EX.No.:1(a)

MANIPULATE A DATABASE BY CREATING, INSERTING, DELETING, UPDATING AND RETRIEVING TABLES

AIM:

To execute SQL commands for creating tables, retrieving the values, inserting, updating and deleting values from the table.

PROCEDURE:

1. Creating a Database

Create is a DDL SQL command used to create a table or a database in relational database management system.
To create a database in RDBMS, **create** command is used.

Syntax:

```
CREATE DATABASE <DB_NAME>;
```

Example:

```
CREATE DATABASE Test;
```

The above command will create a database named Test, which will be an empty schema without any table.

2. Creating a Table

Create command can also be used to create tables. Now when we create a table, we have to specify the details of the columns of the tables too. We can specify the **names** and **data types** of various columns in the create command itself.

Syntax:

```
CREATE TABLE <TABLE_NAME> (column_name1 datatype1, column_name2 datatype2, column_name3 datatype3, column_name4 datatype4);
```

Example:

```
        CREATE TABLE Employee
```

```
(
    EmployeeNo char(4),
    EmployeeName varchar2(30),
    EmployeeSal number(10,2),
    EmployeeCity varchar2(30),
    EmployeeDob date
);
```

The above command will create a table named emp.

3.INSERT SQL command

Data Manipulation Language (DML) statements are used for managing data in database. DML commands are not auto-committed. It means changes made by DML command are not permanent to database, it can be rolled back.

Syntax:

```
INSERT INTO table_name VALUES(data1, data2, ...)
```

Example:

```
INSERT INTO Employee(EmployeeNo, EmployeeName, EmployeeSal, EmployeeCity,
EmployeeDob) Values(('1', 'Arvind', 5000, 'Mumbai','23-DEC-1992');
```

Other Options to insert records, using this technique all the table's columns are required.

```
INSERT INTO Employee values('2', 'Santosh', 5000, 'Delhi','23-DEC-1994');
```

4.Select Command

The SQL SELECT statement is used to fetch the data from a database table which returns this data in the form of a result table. These result tables are called result-sets.

Syntax :

The basic syntax of the SELECT statement is as follows −

SELECT column1, column2, columnN FROM table_name;

Here, column1, column2... are the fields of a table whose values you want to fetch. If you want to fetch all the fields available in the field, then you can use the following syntax.

SELECT * FROM table_name;

Example:

select * from Employee

select EmployeeNo, EmployeeName, EmployeeSal,EmployeeCity,EmployeeDob from Employee

5.UPDATE Command

UPDATE command is used to update any record of data in a table.

Syntax:

UPDATE table_name SET column_name = new_value WHERE some_condition;

WHERE is used to add a condition to any SQL query.

Example:

UPDATE Employee SET EmployeeName='KASHISH' WHERE EmployeeNo=1

5.DELETE Command

DELETE command is used to delete data from a table.

Syntax:

DELETE FROM table_name;

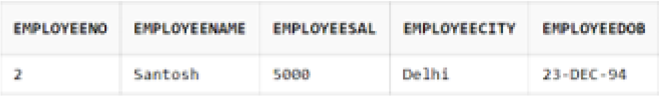
Example :

DELETE FROM EMPLOYEE WHERE employeeNo=1

1(a).MANIPULATE A DATABASE BY CREATING, INSERTING, DELETING, UPDATING AND RETRIEVING TABLES.

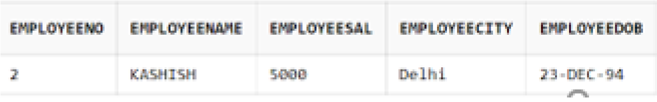
COMMANDS:

```
SQL> CREATE DATABASE Test;
Database Created
SQL> CREATE TABLE Employee(EmployeeNo char(4), EmployeeName varchar2(30),
EmployeeSal number(10,2),   EmployeeCity varchar2(30), EmployeeDob date);
Table Created
SQL> INSERT INTO Employee values('2', 'Santosh', 5000, 'Delhi','23-DEC-1994');
 1 row inserted
 SQL>select * from Employee;
```



EMPLOYEENO	EMPLOYEENAME	EMPLOYEESAL	EMPLOYEECITY	EMPLOYEEDOB
2	Santosh	5000	Delhi	23-DEC-94

```
SQL> UPDATE Employee SET EmployeeName='KASHISH' WHERE EmployeeNo=1;
SQL>SELECT * from Employee;
```



EMPLOYEENO	EMPLOYEENAME	EMPLOYEESAL	EMPLOYEECITY	EMPLOYEEDOB
2	KASHISH	5000	Delhi	23-DEC-94

```
SQL>DELETE * from Employee;
0 row(s) deleted
```

RESULT:

Thus, the SQL commands for creating tables, retrieving the values, inserting, updating and deleting values from the table is executed successfully.

EX.No.:1(b) IMPLEMENTATION OF DDL COMMANDS TO CREATE ,A LTER AND DROP TABLE

AIM:

To execute SQL commands for creating tables, altering and dropping the table from a database.

PROCEDURE:

1. ALTER command

alter command is used for altering the table structure, such as,

1. to add a column to existing table
2. to rename any existing column
3. to change datatype of any column or to modify its size.
4. to drop a column from the table.

ALTER Command: Add a new Column

Using ALTER command, we can add a column to any existing table.

Syntax:

```
ALTER TABLE table_name ADD(column_name  datatype);
```

Example:

ALTER TABLE

ALTER Command: Add multiple new Columns

Using ALTER command we can even add multiple new columns to any existing table.

Syntax:

```
ALTER TABLE table_name ADD(  column_name1 datatype1,column-name2 datatype2, );
```

ALTER Command: Add Column with default value

ALTER command can add a new column to an existing table with a default value too. The default value is used when no value is inserted in the column.

Syntax:

ALTER TABLE table_name ADD(column-name1 datatype1 DEFAULT some_value);

ALTER Command: Modify an existing Column

ALTER command can also be used to modify data type of any existing column.

Syntax:

ALTER TABLE table_name modify(column_name datatype);

ALTER Command: Rename a Column

Using ALTER command you can rename an existing column.

Syntax:

ALTER TABLE table_name RENAME COLUMN old_column_name TO new_column_name;

ALTER Command: Drop a Column

ALTER command can also be used to drop or remove columns.

Syntax:

ALTER TABLE table_name DROP column (column_name);

2..TRUNCATE command

TRUNCATE command removes all the records from a table. But this command will not destroy the table's structure. When we use TRUNCATE command on a table its (auto-increment) primary key is also initialized.

Syntax:

TRUNCATE TABLE table_name;

Example:

TRUNCATE TABLE EMPLOYEE;

3.DROP command

DROP command completely removes a table from the database. This command will also destroy the table structure and the data stored in it.

Syntax:

```
DROP TABLE table_name;
```

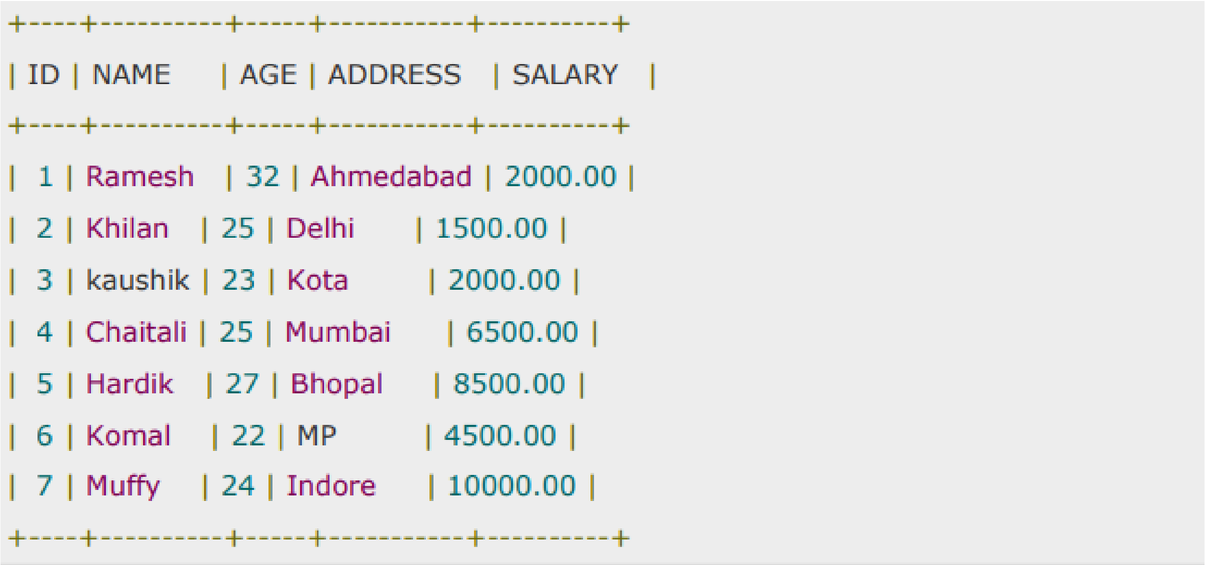
Example:

```
DROP TABLE EMPLOYEE;
```

1(b).IMPLEMENTATION OF DDL COMMANDS TO CREATE ,A LTER AND DROP TABLE.

COMMANDS:

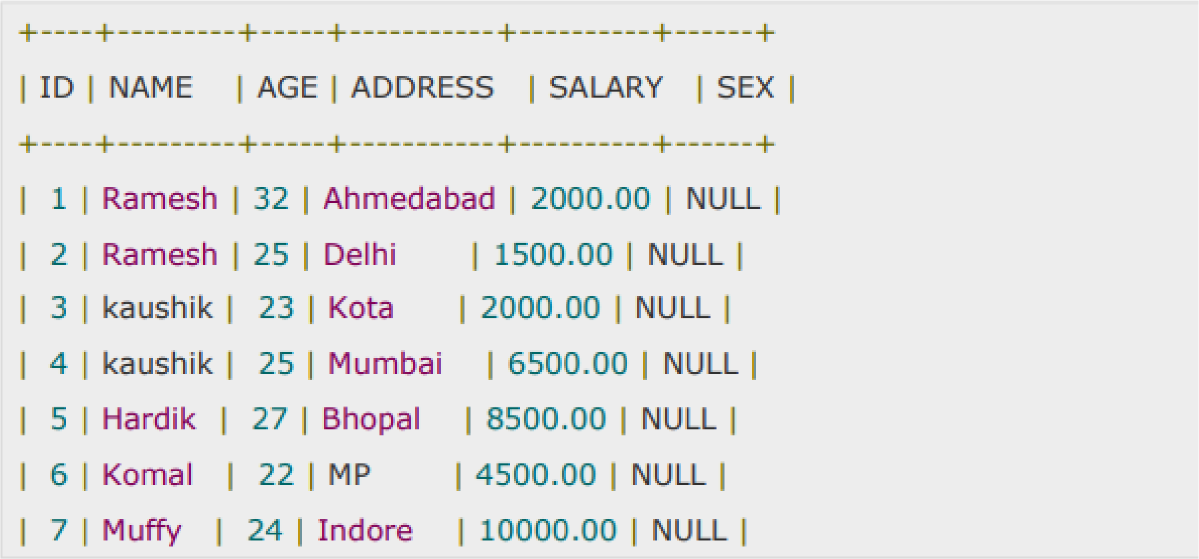
Consider the below table:



```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   | 32 | Ahmedabad | 2000.00 |
|  2 | Khilan   | 25 | Delhi     | 1500.00 |
|  3 | kaushik | 23 | Kota      | 2000.00 |
|  4 | Chaitali | 25 | Mumbai    | 6500.00 |
|  5 | Hardik   | 27 | Bhopal    | 8500.00 |
|  6 | Komal    | 22 | MP        | 4500.00 |
|  7 | Muffy    | 24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

SQL> ALTER TABLE CUSTOMERS ADD SEX char(1);

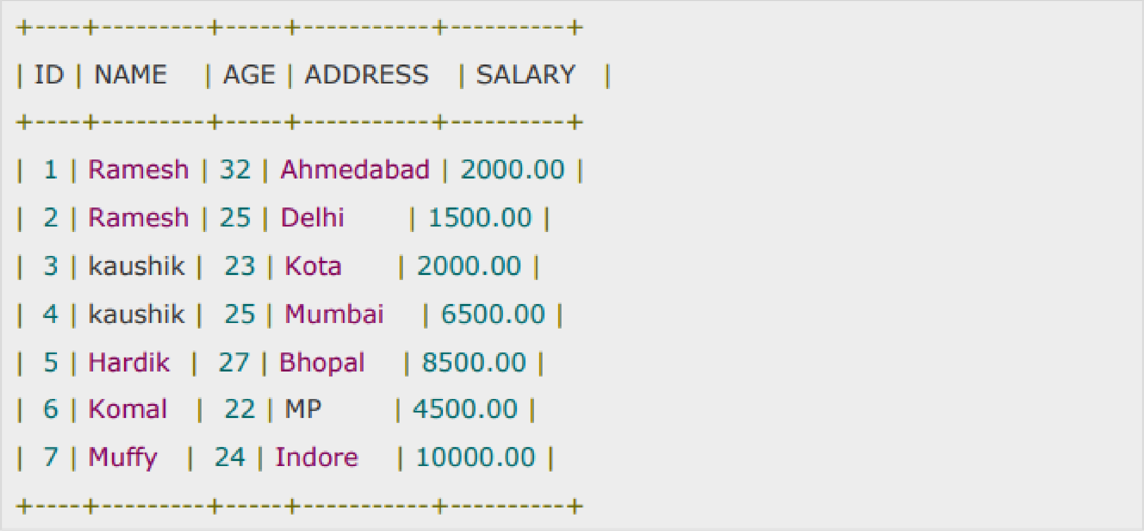
SQL>SELECT * FORM CUSTOMERS;



```
+----+---------+-----+-----------+----------+------+
| ID | NAME    | AGE | ADDRESS   | SALARY   | SEX |
+----+---------+-----+-----------+----------+------+
|  1 | Ramesh | 32 | Ahmedabad | 2000.00 | NULL |
|  2 | Ramesh | 25 | Delhi     | 1500.00 | NULL |
|  3 | kaushik |  23 | Kota     | 2000.00 | NULL |
|  4 | kaushik |  25 | Mumbai   | 6500.00 | NULL |
|  5 | Hardik  |  27 | Bhopal   | 8500.00 | NULL |
|  6 | Komal   |  22 | MP       | 4500.00 | NULL |
|  7 | Muffy   |  24 | Indore   | 10000.00 | NULL |
```

SQL> ALTER TABLE CUSTOMERS DROP SEX;

SQL>SELECT * FORM CUSTOMERS;



```
+----+---------+-----+----------+----------+
| ID | NAME    | AGE | ADDRESS   | SALARY   |
+----+--------+-----+----------+----------+
|  1 | Ramesh | 32 | Ahmedabad | 2000.00 |
|  2 | Ramesh | 25 | Delhi      | 1500.00 |
|  3 | kaushik |  23 | Kota      | 2000.00 |
|  4 | kaushik |  25 | Mumbai    | 6500.00 |
|  5 | Hardik  |  27 | Bhopal    | 8500.00 |
|  6 | Komal   |  22 | MP        | 4500.00 |
|  7 | Muffy   |  24 | Indore    | 10000.00 |
+----+--------+-----+----------+----------+
```

SQL> TRUNCATE TABLE CUSTOMERS;

SQL>SELECT * FORM CUSTOMERS;

Empty set (0.00 sec)

SQL> DROP TABLE CUSTOMERS;

Query OK, 0 rows affected (0.01 sec)

SQL> DESC CUSTOMERS;

ERROR 1146 (42S02): Table 'TEST.CUSTOMERS' doesn't exist

RESULT:

Thus, the SQL commands for creating tables, altering and dropping the table from a database was executed successfully.

EX.No.:2

IMPLEMENTATION OF DML COMMANDS FOR DATA INSERTION USING DIFFERENT WAYS, INTEGRITY CONSTRAINTS AND TRUNCATE

AIM:

To implement commands for data insertion using different ways, integrity constraints and truncate commands

PROCEDURE:

1.DIFERENT WAYS TO INSERT A DATA INTO TABLE:

Method 1: The first way specifies both the column names and the values to be inserted.

Syntax:

INSERT INTO table-name (column-names) VALUES (values) ;

Method 2: Insert Data Only in Specified Columns**.**

Method 3: If you are adding the values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table.

2.INTEGRITY CONSTRAINTS:

- The Set of rules which is used to maintain the quality of information are known as integrity constraints.
- Integrity constraints make sure about data intersection, update and so on.
- Integrity constraints can be understood as a guard against unintentional damage to the database.

Domain Constraint

- The Definition of an applicable set of values is known as domain constraint.
- Strings, character, time, integer, currency, date etc. Are examples of the data type of domain constraints

Entity Integer Constraint

- Entity Integrity Constraints states that the primary value key cannot be null because the primary value key is used to find out individual rows in relation and if the value of the primary key is null then it is not easy to identify those rows.
- There can be a null value in the table apart from the primary key field.

Referential Integrity Constraint

1. Referential Integrity Constraint is specific between two tables.
2. A foreign key in the 1st table refers to the primary key of the 2nd table, in this case each value of the foreign key in the 1st table has to be null or present in the 2nd table.

Key Constraints

- The Entity within its entity set is identified uniquely by the key which is the entity set.
- There can be a number of keys in an entity set but only one will be the primary key out of all keys. In a relational table a primary key can have a unique as well as a null value.

<u>3.TRUNCATE :</u>

TRUNCATE command removes all the records from a table. But this command will not destroy the table's structure. When we use TRUNCATE command on a table its (auto-increment) primary key is also initialized.

Syntax:

TRUNCATE TABLE table_name;

Example:

TRUNCATE TABLE EMPLOYEE;

2. IMPLEMENTATION OF DML COMMANDS FOR DATA INSERTION USING DIFFERENT WAYS, INTEGRITY CONSTRAINTS AND TRUNCATE

<u>COMMANDS:</u>

SQL>**CREATE DATABASE** Organization;

Database Created

```
SQL>CREATE TABLE Persons
(
   PersonID int,
   LastName varchar(255),
   FirstName varchar(255),
   Address varchar(255),
   City varchar(255)
);
```

SQL>INSERT INTO Persons (PersonID, LastName, FirstName, Address, City) VALUES ('101', 'Erichsen', 'Tom', 'Street no-21', 'New York');

SQL>INSERT INTO Persons (PersonID, LastName, FirstName, Address, City)

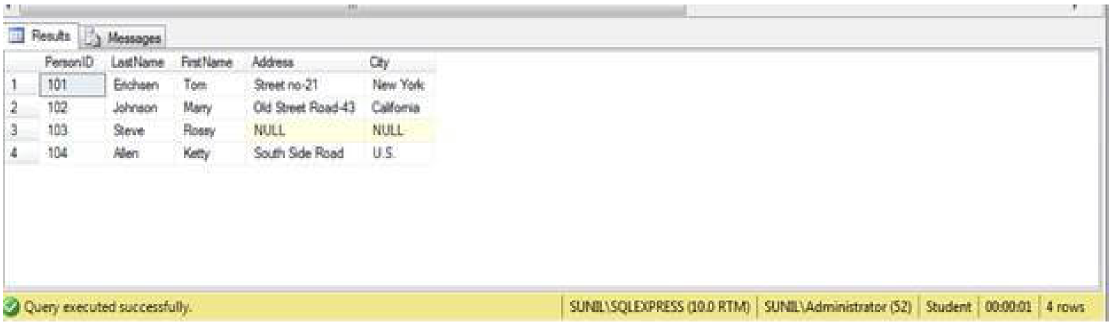
VALUES ('102', 'Johnson', 'Marry', 'Old Street Road-43', 'California');

SQL>INSERT INTO Persons (PersonID, LastName,FirstName)

  VALUES ('103', 'Steve','Rossy')

SQL>INSERT INTO Persons VALUES ('104', 'Allen', 'Ketty', 'South Side Road', 'U.S.');

SQL>**select** * **from** persons



	PersonID	LastName	FirstName	Address	City
1	101	Erichsen	Tom	Street no-21	New York
2	102	Johnson	Marry	Old Street Road-43	California
3	103	Steve	Rossy	NULL	NULL
4	104	Allen	Ketty	South Side Road	U.S.

Query executed successfully. | SUNIL\SQLEXPRESS (10.0 RTM) | SUNIL\Administrator (52) | Student | 00:00:01 | 4 rows

SQL> **TRUNCATE TABLE Persons;**
SQL>SELECT * FORM Persons;

Empty set (0.00 sec)

SQL> CREATE TABLE customer_details(

customer_id character varying(255) NOT NULL,

customer_name character varying(255) NOT NULL,

quantity integer NOT NULL,

date_purchased date

);

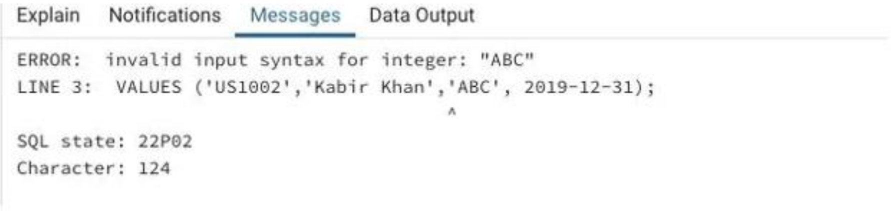
Table Created.

SQL> INSERT INTO public.customer_details(

customer_id, customer_name, quantity, date_purchased)

VALUES ('US1002','Kabir Khan','ABC', 2019-12-31);

OUTPUT:



```
ERROR:  invalid input syntax for integer: "ABC"
LINE 3:  VALUES ('US1002','Kabir Khan','ABC', 2019-12-31);
                                        ^

SQL state: 22P02
Character: 124
```

SQL> CREATE TABLE Students(

Student_ID int NOT NULL,

Student_Name varchar(255) NOT NULL,

Class_Name varchar(255) UNIQUE,

Age int,

PRIMARY KEY (Student_ID)

);

Table Created.

SQL> INSERT INTO public.students(

student_id, student_name, class_name, age)

VALUES (32,'ABC','V',12),(32,'XYZ','V',11);

SQL> CREATE TABLE Department(

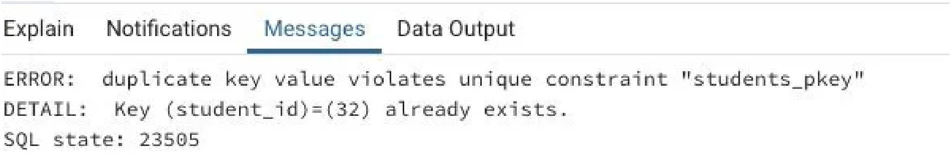
Department_ID int NOT NULL,

Department_Name varchar(255) NOT NULL,

PRIMARY KEY(Department_ID)

);

OUTPUT:



```
ERROR:  duplicate key value violates unique constraint "students_pkey"
DETAIL:  Key (student_id)=(32) already exists.
SQL state: 23505
```

SQL>CREATE TABLE Employees(

Employee_ID int NOT NULL,

Employee_Name varchar(255) NOT NULL,

Department int NOT NULL,

Age int,

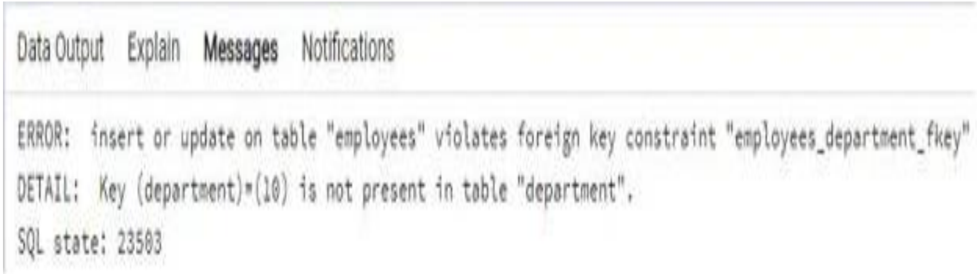
FOREIGN KEY (Department) REFERENCES Department(Department_ID)

);

Table Created.

SQL> INSERT INTO public.employees(
employee_id, employee_name, department, age)
VALUES (1002,'K K Davis',10,43);

OUTPUT:



```
ERROR:  insert or update on table "employees" violates foreign key constraint "employees_department_fkey"
DETAIL:  Key (department)=(10) is not present in table "department".
SQL state: 23503
```

RESULT:

Thus the commands for data insertion using different ways, integrity constraints and truncate has been implemented and executed successfully.

EX.No.:3

MANIPULATE TABLES IN A DATABASE USING SIMPLE QUERIES, NESTED QUERIES, SUB QUERIES AND JOINS

AIM:

To create simple queries,nested queries,sub
queries and joins using tables.

PROCEDURE:

STEP 1: Start the program.

STEP 2: Create two different tables with its essential attributes.

STEP 3: Insert attribute values into the table.

STEP 4: Create the Nested query and join from the above created table.

STEP 5: Execute Command and extract information from the tables.

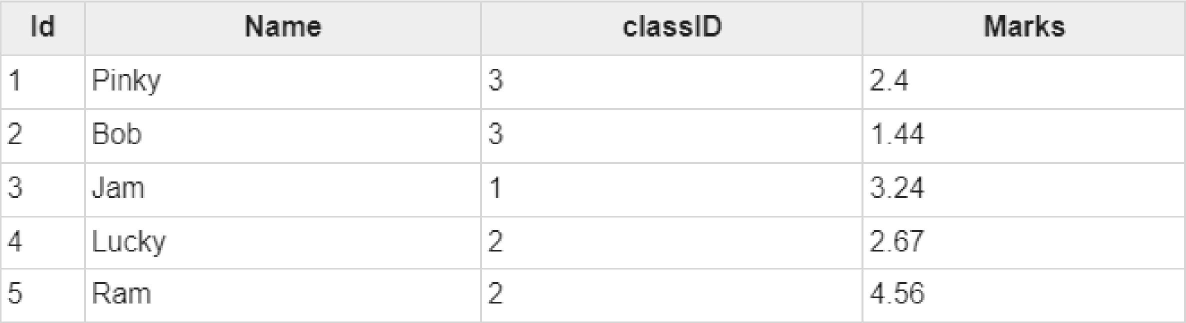
STEP 6: Stop the program.

3. MANIPULATE TABLES IN A DATABASE USING SIMPLE QUERIES, NESTED QUERIES, SUB QUERIES AND JOINS

COMMANDS:

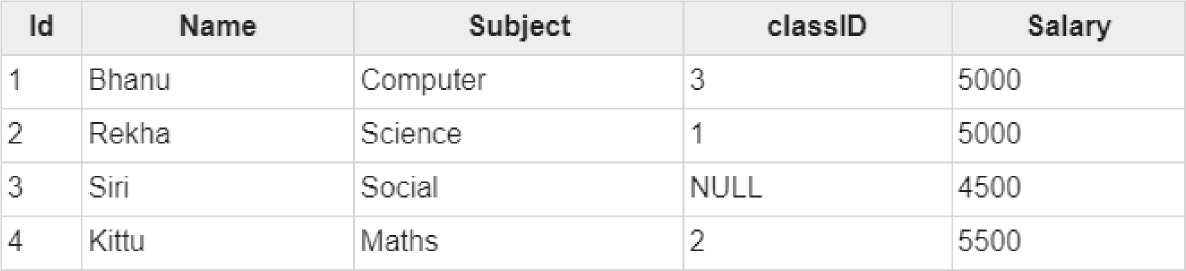
NESTED QUERY

```
SQL>create table student(id number(10), name varchar2(20),classID number(10), marks varchar2(20));
SQL>insert into student values(1,'pinky',3,2.4);
SQL>insert into student values(2,'bob',3,1.44);
SQL>insert into student values(3,'Jam',1,3.24);
SQL>insert into student values(4,'lucky',2,2.67);
SQL>insert into student values(5,'ram',2,4.56);
SQL>select * from student;
```



Id	Name	classID	Marks
1	Pinky	3	2.4
2	Bob	3	1.44
3	Jam	1	3.24
4	Lucky	2	2.67
5	Ram	2	4.56

```
SQL>Create table teacher(id number(10), name varchar(20), subject varchar2(10), classID number(10), salary number(30));
SQL>insert into teacher values(1,’bhanu’,’computer’,3,5000);
SQL>insert into teacher values(2,'rekha','science',1,5000);
SQL>insert into teacher values(3,'siri','social',NULL,4500);
SQL>insert into teacher values(4,'kittu','mathsr',2,5500);
SQL>select * from teacher;
```



Id	Name	Subject	classID	Salary
1	Bhanu	Computer	3	5000
2	Rekha	Science	1	5000
3	Siri	Social	NULL	4500
4	Kittu	Maths	2	5500

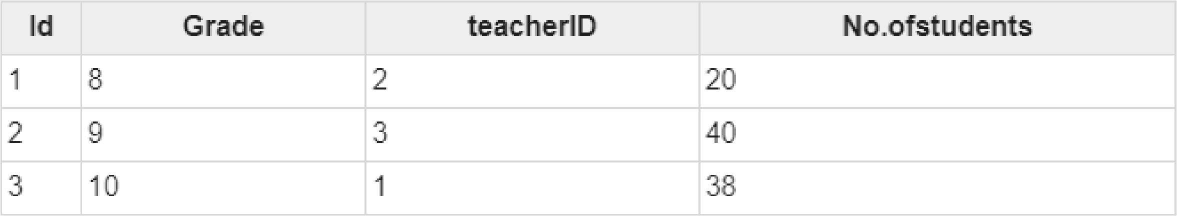
SQL>Create table class(id number(10), grade number(10), teacherID number(10), noofstudents number(10));

SQL>insert into class values(1,8,2,20);

SQL>insert into class values(2,9,3,40);

SQL>insert into class values(3,10,1,38);

SQL>select * from class;



Id	Grade	teacherID	No.ofstudents
1	8	2	20
2	9	3	40
3	10	1	38

SQL> Select AVG(noofstudents) from class where teacherID IN(Select id from teacher Where subject='science' OR subject='maths');

20.0

SQL> SELECT * FROM student WHERE classID = (SELECT id FROM class WHERE 2 noofstudents = (SELECT MAX(noofstudents) FROM class));

4|lucky |2|2.67

5|ram |2|4.56

JOINS:

EQUIJOIN.

Table 1 − CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS INNER JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

LEFT JOIN

SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS LEFT JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

Output:

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   NULL | NULL                |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   NULL | NULL                |
|  6 | Komal    |   NULL | NULL                |
|  7 | Muffy    |   NULL | NULL                |
+----+----------+--------+---------------------+
```

RIGHT JOIN

SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS RIGHT JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

Output:

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

FULL JOIN

SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS FULL JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

Output:

```
+-------+----------+--------+---------------------+
| ID    | NAME     | AMOUNT | DATE                |
+-------+----------+--------+---------------------+
|    1 | Ramesh   |   NULL | NULL                |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|    5 | Hardik   |   NULL | NULL                |
|    6 | Komal    |   NULL | NULL                |
|    7 | Muffy    |   NULL | NULL                |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

SELF JOIN:

SQL> SELECT a.ID, b.NAME, a.SALARY FROM CUSTOMERS a, CUSTOMERS b WHERE a.SALARY < b.SALARY;

Output:

```
+----+----------+---------+
| ID | NAME     | SALARY  |
+----+----------+---------+
|  2 | Ramesh   | 1500.00 |
|  2 | kaushik  | 1500.00 |
|  1 | Chaitali | 2000.00 |
|  2 | Chaitali | 1500.00 |
|  3 | Chaitali | 2000.00 |
|  6 | Chaitali | 4500.00 |
|  1 | Hardik   | 2000.00 |
|  2 | Hardik   | 1500.00 |
|  3 | Hardik   | 2000.00 |
|  4 | Hardik   | 6500.00 |
|  6 | Hardik   | 4500.00 |
|  1 | Komal    | 2000.00 |
|  2 | Komal    | 1500.00 |
|  3 | Komal    | 2000.00 |
|  1 | Muffy    | 2000.00 |
|  2 | Muffy    | 1500.00 |
|  3 | Muffy    | 2000.00 |
|  4 | Muffy    | 6500.00 |
|  5 | Muffy    | 8500.00 |
|  6 | Muffy    | 4500.00 |
+----+----------+---------+
```

CARTESIAN JOIN :

SQL> SELECT ID, NAME, AMOUNT, DATE   FROM CUSTOMERS, ORDERS;

Output:

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   3000 | 2009-10-08 00:00:00 |
|  1 | Ramesh   |   1500 | 2009-10-08 00:00:00 |
|  1 | Ramesh   |   1560 | 2009-11-20 00:00:00 |
|  1 | Ramesh   |   2060 | 2008-05-20 00:00:00 |
|  2 | Khilan   |   3000 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  2 | Khilan   |   2060 | 2008-05-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   2060 | 2008-05-20 00:00:00 |
|  4 | Chaitali |   3000 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   3000 | 2009-10-08 00:00:00 |
|  5 | Hardik   |   1500 | 2009-10-08 00:00:00 |
|  5 | Hardik   |   1560 | 2009-11-20 00:00:00 |
|  5 | Hardik   |   2060 | 2008-05-20 00:00:00 |
|  6 | Komal    |   3000 | 2009-10-08 00:00:00 |
|  6 | Komal    |   1500 | 2009-10-08 00:00:00 |
|  6 | Komal    |   1560 | 2009-11-20 00:00:00 |
|  6 | Komal    |   2060 | 2008-05-20 00:00:00 |
|  7 | Muffy    |   3000 | 2009-10-08 00:00:00 |
|  7 | Muffy    |   1500 | 2009-10-08 00:00:00 |
|  7 | Muffy    |   1560 | 2009-11-20 00:00:00 |
|  7 | Muffy    |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

RESULT:

Thus the simple queries, nested queries ,sub queries and joins has been executed successfully.

EX.No.:4

IMPLENTATION OF AGGREGATION FUNCTIONS, GROUPING AND ORDERING COMMANDS TO MANIPULATE TABLES IN A DATABASE

AIM:

To implement on aggregation functions, grouping and ordering commands to manipulate tables in a database.

PROCEDURE:

An aggregate function allows you to perform a calculation on a set of values to return a single scalar value. We often use aggregate functions with the GROUP BY and HAVING clauses of the SELECT statement.

The following are the most commonly used SQL aggregate functions:

AVG – calculates the average of a set of values.

COUNT – counts rows in a specified table or view.

MIN – gets the minimum value in a set of values.

MAX – gets the maximum value in a set of values.

SUM – calculates the sum of values.

The SQL GROUP BY clause is used in collaboration with the SELECT statement to arrange identical data into groups. This GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax:

```
SELECT column1, column2
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2
ORDER BY column1, column2
```

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

Syntax;

SELECT *column1*, *column2, ...*

FROM *table_name*

ORDER BY *column1, column2, ...* ASC|DESC;

4. IMPLENTATION OF AGGREGATION FUNCTIONS, GROUPING AND ORDERING COMMANDS TO MANIPULATE TABLES IN A DATABASE

COMMANDS:

Let’s consider an employee table. We will perform the calculations on this table by using aggregate functions.

Eid	Ename	Age	City	Salary
E001	ABC	29	Pune	20000
E002	PQR	30	Pune	30000
E003	LMN	25	Mumbai	5000
E004	XYZ	24	Mumbai	4000
E005	STU	32	Bangalore	25000

```
SQL> select AVG(salary) from employee;
16800
SQL> select MAX(salary) from employee;
30000
SQL> select MIN (salary) from employee;
4000
SQL>select SUM (salary) from employee where city=’Pune’;
50000
SQL> select COUNT(Empid) from employee;
5
SQL> select COUNT(*) from employee;
5
```

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
   GROUP BY NAME;
+----------+-------------+
| NAME     | SUM(SALARY) |
+----------+-------------+
| Chaitali |     6500.00 |
| Hardik   |     8500.00 |
| kaushik  |     2000.00 |
| Khilan   |     1500.00 |
| Komal    |     4500.00 |
| Muffy    |    10000.00 |
| Ramesh   |     2000.00 |
+----------+-------------+
SQL> SELECT * FROM CUSTOMERS
   ORDER BY NAME, SALARY;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
+----+----------+-----+-----------+----------+
SQL> SELECT * FROM CUSTOMERS
   ORDER BY NAME DESC;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
+----+----------+-----+-----------+----------+
```

RESULT:

Thus the aggregation functions, grouping and ordering commands to manipulate tables in a database has been implemented and executes successfully.

EX.No.:5 a IMPLEMENT DCL COMMANDS TO SET AND REVOKE PRIVILEGES

AIM:

To implement DCL Commands to set and revoke privileges.

PROCEDURE:

Data control language (DCL) is used to access the stored data. It is mainly used for revoke and to grant the user the required access to a database. In the database, this language does not have the feature of rollback.

It is a part of the structured query language (SQL).

It helps in controlling access to information stored in a database. It complements the data manipulation language (DML) and the data definition language (DDL).

It is the simplest among three commands.

It provides the administrators, to remove and set database permissions to desired users as needed.

These commands are employed to grant, remove and deny permissions to users for retrieving and manipulating a database.

GRANT Command

It is employed to grant a privilege to a user. GRANT command allows specified users to perform specified tasks

Syntax

GRANT privilege_name on objectname to user;

Here,

privilege names are SELECT,UPDATE,DELETE,INSERT,ALTER,ALL

objectname is table name

user is the name of the user to whom we grant privileges

REVOKE Command

It is employed to remove a privilege from a user. REVOKE helps the owner to cancel previously granted permissions.

Syntax

REVOKE privilege_name on objectname from user;

Here,

privilege names are SELECT,UPDATE,DELETE,INSERT,ALTER,ALL

object name is table name user is the name of the user whose privileges are removing

5(a). IMPLEMENT DCL COMMANDS TO SET AND REVOKE PRIVILEGES

COMMANDS:

```
SQL> Grant Create session to student;
 SQL> Grant create table to student;
SQL> Connect student/young;
SQL> Connect system/managers;
SQL> Create user staff identified by guru;
 SQL> Grant resource to staff;
SQL> Connect staff/guru;
SQL> Select * from staff;
```

Staff master in a table in the user staff we first log on the staff [SQL> Connect staff/guru;]

```
SQL> Grant select insert on staff master to student;
```

Now log on to student and try the select command

```
SQL> Connect student/young;
SQL> Select * from staff;
SQL> Grant select, update, delete on student-master to staff;.
SQL> REVOKE SELECT, INSERT ON STUDENT_MASTER from staff;
SQL> REVOKE SELECT ON STUDENT_MASTER from rajan;
```

RESULT:

Thus the DCL Commands to set and revoke privileges has been executed successfully.

EX.No.:5b IMPLEMENTATION OF TCL COMMANDS SAVE-POINT, ROLL BACK AND ROLL BACK TO COMMANDS

AIM:

To execute Transactional Control Commands such as Commit, Rollback and Savepoint.

ALGORITHM:

STEP 1: Start the DMBS.

STEP 2: Connect to the existing database (DB)

STEP 3: Create the table with its essential attributes.

STEP 4: Insert record values into the table or perform any kind of DML operation.

STEP 5: Create the SAVE POINTs for some set of statement on the transaction of database object. STEP 6: Use the COMMIT command to save the effect of the previous command operation except DDL command

STEP 7: Use the ROLLBACK TO SP_LABLE / ROLLBACK command for restore the database status up to the save point

STEP 8: Check the status of the database.

STEP 9: Stop the DBMS.

THEORY:

Transaction Control Language(TCL) commands are used to manage transactions in the database.These are used to manage the changes made to the data in a table by DML statements. It also allowsstatements to be grouped together into logical transactions.

COMMIT command

COMMIT command is used to permanently save any transaction into the database.

To avoid that, we use the COMMIT command to mark the changes as permanent.

Syntax:

COMMIT;

ROLLBACK command

This command restores the database to last commited state. It is also used with

SAVEPOINT

command to jump to a savepoint in an ongoing transaction.

Syntax:

ROLLBACK TO savepoint_name;

SAVEPOINT command

SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point when ever required.

Syntax:

SAVEPOINT savepoint_name;

5(b).IMPLEMENTATION OF TCL COMMANDS SAVE-POINT, ROLL BACK AND ROLL BACK TO COMMANDS

COMMANDS:

Consider the CUSTOMERS table having the following records –

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

```
SQL> DELETE FROM CUSTOMERS  WHERE AGE = 25;

SQL> ROLLBACK;

SQL> select * from customers
```

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

```
SQL> DELETE FROM CUSTOMERS   WHERE AGE = 25;

SQL> COMMIT;

SQL> select * from customers
```

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
SQL> SAVEPOINT SP1;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=1;

1 row deleted.

SQL> SAVEPOINT SP2;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=2;

1 row deleted.

SQL> SAVEPOINT SP3;

Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=3;

1 row deleted.

SQL> ROLLBACK TO SP2;

Rollback complete.

SQL> SELECT * FROM CUSTOMERS;
    +----+----------+-----+-----------+----+
    | ID | NAME     | AGE | ADDRESS   | SALARY   |
    +----+----------+-----+-----------+----------+
    |  2 | Khilan   |  25 | Delhi     |  1500.00 |
    |  3 | kaushik  |  23 | Kota      |  2000.00 |
    |  4 | Chaitali |  25 | Mumbai    |  6500.00 |
    |  5 | Hardik   |  27 | Bhopal    |  8500.00 |
    |  6 | Komal    |  22 | MP        |  4500.00 |
    |  7 | Muffy    |  24 | Indore    | 10000.00 |
    +----+----------+-----+-----------+----------+
    6 rows selected.
```

RESULT:

Thus the Transactional Control Commands such as Commit, Rollback and Savepoint has been executed successfully.

EX.No.:6 IMPLEMENTATION OF PL/SQL USING CONDITIONAL STATEMENTS

AIM:

To implement the conditional selection statement in PL/SQL block.

ALGORITHM:

STEP 1: Start the program.

STEP 2: Create the PL/SQL Block with necessary blocks.

STEP 3: Declare the necessary variable in the declaration section

STEP 4: write the main program logics in the begin block.

STEP 5: if you want to access the table use the SQL statement.

STEP 6: if you want to solve any exception, write the exception name with WHEN statement

STEP 7: Execute the PL/SQL block.

STEP 8: Give the input values or validate the information from the tables.

STEP 9: Stop the program.

The PL/SQL stands for **Procedural Language extensions to Structured Query Language**. Basically, SQL is used to perform basic operations of creating a database, storing data in the database, updating data in the database, retrieving the stored data of database, etc, whereas PL/SQL is a fully Structured **Procedural** language which enables the developer to combine the powers of SQL with its procedural statements.

PL/SQL Block

In a PL/SQL program, code is written in blocks. Each PL/SQL block has 3 sections, which are:

1. Declare section
2. Begin section
3. Exception section

Followed by **END** statement at the end

PL/SQL Block

PL/SQL block creates the structured logical blocks of code that describes the process to be executed. Such a block consists of SQL statements and PL/SQL instructions that are then passed to the oracle engine for execution. PL/SQL block consists of the following four sections:

- **DECLARE Section:**

  PL/SQL code starts with a declaration section in which memory variables and other oracle objects like cursor, triggers etc can be declared and if required can be initialized as well. Once declared/initialised we can use them in SQL statements for data manipulation. As it is not necessary that we would require variables etc in every PL/SQL code, hence this section is an optional section.

- **BEGIN Section:**

  This section contains the SQL and PL/SQL statements that are required to be executed and contains the main logic. This section is responsible for handling the data retrieval and manipulation, may be working with branching, can use looping and conditional statements, etc.

- **EXCEPTION Section:**

  This section is optional. It is mainly used to handle the errors that may occur between **BEGIN** and **EXCEPTION** sections.

- **END Section:**

  This section is the indication of the end of the PL/SQL block.

PL/SQL Conditional Statements

Decision making statements are those statements which are in charge of executing a statement out of multiple given statements based on some condition. The condition will return either true or false. Based on what the condition returns, the associated statement is executed.

For example, if someone says, If I get 40 marks, I will pass the exam, else I will fail. In this case condition is getting 40 marks, if its true then the person will pass else he/she will fail.

This can be logically implemented in PL/SQL block using decision making statements.

The decision making statements in PL/SQL are of two types:

1. If Else statements
2. Case statement

Let's see them all one by one with examples.

PL/SQL: if Statement

The if statement, or the if...then statement can be used when there is only a single condition to be tested. If the result of the condition is **TRUE** then certain specified action will be performed otherwise if it is **FALSE** then no action is taken and the control of program will just move out of the if code block.

Syntax:

```
if <test_condition> then
        body of action
end if;
```

PL/SQL: if...then...else statement

Using this statement group we can specify two statements or two set of statements, dependent on a condition such that when the condition is true then one set of statements is executed and if the condition is false then the other set of statements is executed.

Syntax:

```
if <test_condition> then
        statement 1/set of statements 1
else
```

```
        statement 2/set of statements 2
end if;
```

PL/SQL: if...then...elsif...else statement

It is used to check multiple conditions. Sometimes it is required to test more than one condition in that case if...then...else statement cannot be used. For this purpose, if...then...elsif...else statement is suitable in which all the conditions are tested one by one and whichever condition is found to be **TRUE**, that block of code is executed. And if all the conditions result in **FALSE** then the else part is executed.

In the following syntax, it can be seen firstly **condition1** is checked, if it is true, the statements following it are executed and then control moves out of the complete if block but if the condition is false then the control checks **condition2** and repeats the same process. If all the conditions fail then the else part is executed.

Syntax:

```
if <test_condition1> then
        body of action
elsif <test_condition2>then
        body of action
elsif<test_condition3>then
        body of action
...
...
...
else
        body of action
end if;
```

PL/SQL: Case Statement

If we try to describe the case statement in one line then, then we can say means "**one out of many**". It is a decision making statement that selects only one option out of the multiple available options.

It uses a **selector** for this purpose. This selector can be a variable, function or procedure that returns some value and on the basis of the result one of the case statements is executed. If all the cases fail then the else case is executed.

Syntax:

```
CASE selector
        when value1 then Statement1;
        when value2 then Statement2;
        ...
        ...
        else statement;
end CASE;
```

6.IMPLEMENTATION OF PL/SQL USING CONDITIONAL STATEMENTS

PROGRAM:

Program to find whether a given number by user is even or odd:

```
set serveroutput on;

DECLARE
	x int;
BEGIN
	x := 15;
	if mod(x,2) = 0 then
		dbms_output.put_line('Even Number');
	else
		dbms_output.put_line('Odd Number');
	end if;
END;
/
```

Output:

```
Odd Number
```

Program to find whether the two given numbers are equal and if they are not equal then which one is greater:

```
set serveroutput on;

DECLARE
	a int;
	b int;
BEGIN
	a := 10;
	b := 20;
	if(a>b) then
		dbms_output.put_line(‘a is greater than b’);
	elsif(b>a) then
		dbms_output.put_line(‘b is greater than a’);
	else
		dbms_output.put_line(‘Both a and b are equal’);
	end if;
END;
/
```

Output:

```
b is greater than a
```

Program to demonstrate the use of a simple case statement.

```
set serveroutput on;

DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';

  CASE grade
    WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
/
```

Output:

```
Very Good
```

RESULT:

Thus the conditional selection statement in PL/SQL block has beenverified and executed successfully.

EX.No.:7 IMPLEMENTATION OF IMPLICIT AND EXPLICIT CURSOR TO MANIPULATE A TABLE IN PL/SQL

AIM:

To manipulate a table using Implicit and Explicit Cursors.

PROCEDURE:

PL/SQL - Cursors

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**.

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

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;
```

7. IMPLEMENTATION OF IMPLICIT AND EXPLICIT CURSOR TO MANIPULATE A TABLE IN PL/SQL

PROGRAM :

1.Program to update the table and increase the salary of each customer by 500

Using implicit Cursor

```
SQL>Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+

SQL>DECLARE
  2 total_rows number(2);
  3 BEGIN
  4 UPDATE customers
  5 SET salary = salary + 500;
  6 IF sql%notfound THEN
  7 dbms_output.put_line('no customers selected');
  8 ELSIF sql%found THEN
  9 total_rows := sql%rowcount;
 10 dbms_output.put_line( total_rows || ' customers selected ');
 11 END IF;
 12 END;
```

Output:

```
6 customers selected
PL/SQL procedure successfully completed.
SQL>Select * from customers;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 |
|  2 | Khilan   |  25 | Delhi     |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2500.00 |
|  4 | Chaitali |  25 | Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal    |  9000.00 |
|  6 | Komal    |  22 | MP        |  5000.00 |
+----+----------+-----+-----------+----------+
```

2. Program to illustrate the concepts of explicit cursors &minua;

```
SQL>DECLARE
   2 c_id customers.id%type;
   3 c_name customers.name%type;
  4 c_addr customers.address%type;
   5 CURSOR c_customers is
   6 SELECT id, name, address FROM customers;
   7 BEGIN
   8 OPEN c_customers;
  9 LOOP
  10 FETCH c_customers into c_id, c_name, c_addr;
  11 EXIT WHEN c_customers%notfound;
   12cdbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  13  END LOOP;
  14 CLOSE c_customers;
  15 END;
/
```

Output:

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP

PL/SQL procedure successfully completed.
```

RESULT:

Thus the manipulation of a table using Implicit and Explicit Cursors has been verified and executed successfully.

EX.No.:8 IMPLEMENTATION OF CREATING AND DROPING A TRIGGER IN PL/SQL

AIM:

To create and drop triggers in PL/SQL

PROCEDURE:

TRIGGER :A Trigger is a stored procedure that defines an action that the database automatically take when some database-related event such as Insert, Update or Delete occur.

TYPES OF TRIGGERS: The various types of triggers are as follows,

- Before: It fires the trigger before executing the trigger statement.
- After: It fires the trigger after executing the trigger statement.
- For each row: It specifies that the trigger fires once per row.
- For each statement: This is the default trigger that is invoked. It specifies that the trigger fires once per statement.

VARIABLES USED IN TRIGGERS:

:new

:old

These two variables retain the new and old values of the column updated in the database.

The values in these variables can be used in the database triggers for data manipulation

Syntax:

```
Create or replace trigger <trg_name> Before /After Insert/Update/Delete
[of column_name, column_name….]
on <table_name>
[for each row]
[when condition]
begin
---statement
end;
```

Create a trigger that insert current user into a username column of an existing table

Procedure for doing the experiment:

1. Create a table student4 with name and username as arguments
2. Create a trigger for each row that insert the current user as user name into a table
3. Execute the trigger by inserting value into the table

8.IMPLEMENTATION OF CREATING AND DROPING A TRIGGER IN PL/SQL

COMMANDS:

```
SQL> create table itstudent4(name varchar2(15),username varchar2(15));

Table created.

SQL> create or replace trigger student4 before insert on student4 for each row

 2 declare

 3 name varchar2(20);

 4 begin

 5 select user into name from dual;

 6 :new.username:=name;

 7 end;

8 /

Trigger created.
```

Output:

```
SQL> insert into student4 values('&name','&username');

Enter value for name: akbar

Enter value for username: ranjani

old 1: insert into student4 values('&name','&username')

new 1: insert into student4 values('akbar','ranjani')

1 row created.

SQL> /

Enter value for name: suji

Enter value for username: priya

old 1: insert into student4 values('&name','&username')

new 1: insert into student4 values('suji','priya')

1 row created.

SQL> select * from itudent4;
```

```
NAME USERNAME
--------------- ---------------
akbar SCOTT
 suji SCOTT
```

Develop a query to Drop the Created Trigger

```
SQL> drop trigger ittrigg;
 Trigger dropped.
```

RESULT:

Thus the creation and dropping of triggers was performed and executed successfully

EX.No.:9 IMPLEMENTATION OF PROCEDURE AND FUNCTION MANIPULATE A DATABASE USING PL/SQL

AIM:

To develop procedures and function for various operations

PROCEDURE:

A procedure is a block that can take parameters (sometimes referred to as arguments) and be invoked. Procedures promote reusability and maintainability. Once validated, they can be used in number of applications. If the definition changes, only the procedure are affected, this greatly simplifies maintenance. Modularized program development: · Group logically related statements within blocks. · Nest sub-blocks inside larger blocks to build powerful programs. · Break down a complex problem into a set of manageable well defined logical modules and implement the modules with blocks.

KEYWORDS AND THEIR PURPOSES

REPLACE: It recreates the procedure if it already exists.

PROCEDURE: It is the name of the procedure to be created.

ARGUMENT: It is the name of the argument to the procedure. Parenthesis can be omitted if no arguments are present.

IN: Specifies that a value for the argument must be specified when calling the procedure ie., used to pass values to a sub-program. This is the default parameter.

OUT: Specifies that the procedure passes a value for this argument back to it‟s calling environment after execution ie. used to return values to a caller of the sub-program. INOUT: Specifies that a value for the argument must be specified when calling the procedure and that procedure passes a value for this argument back to it‟s calling environment after execution.

RETURN: It is the data type of the function‟s return value because every function must return a value, this clause is required.

PROCEDURES

Syntax :

create or replace procedure <procedure name> (argument {in,out,inout} datatype) {is,as}

variable declaration;

constant declaration;

begin

PL/SQL subprogram body;

exception

exception PL/SQL block;

end;

FUNCTIONS

Syntax:

create or replace function <function name> (argument in datatype,……) return datatype {is,as}

variable declaration;

constant declaration;

begin

PL/SQL subprogram body;

exception

exception PL/SQL block;

end;

9. IMPLEMENTATION OF PROCEDURE AND FUNCTION MANIPULATE A DATABASE USING PL/SQL

Tables used:

```
SQL> select * from ititems;

ITEMID ACTUALPRICE ORDID PRODID

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

 101 2000 500 201

 102 3000 1600 202

 103 4000 600 202
```

<u>Program For General Procedure – Selected Record‘S Price Is Incremented By 500 , Executing The Procedure Created And Displaying The Updated Table</u>

```
SQL> create procedure itsum(identity number, total number) is price number;
 2 null_price exception;
 3 begin
 4 select actualprice into price from ititems where itemid=identity;
 5 if price is null then
 6 raise null_price;
 7 else
 8 update ititems set actualprice=actualprice+total where itemid=identity;
 9 end if;
10 exception
11 when null_price then
12 dbms_output.put_line('price is null');
13 end;
14 /
Procedure created.
SQL> exec itsum(101, 500);
```

```
PL/SQL procedure successfully completed.

SQL> select * from ititems;

ITEMID ACTUALPRICE ORDID PRODID

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

 101 2500 500 201

 102 3000 1600 202

 103 4000 600 202
```

Procedure For IN Parameter – Creation, Execution

```
SQL> set serveroutput on;

SQL> create procedure yyy (a IN number) is price number;

 2 begin

 3 select actualprice into price from ititems where itemid=a;

4 dbms_output.put_line('Actual price is ' || price);

 5 if price is null then

 6 dbms_output.put_line('price is null');

 7 end if;

 8 end;

 9 /

Procedure created.

SQL> exec yyy(103);

Actual price is 4000

PL/SQL procedure successfully completed.
```

Procedure For OUT Parameter – Creation, Execution

```
SQL> set serveroutput on;

SQL> create procedure zzz (a in number, b out number) is identity number;

 2 begin

 3 select ordid into identity from ititems where itemid=a;

 4 if identity<1000 then
 5 b:=100;
 6 end if;
 7 end;
 8 /
Procedure created.
SQL> declare
 2 a number;
 3 b number;
 4 begin
 5 zzz(101,b);
 6 dbms_output.put_line('The value of b is '|| b);
 7 end;
 8 /
The value of b is 100
PL/SQL procedure successfully completed.
```

<u>Procedure For INOUT Parameter – Creation, Execution</u>

```
SQL> create procedure itit ( a in out number) is
 2 begin
 3 a:=a+1;
 4 end;
 5 /
 Procedure created.
SQL> declare
 2 a number:=7;
 3 begin
 4 itit(a);
 5 dbms_output.put_line(„The updated value is „||a);
 6 end;
 7 /
The updated value is 8
PL/SQL procedure successfully completed.
```

Tables used:

```
SQL>select * from ittrain;
 TNO TFARE
 --------- ------------
 1001 550
 1002 600
```

Program For Function And It‘s Execution

```
SQL> create function trainfn (trainnumber number) return number is
 2 trainfunction ittrain.tfare % type;
 3 begin
 4 select tfare into trainfunction from ittrain where tno=trainnumber;
 5 return(trainfunction);
 6 end;
 7 /
Function created.
SQL> declare
 2 total number;
 3 begin
 4 total:=trainfn (1001);
 5 dbms_output.put_line('Train fare is Rs. '||total);
 6 end;
 7 /
Train fare is Rs.550
PL/SQL procedure successfully completed.
```

Factorial Of A Number Using Function — Program And Execution

```
 SQL> create function itfact (a number) return number is
 2 fact number:=1;
 3 b number;
 4 begin
 5 b:=a;
 6 while b>0
 7 loop
 8 fact:=fact*b;
 9 b:=b-1;
10 end loop;
11 return(fact);
12 end;
13 /
Function created.
SQL> declare
 2 a number:=7;
 3 f number(10);
 4 begin
 5 f:=itfact(a);
 6 dbms_output.put_line(„The factorial of the given number is‟||f);
 7 end;
 8 /
The factorial of the given number is 5040
```

Procedure to calculate total for the all the students and pass regno, mark1, & mark2 as arguments.

```
SQL> create table student2(regno number(3),name varchar(9),mark1 number(3),mark2
number(3));
Table created.
```

```
SQL> insert into student2
 2 values(&a,'&b',&c,&d);
Enter value for a: 110
Enter value for b: arun
Enter value for c: 99 Enter value for d: 100
old 2: values(&a,'&b',&c,&d)
new 2: values(110,'arun',99,100)
1 row created.
SQL> /
Enter value for a: 112 Enter value for b: siva Enter value for c: 99 Enter value
for d: 90
old 2: values(&a,'&b',&c,&d)
new 2: values(112,'siva',99,90)
1 row created.
SQL> select * from student2;
 REGNO NAME MARK1 MARK2
 110 arun 99 100
 112 siva 99 90
SQL> alter table student2 add(total number(5)); Table altered.
SQL> select * from student2;
REGNO NAME MARK1 MARK2 TOTAL
 110 arun 99 100
 112 siva 99 90
SQL> create or replace procedure p1(sno number,mark1 number,mark2 number) is
 2 tot number(5);
 3 begin
 4 tot:=mark1+mark2;
 5 update itstudent2 set total=tot where regno=sno;
 6 end;
 7 /
Procedure created.
SQL> declare
 2 cursor c1 is select * from student2;
 3 rec itstudent2 % rowtype;
 4 begin
 5 open c1;
 6 loop
 7 fetch c1 into rec;
8 exit when c1%notfound;
 9 p1(rec.regno,rec.mark1,rec.mark2);
10 end loop;
11 close c1;
12 end;
13 /
PL/SQL procedure successfully completed.
```

Output:

```
SQL> select * from student2;
REGNO NAME MARK1 MARK2 TOTAL
--------- --------- ---------- ---------- ----------
 110 arun 99 100 199
112 va 99 90 189
```

PL/SQL procedure that takes two numbers as parameter and displays the multiplication of the first parameter till the second parameter.

//p2.sql

```
create or replace procedure multi_table (a number, b number) as
mul number;
begin
```

```
for i in 1. .b
loop
mul : = a * i;
dbms_output.put_line (a || „*‟ || i || „=‟ || mul);
end loop;
end;
```

//pq2.sql

```
declare
a number; b number;
begin
a:=&a; b:=&b; multi_table(a,b);
end;
```

Output:

```
SQL> @p2.sql;
 Procedure created.
SQL> @pq2.sql;
Enter value for a: 4
old 5: a:=&a; new 5: a:=4;
Enter value for b: 3
old 6: b:=&b; new 6: b:=3;
4*1=4
4*2=8
4*3=12
```

Consider the EMPLOYEE (EMPNO, SALARY, ENAME) Table.

Write a procedure raise_sal which increases the salary of an employee. It accepts an employee number and salary increase amount. It uses the employee number to find the current salary from the EMPLOYEE table and update the salary.

//p3.sql

```
create or replace procedure raise_sal( mempno employee . empno % type, msal_percent
number ) as
begin
update employee set salary = salary + salary*msal_percent /100 where empno = mempno;
end;
/
```

//pq3.sql

```
declare
cursor c1 is select * from emp;
 rec emp % rowtype;
 begin
 open c1;
 loop
 fetch c1 into rec;
 exit when c1%notfound;
 raisal(rec.empno,10);
 end loop;
 close c1;
 end;
 /
```

Output:

```
SQL> @p3.sql;
 Procedure created.
SQL> select * from emp;
 EMPNO ENAME JOB DEPTNO SAL
---------- ------------------- ------------- ---------- ----------
 1 Mathi AP 1 10000
 2 Arjun ASP 2 15000
 3 Gugan ASP 1 15000
 4 Karthik Prof 2 30000
 5 Akalya AP 1 10000
SQL> @pq3.sql;
PL/SQL procedure successfully completed.
SQL> select * from emp;
 EMPNO ENAME JOB DEPTNO SAL
---------- ------------------- ------------- ---------- ----------
 1 Mathi AP 1 11000
 2 Arjun ASP 2 16500
 3 Gugan ASP 1 16500
 4 Karthik Prof 2 33000
 5 Akalya AP 1 11000
```

Write a PL/SQL function CheckDiv that takes two numbers as arguments and returns the values 1 if the first argument passed to it is divisible by the second argument, else will return the value 0;

//p4.sql

```
create or replace function checkdiv (n1 number, n2 number) return number as res
number;
begin
if mod (n1, n2) = 0 then
res := 1;
else
res:= 0;
end if;
return res;
end;
/
```

//pq4.sql

```
declare
```

```
a number;

b number;

begin

a:=&a; b:=&b;

dbms_output.put_line(„result=‟||checkdiv(a,b));

end;

/
```

Output:

```
SQL> @p4.sql;

Function created.

SQL> @pq4.sql;

Enter value for a: 4

old 5: a:=&a; new 5: a:=4;

Enter value for b: 2

old 6: b:=&b; new 6: b:=2;

result=1
```

<u>Write a PL/SQL function called POW that takes two numbers as argument and return the value of the first number raised to the power of the second .</u>

//p5.sql

```
create or replace function pow (n1 number, n2 number) return number as

res number;

begin

select power ( n1, n2) into res from dual; return res;

end;
```

or

```
create or replace function pow (n1 number, n2 number) return number as

res number : =1;

begin

for res in 1..n2
```

```
loop
res : = n1 * res;
end loop;
return res;
end;
```

//pq5.sql

```
declare
a number;
b number;
begin
a:=&a; b:=&b;
dbms_output.put_line('power(n1,n2)='||pow(a,b));
end;
/
```

Output:

```
SQL> @p5.sql;
Function created.
SQL> @ pq5.sql;
Enter value for a: 2
old 5: a:=&a;
new 5: a:=2;
Enter value for b: 3
old 6: b:=&b;
new 6: b:=3;
power(n1,n2)=8
```

Write a PL/SQL function ODDEVEN to return value TRUE if the number passed to it is EVEN else will return FALSE.

//p6.sql

```
create or replace function oddeven (n number) return boolean as
```

```
begin
if mod (n, 2) = 0 then return true;
else
return false;
end if;
end;
/
```

//pq6.sql

```
declare
a number; b boolean;
begin
a:=&a; b:=oddeven(a);
if b then
dbms_output.put_line('The given number is Even');
else
dbms_output.put_line('The given number is Odd');
end if;
end;
/
```

Output:

```
SQL> @p6.sql;
Function created.
SQL> @pq6.sql;
Enter value for a: 5
old 5: a:=&a; new 5: a:=5;
The given number is Odd
```

RESULT:

Thus the procedures and function for various operations was developed and executed successfully.

EX.No.:10 IMPLEMENTATION OF HANDLING EXCEPTION IN QUERY

AIM:

To handle exceptions in PL/SQL Program

PROCEDURE:

An error occurs during the program execution is called Exception in PL/SQL.

PL/SQL facilitates programmers to catch such conditions using exception block in the program and an appropriate action is taken against the error condition.

There are two type of exceptions:

- **System-defined Exceptions**
- **User-defined Exceptions**

SYNTAX:

```
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;
```

10. IMPLEMENTATION OF HANDLING EXCEPTION IN QUERY

COMMANDS:

```
SET SERVEROUTPUT ON;
DECLARE
   c_id customers.id%type := 8;
   c_name customerS.Name%type;
   c_addr customers.address%type;
BEGIN
   SELECT  name, address INTO  c_name, c_addr
   FROM customers
   WHERE id = c_id;
   DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
   DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

OUTPUT:

```
No such customer!
PL/SQL procedure successfully completed.
```

User-defined Exceptions

```
DECLARE
   c_id customers.id%type := &cc_id;
   c_name customerS.Name%type;
```

```
  c_addr customers.address%type;
  -- user defined exception
  ex_invalid_id  EXCEPTION;
BEGIN
  IF c_id <= 0 THEN
    RAISE ex_invalid_id;
  ELSE
    SELECT  name, address INTO  c_name, c_addr
    FROM customers
    WHERE id = c_id;
    DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
    DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
  END IF;
EXCEPTION
  WHEN ex_invalid_id THEN
    dbms_output.put_line('ID must be greater than zero!');
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

OUTPUT:

```
Enter value for cc_id: -6 (let's enter a value -6)
old  2: c_id customers.id%type := &cc_id;
new  2: c_id customers.id%type := -6;
ID must be greater than zero!
PL/SQL procedure successfully completed.
```

RESULT:

Thus the exceptions are handled in PL/SQL Program verified and executed successfully.

EX.No.:11 DESIGNING A DATABASE USING ER MODELLING AND NORMALIZATION

AIM:

To design a database using ER Modeling and Normalization

PROCEDURE:

NORMALIZATION

Normalization is the analysis of functional dependencies between attributes/data items of user

views. It reduces a complex user view to a set of small and stable subgroups of the fields and relations.

This process helps to design a logical data model known as conceptual data model.

There are different normal forms

1. First Normal Form(1NF)
2. Second Normal Form(2NF)
3. Third Normal Form(3NF)

FIRST NORMAL FORM (1NF)

1NF states that the domain of an attribute must include only atomic values and that value of any

attribute in a tuple must be a single value from the domain of that attribute. Hence 1NF disallows

multivalued attributes, composite attributes. It disallows “relations within relations”.

SECOND NORMAL FORM (2NF)

A relation is said to be in 2NF if it is already in 1NF and it has no partial dependency. 2NF is based

on the concept of full functional dependency.

A functional dependency(FD) x→y is fully functional dependency is (x-(A))→y does not hold dependency any more if A→x.

A functional dependency x→y is partial dependency if A can be removed which does not affect the

dependency ie (x-(A))→y holds.

A relation is in 2NF if it is in 1NF and every non-primary key attribute is fully and functionally

dependent on primary key.

A relation is in 1NF will be in the 2NF if one of the following conditions is satisfied:

1. The primary key consist of only one attribute.

2. No non-key attribute exist in relation ie all the attributes in the relation are components of the primary

key.

Every non-key attribute is functionally dependent on full set of primary key attributes.

THIRD NORMAL FORM (3NF)

A relation is said to be in 3NF if it is already in 2NF and it has no transitive dependency.

A FD x→y in a relation schema R is a transitive dependency if there is a set of attributes z that is neither a candidate key nor a subset of any key of the relation and both x→z and z→y hold.

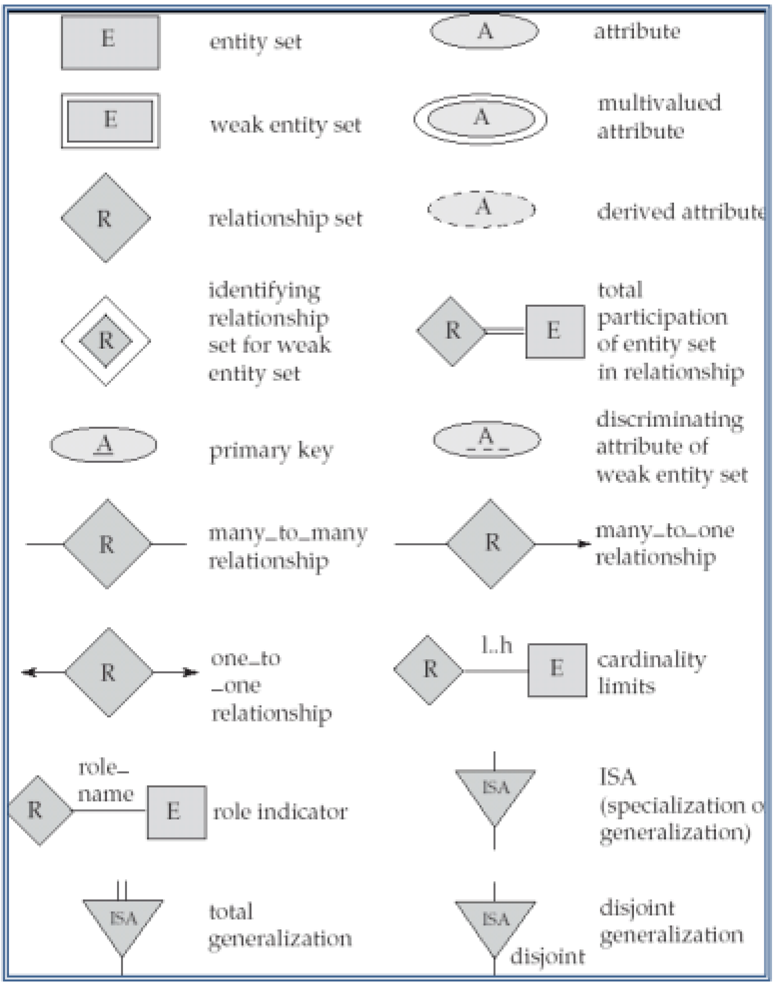
Entity relationship diagram (ERD):

An entity relationship diagram (ERD) shows the relationships of entity sets stored in a database. An

entity in this context is an object, a component of data. An entity set is a collection of similar entities.

These entities can have attributes that define its properties.

Notation


E
entity set
A
attribute
E
weak entity set
A
multivalued attribute
R
relationship set
A
derived attribute
R
identifying relationship set for weak entity set
R
E
total participation of entity set in relationship
A
primary key
A
discriminating attribute of weak entity set
R
many_to_many relationship
R
many_to_one relationship
R
one_to _one relationship
R
l..h
E
cardinality limits
R
role_ name
E
role indicator
ISA
ISA (specialization o generalization)
ISA
total generalization
ISA
disjoint
disjoint generalization

11.DESIGNING A DATABASE USING ER MODELLING AND NORMALIZATION

First Normal Form

1. Create a property table with the following fields: property id, country name, padd, area, price, tax rate and having property id as the primary key.

```
SQL> create table prop(propid number(2) primary key, cname varchar(20), padd varchar(50), area int,
price number(9,2),tax_rate number(2));
SQL> desc prop;
Name Null? Type
--------------------------------------- -------- ----------------------------
PROPID NOT NULL NUMBER(2)
CNAME VARCHAR2(20)
PADD VARCHAR2(50)
AREA NUMBER(38)
PRICE NUMBER(9,2)
TAX_RATE NUMBER(2)
```

2. Insert values in the property table.

```
SQL> insert into prop values('34','india','ganthi nagar,Coimbatore, india','500','500000','2');
1 row created.
SQL> insert into prop values('45','united states','first street southeast, Washington, United
states','400','2550000','5');
1 row created.
SQL> insert into prop values('39','scotland','capelrig road, Glasgow, scotland','600','2500000','4');
1 row created.
```

Before Normalization

prop

Propid Cname Padd Area Price Tax_rate

Normalization to first normal form

1. Creating the prop11 tabale with propid, cname, area,price, tax_rate from prop.

```
SQL> create table prop11 as select , cname, area,price, tax_rate from prop;
```

2. Creating the table prop12 with propid, sname,city,country from prop

```
SQL> create table prop12 as select propid,padd from emp;
```

3. Altering the table prop11 with primary key on prop.

```
SQL> alter table prop12 add constraint c1 foreign key(propid) references prop11(propid);
```

4. Altering the table prop12 with foreign key on propid with reference from prop11.

```
SQL> alter table prop12 add constraint c1 foreign key(propid) references prop11(propid);
```

After Normalization

Prop11

Propid Cname Area Price Tax_rate

Prop12

Propid sname City country

SECOND NORMAL FORM

Normalization to Second Normal Form

1. Create the table prop21 with propid, cname, area, price from the table prop.

```
SQL> create table prop21 as select propid,cname,area, price from prop;
```

2. Create the table prop22 with cname, tax_rate from the table prop.

```
SQL> create table prop22 as select cname,tax_rate from prop;
```

3. Alter table prop21 with a primary key constraint on propid.

```
SQL> alter table prop21 add constraint prop21 primary key(propid);
```

4. Alter table prop22 with a primary key constraint on cname.

```
SQL> alter table prop22 add constraint prop22 primary key(cname);
```

5. Alter table prop21 with foreign key on cname with references on cname from prop22.

```
SQL> alter table prop21 add constraint prop212 foreign key(cname) references prop22(cname);
```

After normalization

Prop21 prop22

Propid Cname Area Price

THIRD NORMAL FORM

The 2NF table is given as input here and convert it to 3NF.

Input: prop21, prop22 tables.

For converting to 3NF it is enough making changes in prop21 table.

Before Normalization

Prop21

Propid Cname Area Price

1. Create table prop31 with propid, cname, area from prop21.

SQL> create table prop31 as select propid,cname,area from prop21;

2. Create table prop32 with area, price from prop21.

SQL> create table prop32 as select area, price from prop21;

3. Alter table prop31 with the constraint primary key on propid.

SQL> alter table prop31 add constraint prop31 primary key(propid);

4. Alter table prop32 with the constraint primary key on area.

SQL> alter table prop32 add constraint prop32 primary key(area);

5. Alter table prop31 with the constraint foreign key on area with refernce from area in prop32.

SQL> alter table prop31 add constraint prop311 foreign key(area) references prop32(area);

After Normalization

Prop31 prop32

Propid Cname Area

RESULT:

Thus the database has been designed using ER Modeling and Normalization successfully.

EX.No.:12

DEVELOPING AN ENTERPRISE APPLICATION USING USER INTERFACE AND DATABASE

AIM:

To design the payroll processing system in visual basic using ORACLE as backend

PROCEDURE :

1. Create table with following fields.

NAMENULL?	TYPE
EID	NUMBER(10)
ENAME	VARCHAR2(1 0)
DES	VARCHAR2(10)
BASICPAY	NUMBER(10)
HRA	NUMBER(10)
DA	NUMBER(10)
MA	NUMBER(10)
GROSSPAY	NUMBER(10)
DEDUCTION	NUMBER(10)
NETPAY	NUMBER(10)

2. Insert all possible values into the table.
3. Enter commit work command.
4. Go to Start --> Settings --> Control Panel --> Administrative tools --> Data Sources(ODBC) --> User DSN --> Add --> Select ORACLE database driver --> OK.
5. One new window will appear. In that window, type data source name as table name created in ORACLE. Type user name as secondcsea.

Procedure for adodc in visual basic:

1. In visual basic create tables, command buttons and then text boxes.
2. In visual basic, go to start menu.
3. Projects --> Components --> Microsoft ADO Data Control 6.0 for OLEDB --> OK.
4. Now ADODC Data Control available in tool box.
5. Drag and drop the ADODC Data Control in the form.
6. Right click in ADODC Data Control, then click ADODC properties.
7. One new window will appear.
8. Choose general tab, select ODBC Data Sources name as the table name created in ORACLE
9. Choose authentication tab and select username password as secondcsea and secondcsea
10. Choose record name-->select command type as adcmdTable.
11. Select table or store procedure name as table created in ORACLE.
12. Click Apply-->OK
13. Set properties of each text box.
14. Select the data source as ADODC1.
15. Select the Data field and set the required field name created in table

12. DEVELOPING AN ENTERPRISE APPLICATION USING USER INTERFACE AND DATABASE

VB SCRIPT:

```
ADD:
Private Sub Add_Click() Adodc1.Recordset.AddNew Textl.SetFocus End Sub
DELETE:
Private Sub Delete_Click()
If MsgBox ("DELETE IT?",vb OKCancel)= vbOK Then

Adodc1.Recordset.Delete End If MsgBox "ONE ROW DELETED" Textl.Text - " "
Text2.Text - " "
Text3.Text - " "
Text4.Text - " "
Text5.Text - " "
Text6.Text - " "
Text7.Text - " "
Text8.Text - " "
Text9.Text - " " Textl0.Text - " " End Sub SAVE:
Private Sub Save_Click()
If MsgBox ("SAVE IT?",vbOKCancel ) = vbOK Then Adodc1.Recordset.Update Else Adodc1.Recordset.CancelUpdate
End If End Sub FIND:
Private Sub Find_Click() Dim N as string
N = InputBox ("Enter the accno") Adodc1.Recordset.Find "accno=" & N
If Adodcl.Recordset.BOF or Adodc1.Recordset.EOF Then MsgBox "Record not found" End If End Sub
```

UPDATE:

Private Sub Update_Click() Adodc1.Recordset.EditMode Adodc1.Recordset.Update End Sub FIRST:

Private Sub First_Click() Adodc1.Recordset.MoveFirst End Sub LAST:

Private Sub Last_Click() Adodc1.Recordset.MoveLast End Sub NEXT:

Private Sub Next_Click() Adodc1.Recordset.MoveNext End Sub PREVIOUS:

Private Sub Previous_Click() Adodc1.Recordset.MovePrevious End Sub DEPOSIT:

Private Sub Deposit_Click0 Dim N1 as string

N = InputBox ("Enter the accno") Adodcl.Recordset.Find "accno=" & N Nl = InputBox ("Enter the amount") Text4.Text= val (Text4.Text) + Nl Adodc1.Recordset.Update

End Sub

WITHDRAW:

Private Sub Withdraw_Click() Dim Nl as string

N = InputBox ("Enter the accno") Adodcl.Recordset.Find "accno=" & N Nl = InputBox ("Enter the amount") Text4.Text= val (Text4.Text) - Nl Adodc l.Recordset.Update

End Sub EXIT:

Private Sub Add_Click() Unload Me End Sub FUNCTION:

Function Calculate()

Text8.Text=val(Text4.Text) + val (Text5.Text) + val (Text6.Text) + val (Text7.Text) Text9.Text=val(Text5.Text) + val (Text6.Text) + val (Text7.Text)

Text 10.Text=val(Text8.Text) + val (Text9.Text) End Function BASICPAY,HRA,DA,MA,GROSSPAY,DEDUCTION,NETPAY:

Private Sub Basicpay_Change() Call Calculate End Sub

Private Sub HRA_Change() Call Calculate End Sub

Private Sub DA_Change() Call Calculate End Sub

Private Sub MA_Change() Call Calculate

End Sub

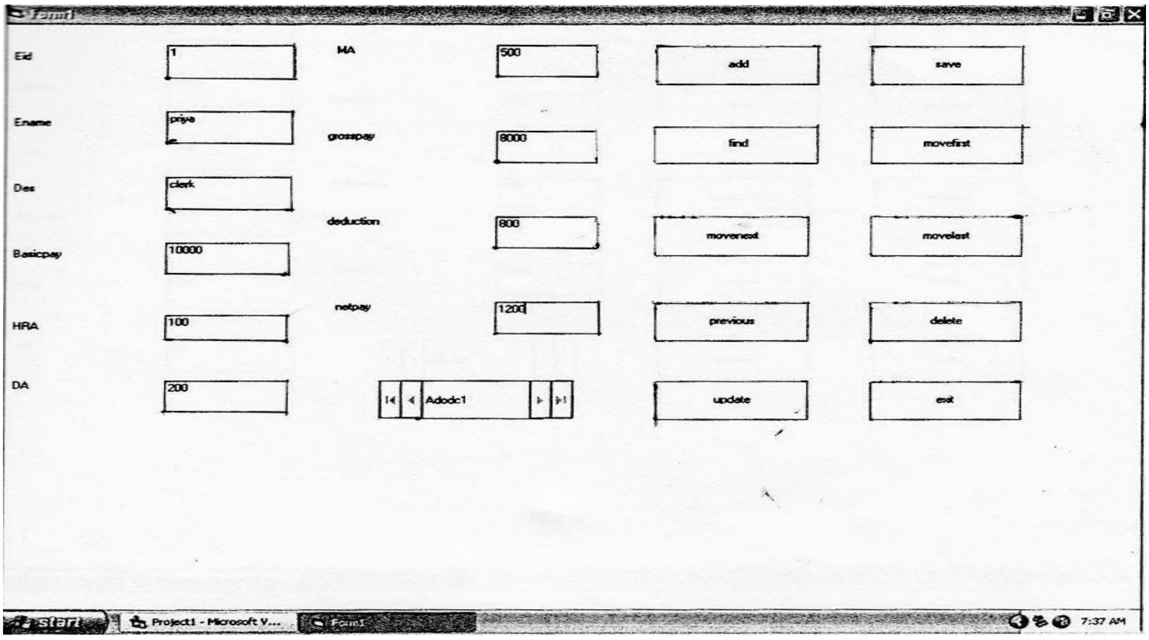
Private Sub Grosspay_Change() Call Calculate End Sub

```
Private Sub Deduction_Change() Call Calculate End Sub

Private Sub Netpay_Change() Call Calculate End Sub
```

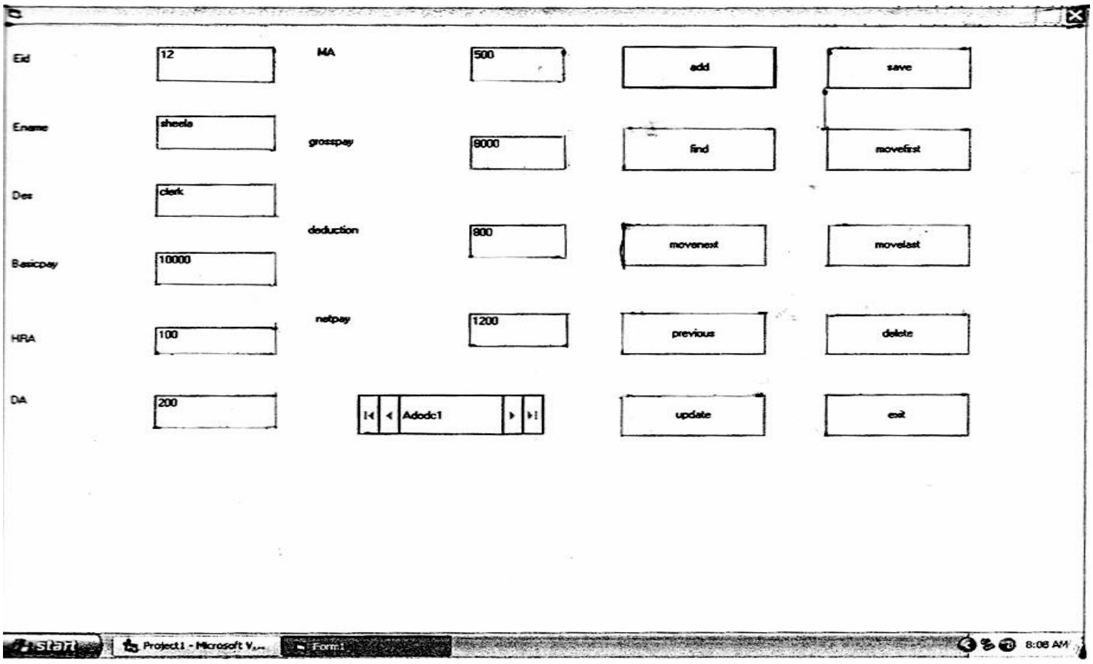
OUTPUT:

STARTUP FORM



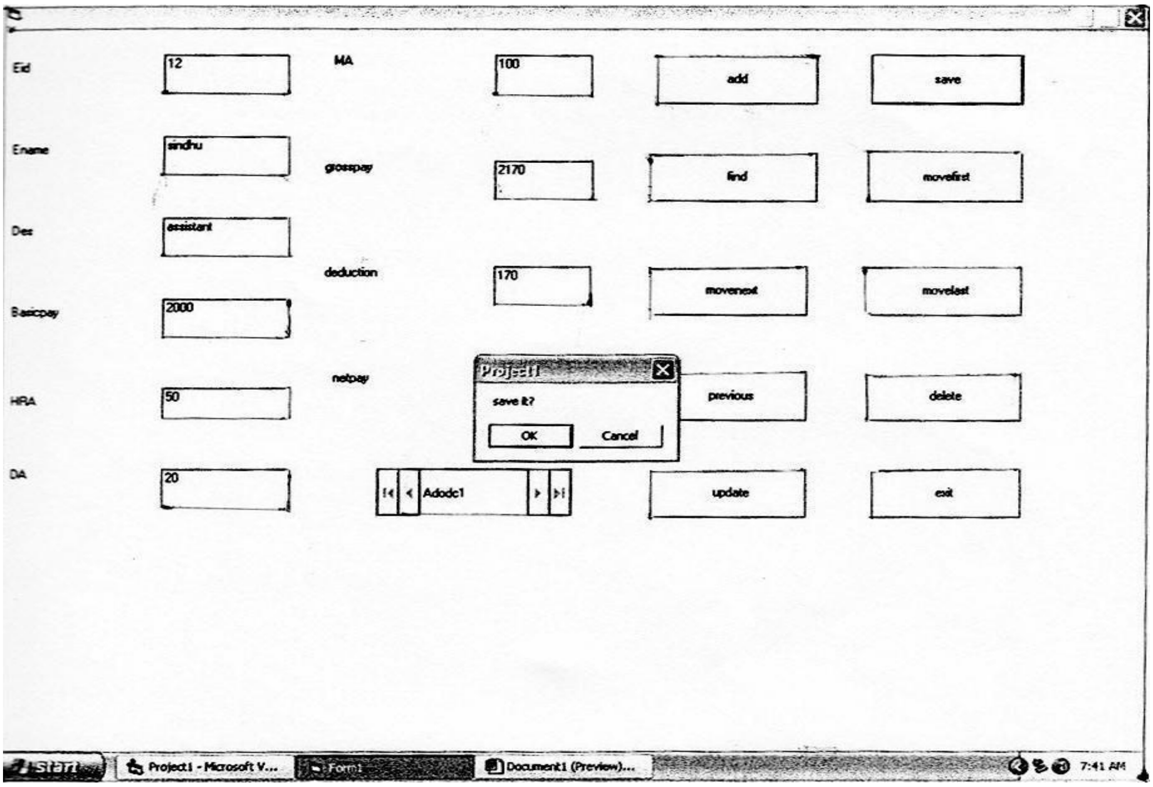
Eid 1
Ename priya
Des clerk
Basicpay 10000
HRA 100
DA 200
MA 500
grosspay 8000
deduction 800
netpay 1200

ADD FORM



Eid 12
Ename sheela
Des clerk
Basicpay 10000
HRA 100
DA 200
MA 500
grosspay 8000
deduction 800
netpay 1200

SAVE FORM



Eid 12

Ename sindhu

Des assistant

Basicpay 2000

HRA 50

DA 20

MA 100

grosspay 2170

deduction 170

netpay

Project1

save it?

RESULT:

Thus the payroll processing system was designed in Visual Basic using ORACLE as backendand the output was verified

EX.No.:13

IMPLEMENTATION OF CREATING INDEX

AIM:

To create and drop index in a table

PROCEDURE:

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. An index is a pointer to data in a table. An index in a database is very similar to an index in the back of abook. An index helps speed up SELECT queries and WHERE clauses, but it slows down data input, with UPDATE and INSERT statements. Indexes can be created or dropped with no effect on the data. Index in sql is created on existing tables to retrieve the rows quickly. When there are thousands of records in a table, retrieving information will take a long time. When an index is created, it first sorts the data and then it assigns a ROWID for each row.

An index can be created in a table to find data more quickly and efficiently.

The users cannot see the indexes, they are just used to speed up searches/queries

1.Syntax to create Index

CREATE INDEX index_name ON table_name (column_name1,column_name2...);

2.Syntax to create SQL unique index

CREATE UNIQUE INDEX index_name ON table_name (column_name1, column_name2...);

•index_name is the name of the INDEX.

•table_name is the name of the table to which the indexed column belongs.

•column_name1, column_name2.. is the list of columns which make up the INDEX.

3.The Drop Index Command

An index can be dropped using SQL DROP command. Care should be taken when dropping an index because performance may be slowed or improved.

DROP INDEX index_name;

13.IMPLEMENTATION OF CREATING INDEX

COMMANDS:

```
SQL> create table persons (first name varchar (20), last name varchar(10));

Table created;
```

Create an index for the above relation based on last name

```
SQL> create index plndex on persons (last name);

Index created.

SQL> select * from persons; No rows selected.

SQL> drop plndex on persons;

Drop index plndex on persons

* ERROR at line1:

ORA_00950: Invalid DROP option
```

RESULT:

Thus the index has been created and dropped in a table has been implemented and executed successfully.

EX.No.:14 INTRODUCTION TO NOSQL DATABASES USING MONGODB

Aim:

The objective is to introduce some features of non-relational or NoSQL databases using MongoDB. MongoDB stores data in JSON objects which it calls documents and uses a custom language for queries.

Installation

Option 1:

1. Set up a free cluster on Mongo Atlas by following the instructions here: https://docs.atlas.mongodb.com/tutorial/deploy-free-tier-cluster/
2. Install the mongo shell on your machine and connect to your cluster following the instructions on the dashboard.

Option 2:

1. Download and setup MongoDB for your OS following these instructions https://www.mongodb.com/download-center/community

2. Start the server. Then use mongo shell to connect your server: https://docs.mongodb.com/manual/mongo/ Preparation The instructions below assume you have MongoDB installed and started on your local machine. For Atlas, create a database and a user by following the instructions on the dashboard.

1. Once you have installed and started the Mogodb you can log into your server as root. mongo -u root

2. To create a new database do use mongolab

3. Now create a new user to access the database.

db.createUser({user:"e14xxx", pwd:"abc123", roles:[{role: "dbOwner" , db:"mongolab"}]})

4. Log out of the mongo shell and log back in using the user you created. mongo localhost/mongolab -u e14xxx

Data Validation:

Document databases are a flexible alternative to the predefined schemas of relational databases. Each document in a collection can have a unique set of fields, and those fields can be added or removed from documents once they are inserted. Since the data fields can be changed for each document in a collection, data validation is extremely important to ensure queries run predictability.

To create the customer collection with a custom data validation function, enter the following code.

```
db.createCollection("customers", {
validator: {
$and: [
{
"firstName": {$type: "string", $exists: true}
 },
{
"lastName": { $type: "string", $exists: true}
},
{
"phoneNumber":
{
$type: "string",
$exists: true,
$regex: /^[0-9]{3}-[0-9]{3}-[0-9]{4}$/
}
},
{
"email": {
$type: "string",
$exists: true
}
}    ]
} })
```

RESULT:

Thus the Data validation in NoSQL databases using MongoDB has been implemented and executed successfully.