EX: NO: 1 DATE:	DDL& DML COMMANDS

AIM

To execute and verify the Data Definition Language and Data Manipulation Language commands.

COMMANDS:

DDL (DATA DEFINITION LANGUAGE**) & DML (**DATA MANIPULATION LANGUAGE**)**

- INSERT
- DELETE
- MODIFY
- ALTER
- UPDATE
- VIEW

CREATE COMMANDS

DESCRIPTION :

This command is used to create the table.

SYNTAX :

Create table<table name>(attribute1 datatype 1…..attribute n datatype n);

Example:

SQL> create table stu(name varchar(10),dept varchar2(4),mark number(3),total number(3));

Table created.

INSERT COMMANDS

DESCRIPTION :

This command is used to insert the values in the table.

SYNTAX :

Insert into tablename values('&column1',…..'&column n');

Example

SQL> insert into stu values('neema','cse',89,500);
1 row created.

SQL> insert into stu values('ramu','it',80,600);
1 row created.

SQL> select * from stu;

NAME	DEPT	MARK	TOTAL
neema	cse	89	500
ramu	it	80	600

2 rows selected.

DELETE COMMAND

Description :

This command is used to delete particular row based on condition.

Syntax :

Delete from <table name> where condition;

Example:

SQL> delete stu where total=550;
1 row deleted.

ALTER & MODIFY COMMANDS :

DESCRIPTION :

This command is used to alter table such as

a). To modify datatype, size:

Description :

This command is used to modify the datatype of the column.

Syntax :

Alter table<table name> modify(attribute name datatype);

Example:

SQL> alter table stu modify(dept varchar(5));

Table altered.

b). To add column :

Description :

This command is used to add new column in the table.

Syntax :

Alter table<table name>add (attribute name,datatype);

Example:

SQL> alter table stu add (avg float(5));

Table altered.

SQL> desc stu;

Name	Null?	Type

NAME	NOT NULL	VARCHAR2(20)
DEPT		VARCHAR2(5)
MARK		NUMBER(3)
TOTAL		NUMBER(3)
ROLLNO		NUMBER(3)
AVG		FLOAT(5)

UPDATE COMMAND

Description :

This command is used to update the single row/ multiple rows in the table.

Syntax :

Update <table name>set field=values where condition; //single

Update <table name>set field=values where (conditions);//multiple

Example:

SQL> update stu set dept='ece' where mark=65;
1 row updated.

SQL> select * from stu;

NAME	DEPT	MARK	TOTAL
neema	cse	89	500
neema	cse	89	500
ramu	it	80	600
raju	cse	60	700
ramya	ece	65	500

SQL> update stu set dept='eee' where (mark=65 or mark=60);

2 rows updated.

SQL> select * from stu;

NAME	DEPT	MARK	TOTAL
neema	cse	89	500
neema	cse	89	500
ramu	it	80	600
raju	eee	60	700
ramya	eee	65	500

VIEW COMMAND

Description :

This command is used to update the single row/ multiple rows in the table.

Syntax :

Create view <view name> as select <attribute name> from <table name> where condition;

OR

Create or replace view <view name> as select <attribute name> from <table name> where
condition;

```
SQL> create view v1 as select name from stu where mark=89;
View created.

SQL> select * from v1;
NAME
----------
neema
neema

SQL> create or replace view v1 as select name from stu where mark>80;
View created.

SQL> select * from v1;

NAME
----------
neema
neema
```

TCL statements

The TCL language is used for controlling the access to the table and hence securing the database. TCL is used to provide certain privileges to a particular user. Privileges are rights to be allocated. The privilege commands are namely,

- Grant
- Revoke
- Commit
- Savepoint
- Rollback

GRANT COMMAND: It is used to create users and grant access to the database. It requires database administrator (DBA) privilege, except that a user can change their password. A user can grant access to their database objects to other users.

REVOKE COMMAND: Using this command, the DBA can revoke the granted database privileges from the user.

COMMIT: It is used to permanently save any transaction into database.

SAVEPOINT: It is used to temporarily save a transaction so that you can rollback to that point whenever necessary

ROLLBACK: It restores the database to last committed state. It is also use with savepoint

command to jump to a savepoint in a transaction

GRANT COMMAND

Grant < database_priv [database_priv…..] > to <user_name> identified by <password> [,<password…..];
Grant <object_priv> | All on <object> to <user | public> [With Grant Option];

REVOKE COMMAND

Revoke <database_priv> from <user [, user] >;
Revoke <object_priv> on <object> from < user | public >;
<database_priv> -- Specifies the system level priveleges to be granted to the users or roles. This includes create / alter / delete any object of the system.
<object_priv> -- Specifies the actions such as alter / delete / insert / references / execute / select / update for tables.
<all> -- Indicates all the priveleges.
[With Grant Option] – Allows the recipient user to give further grants on the objects.
The priveleges can be granted to different users by specifying their names or to all users by using the "Public" option.

Examples

Consider the following tables namely "DEPARTMENTS" and "EMPLOYEES" Their schemas are as follows ,
Departments (dept _no , dept_ name , dept_location);
Employees (emp_id , emp_name , emp_salary);
SQL> Grant all on employees to abcde;
Grant succeeded.
SQL> Grant select , update , insert on departments to abcde with grant option;
Grant succeeded.

SQL> Revoke all on employees from

abcde; Revoke succeeded.

SQL> Revoke select , update , insert on departments from abcde;

Revoke succeeded.

RESULT:

Thus the Data Definition Language and Data Manipulation Language commands are executed and verified.

EX: NO: 2 DATE:	QUERIES AND JOINS

AIM:

To perform join operations using SQL.

EQUIJOIN:

SYNTAX:

Select attribute 1, 2, …., from table name1,table name2 where t1=t2 attribute;

DESCRIPITION:

A equi join is a SQL join.

NON EQUI JOIN:

SYNTAX:

Select attribute1, attribute2.....from table 1, table2 where table1 attribute <> table2 attribute

DESCRIPTION:

A non equi join is a SQL join whose condition is established using all comparison operator except the equal (=) operations like <=,>=,<,>

SELF JOIN

SYNTAX:

Select distinct t1 attributes _name1 t1 attributes name from table name t1 where attribute name1=t2.attribute_name and t1.attribute_name in<> t2.attribute_name in

DESCRIPTION:

A self join is a type of SQL join which is used to join a table to itself particularly when the used table has a foreign key the references its own primary key. It compares values with the column of a single table.

INNER JOIN:

SYNTAX:

Select attribute1, attribute 2.....from table 1 left inner join table2 on condition.

DESCRIPTION:

Inner join returns the matching rows from the tables that are being joined.

OUTER JOIN:

1. **Left outer join**

SYNTAX:

Select attribute1, attribute 2.....from table 1 left outer join table2 on condition.

DESCRIPTION:

Left outer join returns the matching rows from the tables that are being joined and also non matching rows from the left table in the result and the place NULL values in the attribute that came from the right table.

2. Right outer join

SYNTAX:

Select attribute1, attribute 2.....from table 1 right outer join table2 on condition.

DESCRIPTION:

Right outer join returns the matching rows from the tables that are being joined and also non matching rows from the right table in the result and the place NULL values in the attribute that came from the left table.

3. Full outer join

SYNTAX:

Select attribute1, attribute 2.....from table 1 full outer join table2 on condition.

DESCRIPTION:

The full outer join returns the matching rows from the tables that are being joined and also non matching rows from the left and right table in the result and the place NULL values in the attribute that came from the right & left table.

Example:

```
SQL> create table eem(eno number(3),ename varchar2(10),dept varchar2(5));
Table created.
SQL> create table sal(eno number(3),sal number(6),dept varchar(10));
Table created.
SQL> select * from eem;
    ENO   ENAME      DEPT
   --------- ----------     -----
      1          aaa       cse
      2          bbb       it
      3          ccc       cse
SQL> select * from sal;
    ENO        SAL            DEPT
---------        ---------          ----------
      1          30000          cse
      2          50000          it
      5          60000          cse
```

Equi join:SQL> select eem.eno,ename,sal from eem,sal where eem.eno=sal.eno;

```
    ENO              ENAME              SAL
   ---------          ----------          ---------
     1                aaa                30000
```

2 bbb 50000

Non equi join:

SQL> select eem.eno,ename,sal from eem,sal where eem.eno<>sal.eno;

ENO	ENAME	SAL
1	aaa	50000
1	aaa	60000
2	bbb	30000
2	bbb	60000
3	ccc	30000
3	ccc	50000
3	ccc	60000

7 rows selected.

SQL> select * from eem,sal where eem.eno<>sal.eno;

ENO	ENAME	DEPT	ENO	SAL	DEPT
1	aaa	cse	2	50000	it
1	aaa	cse	5	60000	cse
2	bbb	it	1	30000	cse
2	bbb	it	5	60000	cse
3	ccc	cse	1	30000	cse
3	ccc	cse	2	50000	it
3	ccc	cse	5	60000	cse

7 rows selected.

Inner join:

SQL> select eem.eno,ename,sal from eem inner join sal on eem.eno=sal.eno;

ENO	ENAME	SAL
1	aaa	30000
2	bbb	50000

Outer join:

1. **Left outer join**

SQL> select eem.eno,ename,sal from eem left join sal on eem.eno=sal.eno;

ENO	ENAME	SAL
1	aaa	30000
2	bbb	50000
3	ccc	

2. **Right outer join**

SQL> select eem.eno,ename,sal from eem right join sal on eem.eno=sal.eno;

ENO	ENAME	SAL

1	aaa	30000
2	bbb	50000
		60000

Full outer join

SQL> select eem.eno,ename,sal from eem full join sal on eem.eno=sal.eno;

ENO	ENAME	SAL
1	aaa	30000
2	bbb	50000
3	ccc	60000

RESULT:

Thus the join operations has been verified and executed successfully.

EX: NO: 3 DATE:	VIEWS, SYNONYMS AND SEQUENCE

AIM

To execute and verify the SQL commands for Views, indexes, sequence and synonyms.

OBJECTIVE

- Views Helps to encapsulate complex query and make it reusable.
- Provides user security on each view - it depends on your data policy security.
- Using view to convert units - if you have a financial data in US currency, you can create view to convert them into Euro for viewing in Euro currency.

PROCEDURE

1. Start.
2. Create the table with its essential attributes.
3. Insert attribute values into the table.
4. Create the view from the above created table.
5. Execute different Commands and extract information from the View.
6. Stop.

SQL COMMANDS

Views:

- A database view is a *logical* or *virtual table* based on a query. It is useful to think of a
  view as a stored query. Views are queried just like tables.
- A DBA or view owner can drop a view with the DROP VIEW command.

Types of views

- Updatable views – Allow data manipulation
- Read only views – Do not allow data manipulation

Creating views:

Syntax:

Create view<view name>;

Description:

This command is used to create view by combining two tables.

Viewing single row of table:

Syntax: Create view<view name> as select from <table name>;

Description:

This command is used to view a single row from a particular table.

Viewing all columns from a single table:

Syntax:

Create view<view name> as select * from <table name>;

Description :

This is used to create view which displays all columns from a single table.

View specified column from a single table:
Syntax:

Create view<view table name> as select column1,column2 from <tablename>;

Description:

This command is used to create view which displays on a specifiedcolumn from a single table.

View specified column from a muliple table:
Syntax:

Create view<view table name> as select column1,column2,….columnn where 'condition';

Description:

This is used to create view to display specified columns from multiple tables.

View all column from a muliple table:
Syntax:

Create view<view table name> as select * from <table name> where 'condition';

Description:

This is used to create view which displays all the columns of a table.

Inserting into views:

Syntax:

Insert into <view name> values <'data1','data2',……>;

Description:

This is used to do inserting of information or data into values.

Updating in view:

Syntax:

Alter table <table name> add constraint;

Description:

This is used for updating of values by specifying the constraints.

Deleting a view:

Syntax:

Drop view <view name>;

Description:

This is used to delete a developed view.

Index:

Types of index:

- Simple index
- Composite index
- Unique index
- Bit map index
- B-tree index
  - Function based index
  - Reverse index

    This is created on columns containing data in sequential order.

    create index rid on emp(empno) reverse;
  - Descending index

    This is created on columns which are displayed in sorted order.

    create index dixd on emp(ename asc, sal desc);

Creating simple Index:

Syntax:

CREATE INDEX index_name ON table_name (column_name);

SQL> create index de on student4 (dept);

Index created.

Description:

Creates an index on a table. Duplicate values are allowed.

Creating composite Index:

Syntax:

CREATE INDEX index_name ON table_name (column_name1,column_name2);

SQL> create index cid on student4 (regno,name);

Index created.

Description: An index created on more than one column of table.

Creating UNIQUE Index :

Syntax:

CREATE UNIQUE INDEX index_name ON table_name (column_name);

SQL> create unique index re on student4 (regno);

Index created.

Description:

Creates a unique index on a table. Duplicate values are not allowed

Creating Bit-Map Index :

Syntax:

CREATE bit map INDEX index_name ON table_name (column_name);

SQL> create bit map index re on student4 (regno);

Index created.

Description:

Bitmap indexes are normally used to index low cardinality columns in a warehouse environment.

Sequences:

Syntax:

CREATE SEQUENCE <sequence_name>
INCREMENT BY <integer>
START WITH <integer>
MAXVALUE <integer> / NOMAXVALUE
MINVALUE <integer> / NOMINVALUE
CYCLE / NOCYCLE
CACHE <#> / NOCACHE
ORDER / NOORDER;

Description:

A sequence is an object in Oracle that is used to generate an auto number field (a number sequence). This is often used to create a unique number to act as a primary key.

Create Sequence:

SQL> create table testbad(user_id number(5),classtpe varchar2(10),roomlocation varchar2(10));

Table created.

```
SQL> create sequence se_user_id;

Sequence created.

SQL> select se_user_id.nextval from dual;

  NEXTVAL
---------
       1

SQL> insert into
testbad(user_id,classtpe,roomlocation)values(se_user_id.nextval,'underwater','rm12
  2  3');

1 row created.

SQL> select * from testbad;

  USER_ID     CLASSTPE     ROOMLOCATI
---------     ----------     ----------
       2      underwater     rm123

SQL> CREATE SEQUENCE user_id
  2      MINVALUE 10
  3      MAXVALUE 100
  4      START WITH 10
  5      INCREMENT BY 5
  6      CACHE 20;

Sequence created.

SQL> select user_id.nextval from dual;

  NEXTVAL
---------
      10

SQL> select user_id.nextval from dual;

  NEXTVAL
---------
      15

SQL> select user_id.currval from dual;

  CURRVAL
---------
      40
```

```
SQL> select rowid from dual;

ROWID
------------------
AAAAECAABAAAAgiAAA

SQL> select rownum from dual;

   ROWNUM
---------
       1

SQL> select sysdate from dual;

SYSDATE
---------
03-JUN-14
```

Alter Sequence:

```
SQL> create sequence user_id
  2      MINVALUE 10
  3      MAXVALUE 200
  4      START WITH 10
  5      INCREMENT BY 5
  6      CACHE 20;

Sequence created.

SQL> select user_id.nextval from dual;

  NEXTVAL
---------
      10

SQL> select user_id.nextval from dual;

  NEXTVAL
---------
      15

SQL> alter sequence user_id
  2  increment by 10;

Sequence altered.

SQL> select user_id.nextval from dual;
```

```
  NEXTVAL
---------
      25
SQL> select user_id.nextval from dual;

  NEXTVAL
---------
      35
```

Drop Sequence:

```
SQL>drop sequence user_id;

Sequence dropped.
```

Synonyms:

A **synonym** is an alternative name for objects such as tables, views, sequences, stored procedures, and other database object type, or another synonym.

Two types of synonyms

1. Private- private synonym is a synonym within a database schema.
2. Public-public synonyms can be referenced by all users in the database.

Create synonyms

```
SQL> CREATE TABLE product (
  2      product_name     VARCHAR2(25) PRIMARY KEY,
  3      product_price    NUMBER(4,2),
  4      quantity_on_hand NUMBER(5,0),
  5      last_stock_date  DATE
  6      );

Table created.

SQL> INSERT INTO product VALUES ('Product 1', 99, 1,    '15-JAN-03');

1 row created.

SQL> INSERT INTO product VALUES ('Product 2', 75, 1000, '15-JAN-02');

1 row created.
```

Create private synonym:

```
SQL> CREATE SYNONYM prod FOR product;

Synonym created.
SQL> SELECT * FROM prod;
```

```
PRODUCT_NAME    PRODUCT_PRICE QUANTITY_ON_HAND       LAST_STOC
------------------------ ------------- ---------------  ---------
Product 1                99              1              15-JAN-03
Product 2                75              1000           15-JAN-02
2 rows selected.

SQL> drop SYNONYM prod;
Synonym dropped.

SQL> drop table product;
Table dropped.
```

Create public synonym:

```
SQL> CREATE or replace PUBLIC SYNONYM source_log FOR source_log;
Synonym created.
SQL> GRANT SELECT, INSERT ON source_log to PUBLIC;
Grant succeeded.
```

Drop synonyms

Synonyms, both private and public, are dropped in the same manner by using the DROP SYNONYM command, but there is one important difference. If you are dropping a public synonym; you need to add the keyword PUBLIC after the keyword DROP.

Syntax

```
DROP SYNONYM addresses;
```

RESULT:

Thus the SQL commands for Views, indexes, sequence and synonyms was executed and verified.

EX: NO:4 DATE:	IMPLICIT AND EXPLICIT CURSORS

AIM

To write a PL/SQL block to Implicit and Explicit Cursors.

PROCEDURE FOR CURSOR IMPLEMENATION

```
SQL> create table student (regno number (4), name varchar2)20), mark1 number (3), mark2
     number (3), mark3 number (3), mark4 number(3), mark5 number(3));
Table created
SQL> insert into student values (101,'priya', 78, 88,77,60,89);
1 row created.
SQL> insert into student values (102,'surya', 99,77,69,81,99);
1 row created.
SQL> insert into student values (103,'suryapriya', 100,90,97,89,91);
1 row created.

SQL> select * from student;
regno name mark1 mark2 mark3 mark4 mark5
---------------------------------------------------------------------
101 priya 78 88 77 60 89
102 surya 99 77 69 81 99
103 suryapriya 100 90 97 89 91

SQL> declare
ave number(5,2);
tot number(3);
cursorc_mark is select*from student where mark1>=40 and mark2>=40 and
mark3>=40 and mark4>=40 and mark5>=40;
begin
dbms_output.put_line('regno name mark1 mark2 mark3 mark4 mark4 mark5 total
average');
dbms_output.put_line('-----------------------------------------------------------');
for student in c_mark
loop
tot:=student.mark1+student.mark2+student.mark3+student.mark4+student.mark5;
ave:=tot/5;
dbms_output.put_line(student.regno||rpad(student.name,15)
||rpad(student.mark1,6)||rpad(student.mark2,6)||rpad(student.mark3,6)
||rpad(student.mark4,6)||rpad(student.mark5,6)||rpad(tot,8)||rpad(ave,5));
end loop;
end;
/
```

SAMPLEOUTPUT:

regno name mark1 mark2 mark3 mark4 mark5 total average

---

101 priya 78 88 77 60 89 393 79
102 surya 99 77 69 81 99 425 85
103 suryapriya 100 90 97 89 91 467 93
PL/SQL procedure successfully completed.

EXPLICIT CURSORS AND EXPLICIT CURSORS IMPLEMENTATION
CREATING A TABLE EMP IN ORACLE

SQL> select * from EMP;

EMPNO	ENAME	JOB	MGR	HIRE DATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SAL	76	98	20-FEB-81	1600	30030
7521	WARD	SALESMAN	76	98	22-FEB-81	1250	50030

EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO

7566 JONES MANAGER 7839 02-APR-81 297520
7654 MARTIN SALESMAN 7698 28-SEP-81 1250 140030
7698 BLAKE MANAGER 7839 01-MAY-81 285030

EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO

7782 CLARK MANAGER 7839 09-JUN-81 245010
7788 SCOTT ANALYST 7566 09-DEC-82 300020
7839 KING PRESIDENT 17-NOV-81 500010

EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO

7844 TURNER SALESMAN 7698 08-SEP-81 1500 030
7876 ADAMS CLERK 7788 12-JAN-83 110020
7900 JAMES CLERK 7698 03-DEC-81 95030

EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO

7902 FORD ANALYST 7566 03-DEC-81 300020
7934 MILLER CLERK 7782 23-JAN-82 130010
14 rows selected.

IMPLICIT CURSORS:

```
SQL> DECLARE
2 ena EMP.ENAME%TYPE;
3 esa EMP.SAL%TYPE;
4 BEGIN
5 SELECT ENAME,SAL INTO ENA,ESA FROM EMP
6 WHERE EMPNO = &EMPNO;
7 DBMS_OUTPUT.PUT_LINE (‘NAME:' || ENA);
8 DBMS_OUTPUT.PUT_LINE (‘SALARY:' || ESA);
9 EXCEPTION
11WHEN NO_DATA_FOUND THEN
11 DBMS_OUTPUT.PUT_LINE ('Employee no does not exits');
12 END;
13 /
```

OUTPUT:

```
Enter value for empno: 7844
old 6: WHERE EMPNO = &EMPNO;
new 6: WHERE EMPNO = 7844;
PL/SQL procedure successfully completed.
```

EXPLICIT CURSORS:

```
SQL> DECLARE
2 ena EMP.ENAME%TYPE;
3 esa EMP.SAL%TYPE;
4 CURSOR c1 IS SELECT ename,sal FROM EMP;
5 BEGIN
6 OPEN c1;
7 FETCH c1 INTO ena,esa;
8 DBMS_OUTPUT.PUT_LINE(ena || ' salry is $ ' || esa);
9
10 FETCH c1 INTO ena,esa;
11 DBMS_OUTPUT.PUT_LINE(ena || ' salry is $ ' || esa);
12
13 FETCH c1 INTO ena,esa;
14 DBMS_OUTPUT.PUT_LINE(ena || ' salry is $ ' || esa);
15 CLOSE c1;
16 END;
17 /
```

OUTPUT:

```
SMITH salry is $ 800
ALLEN salry is $ 1600
WARD salry is $ 1250
```

RESULT:

Thus the Implicit and Explicit Cursors has been executed successfully.

EX: NO:5 DATE:	PROCEDURES AND FUNCTIONS

AIM

To write a PL/SQL block to display the student name marks whose average mark is above 60% and write a Functional procedure to search an address from the given database.

PROCEDURE

1.Start.
2.Create a table with table name stud_exam.
3.Insert the values into the table and calculate total and average of each student.
4.Execute the procedure function the student who gets above 60%.
5.Display the total and average of student.
6.Stop.

EXECUTION:

SETTING SERVEROUTPUT ON:

```
SQL> SET SERVEROUTPUT ON
```

1. PROCEDURE USING POSITIONAL PARAMETERS

```
SQL> SET SERVEROUTPUT ON
SQL> CREATE OR REPLACE PROCEDURE PROC1 AS
2 BEGIN
3 DBMS_OUTPUT.PUT_LINE ('Hello from procedure...');
4 END;
5 /
```

OUTPUT:

```
Procedure created.
SQL> EXECUTE PROC1
Hello from procedure...
PL/SQL procedure successfully completed.
```

2. PROCEDURE USING NOTATIONAL PARAMETERS

```
SQL> CREATE OR REPLACE PROCEDURE PROC2
2 (N1 IN NUMBER,N2 IN NUMBER,TOT OUT NUMBER) IS
3 BEGIN
4 TOT: = N1 + N2;
5 END;
6 /
```

OUTPUT:

```
Procedure created.
SQL> VARIABLE T NUMBER
```

```
SQL> EXEC PROC2 (33, 66: T)
PL/SQL procedure successfully completed.
SQL> PRINT T
T
----------
99
```

3. PROCEDURE FOR GCD NUMBERS

```
SQL> create or replace procedure pro
is
a number(3);
b number(3);
c number(3);
d number(3);
begin
a:=&a;
b:=&b;
if(a>b) then
c:=mod(a,b);
if(c=0) then
dbms_output.put_line('GCD is');
dbms_output.put_line(b);
else
dbms_output.put_line('GCD is');
dbms_output.put_line(c);
end if;
else
d:=mod(b,a);
if(d=0) then
dbms_output.put_line('GCD is');
dbms_output.put_line(a);
else
dbms_output.put_line('GCD is');
dbms_output.put_line(d);
end if;
end if;
end;
/
```

INPUT:

```
Enter value for a: 8
old 8: a:=&a;
new 8: a:=8;
Enter value for b: 16
old 9: b:=&b;
new 9: b:=16;
```

OUTPUT:

```
Procedure created.
```

SQL> set serveroutput on;
SQL> execute pro;
GCD is
8
PL/SQL procedure successfully completed.

PROCEDURE

1. Start.
2. Create the table with essential attributes.
3. Initialize the Function to carry out the searching procedure.
4. Frame the searching procedure for both positive and negative searching.
5. Execute the Function for both positive and negative result.
6. Stop.

EXECUTION:

SETTING SERVEROUTPUT ON:

SQL> SET SERVEROUTPUT ON

1. IMPLEMENTATION OF FACTORIAL USING FUNCTION

```
SQL>create function fnfact (n number)
return number is
b number;
begin
b:=1;
for i in 1..n
loop
b:=b*i;
end loop;
return b;
end;
/
SQL>Declare
n number:=&n;
y number;
begin
y:=fnfact(n);
dbms_output.put_line(y);
end;
/
```

Function created.

```
Enter value for n: 5
old 2: n number:=&n;
new 2: n number:=5;
120
```

PL/SQL procedure successfully completed.

2. PROGRAM

SQL> create table phonebook (phone_no number (6) primary key,username varchar2(30),doorno

varchar2(10),street varchar2(30),place varchar2(30),pincode char(6));

Table created.

SQL> insert into phonebook values(20312,'vijay','120/5D','bharathi street','NGOcolony','629002');
1 row created.

SQL> insert into phonebook values(29467,'vasanth','39D4','RK bhavan','sarakkal vilai','629002');
1 row created.

SQL> select * from phonebook;

PHONE_NO USERNAME DOORNO STREET PLACE PINCODE

20312 vijay 120/5D bharathi street NGO colony 629002
29467 vasanth 39D4 RK bhavansarakkalvilai 629002

SQL> create or replace function findAddress(phone in number) return varchar2 as
address varchar2(100);
begin
select username||','||doorno ||','||street ||','||place||','||pincode into address from phonebook
wherephone_no=phone;
return address;
exception
whenno_data_found then return 'address not found';
end;
/
Function created.
SQL>declare
2 address varchar2(100);
3 begin
4 address:=findaddress(20312);
5 dbms_output.put_line(address);
6 end;
7 /

OUTPUT 1:
Vijay,120/5D,bharathi street,NGO colony,629002
PL/SQL procedure successfully completed.

SQL> declare
2 address varchar2(100);
3 begin
4 address:=findaddress(23556);
5 dbms_output.put_line(address);
6 end;
7 /

OUTPUT2:

Address not found

PL/SQL procedure successfully completed.

RESULT:

Thus the PL/SQL block to display the student name, marks, average is verified and Function are executed.

EX:NO:6 DATE:	CREATION OF TRIGGER AND FUNCTION

AIM

To write a PL/SQL query to create a trigger and function.

TRIGGERS:

A trigger is a statement that the system automatically as a side effect of a modification

SYNTAX FOR CREATING A TRIGGER:

```
Create or replace trigger<trigger_name>
{Before/After}
{Delete Insert Update}
      On<Table name>
[Referencing {OLD[AS] OLD new [AS] new}]
[For each row]
When (condition)
Declare
     <declaration section>
Begin
      <executable statement>
End;
```

Component of Trigger

Triggering SQL statement : SQL DML (INSERT, UPDATE and DELETE) statement that execute and implicitly called trigger to execute.
Trigger Action : When the triggering SQL statement is execute, trigger automatically call and PL/SQL trigger block execute.
Trigger Restriction : We can specify the condition inside trigger to when trigger is fire.

Type of Triggers

1. **BEFORE Trigger** : BEFORE trigger execute before the triggering DML statement (INSERT, UPDATE, DELETE) execute. Triggering SQL statement is may or may not execute, depending on the BEFORE trigger conditions block.
2. **AFTER Trigger** : AFTER trigger execute after the triggering DML statement (INSERT, UPDATE, DELETE) executed. Triggering SQL statement is execute as soon as followed by the code of trigger before performing Database operation.
3. **ROW Trigger** : ROW trigger fire for each and every record which are performing INSERT, UPDATE, DELETE from the database table. If row deleting is define as trigger event, when trigger file, deletes the five rows each times from the table.
4. **Statement Trigger** : Statement trigger fire only once for each statement. If row deleting is define as trigger event, when trigger file, deletes the five rows at once from the table.
5. **Combination Trigger** : Combination trigger are combination of two trigger type,
    1. **Before Statement Trigger** : Trigger fire only once for each statement before the triggering DML statement.

2. **Before Row Trigger** : Trigger fire for each and every record before the triggering DML statement.
3. **After Statement Trigger** : Trigger fire only once for each statement after the triggering DML statement executing.
4. **After Row Trigger** : Trigger fire for each and every record after the triggering DML statement executing.

PL/SQL Triggers

Inserting Trigger

```
SQL> CREATE or REPLACE TRIGGER  trg1
  2   BEFORE
  3   INSERT ON customer
  4   FOR EACH ROW
  5  BEGIN
  6   :new.name := upper(:new.name);
  7  END;
  8  /
```

Trigger created.

```
SQL> select * from customer;

NAME          CID        ADDRESS          SALARY
----------    ---------   ----------      ---------
mani          200             erode          30000
abi           201             namakkal       40000
rani          301             salem          40000
```

```
SQL> insert into customer values('&name',&cid,'&address',&salary);
Enter value for name: logu
Enter value for cid: 306
Enter value for address: salem
Enter value for salary: 40000
old   1: insert into customer values('&name',&cid,'&address',&salary)
new   1: insert into customer values('logu',306,'salem',40000)
```

1 row created.

```
SQL> select * from customer;

NAME          CID          ADDRESS      SALARY
----------    ----------   -----------    ------
LOGU          306           salem         40000
mani          200           erode         30000
abi           201           namakkal      40000
rani          301           salem         40000
```

This trigger execute BEFORE to convert name field lowercase to uppercase.

CREATION OF FIRST TABLE

SQL> create table student(regno number(10),name varchar2(10),dept varchar2(10),percentage number(10));

Table created

SQL> select * from student;

REGNO	NAME	DEPT	PERCENTAGE
1001	kani	cse	70
1002	madhi	cse	80
1003	sharan	it	85
1004	sathya	it	90

CREATION OF SECOND TABLE

SQL> create table bckupstu2(regno number(10),name varchar2(10),dept varchar2(10),percentage number(10));

Table created.

TRIGGER -1:

```
SQL> create or replace trigger t1 after delete on student for each row
  2  begin
  3  insert into bckupstu2 values(:old.regno,:old.name,:old.dept,:old.percentage);
  4  end;
  5  /
```

Trigger created.

SQL> select * from bckupstu2;

no rows selected

SQL> delete from student where regno=1001;

1 row deleted.

SQL> select * from bckupstu2;

REGNO	NAME	DEPT	PERCENTAGE
1001	kani	cse	70

SQL> select * from student;

REGNO	NAME	DEPT	PERCENTAGE
1002	madhi	cse	80
1003	sharan	it	85
1004	sathya	it	90

TRIGGER-2

```
SQL> create or replace trigger t2 after update of regno on student for each row
  2  begin
  3  insert into bckupstu2 values(:old.regno,:old.name,:old.dept,:old.percentage);
  4  end;
  5  /
```

Trigger created.

SQL> update student set name='mozhi' where regno=1002;

1 row updated.

SQL> select * from student;

REGNO	NAME	DEPT	PERCENTAGE
1002	mozhi	cse	80
1003	sharan	it	85
1004	sathya	it	90

SQL> select * from bckupstu2;

REGNO	NAME	DEPT	PERCENTAGE
1001	kani	cse	70

SQL> update student set regno=1005 where regno=1002;

1 row updated.

SQL> select * from student;

REGNO	NAME	DEPT	PERCENTAGE
1005	mozhi	cse	80
1003	sharan	it	85
1004	sathya	it	90

SQL> select * from bckupstu2;

REGNO	NAME	DEPT	PERCENTAGE
1001	kani	cse	70
1002	mozhi	cse	80

TRIGGER-3

```
SQL> create or replace trigger t3 before insert or update of name on student for each row
  2  begin
  3  :new.name:=upper(:new.name);
  4  end;
  5  /
```

Trigger created.

SQL> insert into student values(1006,'anu','ece',91);

1 row created.

SQL> select * from student;

REGNO	NAME	DEPT	PERCENTAGE
1005	mozhi	cse	80
1003	sharan	it	85
1004	sathya	it	90
1006	anu	ece	91

SQL> select * from bckupstu2;

REGNO	NAME	DEPT	PERCENTAGE
1001	kani	cse	70
1002	mozhi	cse	80

CREATION OF THIRD TABLE:

SQL> create table voter_master(voterid number(5),name varchar2(10),wardno number(4),dob date,address varchar2(15),primary key(voterid,wardno));

Table created.

SQL> select * from voter_master;

VOTERID	NAME	WARDNO	DOB	ADDRESS
1000	Saran	5	03-JAN-91	Neru street erode
1001	Madhi	12	15-MAY-89	Gandhi nagar erode
2000	Priya	6	05-FEB-84	Yal street erode

CREATION OF FOURTH TABLE:

SQL> create table new_list(voterid number(5),wardno number(4),name char(20),description char(20);

Table created.

SQL> select * from new_list;

VOTERID	WARDNO	NAME	DESCRIPTION
1000	5	Saran	Neru street erode
1001	12	Madhi	Gandhi nagar erode
5000	21	Roja	Ashok street erode

TRIGGER-4

```
SQL> create or replace trigger vot_trig  after delete on voter_master for each row
      declare
  2  v_id number(5);
  3  w_id number(4);
  4  begin
  5  v_id:=:old.voterid;
  6  update new_list set description='shifted' where voterid=v_id;
  7  end;
  8  /
```

Trigger created.

SQL> delete from voter_master where voterid=1000;

1 row deleted.

SQL> select * from voter_master;

VOTERID	NAME	WARDNO	DOB	ADDRESS
1001	Madhi	12	15-MAY-89	Gandhi nagar erode
2000	Priya	6	05-FEB-84	Yal street erode

SQL> select * from new_list;

VOTERID	WARDNO	NAME	DESCRIPTION
1000	5	Saran	shifted

1001	12	Madhi	Gandhi nagar erode
5000	21	Roja	Ashok street erode

SQL> desc phonebook;

Name	Null?	Type
PHONE_NO	NOT NULL	NUMBER (6)
USERNAME		VARCHAR2 (13)
DOORNO		VARCHAR2 (5)
STREET		VARCHAR2 (13)
PLACE		VARCHAR2 (13)
PINCODE		CHAR (6)

SQL> select * from phonebook;

PHONE_NO	USERNAME	DOORN	STREET	PLACE	PINCOD
25301	priya	390	main street	SIT colony	659002
25401	murthy	39D9	MS bhavan	sai nagar	689002
25701	karthi	7	jay nagar	chennai	600002

SQL> create or replace function findAddress(phone in number) return varchar2 as address varchar2(100);

```
 2   begin
 3   select username||','||door no ||','||street ||','||place||','||Pincode  into address from  phonebook
 4   where phone_no=phone;
 5   return address;
 6 exceptions
 7   when no_data_found then return 'address not found';
 8   end;
 9 /
```

Function created.

SQL> declare

```
 2 address varchar2 (100);
 3 begin
 4 address:=find address(25301);
 5 dbms_output.put_line (address);
 6 end;
```

Priya,390,main street,SIT colony,659002

PL/SQL procedure successfully completed.

SQL> declare

```
2 address varchar2(100);
3 begin
4 address:=findaddress(25601);
5 dbms_output.put_line(address);
6 end;
 Address not found

PL/SQL procedure successfully completed.
```

RESULT:

Thus the program for creation of function and trigger is executed successfully

EX:NO:7 DATE:	EXCEPTIONS

AIM

To write a PL/SQL block that handles all types of exceptions.

DESCRIPTION

An error condition during a program execution is called an exception in PL/SQL. 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

General Syntax for Exception Handling:

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 exception3 THEN
    exception3-handling-statements
  ........
  WHEN others THEN
    exception3-handling-statements
END;
```

```
SQL> create table customer (name varchar2(10),cid number(3),address varchar2(10),salary
number(5));

Table created.

SQL> insert into customer values('&name',&cid,'&address',&salary);
Enter value for name: mani
Enter value for cid: 200
Enter value for address: erode
Enter value for salary: 30000
old   1: insert into customer values('&name',&cid,'&address',&salary)
```

```
new   1: insert into customer values('mani',200,'erode',30000)

1 row created.

SQL> insert into customer values('&name',&cid,'&address',&salary);
Enter value for name: abi
Enter value for cid: 201
Enter value for address: namakkal
Enter value for salary: 40000
old   1: insert into customer values('&name',&cid,'&address',&salary)
new   1: insert into customer values('abi',201,'namakkal',40000)

1 row created.

SQL> insert into customer values('&name',&cid,'&address',&salary);
Enter value for name: rani
Enter value for cid: 301
Enter value for address: salem
Enter value for salary: 40000
old   1: insert into customer values('&name',&cid,'&address',&salary)
new   1: insert into customer values('rani',301,'salem',40000)

1 row created.

SQL> desc customer;
 Name                      Null?    Type
 ------------------------- -------- ----------------
 NAME                               VARCHAR2(10)
 CID                                NUMBER(3)
 ADDRESS                            VARCHAR2(10)
 SALARY                             NUMBER(5)

SQL> select * from customer;

NAME       CID        ADDRESS    SALARY
---------- ---------- ---------- ---------
mani       200        erode      30000
abi        201        namakkal   40000
rani       301        salem      40000
```

USER DEFINED EXCEPTION:

```
SQL> set serveroutput on
SQL> DECLARE
  2     c_id customer.cid%type := 8;
  3     c_name  customer.name%type;
  4     c_addr customer.address%type;
  5  BEGIN
```

```
 6    SELECT  name, address INTO  c_name, c_addr
 7    FROM customer
 8    WHERE cid = c_id;
 9    DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
10     DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
11  EXCEPTION
12     WHEN no_data_found THEN
13        dbms_output.put_line('No such customer!');
14     WHEN others THEN
15        dbms_output.put_line('Error!');
16  END;
17  /
No such customer!

PL/SQL procedure successfully completed.
```

NOTE: The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO_DATA_FOUND**, which is captured in **EXCEPTION** block.

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 of raising an exception:

```
DECLARE
  exception_name EXCEPTION;
BEGIN
  IF condition THEN
    RAISE exception_name;
  END IF;
EXCEPTION
  WHEN exception_name THEN
  statement;
END;
```

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;

SQL> DECLARE
```

```
 2    c_id customer.cid%type := &cc_id;
 3    c_name  customer.name%type;
 4    c_addr customer.address%type;
 5    -- user defined exception
 6    ex_invalid_id  EXCEPTION;
 7  BEGIN
 8    IF c_id <= 0 THEN
 9       RAISE ex_invalid_id;
10    ELSE
11       SELECT  name, address INTO  c_name, c_addr
12       FROM customer
13       WHERE cid = c_id;
14       DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
15       DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
16    END IF;
17  EXCEPTION
18    WHEN ex_invalid_id THEN
19       dbms_output.put_line('ID must be greater than zero!');
20    WHEN no_data_found THEN
21       dbms_output.put_line('No such customer!');
22    WHEN others THEN
23       dbms_output.put_line('Error!');
24  END;
25  /
```

OUTPUT:

```
Enter value for cc_id: -3
old   2:    c_id customer.cid%type := &cc_id;
new   2:    c_id customer.cid%type := -3;
ID must be greater than zero!
PL/SQL procedure successfully completed.
```

RESULT:

Thus the PL/SQL blocks to handle all types of exceptions have been executed and verified.

Ex.No.8 DATE:	ER MODEL

AIM

To design tables by applying normalization principles and following fields,

- empid
- empname
- empstreet
- empdesig
- empsalary
- cmpname
- cmpname
- cmpcity

  where emp-lives manager of the company.

PROCEDURE

- we can identify manager based on company, city and name
- we can identify employee personal details using empid
- we need the employee's company name where salary, design and manager of the company

IDENTIFYING MANAGER BASED ON COMPANY CITY AND COMPANY NAME:

FIRST NORMAL FORM:

The domain up all attributes in the company relation or automatic so it is satisfy of first normal form.

SECOND NORMAL FORM:

The every non prime attributes of the company if fully functional depend on the primary key. Here there is no direct primary key there but cmpname and cmpcity forms a super key. That act as a primary key so the Second normal form condition is satisfied.

THIRD NORMAL FORM:

The relation between cmpname and cmpcity and manager of company is fully functional depend on any company name and city are two candidate key that form a super key. Thus it satisfies the request of third normal form.

BOYCEE CODED NORMAL FORM:

Here the cmpname and city from a super key and the relation b/w other is also function depend thus it satisfies the condition of BCNF.

IDENTIFYING EMPLOYEE PERSONAL DETAILS USING EMPID:

FIRST NORMAL FORM:

The domain up all attributes in the personal relation or automatic so it is satisfy of first normal form.

SECOND NORMAL FORM:

The every non prime attributes of the personal relation is in functional dependency with the primary key of empid. so the Second normal form condition is satisfied.

THIRD NORMAL FORM:

Here there is no primary key but empid is a super key of the personal relation. Thus it satisfies the request of third normal form.

BOYCEE CODED NORMAL FORM:

As said before empid the super key of the personal relation are all the attributes of personal relation are fully function depend each other. thus it satisfies the condition of BCNF.

USING EMPLOYEE COMPANEY NAME AND CITY FINDING ITS SALARY, DESIGNATION AND ITS MANAGER:

FIRST NORMAL FORM:

The domain up all attributes in the employee relation or automatic. thus it is satisfy of first normal form.

SECOND NORMAL FORM:

The every non prime attributes of the employee relation is in functional dependency with the primary key of empid. so the Second normal form condition is satisfied.

THIRD NORMAL FORM:

Empid, cmpname and city are functional dependent to each other and cmpname and city are primary keys. Thus it satisfies the request of third normal form.

BOYCEE CODED NORMAL FORM:

Cmpname and city forms a super key and the empid and the employee relation is also a super key. thus it satisfies the condition of BCNF.

OUTPUT:

COMPANY RELATION:

Company name	Company city	Manager of company

PERSONAL RELATION:

Employee id	Employee name	Employee street	Employee designation	Employee salary	City whereEmployee works

EMPLOYEE RELATION:

Employee id	Employee name	Employee street	Employee designation	Employee salary	City whereEmployee works

RESULT:

Thus the design of tables by applying normalization principle is executed and verified successfully

Ex.No.9 DATE:	PAY ROLL PROCESSING (MINI PROJECT)

AIM:

To create a database for payroll processing system using SQL and implement it using VB.

PROCEDURE:

1. Create a database for payroll processing which request the using SQL .
2. Establish ODBC connection
3. In the administrator tools open data source ODBC
4. Click add button and select oracle in ORA and click finish
5. A window will appear given the data source home as oracle and select source name and user id.
6. ADODC CONTROL FOR SALARY FORM
7. The above procedure must be follow except the table ,A select the table as salary
8. Write appropriate Program in form each from created in VB from each from created in VB form project.

PROGRAM:

SQL>create table emp(eno number primary key,enamr varchar(20),age number,addr varchar(20),DOB date,phno number(10));

Table created.

SQL>create table salary(eno number,edesig varchar(10),basic number,da number,hra number,pf number,mc number,met number,foreign key(eno) references emp);

Table created.

TRIGGER to calculate DA,HRA,PF,MC

```
SQL> create or replace trigger employ
2 after insert on salary
3 declare
4 cursor cur is select eno,basic from salary;
5 begin
6 for cur1 in cur loop
7 update salary set
8 hra=basic*0.1,da=basic*0.07,pf=basic*0.05,mc=basic*0.03 where hra=0;
 9 end loop;
10 end;
11 /
```

Trigger created.

PROGRAM FOR FORM 1

```
Private Sub emp_Click()
Form2.Show
End Sub
Private Sub exit_Click()
Unload Me
 End Sub
Private Sub salary_Click()
Form3.Show
End Sub
```

PROGRAM FOR FORM 2

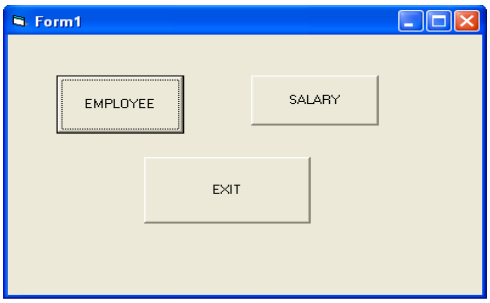
```
Private Sub add_Click()
Adodc1.Recordset.AddNew MsgBox "Record added"
End Sub
Private Sub clear_Click()
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
Text4.Text = ""
Text5.Text = ""
Text6.Text = ""
End Sub
 Private Sub delte_Click()
Adodc1.Recordset.Delete MsgBox "Record Deleted"
If Adodc1.Recordset.EOF = True Then
Adodc1.Recordset.MovePrevious
End If
 End Sub
Private Sub exit_Click()
Unload Me
 End Sub
Private Sub main_Click()
Form1.Show
End Sub
Private Sub modify_Click()
Adodc1.Recordset.Update
End Sub
```

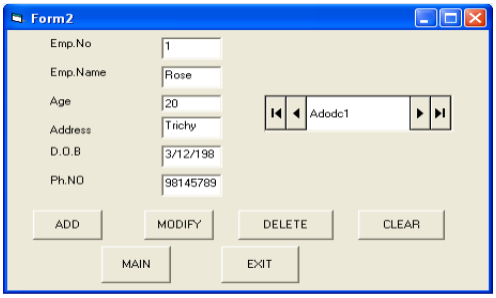
PROGRAM FOR FORM 3

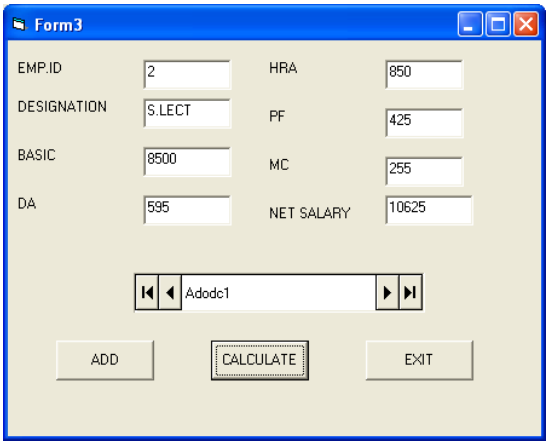
```
Private Sub add_Click()
Adodc1.Recordset.AddNew MsgBox "Record added"
End Sub
 Private Sub clear_Click()
Text1.Text = "" Text2.Text = "" Text3.Text = "" Text4.Text = "" Text5.Text = "" Text6.Text =
""
End Sub
```

```
 Private Sub delte_Click()
Adodc1.Recordset.Delete MsgBox "Record Deleted"
If Adodc1.Recordset.EOF = True Then
Adodc1.Recordset.MovePrevious
End If
End Sub
Private Sub exit_Click()
Unload Me
End Sub
Private Sub main_Click()
Form1.Show
 End Sub
Private Sub modify_Click()
Adodc1.Recordset.Update
End Sub
```

Output:







RESULT:

Thus the design and implementation of payroll processing system using SQL, VB was successfully done.

Ex.No. 10 DATE:	BANKING SYSTEM (MINI PROJECT)

AIM:

To implement the banking project by using Visual Basic as front end & Oracle as back end.

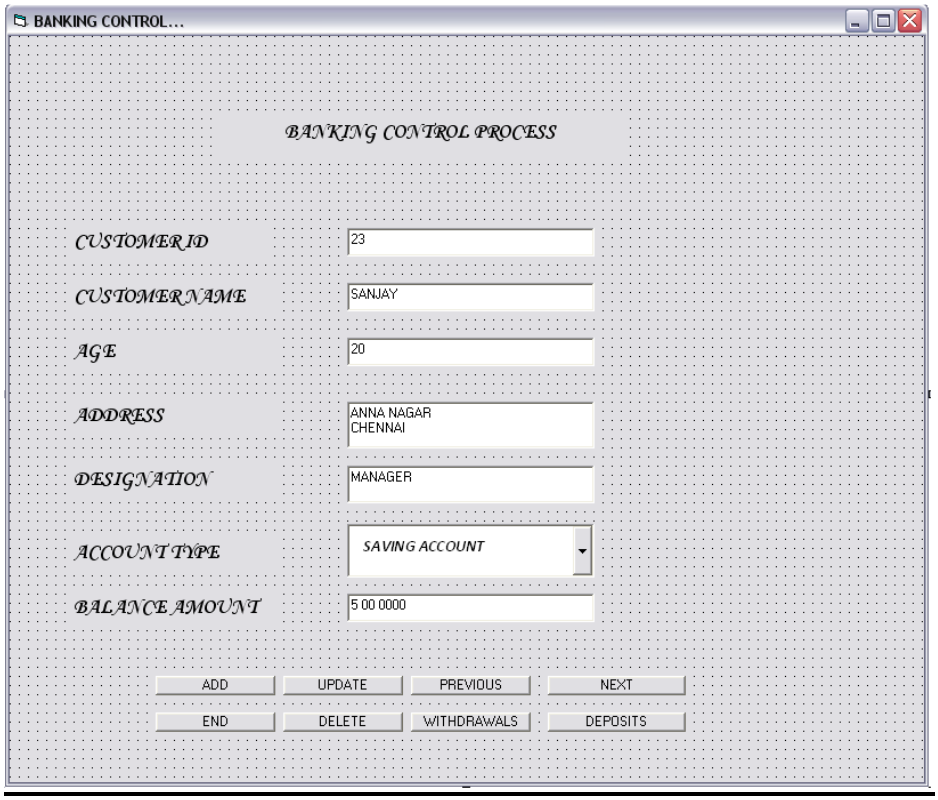
PROGRAM CODE:

```
Dim cn As New ADODB.Connection
Dim rc As New ADOB.Recordset
Private Sub cmdadd_Click()
clear
rc.AddNew
cmdadd.Enabled = False
End Sub
Private Sub cmdupdate_Click()
rc(0) = Txtid.Text
rc(1) = txtname.Text
rc(2) = txtage.Text
rc(3) = txtadd.Text
rc(4) = txtdes.Text
rc(5) = cmbacc.Text
rc(6) = txtamount.Text
rc.Update
ref
If cmdadd.Enabled Then
MsgBox "Record Modified", vbInformation, "Banking Control..."
Else
cmdadd.Enabled = True
MsgBox "Record Saved", vbInformation, "Banking Control..."
End If
End Sub
Private Sub cmdpre_click()
If Not cmdadd.Enabled Then
rc.CancelUpdate
cmdadd.Enabled = True
End If
rc.MovePrevious
If rc.BOF Then rc.MoveFirst
disp
End Sub
Private Sub cmdnext_Click()
If Not cmdadd.Enabled Then
rc.CancelUpdate
cmdadd.Enabled = True
End If
If rc.EOF Then rc.MoveLast
```

```
disp
End Sub

Private Sub cmdfind_Click()
If Not cmdadd.Enabled Then
rc.CancelUpdate
cmdadd.Enabled = True
End If
rc.Requery
On Error GoTo AD
FO = InputBox("Enter the Customer ID to Find", "Banking Controls...")
If FO = "" Then
Else
rc.Find "id=""&FO&"""
disp
Exit Sub
AD:
MsgBox "NO RECORD FOUND", vbCritical, "Banking Control..."
End If
End Sub
Private Sub cmddelete_Click()
If Not cmdadd.Enabled Then
rc.CancelUpdate
cmdadd.Enabled = True
End If
If Not rc.EOF Then
b = MsgBox("Do you want to delete record", vbInformation + vbYesNo, "Banking Control...")
If b = vbYes Then
rc.Delete
MsgBox "Record Deleted", vbInformation, "Banking Controls..."
End If
Else
MsgBox "No Records to delete", vbCritical, "Banking Controls..."
End If
ref
End Sub
Private Sub cmdwd_Click()
Dim p As New ADODB.Recordset
wd = InputBox("Enter the Amount to withdraw...", "Banking Control...")
p.open "select balance from pritto 1", cn, adOpenKeyset, adLockOptimistic
If wd <> "" Then
txtamount.Text = Val(txtamount.Text) - Val(wd)
p(0) = Val(txtamount.Text)
p.Update
End If

p.Close
ref
disp
End Sub
```

```
Private Sub cmddep_Click()
Dim p As New ADODB.Recordset
d = InputBox("Enter the Amount to Deposit...", "Banking Control...")
p.open "Select balance from pritto 1", cn, adOpenKeyset, adLockOptimistic
If d <> "" Then
txtamount.Text = Val(txtamount.Text) + Val(d)
p(0) = Val(txtamount.Text)
p.Update
End If
p.Close
ref
disp
End Sub
Private Sub form_Load()
cn.open "dsn=emp", "cse", "cse"
rc.open "select * from pritto", cn, adOpenKeyset, adLockOptimistic
If Not rc.EOF Then
rc.MoveFirst
disp
End If
cmbacc.AddItem "Saving account"
cmbacc.AddItem "Current account"
End Sub
Private Sub disp()
Txtid.Text = rc(0)
txtname.Text = rc(1)
txtage.Text = rc(2)
textadd.Text = rc(3)
txtdes.Text = rc(4)
cmbacc.Text = rc(5)
txtamount.Text = rc(6)
End Sub
Private Sub clear()
Txtid.Text = ""
txtname.Text = ""
txtage.Text = ""

txtadd.Text = ""
txtdes.Text = ""
cmbacc.Text = ""
txtamount.Text = ""
End Sub
Private Sub txtid_GotFocus()
'txtname.setFocus
End Sub
Private Sub ref()
rc.Close
rc.open "select * from pritto 1", cn, adOpenKeyset, adLockOptimistic
End Sub
Private Sub txtamount_GotFocus()
```

```
If Not cmdadd.Enabled Then
txtamount.SetFocus
Else
cmdwd.SetFocus
End If
End Sub
```



RESULT:

Thus the banking project has been executed and verified successfully.