.

```
SQL> declare
   n number:=&n;
   f number;
   begin
   fact(n,f);
   dbms_output.put_line('the factorial is'||f);
   end;
   /
```

OUTPUT:

```
Enter value for n: 5
old   2: n number:=&n;
new   2: n number:=5;
the factorial is120
```

PL/SQL procedure successfully completed.

5. **PL/SQL Program to illustrate Procedure for passing parameters with IN and IN OUT of Procedures.**

```
SQL> create or replace procedure fact(n in number,f in out number) is
   f1 number;
   i number;
   begin
   f1:=f;
   for i in 1..n loop
   f1:=f1 * i;
   end loop;
   f:=f1;
  end;
  /
```

OUTPUT:

Procedure created.

```
SQL> declare
   n number:=&n;
   f number:=1;
   begin
   fact(n,f);
   dbms_output.put_line('factorial value is:'||f);
   end;
    /
```

OUTPUT:

```
Enter value for n: 6
old   2: n number:=&n;
new   2: n number:=6;
```

factorial value is:720

PL/SQL procedure successfully completed.

EXPERIMENT-9

AIM: Program development using creation of stored functions, invoke functions in SQL Statements and write complex functions.

Creating a Function

A standalone function is created using the **CREATE FUNCTION** statement.

CREATE OR REPLACE FUNCTION <function_name>

(<variable_name> IN <datatype>,

<variable_name> IN <datatype>,...)

RETURN <datatype> IS/AS

variable/constant declaration;

BEGIN

    -- PL/SQL subprogram body;

EXCEPTION

    -- Exception Handling block ;

END <function_name>;

Let's understand the above code,

- **function_name** is for defining function's name and **variable_name** is the variable name for variable used in the function.
- CREATE or REPLACE FUNCTION is a keyword used for specifying the name of the function to be created.
- **IN** mode refers to **READ ONLY mode** which is used for a variable by which it will accept the value from the user. It is the default parameter mode.
- RETURN is a keyword followed by a datatype specifying the datatype of a value that the function will return.

Example1: The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

SQL> select * from customers;

ID	NAME	AGE	ADDRESS	SALARY
1	Abhi	32	Hyderabad	7000
2	John	25	Mumbai	5000
3	Geeta	23	Chennai	9000
4	Sana	26	Bangalore	8000
5	Sai	27	Delhi	7000

SQL> desc customers;

Name	Null?	Type
ID		NUMBER(38)
NAME		VARCHAR2(20)
AGE		NUMBER(38)
ADDRESS		VARCHAR2(20)
SALARY		NUMBER(6,2)

```
--PL/SQL Program to create function
set serveroutput on;
CREATE or REPLACE FUNCTION totalCustomers
return number IS
        total number(2):=0;
BEGIN
        select count(*) into total from customers;
        return total;
END;
/
```

```
--Calling function
set serveroutput on;
DECLARE
        c number(2);
BEGIN
        c:=totalCustomers();
        dbms_output.put_line('Total number of CUstomers = '||c);
END;
/
```

Output:

```
SQL> @E:\DIET\Dbmsexp9\9a.sql;

Function created.


SQL> @E:\DIET\Dbmsexp9\9b.sql;

Total number of CUstomers = 5

PL/SQL procedure successfully completed.
```

Example2: The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the addition of two values.

```
--Creating function
set serveroutput on;
CREATE OR REPLACE FUNCTION Addition(a IN number, b IN number) RETURN Number
IS c number;
BEGIN
        c := a+b;
        RETURN c;
END;
/
```

Output:

```
SQL> @E:\DIET\Dbmsexp9\9e.sql;
Function created.
--Function Calling
set serveroutput on;
DECLARE
        no1 number;
        no2 number;
        result number;
BEGIN
        no1 := &no1;
        no2 := &no2;
        result := Addition(no1,no2);
dbms_output.put_line('Sum of two nos='||result);
END;
/
```

Output:

```
SQL> @E:\DIET\Dbmsexp9\9e2.sql;

Enter value for no1: 15

old   6:       no1 := &no1;

new   6:        no1 := 15;

Enter value for no2: 18

old   7:       no2 := &no2;

new   7:        no2 := 18;

Sum of two nos=33

PL/SQL procedure successfully completed.
```

EXPERIMENT-10

AIM: Develop programs using features parameters in a CURSOR, FOR UPDATE CURSOR, WHERE CURRENT of clause and CURSOR variables.

A **cursor** is a pointer to this context area. 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	**%FOUND** 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	**%NOTFOUND** 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.

%ISOPEN

3 Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.

%ROWCOUNT

4 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**

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 c_customers IS
  SELECT id, name, address FROM customers;
```

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 c_customers;
```

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 c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

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

```
CLOSE c_customers;
```

Creating customers table:

```
SQL> select * from customers;

      ID NAME                 AGE ADDRESS              SALARY
---------- -------------------- ---------- -------------------- ----------
         1 Abhi                 32 Hyderabad             7000
         2 John                 25 Mumbai                5000
         3 Geeta                 23 Chennai              9000
         4 Sana                 26 Bangalore             8000
         5 Sai                  27 Delhi                 7000
```

Implicit Cursors :

```
--Example for implicit cursor
set serveroutput on;
DECLARE
  total_rows number(2);
BEGIN
  UPDATE customers
  SET salary = salary + 500;
```

```
  IF sql%notfound THEN
    dbms_output.put_line('no customers selected');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers selected ');
  END IF;
END;
/
```

OUTPUT:

```
SQL> @E:\DIET\Dbmsexp10\10a.sql;

5 customers selected

PL/SQL procedure successfully completed.
```

Explicit Cursors:

```
--Example for explicit cursors

set serveroutput on;
DECLARE
  c_id customers.id%type;
  c_name customers.name%type;
  c_addr customers.address%type;
  CURSOR c_customers is
    SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
  FETCH c_customers into c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
```

```
  CLOSE c_customers;
END;
/
```

OUTPUT:

```
SQL> @E:\DIET\Dbmsexp10\10b.sql;

1 Abhi Hyderabad

2 John Mumbai

3 Geeta Chennai

4 Sana Bangalore

5 Sai Delhi


PL/SQL procedure successfully completed.
```

Exercise - 11

Develop programs using before and after triggers, row and statement triggers and instead of triggers.

Triggers in oracle are blocks of PL/SQL code which oracle engine can execute automatically based on some action or event.

These events can be:

- DDL statements (CREATE, ALTER, DROP, TRUNCATE)
- DML statements (INSERT, SELECT, UPDATE, DELETE)
- Database operation like connecting or disconnecting to oracle (LOGON, LOGOFF, SHUTDOWN)

Triggers are automatically and repeatedly called upon by oracle engine on satisfying certain condition.

Triggers can be activated or deactivated depending on the requirements.

If triggers are activated then they are executed implicitly by oracle engine and if triggers are deactivated then they are executed explicitly by oracle engine.

Types of Triggers in Oracle

Triggers can be classified based on the following parameters.

- Classification based on the **timing**
  - BEFORE Trigger: It fires before the specified event has occurred.
  - AFTER Trigger: It fires after the specified event has occurred.
  - INSTEAD OF Trigger: A special type. You will learn more about the further topics. (only for DML)
- Classification based on the **level**
  - STATEMENT level Trigger: It fires once for the specified event statement.
  - ROW level Trigger: It fires for each record that got affected in the specified event. (only for DML)
- Classification based on the **Event**
  - DML Trigger: It fires when the DML event is specified (INSERT/UPDATE/DELETE)
  - DDL Trigger: It fires when the DDL event is specified (CREATE/ALTER)

- DATABASE Trigger: It fires when the database event is specified (LOGON/LOGOFF/STARTUP/SHUTDOWN).

How to Create Trigger

Below is the syntax for creating a trigger.

Syntax:

```
CREATE [ OR REPLACE ] TRIGGER <trigger_name>

[ BEFORE | AFTER | INSTEAD OF ]

[ INSERT | UPDATE | DELETE.....]

ON <name of underlying object>

[ FOR EACH ROW ]

[ WHEN <condition for trigger to get execute> ]

DECLARE
    <Declaration part>
BEGIN
    <Execution part>
EXCEPTION
    <Exception handling part>
END;
```

Trigger Timing

Event

Row Level

Conditional Clause

Syntax Explanation:

- The above syntax shows the different optional statements that are present in trigger creation.
- BEFORE/ AFTER will specify the event timings.
- INSERT/UPDATE/LOGON/CREATE/etc. will specify the event for which the trigger needs to be fired.
- ON clause will specify on which object the above-mentioned event is valid. For example, this will be the table name on which the DML event may occur in the case of DML Trigger.
- Command “FOR EACH ROW” will specify the ROW level trigger.
- WHEN clause will specify the additional condition in which the trigger needs to fire.
- The declaration part, execution part, exception handling part is same as that of the other PL/SQL blocks. Declaration part and exception handling part are optional.

:NEW and :OLD Clause

In a row level trigger, the trigger fires for each related row. And sometimes it is required to know the value before and after the DML statement.

Oracle has provided two clauses in the RECORD-level trigger to hold these values. We can use these clauses to refer to the old and new values inside the trigger body.

- :NEW – It holds a new value for the columns of the base table/view during the trigger execution
- :OLD – It holds old value of the columns of the base table/view during the trigger execution

This clause should be used based on the DML event. Below table will specify which clause is valid for which DML statement (INSERT/UPDATE/DELETE).

	INSERT	UPDATE	DELETE
:NEW	VALID	VALID	INVALID. There is no new value in delete case.
:OLD	INVALID. There is no old value in insert case	VALID	VALID

Drop Trigger

To drop a trigger

Syntax:

Drop trigger trigger_name;

SQL> create table customers(id number(3), name varchar2(10), age number(3), address varchar2(10), salary number(10,2));

Table created.

SQL> insert into customers values(1,'ramesh',32,'ahmedabad',2000);

1 row created.

SQL> insert into customers values(2,'khilan',25,'Delhi',1500);

1 row created.

SQL> insert into customers values(3,'kaushik',23,'Kota',2000);

1 row created.

SQL> insert into customers values(4,'chitali',25,'Mumbai',6500);

1 row created.

SQL> select * from customers;

ID	NAME	AGE	ADDRESS	SALARY
1	ramesh	32	ahmedabad	2000
2	khilan	25	Delhi	1500
3	kaushik	23	Kota	2000
4	chitali	25	Mumbai	6500

4 rows selected.

PL/SQL Code for creation of trigger while insert / update records into a table.

SQL>

```
CREATE OR REPLACE TRIGGER display_salary_changes

BEFORE DELETE OR INSERT OR UPDATE ON customers

FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
sal_diff number;
BEGIN
  sal_diff := :NEW.salary - :OLD.salary;
  dbms_output.put_line('Old salary: ' || :OLD.salary);
  dbms_output.put_line('New salary: ' || :NEW.salary);
  dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

SQL> Trigger created

SQL> insert into customers values(5,'Hardik',27,'Mumbai',5500);

Old salary:

New salary: 5500

Salary difference:

1 row created.

SQL> update customers set salary=salary+500 where id=2;

Old salary: 1500

New salary: 2000

Salary difference: 500

1 row updated.

SQL> select * from customers;

ID	NAME	AGE	ADDRESS	SALARY
1	ramesh	32	ahmedabad	2000
2	khilan	25	Delhi	2000
3	kaushik	23	Kota	2000
4	chitali	25	Mumbai	6500
5	Hardik	27	Mumbai	5500

SQL> delete from customers where id=5;

1 row deleted.

SQL> select * from customers;

ID	NAME	AGE	ADDRESS	SALARY
1	ramesh	32	ahmedabad	2000
2	khilan	25	Delhi	2000
3	kaushik	23	Kota	2000
4	chitali	25	Mumbai	6500

SQL> drop trigger display_salary_changes;

Trigger dropped.

Exercise - 12

Create a table and perform the search operation on table using indexing and non-indexing techniques.

The SQL Indexes

SQL Indexes are special lookup tables that are used to speed up the process of data retrieval. They hold pointers that refer to the data stored in a database, which makes it easier to locate the required data records in a database table.

The CREATE INDEX Statement

An index in SQL can be created using the **CREATE INDEX** statement. This statement allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in an ascending or descending order.

Preferably, an index must be created on column(s) of a large table that are frequently queried for data retrieval.

Syntax

The basic syntax of a **CREATE INDEX** is as follows –

Create index index_name on Table_name(column_name asc/desc);

Drop Index

Index can be drop by using the syntax:

SQL>drop index index_name;

```
CREATE TABLE Books (id INT PRIMARY KEY NOT NULL, name
VARCHAR(50) NOT NULL, category VARCHAR(50) NOT NULL, price INT
NOT NULL);

SQL> INSERT INTO Books VALUES(1, 'Book1', 'Cat1', 1800);

1 row created.

SQL> INSERT INTO Books VALUES(2, 'Book2', 'Cat2', 1500);

1 row created.

SQL> INSERT INTO Books VALUES (3, 'Book3', 'Cat3', 2000);

1 row created.

SQL> INSERT INTO Books VALUES (4, 'Book4', 'Cat4', 1300);

1 row created.


SQL> INSERT INTO Books VALUES (5, 'Book5', 'Cat5', 1500);

1 row created.

SQL> Set timing on;

SQL> select * from Books;

        ID NAME

---------- --------------------------------------------------

CATEGORY                                             PRICE

-------------------------------------------------- ----------

         1 Book1

Cat1                                               1800
```

```
         2 Book2
Cat2                                    1500

         3 Book3
Cat3                                    2000
        ID NAME
---------- --------------------------------------------------
CATEGORY                                 PRICE
-------------------------------------------------- ----------
         4 Book4
Cat4                                    1300

         5 Book5
Cat5                                    1500

Elapsed: 00:00:00.05
SQL> CREATE INDEX indbooks on Books (price ASC);
Index created.
Elapsed: 00:00:00.05
SQL> select * from books;
        ID NAME
---------- --------------------------------------------------
CATEGORY                                 PRICE
-------------------------------------------------- ----------
         1 Book1
Cat1                                    1800
```

```
         2 Book2
Cat2                                    1500

         3 Book3
Cat3                                    2000
        ID NAME
---------- --------------------------------------------------
CATEGORY                                 PRICE
-------------------------------------------------- ----------
         4 Book4
Cat4                                    1300

         5 Book5
Cat5                                    1500

Elapsed: 00:00:00.03

 SQL> drop index indbooks;

Index dropped.
```