UNIVERSITY COLLEGE OF ENGINEERING NAGERCOIL

(ANNA UNIVERSITY CONSTITUENT COLLEGE)

KONAM, NAGERCOIL – 629 004

RECORD NOTE BOOK

CS 3481-DATABASE MANAGEMENT SYSTEMS LABORATORY

Register No :

Name :

Year/Semester :

Department :

UNIVERSITY COLLEGE OF ENGINEERING NAGERCOIL

(ANNA UNIVERSITY CONSTITUENT COLLEGE)

KONAM, NAGERCOIL – 629 004

Register No:

Certified that, this is the bonafide record of work done by ***Mr./Ms.*** *.....* *of IV Semester in Computer Science and Engineering of this college, in the* ***CS3481-Database Management Systems Laboratory*** *during academic year 2022-2023 in partial fulfillment of the requirements of the B.E Degree course of the Anna University Chennai.*

Staff-in-charge **Head of the Department**

This record is submitted for the University Practical Examination held on .....

Internal Examiner **External Examiner**

INDEX

EX NO:01	CREATE DATABASE TABLE, ADD CONSTRAINTS (PRIMARY KEY, UNIQUE, CHECK, NOT NULL), INSERT ROWS, UPDATE AND DELETE ROWS USING SQL DDL AND DML COMMANDS
DATE:	

AIM:

To create database table, add constraints (primary key, unique, check, not null), insert rows, update and delete rows using sql DDL and DML commands.

USER CREATION:

```
SQL*Plus: Release 11.2.0.2.0 Production on Fri Apr 28 21:03:05 2023

Copyright (c) 1982, 2014, Oracle.  All rights reserved.

SQL> connect sys as sysdba
Enter password:
Connected.
SQL> create user gokila identified by goki;

User created.

SQL> grant connect to gokila;

Grant succeeded.

SQL> grant all privileges to gokila;

Grant succeeded.

SQL> connect gokila;
Enter password:
Connected.
```

DDL COMMANDS

SQL:Create command:

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

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 **datatypes** of various columns in the create command itself.

Create table command will tell the database system to create a new table with the given table name and column information.

SYNTAX:

CREATE TABLE<table_name>

(column_name1 datatype1,

column_name2 datatype2,

column_name3 datatype3,

column_name4 datatype4**);**

QUERY AND OUTPUT:

```
SQL> create table student(name varchar(20),id int,reg int);

Table created.

SQL> desc student;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 NAME                                               VARCHAR2(20)
 ID                                                 NUMBER(38)
 REG                                                NUMBER(38)

SQL>
```

SQL: ALTER command

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

- To add a column to existing table
- To rename any existing column
- To change data type of any column or to modify its size.
- 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 data type);

ALTER Command: Add multiple new Columns

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

SYNTAX:

ALTER TABLE table_name **ADD**

(Column_name1 datatype1**,**

Column_name2 datatype2**);**

QUERY AND OUTPUT:

```
SQL> alter table student add(phone_no int);

Table altered.

SQL> desc student;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
 NAME                                               VARCHAR2(20)
 ID                                                 NUMBER(38)
 REG                                                NUMBER(38)
 PHONE_NO                                           NUMBER(38)
```

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 **);**

QUERY AND OUTPUT:

```
SQL> alter table student add(gender varchar(20) default'male');

Table altered.

SQL> desc student;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
 NAME                                               VARCHAR2(20)
 ID                                                 NUMBER(38)
 REG                                                NUMBER(38)
 PHONE_NO                                           NUMBER(38)
 GENDER                                             VARCHAR2(20)
```

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 **);**

QUERY AND OUTPUT:

```
SQL> alter table student modify id varchar(20);

Table altered.

SQL> desc student;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------
 NAME                                               VARCHAR2(20)
 ID                                                 VARCHAR2(20)
 REG                                                NUMBER(38)
 PHONE_NO                                           NUMBER(38)
 GENDER                                             VARCHAR2(20)
```

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**;**

QUERY AND OUTPUT:

```
SQL> alter table student rename column phone_no to phone;

Table altered.

SQL> desc student;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------
 NAME                                               VARCHAR2(20)
 ID                                                 VARCHAR2(20)
 REG                                                NUMBER(38)
 PHONE                                              NUMBER(38)
 GENDER                                             VARCHAR2(20)
```

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**);**

QUERY AND OUTPUT:

```
SQL> alter table student drop column phone;

Table altered.

SQL> desc student;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
 NAME                                               VARCHAR2(20)
 ID                                                 VARCHAR2(20)
 REG                                                NUMBER(38)
 GENDER                                             VARCHAR2(20)
```

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;

QUERY AND OUTPUT:

```
SQL> truncate table student;

Table truncated.

SQL> select*from student;

no rows selected

SQL> desc student;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------
 NAME                                               VARCHAR2(20)
 ID                                                 VARCHAR2(20)
 REG                                                NUMBER(38)
 GENDER                                             VARCHAR2(20)
```

RENAME query:

RENAME command is used to set a new name for any existing table.

SYNTAX:

RENAME old_table_name TO new_table_name;

QUERY AND OUTPUT:

```
SQL> rename student to class;

Table renamed.

SQL> desc class;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
 NAME                                               VARCHAR2(20)
 ID                                                 VARCHAR2(20)
 REG                                                NUMBER(38)
 GENDER                                             VARCHAR2(20)
```

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;

QUERY AND OUTPUT:

```
SQL> drop table class;

Table dropped.

SQL> desc class;
ERROR:
ORA-04043: object class does not exist
```

DML COMMANDS

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

INSERT command:

Insert command is used to insert data into a table.

SYNTAX:

INSERT INTO table_name **VALUES (**data1**,**

QUERY AND OUTPUT:

```
SQL> insert into class3 values('gokul',1,22,'male');

1 row created.

SQL> select*from class3;

NAME                          ID        AGE GENDER
-------------------- ---------- ---------- ------------
gokul                          1         22 male
```

INSERT VALUE INTO ONLY SPECIFIC COLUMNS:

We can use the **INSERT** command to insert values for only some specific columns of a row. We can specify the column name along with the values to be inserted.

SYNTAX:

INSERT INTO table_name **(**column1, column2, column3**….)** **VALUES (**value1, value2, value3**…);**

QUERY AND OUTPUT:

```
SQL> insert into class3(name,id)values('ravi',2);

1 row created.

SQL> select*from class3;

NAME                          ID        AGE GENDER
-------------------- ---------- ---------- ------------
gokul                          1         22 male
ravi                           2
```

INSERT NULL value to a column:

We can insert NULL values to a specific column using **INSERT** command.

SYNTAX:

INSERT INTO table_name
VALUES (value1, value2, NULL**);**

QUERY AND OUTPUT:

```
SQL> insert into class2 values('alan',NULL,20,'male');

1 row created.

SQL> select*from class2;

NAME                          ID        AGE GENDER
-------------------- ---------- ---------- ---------------
Gokul                          1         22 male
ravi                           2
alan                                     20 male
```

Insert DEFAULT value to a column:

We can set the default values to a particular column using the following syntax.

SYNTAX:

ALTER TABLE table_name **MODIFY** column_name
DEFAULT value**;**

QUERY AND OUTPUT:

```
SQL> alter table class2 modify gender default'male';

Table altered.

SQL>
```

If we run the below query, it'll automatically set default value in the gender column.

```
SQL> insert into class2 values('Egan',4,25,default);

1 row created.

SQL> select*from class2;

NAME                          ID        AGE GENDER
-------------------- ---------- ---------- -----------
Gokul                          1         22 male
ravi                           2
alan                                     20 male
Egan                           4         25 male
```

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**;**

QUERY AND OUTPUT:

```
SQL> update class2 set id=3 where name='alan';

1 row updated.

SQL> select*from class2;

NAME                          ID        AGE GENDER
------------------- ---------- ---------- ---------
Gokul                          1         22 male
ravi                           2
alan                           3         20 male
Egan                           4         25 male
```

Updating Multiple Columns:

We can also update values of multiple columns using a single **UPDATE** statement.

SYNTAX:

UPDATE table_name SET Column1_name=new_value,

Column2_name= **new_value WHERE** some_condition**;**

QUERY AND OUTPUT:

```
SQL> update class2 set id=3,age=25 where name='Egan';

1 row updated.

SQL> select*from class2;

NAME                          ID        AGE GENDER
------------------- ---------- ---------- ------------
Gokul                          1         22 male
ravi                           2
alan                           3         20 male
Egan                           3         25 male
```

DELETE Command:

DELETE command is used to delete data from a table.

Delete a particular Record from a Table:

If we want to delete a single record,we can use the WHERE clause to provide a condition in our DELETE statement.

SYNTAX:

DELETE FROM table_name **WHERE** some_condition**;**

Delete all Records from a table:

We can also delete all records from the table at a time using DELETE command.

SYNTAX:

DELETE FROM table_name**;**

QUERY AND OUTPUT:

Delete all records from a Table:

```
SQL> delete from class2;

4 rows deleted.

SQL> select*from class2;

no rows selected
```

Delete particular record from a table:

```
SQL> delete from class where
  2  age=21;

1 row deleted.

SQL> select *from class;

NAME       ID                    AGE GENDER
---------- ---------- ---------- ----------
akshay     1                      19 male
avinav     2                         male
alan       3                      20 male
```

CONSTRAINTS

Check constraint:

Whenever a CHECK constrain is applied to the table's column, and the user wants to insert the value in it, then the value will first be checked for certain conditions before inserting the value into that column.

SYNTAX:

CREATE TABLE table_name(column1 data_type,column2 data_type,…….CONSTRAINT constraint_name **CHECK**(c_column_name condition)[DISABLE]);

QUERY AND OUTPUT:

```
SQL> create table book6
  2  (BOOKNO NUMBER(5),
  3  ANAME VARCHAR(15),
  4  ACOUNTRY VARCHAR(15),
  5  CHECK (ACOUNTRY IN
  6  ('INDIA','US')));

Table created.

SQL> DESC BOOK6;
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------
 BOOKNO                                             NUMBER(5)
 ANAME                                              VARCHAR2(15)
 ACOUNTRY                                           VARCHAR2(15)

SQL> INSERT INTO BOOK6 VALUES
  2  (1,'GOKILA','INDIA');

1 row created.

SQL> INSERT INTO BOOK6 VALUES
  2  (1,'GOKILA','MALAYSIA');
INSERT INTO BOOK6 VALUES
*
ERROR at line 1:
ORA-02290: check constraint (RECORD.SYS_C007065) violated
```

Not Null Constraint:

The **NOT NULL** constraint is used to ensure that a given column of a table is never assigned the null value.

NOT NULL constraint has defined for a particular column, any insert or update operation that attempts to place a null value in that column will fail.

SYNTAX:

CREATE TABLE Table_**NAME**
(Column_Name_1 DataType (character_size of the column_1**)**
NOT NULL,
Column_Name_2 DataType(character_size of the column_2)
NOT NULL,……..,
Column_Name_N DataType(character_size of the column_N)**NOT NULL);**

QUERY AND OUTPUT:

```
SQL> CREATE TABLE BOOK9
  2  (BOOKNO NUMBER(5)NOT NULL,
  3  TITLE VARCHAR2(15) NOT NULL);

Table created.

SQL> DESC BOOK9;
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------
 BOOKNO                                    NOT NULL NUMBER(5)
 TITLE                                     NOT NULL VARCHAR2(15)

SQL> INSERT INTO BOOK9 VALUES
  2  (1,'GOKILA');

1 row created.

SQL> SELECT*FROM BOOK9;

    BOOKNO TITLE
---------- ---------------
         1 GOKILA

SQL> INSERT INTO BOOK9 VALUES
  2  ('GOKILA');
INSERT INTO BOOK9 VALUES
            *
ERROR at line 1:
ORA-00947: not enough values
```

Unique Constraint:

Unique Constraint can be applied to the table's column, and the user has not specified the value to be inserted in it.

SYNTAX:

CREATE TABLE table_name(Column_Name_1 DataType **NOT NULL**
UNIQUE,Column_Name_2 DataType
UNIQUE,Column_Name_3 DataType);

QUERY AND OUTPUT:

```
SQL> create table book4
  2  (bookno number(5)
  3  unique,Aname varchar(15));

Table created.

SQL> insert into book4 values(1,'Mary');

1 row created.

SQL> select*from book4;

    BOOKNO ANAME
---------- ---------------
         1 Mary

SQL> insert into book4 values(1,'Mary');
insert into book4 values(1,'Mary')
*
ERROR at line 1:
ORA-00001: unique constraint (RECORD.SYS_C007063) violated
```

Primary Constraint:

- Primary key constraint is a combination of NOT NULL and unique constraints.
- **NOT NULL** constraint and a unique constraint together forms a primary constraint.

SYNTAX:

CREATE TABLE table_name(column1 datatype null/not null,column2 datatype null/not null,…..

CONSTRAINT constraint-name

PRIMARY KEY (column1, column2…column_n));

QUERY AND OUTPUT:

```
SQL> create table book12(bookno number(5) primary key, Aname varchar(15));

Table created.

SQL> insert into book12 values(36,'Jani');

1 row created.

SQL> select * from book12;

    BOOKNO ANAME
---------- ---------------
        36 Jani

SQL> insert into book12 values(NULL,'Nehru');
insert into book12 values(NULL,'Nehru')
                          *
ERROR at line 1:
ORA-01400: cannot insert NULL into ("GOKILA"."BOOK12"."BOOKNO")
```

S

RESULT:

Thus, the query to create database table, add constraints (primary key, unique, check, not null), insert rows, update and delete rows using sql DDL DML and Constraints commands was executed successfully.

EX NO: 02	CREATE A SET OF TABLES, ADD FOREIGN KEY CONSTRAINTS AND INCORPORATE REFERENTIAL INTEGRITY
DATE:	

AIM:

To create a set of related tables in a relational database management system, define foreign key constraints between them, and ensure referential integrity by enforcing those constraints.

CREATE COMMAND:

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 datatypes of various columns in the create command itself.

Create table command will tell the database system to create a new table with the given table name and column information.

SYNTAX:

```
CREATE TABLE<table_name>
  (column_name1 datatype1,
   column_name2 datatype2,
   column_name3 datatpe3,
   column_name4 datatype4);
```

QUERY AND OUTPUT:

```
SQL> CREATE TABLE Persons(
  2  PersonID int PRIMARY KEY,
  3  Firstname varchar(15),
  4  Lastname varchar(15),
  5  Age int);

Table created.

SQL> desc Persons;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 PERSONID                                  NOT NULL NUMBER(38)
 FIRSTNAME                                          VARCHAR2(15)
 LASTNAME                                           VARCHAR2(15)
 AGE                                                NUMBER(38)
```

```
SQL> SELECT * FROM Persons;

  PERSONID FIRSTNAME       LASTNAME               AGE
---------- --------------- --------------- ----------
         1 John            Wick                    26
         2 Park            Jimin                   25
         3 Mark            Lee                     23
         4 George          Washington              35
```

FOREIGN KEY:

A foreign key is a column or set of columns in a table that is used to establish a link or relationship between that table and another table. It creates a logical association between the data in two tables by enforcing a referential integrity constraint, ensuring that the data in one table's foreign key column(s) matches the data in another table's primary key column(s).

SYNTAX:

```
CREATE TABLE table_name (
  column1 datatype,
  column2 datatype,
  ...
  CONSTRAINT fk_constraint_name
    FOREIGN KEY (column_name)
    REFERENCES referenced_table_name
(referenced_column_name)
);
```

QUERY AND OUTPUT:

```
SQL> CREATE TABLE orders (
  2    orderID int NOT NULL,
  3    orderNumber int NOT NULL,
  4    PersonID int,
  5    CONSTRAINT Fk_Personorder
  6    FOREIGN KEY(PersonID)
  7    REFERENCES persons(PersonID)
  8  );

Table created.

SQL> desc orders;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ORDERID                                   NOT NULL NUMBER(38)
 ORDERNUMBER                               NOT NULL NUMBER(38)
 PERSONID                                           NUMBER(38)
```

```
SQL> SELECT * FROM orders;

   ORDERID ORDERNUMBER   PERSONID
---------- ----------- ----------
         1       47837          1
         2       47838          2
         3       47654          3
         4       85856          4
```

ALTER COMMAND:

The ALTER command in SQL is used to modify an existing database object, such as a table, column, or constraint. When it comes to foreign keys, the ALTER command can be used to add, drop, or modify a foreign key constraint on an existing table.

There are two types of syntax to add foreign key constraints using ALTER command.

SYNTAX:

```
ALTER TABLE child_table
ADD CONSTRAINT constraint_name
FOREIGN KEY (child_column)
REFERENCES parent_table (parent_column)
[ON DELETE {CASCADE|SET NULL}]
[ON UPDATE {CASCADE|SET NULL}];
```

```
ALTER TABLE child_table
ADD FOREIGN KEY (child_column)
REFERENCES parent_table (parent_column)
[ON DELETE {CASCADE|SET NULL}]
[ON UPDATE {CASCADE|SET NULL}];
```

QUERY AND OUTPUT:

```
SQL> ALTER TABLE orders
  2  ADD CONSTRAINTS Fk_PersonOrders
  3  FOREIGN KEY(PersonID)
  4  REFERENCES Persons(PersonID);

Table altered.
```

```
SQL> ALTER TABLE orders
  2  ADD FOREIGN KEY(PersonID)
  3  REFERENCES Persons(PersonID);

Table altered.
```

REFERENTIAL INTEGRITY:

Referential integrity is a relational database property that ensures data consistency and accuracy between related tables. It is a set of rules that governs the relationships between tables, ensuring that data is always valid and consistent.

Referential integrity is typically enforced through the use of foreign key constraints, which establish a relationship between a primary key column in one table and a matching foreign key column in another table. When a foreign key is defined, it ensures that any value in the foreign key column must exist in the primary key column of the referenced table, or be null if allowed.

QUERY AND OUTPUT:

```
SQL> INSERT INTO orders values
  2  (5,86755,5);
INSERT INTO orders values
*
ERROR at line 1:
ORA-02291: integrity constraint (BALABHARATHY.SYS_C007133) violated - parent
key not found
```

```
SQL> DELETE FROM Persons
  2  WHERE PersonID=1;
DELETE FROM Persons
*
ERROR at line 1:
ORA-02292: integrity constraint (BALABHARATHY.FK_PERSONORDER) violated - child
record found
```

DELETE COMMAND:

When a foreign key constraint is defined between two tables, the delete operation on a row in the parent table can have different effects on the rows in the child table. The delete operation in a foreign key relationship is an important aspect of maintaining referential integrity in a database. It ensures that data is always accurate, consistent, and free of orphaned records.

SYNTAX:

```
DELETE FROM child_table
WHERE child_table.foreign_key_column IN (
  SELECT parent_table.primary_key_column
  FROM parent_table
  WHERE [condition]
)
```

QUERY AND OUTPUT:

```
SQL> DELETE FROM orders
  2  WHERE orders.PersonID IN(
  3  SELECT Persons.PersonID
  4  FROM Persons
  5  WHERE PersonID=1);

1 row deleted.

SQL> SELECT * FROM orders;

   ORDERID ORDERNUMBER   PERSONID
---------- ----------- ----------
         2       47838          2
         3       47654          3
         4       85856          4
```

UPDATE COMMAND:

Updating data in a table with a foreign key constraint is similar to updating data in a regular table but with some additional considerations for maintaining referential integrity.

When updating a column that is part of a foreign key constraint, you must ensure that the new value exists in the referenced table, or update the referenced value as well. Otherwise, the update operation may violate the referential integrity constraint.

SYNTAX:

```
UPDATE child_table
SET child_table.foreign_key_column = new_value
WHERE child_table.foreign_key_column IN (
  SELECT parent_table.primary_key_column
  FROM parent_table
  WHERE [condition]
)
```

QUERY AND OUTPUT:

```
SQL> CREATE TABLE dob (
  2    person_id NUMBER PRIMARY KEY,
  3    dob DATE
  4  );

Table created.

SQL> desc dob;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 PERSON_ID                                 NOT NULL NUMBER
 DOB                                               DATE
```

```
SQL> CREATE TABLE people (
  2    person_id NUMBER PRIMARY KEY,
  3    name VARCHAR2(50),
  4    age NUMBER,
  5    CONSTRAINT fk_dob_person_id
  6      FOREIGN KEY (person_id)
  7      REFERENCES dob (person_id)
  8  );

Table created.

SQL> desc people;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 PERSON_ID                                 NOT NULL NUMBER
 NAME                                              VARCHAR2(50)
 AGE                                               NUMBER
```

```
SQL> SELECT * FROM dob;

 PERSON_ID DOB
---------- ---------
         1 15-JUN-90
         2 23-AUG-85
         3 01-MAR-93
         4 30-DEC-92
         5 07-APR-03
```

```
SQL> SELECT * FROM people;

 PERSON_ID NAME                                               AGE
---------- -------------------------------------------------- ----------
         1 Alice
         2 Bob
         3 Victoria
         4 Andy
         5 Charlie
```

```
SQL> UPDATE people p
  2  SET p.age = (
  3    SELECT EXTRACT(YEAR FROM sysdate) - EXTRACT(YEAR FROM d.dob)
  4    FROM dob d
  5    WHERE d.person_id = p.person_id
  6  ) ;

5 rows updated.

SQL> SELECT * FROM people;

 PERSON_ID NAME                                               AGE
---------- -------------------------------------------------- ----------
         1 Alice                                              33
         2 Bob                                                38
         3 Victoria                                           30
         4 Andy                                               31
         5 Charlie                                            20
```

DROP COMMAND:

In SQL, the DROP command is used to remove an existing database object, such as a table, index, or foreign key constraint.

When used with a foreign key constraint, the DROP command removes the constraint from the table that it was defined on. This can be useful when you need to modify the table or remove the constraint altogether.

SYNTAX:

```
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

QUERY AND OUTPUT:

```
SQL> ALTER TABLE orders
  2  DROP CONSTRAINTS Fk_Personorder;

Table altered.
```

RESULT:

Thus, the SQL query to create a set of tables, add foreign key constraints and incorporate referential integrity is verified and executed successfully.

EX NO: 03	QUERY THE DATABASE TABLE USING DIFFERENT 'WHERE' CLAUSE CONDITIONS AND ALSO IMPLEMENT AGGREGATE FUNCTION
DATE:	

AIM:

To query the database table using different 'where' clause conditions and also implement aggregate functions.

WHERE CLAUSE:

DEFINITION:

The WHERE clause is used to filter records. It is used to extract only those records that fulfil a specified condition.

SYNTAX:

```
SELECT column1, column2...
FROM table_name
WHERE condition;
```

QUERY AND OUTPUT USING 'WHERE' CLASS:

```
SQL> select *from customer;

CUSNO                CUSNAME              CUSADD
-------------------- -------------------- --------------------
1                    john                 chennai
2                    alan                 mumbai
3                    paul                 delhi
4                    akshay               kerala
5                    avinav               banglore

SQL> Select cusadd from customer where cusno=5;

CUSADD
--------------------
banglore
```

```
SQL> Select title from book where price between 300 and 500;

TITLE
--------------------
it ends with us
it starts with us
tom jones
```

```
SQL> select *from book;

TITLE                AUTHORNAME
-------------------- --------------------
it ends with us      colleen hoover
it start with us     colleen hoover
tom jones            Henry
volpone              ben jonson

SQL>  Select authorname from book where authorname like '_e%';

AUTHORNAME
--------------------
Henry
ben jonson
```

AGGREGATE FUNCTION:

SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value. It is also used to summarize the data.

Consider the below table:

```
SQL> select *from book;

TITLE                PRICE
-------------------- --------------------
it ends with us      450
it starts with us    450
tom jones            320
volpone              649
```

AVERAGE:

DEFINITION:

The AVG function is used to calculate the average value of the numeric type. AVG function returns the average of all non-Null values.

SYNTAX:

```
AVG()

or

AVG( [ALL|DISTINCT] expression )
```

QUERY AND OUTPUT:

```
SQL> Select avg(price)"Average price" from book;

Average price
-------------
       467.25
```

MINIMUM:

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

SYNTAX:

```
MIN()
or
MIN( [ALL|DISTINCT] expression )
```

QUERY AND OUTPUT:

```
SQL> Select min(price) "Minimum price" from book;

Minimum price
--------------------
320
```

MAXIMUM:

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.

SYNTAX:

```
MAX()

or

MAX( [ALL|DISTINCT]
```

QUERY AND OUTPUT:

```
SQL> Select max(price) "Maximum price" from book;

Maximum price
--------------------
649
```

SUM:

Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.

SYNTAX:

```
SUM()
or
SUM( [ALL|DISTINCT]
```

QUERY AND OUTPUT:

```
SQL>  Select sum(price) as total from book;

     TOTAL
----------
      1869
```

COUNT:

COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.

SYNTAX:

```
COUNT(*)
or
COUNT( [ALL|DISTINCT]
expression )
```

QUERY AND OUTPUT:

```
SQL> Select count(title)" no of books" from book;

 no of books
------------
           4
```

SUB QUERIES USING WHERE CLAUSE AND AGGREGATE FUNCTION:

A subquery can also be found in the SELECT clause. These are generally used when you wish to retrieve a calculation using an aggregate function such as the SUM, COUNT, MIN, or MAX function, but you do not want the aggregate function to apply to the main query. A subquery can also be found in the SELECT clause. These are generally used when you wish to retrieve a calculation using an aggregate function such as the SUM, COUNT, MIN, or MAX function, but you do not want the aggregate function to apply to the main query.

Consider the below table:

```
SQL> select * from employee;

NAME                          AGE         ID     SALARY
-------------------- ---------- ---------- ----------
john                           23          1      25000
paul                           24          2      25000
akshay                         25          3      30000
avinav                         26          4      30000
```

1.Write a SQL query to display all column of employee table whose getting minimum salary.

QUERY AND OUTPUT:

```
SQL> select * from employee where
  2  salary=(select min(salary)from
  3  employee);

NAME                      AGE         ID     SALARY
-------------------- ---------- ---------- ----------
john                       23          1      25000
paul                       24          2      25000

SQL>
```

2. Write a SQL query selects the id and salary columns from the employee table where the salary is equal to the minimum salary for each id in the same table

QUERY AND OUTPUT:

```
SQL> select e.id,e.salary from employee
  2  e where salary in
  3  (select min(e.salary)from employee
  4  e group by e.id);

        ID     SALARY
---------- ----------
         1      25000
         2      25000
         3      30000
         4      30000
```

3. Write a SQL query retrieves the id, age, and salary columns from the employee table where both the age and salary values match the corresponding values of a specific employee with id 13.

QUERY AND OUTPUT:

```
SQL> select e.id,e.age,e.salary from
  2  employee e where (e.age,e.salary)
  3  in(select e.age,e.salary from
  4  employee e where e.id=3);

        ID        AGE     SALARY
---------- ---------- ----------
         3         25      30000
```

4.write a SQL query to retrieve the name column from the employee table and the age column from the subquery b which calculates the average age for each employee based on their id.

QUERY AND OUTPUT:

```
SQL> select a.name,b.age from employee a,
  2  (select id,avg(age)as age from employee
  3  group by id)b where b.id=a.id;

NAME                        AGE
-------------------- ----------
john                         23
paul                         24
avinav                       26
akshay                       25
```

Consider the below table:

```
SQL> select * from report;

        ID NAME
---------- --------------------
         1 akshay
         2 allan
         3 avinav
         4 john
         5 george
```

5. Write a SQL query retrieves the Id, name, and the number of employees with the same age as each employee in the report table.

QUERY AND OUTPUT:

```
SQL> select id,name,(select count(age)
  2  as age from employee a where a.id
  3  =b.id group by id)sameage from report b;

        ID NAME                 SAMEAGE
---------- -------------------- ----------
         1 akshay                      1
         2 allan                       1
         3 avinav                      1
         4 john                        1
         5 george
```

RESULT:

Thus, the query using different 'where' clause condition and aggregate function was verified and executed successfully.

EX NO:04	QUERY THE DATABASE TABLE AND EXPLORE SUB QUERIES AND SIMPLE JOIN OPERATION
DATE:	

AIM:

To query the database table and explore sub queries and simple join operation.

SUB QUERIES:

A Sub query or Inner query or a nested query is a query within another SQL query and embedded within clauses, most commonly in the WHERE clause. It is used to return data from a table, and this data will be used in the main query as a condition to further restrict the data to be retrieved.

Sub queries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <,>,>=, <=, IN BETWEEN,etc.

Consider the Employee table having the following records:

```
SQL> select * from employee;

COMPNAME             EMPNAME              EMPAGE      EMPID     SALARY
-------------------- -------------- ---------- ---------- ----------
GOOGLE               James                  35        111      50000
AMAZON               Helen                  28        121      48000
ZOHO                 Sofia                  39        131      65000
TCS                  Mathew                 27        141      54000
DELL                 Alice                  34        151      48000
```

Sub query using WHERE clause:

The WHERE clause specifies criteria that field values must meet to the records that contain the values to be included in the query results.

SYNTAX:

```
SELECT column1,column2,….
FROM table_name
WHERE column_name
    (SELECT column_name
     FROM table_name
     WHERE condition);
```

QUERY AND OUTPUT:

```
SQL> SELECT * from employee where compname in
  2  (select compname from employee where salary=48000);

COMPNAME   EMPNAME              EMPAGE      EMPID     SALARY
---------- ---------------- ---------- ---------- ----------
AMAZON     Helen                    28        121      48000
DELL       Alice                    34        151      48000
```

Sub query using FROM clause:

A FROM clause is the source of row set to be operated upon in a Data Manipulation Language (DML) statement.

SYNTAX:

```
SELECT column1, column2,…
FROM (SELECT column_name
        FROM table_name
        WHERE condition);
```

QUERY AND OUTPUT:

```
SQL> SELECT empname,compname from
  2  (select * from employee where salary=48000);

EMPNAME          COMPNAME
---------------- ----------
Helen            AMAZON
Alice            DELL
```

Sub query using IN clause:

An IN clause is used in a sub query to specify a list of values that are compared to a column in the outer query.

SYNTAX:

```
SELECT column1,column2,…
FROM table_name
WHERE colum_name
IN condition;
```

QUERY AND OUTPUT:

```
SQL> SELECT empname,empage,salary from employee
  2  where compname IN('TCS','GOOGLE');

EMPNAME              EMPAGE     SALARY
--------------- ---------- ----------
James                    35      50000
Mathew                   27      68000
```

Sub query using NOT IN clause:

In a sub query NOT IN clause is used to exclude rows that match a set of values returned by the sub query. This is the opposite of IN clause ,which includes rows that match the specified values .

SYNTAX:

```
SELECT column1,column2,…
FROM tale_name
WHERE column_name
NOT IN condition;
```

QUERY AND OUTPUT:

```
SQL> SELECT empname,empage,salary from employee
  2  where compname NOT IN('TCS','GOOGLE');

EMPNAME              EMPAGE     SALARY
--------------- ---------- ----------
Helen                    28      48000
Sofia                    39      54000
Alice                    34      48000
```

Sub query using GROUP BY command:

In the sub query, the GROUP BY command is used to group rows by one or more columns and perform an aggregate function on each group.

SYNTAX:

```
SELECT column_name, aggregate_function(column_name)
AS subquery_alias
FROM table_name
GROUP BY column-name;
```

QUERY AND OUTPUT:

```
SQL> SELECT compname,sum(salary)"Total Salary"
  2  from employee GROUP BY compname;

COMPNAME   Total Salary
---------- ------------
ZOHO              54000
GOOGLE            50000
DELL              48000
AMAZON            48000
TCS               68000
```

Sub query using ORDER BY command:

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 column_name DESC;
```

QUERY AND OUTPUT:

```
SQL> SELECT empname,compname,salary from employee
  2  ORDER BY salary desc;

EMPNAME          COMPNAME       SALARY
---------------- ---------- ----------
Mathew           TCS             68000
Sofia            ZOHO            54000
James            GOOGLE          50000
Alice            DELL            48000
Helen            AMAZON          48000
```

JOIN OPERATIONS:

In SQL, a join operation is used to combine rows from two or more tables based on a related column. The most common type of join operation is the INNER JOIN, which returns only the rows that have matching values in both tables.

Consider the Company table having the following records:

```
SQL> select * from company;

COMPNAME             EMPNAME           EMPAGE      EMPID     SALARY
-------------------- -------------- ---------- ---------- ----------
microsoft            William                34        131      56000
yahoo                Smith                  29        211      67000
saffron              Danny                  42        311      86000
ibm                  Riya                   28        411      56000
```

JOIN USING CLAUSE:

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

SYNTAX:

```
SELECT column_name
FROM table1
JOIN table2 using(column_name);
```

QUERY AND OUTPUT:

```
SQL> SELECT a.empname,b.empage,b.salary from company a
  2  join employee b USING(empid);

EMPNAME            EMPAGE     SALARY
-------------- ---------- ----------
William                39      54000
```

Join using ON clause:

In SQL, the ON clause can be used to join columns that have different names. Use the ON clause to specify conditions or specify columns to join. The join condition is separated from other search conditions.

SYNTAX:

```
SELECT column_name(s)
FROM table1
JOIN table2
ON table1.column_name = table2.column_name;
```

QUERY AND OUTPUT:

```
SQL> SELECT b.empname,b.compname,a.salary from company a
  2  join employee b ON(a.empid=b.empid);

EMPNAME          COMPNAME       SALARY
---------------- ---------- ----------
Sofia            ZOHO            56000
```

JOIN USING INNER JOIN:

Inner join is a type of join used in SQL to retrieve data from two or more tables based on a common column or columns between them. The inner join returns only the rows from both tables that meet the specified join conditions.

SYNTAX:

```
SELECT column_name(s)
FROM table1
IINER JOIN table2
ON table1.column_name = table2.column_name;
```

QUERY AND OUTPUT:

```
SQL> SELECT a.empname,b.salary,b.compname from company a
  2  INNER JOIN employee b USING(empid);

EMPNAME           SALARY COMPNAME
--------------- ---------- ----------
William            54000 ZOHO
```

RESULT:

Thus query the database to explore sub queries and simple join operation was executed successfully and the output was verified.

EX NO:05	QUERY THE DATABASE TABLES AND EXPLORE NATURAL, EQUI AND OUTER JOINS
DATE:	

AIM:

To query the database tables and explore natural, equi and outer joins.

Consider the report table having the following records:

```
SQL> select *from report;

        ID NAME
---------- ----------
         1 Dhoni
         2 Virat
         3 Rohit
         4 Hardik
         5 Raina
```

Consider the employee table having the following records:

```
SQL> select *from employee;

        ID        AGE     SALARY
---------- ---------- ----------
         2         34     150000
         3         37     170000
         4         32     140000
         5         35     200000
         6         30     125000
```

INNER JOIN:

An Inner Join retrieves the matching records, in other words it retrieves all the rows where there is at least one match in the tables.

SYNTAX:

SELECT *FROM table_name1 **INNER JOIN** table_name_2 using (common_column_name1);

QUERY AND OUTPUT:

```
SQL> select a.name,b.age,b.salary from report
  2  inner join employee b using(id);

NAME                     AGE     SALARY
-------------- ---------- ----------
Virat                     34     150000
Rohit                     37     170000
Hardik                    32     140000
Raina                     35     200000
```

OUTER JOIN:

The records that don't match will be retrieved by the Outer join. It is of the following three types:

1.Left Outer Join

2.Right OuterJoin

3.Full Outer Join

LEFT OUTER JOIN:

A Left outer join retrieves all records from the left-hand side of the table with all the matched records.

SYNTAX:

SELECT *FROM table_name_1 **LEFT OUTER JOIN** table_name_2 **ON(**condition_using_common_column_name1);

QUERY AND OUTPUT:

```
SQL> select a.name,a.id,b.age,b.salary from report a
  2  left outer join employee b
  3  on(a.id=b.id);

NAME                      ID        AGE     SALARY
-------------- ---------- ---------- ----------
Virat                      2         34     150000
Rohit                      3         37     170000
Hardik                     4         32     140000
Raina                      5         35     200000
Dhoni                      1
```

RIGHT OUTER JOIN:

A Right Outer Join retrieves the records from the right hand side columns.

SYNTAX:

SELECT *FROM table_name_1 **RIGHT OUTER JOIN** table_name_2 **ON(**condition_using_common_column_name1);

QUERY AND OUTPUT:

```
SQL> select a.name,a.id,b.age,b.salary from report a
  2  right outer join employee b
  3  on(a.id=b.id);

NAME                      ID        AGE     SALARY
-------------- ---------- ---------- ----------
Virat                      2         34     150000
Rohit                      3         37     170000
Hardik                     4         32     140000
Raina                      5         35     200000
                                     30     125000
```

NON EQUI JOIN:

A Non-Equi join is based on a condition using an operator other than equal to "=".

SYNTAX:

SELECT *FROM table_name_1 , table_name_2
WHERE Column_name1 **OPERATOR** column_name2;

OUERY AND OUTPUT:

```
SQL> select a.name,a.id,b.age,b.salary from report a,
  2  employee b where a.id>b.id;

NAME                      ID        AGE     SALARY
-------------- ---------- ---------- ----------
Rohit                      3         34     150000
Hardik                     4         34     150000
Raina                      5         34     150000
Hardik                     4         37     170000
Raina                      5         37     170000
Raina                      5         32     140000

6 rows selected.
```

SELF JOIN:

When a table is joined to itself only then that condition is called self-join.

SYNTAX:

SELECT *FROM table_name_1, table_name_2 **WHERE** (condition_using_common_column_name);

QUERY AND OUTPUT:

```
SQL> select a.name,a.id,b.age,b.salary from report a,
  2  employee b where a.id>b.id;

NAME                    ID        AGE     SALARY
--------------- ---------- ---------- ----------
Rohit                    3         34     150000
Hardik                   4         34     150000
Raina                    5         34     150000
Hardik                   4         37     170000
Raina                    5         37     170000
Raina                    5         32     140000

6 rows selected.
```

NATURAL JOIN:

A natural join is just like an equi-join since it compares the common columns of both tables.

SYNTAX:

SELECT *FROM table_name_1 **NATURAL JOIN** table_name_2;

QUERY AND OUTPUT:

```
SQL> select *from report natural join employee;

        ID NAME                 AGE     SALARY
---------- --------------- ---------- ----------
         2 Virat                   34     150000
         3 Rohit                   37     170000
         4 Hardik                  32     140000
         5 Raina                   35     200000
```

CARTESIAN OR CROSS JOIN:

The CARTESIAN JOIN or CROSS JOIN returns the Cartesian product of the sets of records from two or more joined tables.

SYNTAX:

SELECT *FROM table_name_1

CROSS JOIN table_name_2;

OUERY AND OUTPUT:

```
SQL> select *from report cross join employee;

        ID NAME                       ID        AGE     SALARY
---------- --------------- ---------- ---------- ----------
         1 Dhoni                    2         34     150000
         2 Virat                    2         34     150000
         3 Rohit                    2         34     150000
         4 Hardik                   2         34     150000
         5 Raina                    2         34     150000
         1 Dhoni                    3         37     170000
         2 Virat                    3         37     170000
         3 Rohit                    3         37     170000
         4 Hardik                   3         37     170000
         5 Raina                    3         37     170000
         1 Dhoni                    4         32     140000

        ID NAME                       ID        AGE     SALARY
---------- --------------- ---------- ---------- ----------
         2 Virat                    4         32     140000
         3 Rohit                    4         32     140000
         4 Hardik                   4         32     140000
         5 Raina                    4         32     140000
         1 Dhoni                    5         35     200000
         2 Virat                    5         35     200000
         3 Rohit                    5         35     200000
         4 Hardik                   5         35     200000
         5 Raina                    5         35     200000
         1 Dhoni                    6         30     125000
         2 Virat                    6         30     125000

        ID NAME                       ID        AGE     SALARY
---------- --------------- ---------- ---------- ----------
         3 Rohit                    6         30     125000
         4 Hardik                   6         30     125000
         5 Raina                    6         30     125000

25 rows selected.
```

UNION ALL JOINS:

In MySQL FULL JOIN doesn’t work instead of that we using UNION ALL command for join.

SYNTAX:

SELECT *FROM table_name_1 **LEFT OUTER JOIN** table_name_2
ON(condition_using_common_column_name1)**UNION ALL**
SELECT *FROM table_name_1 **RIGHT OUTER JOIN** table_name_2
ON(condition_using_common_name1);

QUERY AND OUTPUT:

```
SQL> select *from report a left outer join employee  b
  2  on a.id=b.id union all
  3  select *from report a right outer join employee b
  4  on a.id=b.id;

QCSJ_C000000000300000 NAME            QCSJ_C000000000300001        AGE
--------------------- --------------- --------------------- ----------
    SALARY
----------
                    2 Virat                                 2         34
    150000

                    3 Rohit                                 3         37
    170000

                    4 Hardik                                4         32
    140000


QCSJ_C000000000300000 NAME            QCSJ_C000000000300001        AGE
--------------------- --------------- --------------------- ----------
    SALARY
----------
                    5 Raina                                 5         35
    200000

                    1 Dhoni


                    2 Virat                                 2         34
    150000
QCSJ_C000000000300000 NAME            QCSJ_C000000000300001        AGE
--------------------- --------------- --------------------- ----------
    SALARY
----------
                    3 Rohit                                 3         37
    170000

                    4 Hardik                                4         32
    140000

                    5 Raina                                 5         35
    200000


QCSJ_C000000000300000 NAME            QCSJ_C000000000300001        AGE
--------------------- --------------- --------------------- ----------
    SALARY
----------
                                                            6         30
    125000


10 rows selected.

SQL>
```

RESULT:

Thus, the database table querying natural, equi and outer joins was executed successfully and the output is verified.

EX NO: 06	WRITE USER DEFINED FUNCTIONS AND STORED PROCEDURES IN SQL
DATE:	

AIM:

To create table and execute user-defined function and stored procedure in SQL

PROCEDURE:

A procedure is a named PL/SQL block which perform one or more specific task. It does not return a value directly; mainly used to perform an action. They can be used for validation, access control, or to reduce network traffic between client and DBMS server. A procedure is created with the CREATE OR REPLACE PROCEDURE statement.

SYNTAX:

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

Consider the customer table having the following records:

```
SQL> select * from customer;

NAME       GENDER     ADDRESS
---------- ---------- ----------
ajay       male       ahmedabad
akash      male       delhi
alan       male       mumbai
alex       male       kota
ramesh     male       bhopal
```

PROCEDURE FOR INSERTING:

Suppose we want to insert a record in the customer table. The following code is used to create the procedure named insert customer.

QUERY AND OUTPUT:

```
SQL> create or replace procedure insertcustomer(
  2  p_name customer.name%type,
  3  p_gender customer.gender%type,
  4  p_address customer.address%type)
  5  is
  6  begin
  7  insert into customer(name,gender,address)
  8  values(p_name,p_gender,p_address);
  9  commit;
 10  end;
 11  /

Procedure created.

SQL> execute insertcustomer('varun','male','delhi')

PL/SQL procedure successfully completed.

SQL> select * from customer;

NAME       GENDER     ADDRESS
---------- ---------- ----------
ajay       male       ahmedabad
akash      male       delhi
alan       male       mumbai
alex       male       kota
ramesh     male       bhopal
varun      male       delhi
```

PROCEDURE FOR UPDATING VALUES:

Suppose we want to update a record in the customer table. The following code is used to create the procedure named update customer.

QUERY AND OUTPUT:

```
SQL> create or replace procedure updatecustomer(
  2  p_name in customer.name%type,
  3  p_gender in customer.gender%type,
  4  p_address in customer.address%type)
  5  is
  6  begin
  7  update customer set gender = p_gender,
  8  address = p_address where name = p_name;
  9  commit;
 10  end;
 11  /

Procedure created.

SQL> execute updatecustomer('varun','male','agra')

PL/SQL procedure successfully completed.

SQL> select * from customer;

NAME       GENDER     ADDRESS
---------- ---------- ----------
ajay       male       ahmedabad
akash      male       delhi
alan       male       mumbai
alex       male       kota
ramesh     male       bhopal
varun      male       agra
```

PROCEDURE FOR DELETING VALUES:

Suppose we want to delete a record in the customer table. The following code is used to create the procedure named delete customer.

QUERY AND OUTPUT:

```
SQL> create or replace procedure deletecustomer(
  2  p_name in customer.name%type)
  3  is
  4  begin
  5  delete from customer where name=p_name;
  6  end;
  7  /

Procedure created.

SQL> execute deletecustomer('alex')

PL/SQL procedure successfully completed.

SQL> select * from customer;

NAME       GENDER     ADDRESS
---------- ---------- ----------
ajay       male       ahmedabad
akash      male       delhi
alan       male       mumbai
ramesh     male       bhopal
varun      male       agra
```

PROCEDURE FOR SELECTING:

Suppose we want to select a record in the customer table. The following code is used to create the procedure named select customer.

QUERY AND OUTPUT:

```
SQL> set serveroutput on
SQL> create or replace procedure selectcustomer(
  2  p_name in customer.name%type)
  3  is
  4  p_address varchar(20);
  5  begin
  6  select address into p_address
  7  from customer where name=p_name;
  8  dbms_output.put_line('pname='||p_name);
  9  dbms_output.put_line('p_address='||p_address);
 10  end;
 11  /

Procedure created.

SQL> execute selectcustomer('alan')
pname=alan
p_address=mumbai

PL/SQL procedure successfully completed.
```

FUNCTION:

The function is a set of SQL statements that perform a specific task. Functions foster code reusability. If you have to repeatedly write large SQL scripts to perform the same task, you can create a function that performs that task. Next time instead of rewriting the SQL, you can simply call that function. A function accepts inputs in the form of parameters and returns a value.

SYNTAX:

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

FUNCTION PROGRAM 1:

Consider the customer table having the following records:

```
SQL> select * from customer;

NAME       GENDER     ADDRESS
---------- ---------- ----------
ajay       male       ahmedabad
akash      male       delhi
alan       male       mumbai
ramesh     male       bhopal
varun      male       agra
```

QUERY AND OUTPUT:

```
SQL> create or replace function totalcustomers
  2  return number is
  3  total number(2):=0;
  4  begin
  5  select count(*) into total
  6  from customer;
  7  return total;
  8  end;
  9  /

Function created.

SQL> select totalcustomers() from dual;

TOTALCUSTOMERS()
----------------
               5
```

FUNCTION PROGRAM 2:

Consider the student mark table having the following records:

```
SQL> select * from stmark;

NAME            MARK1      MARK2      MARK3      MARK4
---------- ---------- ---------- ---------- ----------
arun                94         87         92         55
anu                 99         92         95         75
ajay                89         72         91         85
asha                91         92         66         87
```

QUERY AND OUTPUT:

```
SQL> create or replace function pmark(sname varchar)
  2  return number
  3  is
  4  m1 number(3);
  5  m2 number(3);
  6  m3 number(3);
  7  m4 number(3);
  8  total number(3);
  9  precentage number(3);
 10  begin
 11  select mark1 into m1 from stmark where name=sname;
 12  select mark2 into m2 from stmark where name=sname;
 13  select mark3 into m3 from stmark where name=sname;
 14  select mark4 into m4 from stmark where name=sname;
 15  total:=m1+m2+m3+m4;
 16  precentage:=total/4;
 17  return precentage;
 18  end;
 19  /

Function created.

SQL> select pmark('anu') from dual;

PMARK('ANU')
------------
          90
```

RESULT:

Thus, the user defined function and stored procedure in SQL was executed successfully.

EX NO:07	EXECUTE COMPLEX TRANSACTIONS AND REALIZE DCL AND TCL COMMANDS
DATE:	

AIM:

To execute complex transactions and realize DCL and TCL commands.

TCL:

TCL stands for Transaction Control Language in SQL. Transaction Control Language (TCL) is a set of special commands that deal with the transactions within the database.

COMMIT:

Commit is a transaction control language in SQL. It lets a user permanently save all the changes made in the transaction of a database or table.

SYNTAX:

```
COMMIT;
```

ROLLBACK:

ROLLBACK is the SQL command that is used for reverting changes performed by a transaction. When a ROLLBACK command is issued it reverts all the changes since last COMMIT or ROLLBACK.

SYNTAX:

```
ROLLBACK;
```

QUERY AND OUTPUT:

COMMIT WITH ROLLBACK:

```
SQL> commit;

Commit complete.

SQL> rollback;

Rollback complete.

SQL> select *from emp;

NAME                    AGE     SALARY
--------------- ---------- ----------
Virat                    33      50000
Kane                     34      45000
Dhoni                    39      55000
```

ROLLBACK WITHOUT COMMIT:

```
SQL> insert into emp values('Rahul',30,40000);

1 row created.

SQL> select *from emp;

NAME                AGE     SALARY
--------------- ---------- ----------
Virat                33      50000
Kane                 34      45000
Dhoni                39      55000
Rahul                30      40000

SQL> rollback;

Rollback complete.

SQL> select *from emp;

NAME                AGE     SALARY
--------------- ---------- ----------
Virat                33      50000
Kane                 34      45000
Dhoni                39      55000
```

SAVEPOINT:

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

SYNTAX:

Creation of **SAVEPOINT**:

SAVEPOINT SAVEPOINT_NAME;

Rolling back to a **SAVEPOINT**:

ROLLBACK TO SAVEPOINT_NAME;

QUERY AND OUTPUT:

```
SQL> savepoint a;

Savepoint created.

SQL> insert into student values(1,'Messi',30);

1 row created.

SQL> insert into student values(2,'Ronaldo',32);

1 row created.

SQL> savepoint b;

Savepoint created.

SQL> insert into student values(3,'Pele',28);

1 row created.

SQL> insert into student values(4,'Neymar',29);

1 row created.
```

```
SQL> rollback to b;

Rollback complete.

SQL> select *from student;

        ID NAME                 AGE
---------- -------------- ----------
         1 Messi                  30
         2 Ronaldo                32

SQL> rollback to a;

Rollback complete.

SQL> select *from student;

no rows selected
```

DCL:

Data Control Language (DCL) deals with the commands used in SQL that permit a user to access, modify or work on the different privileges in order to control the database. It allows the database owner to give access, revoke access, and change the given permissions as and when required.

GRANT:

GRANT command is a part of Data Controlling Language. GRANT command is used to grant SQL SELECT, UPDATE, INSERT, DELETE, and other privileges on tables or views.

SYNTAX:

GRANT privileges_names **ON** object **TO** user;

QUERY AND OUTPUT:

```
SQL> connect asten;
Enter password:
Connected.
SQL> grant select,insert,update on emp to aaron;

Grant succeeded.

SQL> connect aaron;
Enter password:
Connected.
SQL> select *from asten.emp;

NAME                AGE     SALARY
--------------- ---------- ----------
Virat                33      50000
Kane                 34      45000
Dhoni                39      55000
```

```
SQL> insert into asten.emp values('Sachin',44,60000);

1 row created.

SQL> select *from asten.emp;

NAME                      AGE      SALARY
--------------- ---------- ----------
Virat                      33       50000
Kane                       34       45000
Dhoni                      39       55000
Sachin                     44       60000
```

```
SQL> update asten.emp set age=20 where name='Kane';

1 row updated.

SQL> select *from asten.emp;

NAME                    AGE     SALARY
--------------- ---------- ----------
Virat                    33      50000
Kane                     20      45000
Dhoni                    39      55000
Sachin                   44      60000
```

REVOKE:

REVOKE command is used to withdraw or remove the permissions or privileges of a user on database objects set by the GRANT command. REVOKE command is the opposite of the command GRANT.

SYNTAX:

REVOKE privileges_names **ON** object **FROM** user;

QUERY AND OUTPUT:

```
SQL> connect asten;
Enter password:
Connected.
SQL> revoke update,select,insert on emp from aaron;

Revoke succeeded.

SQL> connect aaron;
Enter password:
Connected.
SQL> select *from asten.emp;
select *from asten.emp
                   *
ERROR at line 1:
ORA-00942: table or view does not exist
```

RESULT:

Thus, to execute complex transactions and realize DCL and TCL commands was executed and the output is verified.

EX NO: 08	WRITE SQL TRIGGERS FOR INSERT, DELETE, AND UPDATE OPERATIONS IN A DATABASE TABLE
DATE:	

AIM:

To build database and apply Triggers for insert, delete and update operation in a database table.

PL/SQL-Triggers:

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events: - A database manipulation (DML) statement (DELETE, INSERT or UPDATE) A database definition (DDL) statement (CREATE, ALTER, or DROP). A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

TYPES OF TRIGGERS:

The various types of triggers are as follows: -

- **Before Triggers:** These triggers are fired before the SQL statement trigger (INSERT, UPDATE, DELETE) is executed. The execution of the triggering SQL statement is stopped, depending on the various conditions to be fulfilled in the BEFORE trigger.
- **After Triggers:** These triggers are fired after the triggering SQL statement (INSERT, UPDATE, DELETE) is executed. The triggering SQL statement is executed first, followed by the code of the trigger.
- **ROW Trigger:** The triggers are fired for each and every record, which is inserted or updated or deleted from a table.
- **Statement Trigger:** The trigger is fired for each row of the DML operation, being performed on a table. We cannot access the column values for the records being inserted, updated, deleted on the table nor the individual records.

BENEFITS OF TRIGGERS:

- Triggers can be written for the following purposes: -
- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

SYNTAX:

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

CREATING THE TABLE:

Create table customer (id integer, name varchar (20), age integer, address varchar (30), salary integer);

```
SQL> select * from customer;

        ID NAME                      AGE ADDRESS                SALARY
---------- -------------------- ---------- -------------------- ----------
         1 Akshay                       20 Chennai                 80000
         2 Mathan                       19 Tuticorin               70000
         3 Ismail                       21 Mumbai                  85000
         4 Jenolan                      22 Pune                    82000
         5 Herwin                       23 Delhi                   86000
```

AFTER INSERT, UPDATE AND DELETE:

These triggers are fired after the triggering SQL statement (INSERT, UPDATE, DELETE) is executed. The triggering SQL statement is executed first, followed by the code of the trigger.

QUERY AND OUTPUT:

```
SQL> set serveroutput on;
SQL> create or replace Trigger Trigger1
  2  after update or insert or delete on customer
  3  for each row
  4  begin
  5  if updating then
  6  dbms_output.put_line('Data is updated');
  7  elsif inserting then
  8  dbms_output.put_line('Data is inserted');
  9  elsif deleting then
 10  dbms_output.put_line('Data is deleted');
 11  end if;
 12  end;
 13  /

Trigger created.
```

INSERTION:

```
SQL> insert into customer values(6,'Jenish',25,'Gujarat',70000);
Data is inserted

1 row created.

SQL> select * from customer;

        ID NAME                      AGE ADDRESS                SALARY
---------- -------------------- ---------- -------------------- ----------
         1 Akshay                       20 Chennai                 80000
         2 Mathan                       19 Tuticorin               70000
         3 Ismail                       21 Mumbai                  85000
         4 Jenolan                      22 Pune                    82000
         5 Herwin                       23 Delhi                   86000
         6 Jenish                       25 Gujarat                 70000

6 rows selected.
```

UPDATION:

```
SQL> update customer set age=24 where name='Ismail';
Data is updated

1 row updated.

SQL> select * from customer;

        ID NAME                      AGE ADDRESS                SALARY
---------- -------------------- ---------- -------------------- ----------
         1 Akshay                       20 Chennai                 80000
         2 Mathan                       19 Tuticorin               70000
         3 Ismail                       24 Mumbai                  85000
         4 Jenolan                      22 Pune                    82000
         5 Herwin                       23 Delhi                   86000
```

DELETION:

```
SQL> delete from customer where name='Jenish';
Data is deleted

1 row deleted.

SQL> select * from customer;

        ID NAME                      AGE ADDRESS                SALARY
---------- -------------------- ---------- -------------------- ----------
         1 Akshay                       20 Chennai                 80000
         2 Mathan                       19 Tuticorin               70000
         3 Ismail                       21 Mumbai                  85000
         4 Jenolan                      22 Pune                    82000
         5 Herwin                       23 Delhi                   86000
```

BEFORE UPDATE:

These triggers are fired before the SQL statement trigger (INSERT, UPDATE, DELETE) is executed. The execution of the triggering SQL statement is stopped, depending on the various conditions to be fulfilled in the BEFORE trigger.

QUERY AND OUTPUT:

```
SQL> create or replace trigger display_salary_changes
  2  before delete or insert or update on customer
  3  for each row
  4  when(new.id>0)
  5  declare
  6  sal_diff number;
  7  begin
  8  sal_diff:=:new.salary - :old.salary;
  9  dbms_output.put_line('Old salary:'||:old.salary);
 10  dbms_output.put_line('New salary:'||:new.salary);
 11  dbms_output.put_line('Salary difference:'||sal_diff);
 12  end;
 13  /

Trigger created.
```

```
SQL> update customer set salary=89000 where name='Akshay';
Old salary:80000
New salary:89000
Salary difference:9000
Data is updated

1 row updated.

SQL> select * from customer;

        ID NAME                      AGE ADDRESS                SALARY
---------- -------------------- ---------- -------------------- ----------
         1 Akshay                       20 Chennai                 89000
         2 Mathan                       19 Tuticorin               70000
         3 Ismail                       24 Mumbai                  85000
         4 Jenolan                      22 Pune                    82000
         5 Herwin                       23 Delhi                   86000
```

AFTER UPDATE:

These triggers are fired after the triggering SQL statement (INSERT, UPDATE, DELETE) is executed. The triggering SQL statement is executed first, followed by the code of the trigger.

QUERY AND OUTPUT:

```
SQL> set serveroutput on;
SQL> create trigger trignew
  2  after insert or update of age on customer
  3  for each row
  4  begin
  5  if(:new.age<10)then
  6  raise_application_error(-20015,'Enter valid age');
  7  else
  8  dbms_output.put_line('Valid Age');
  9  end if;
 10  end;
 11  /

Trigger created.
```

```
SQL> update  customer set age=28 where name='Jenolan';
Old salary:82000
New salary:82000
Salary difference:0
Valid Age

1 row updated.

SQL> update  customer set age=9 where name='Jenolan';
Old salary:82000
New salary:82000
Salary difference:0
update  customer set age=9 where name='Jenolan'
        *
ERROR at line 1:
ORA-20015: Enter valid age
ORA-06512: at "AKSHAY.TRIGNEW", line 3
ORA-04088: error during execution of trigger 'AKSHAY.TRIGNEW'
```

```
SQL> select * from customer;

        ID NAME                      AGE ADDRESS                SALARY
---------- -------------------- ---------- -------------------- ----------
         1 Akshay                       20 Chennai                 89000
         2 Mathan                       19 Tuticorin               70000
         3 Ismail                       24 Mumbai                  85000
         4 Jenolan                      28 Pune                    82000
         5 Herwin                       23 Delhi                   86000
```

RESULT:

Thus, the queries to build database and apply triggers was written and executed successfully.

EX NO:09	CREATE VIEW AND INDEX FOR DATABASE TABLES WITH A LARGE NUMBER OF RECORDS
DATE:	

AIM:

To create view and index for database tables with a large number of records.

VIEW:

Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database.

CREATING VIEWS:

Database views are created using the **create view** statement. Views can be created from a single table, multiple tables or another view.

SYNTAX:

```
CREATE  VIEW view_name AS
SELECT  column1,column2,…
FROM   table_name
WHERE  [condition];
```

QUERY AND OUTPUT:

```
SQL> SELECT * from customer;

        ID NAME              AGE ADDRESS        SALARY
---------- ---------- ---------- ---------- ----------
         1 Ben                18 Bhopal          18000
         2 Ram                19 Goa             19000
         3 John               20 Salem           20000

SQL> CREATE VIEW customer_view AS
  2  SELECT name,age
  3  FROM customer;

View created.

SQL> SELECT * from customer_view;

NAME              AGE
---------- ----------
Ben                18
Ram                19
John               20
```

THE WITH CHECK OPTION:

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the with check option is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

SYNTAX:

```
CREATE  VIEW view_name AS
SELECT  column1,column2,…
FROM   table_name
WHERE  [condition];
```

QUERY AND OUTPUT:

```
SQL> SELECT * from cus;

        ID NAME                 AGE ADDRESS             SALARY
---------- ---------- ---------- ---------- ----------
         1 Ram                  19 Assam              18000
         2 Arun                 23 Bengal             20000
         3 Varun                24 Pune               22000

SQL> CREATE VIEW cus_view AS
  2  SELECT name,age,address
  3  FROM cus
  4  WHERE age>20;

View created.

SQL> SELECT * from cus_view;

NAME                AGE ADDRESS
---------- ---------- ----------
Arun                 23 Bengal
Varun                24 Pune
```

UPDATING A VIEW:

The sql UPDATE VIEW command can be used to modify the data of a view.

SYNTAX:

```
UPDATE  view_name
SET  column1,colmn2,..
WHERE  [condition];
```

QUERY AND OUTPUT:

```
SQL> UPDATE customer_view
  2  SET age=26
  3  WHERE name='Ben';

1 row updated.

SQL> SELECT * from customer_view;

NAME              AGE
---------- ----------
Ben                26
Ram                19
John               20
```

DELETING ROWS INTO A VIEW:

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

SYNTAX:

```
DELETE  FROM  view_name
WHERE  [condition];
```

BEFORE DELETING A ROW:

QUERY AND OUTPUT:

```
SQL> SELECT * from customer_view;

NAME              AGE
---------- ----------
Ben                26
Ram                19
John               20
```

AFTER DELETING A ROW:

```
SQL> SELECT * from customer_view;

NAME              AGE
---------- ----------
Ben                26
Ram                19
John               20

SQL> DELETE FROM customer_view
  2  WHERE age=26;

1 row deleted.

SQL> SELECT * from customer_view;

NAME              AGE
---------- ----------
Ram                19
John               20
```

DROPPING VIEWS:

DROP VIEW command is used to delete a view.

SYNTAX:

```
DROP VIEW view_name;
```

QUERY AND OUTPUT:

```
SQL> SELECT * from customer_view;

NAME              AGE
---------- ----------
Ram                19
John               20

SQL> DROP VIEW customer_view;

View dropped.

SQL> SELECT * from customer_view;
SELECT * from customer_view
              *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL> SELECT * from customer;

        ID NAME              AGE ADDRESS        SALARY
---------- ---------- ---------- ---------- ----------
         2 Ram                19 Goa             19000
         3 John               20 Salem           20000
```

INDEXES:

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. Simply put, 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 a book.

CREATE INDEX:

BASE TABLE:

```
SQL> SELECT * from customer;

        ID NAME              AGE ADDRESS        SALARY
---------- ---------- ---------- ---------- ----------
         1 Ram                26 Assam           25000
         2 Varun              19 Bhopal          19000
         3 John               20 Salem           16000
         4 Sam                21 Pune            21000
         5 Kavin              22 Bengal          18000
```

BEFORE CREATING AN INDEX:

QUERY AND OUTPUT:

```
SQL> SET autotrace ON explain;
SQL> SELECT * from customer
  2  WHERE salary > 10000;

        ID NAME              AGE ADDRESS        SALARY
---------- ---------- ---------- ---------- ----------
         1 Ram                26 Assam           25000
         2 Varun              19 Bhopal          19000
         3 John               20 Salem           16000
         4 Sam                21 Pune            21000
         5 Kavin              22 Bengal          18000


Execution Plan
----------------------------------------------------------
Plan hash value: 2844954298

--------------------------------------------------------------------------------
| Id  | Operation         | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |          |     5 |   265 |     2   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| CUSTOMER |     5 |   265 |     2   (0)| 00:00:01 |
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("SALARY">10000)

Note
-----
   - dynamic sampling used for this statement (level=2)
```

SYNTAX:

```
CREATE  INDEX  index_name
ON  table_name (column1,column2,..);
```

AFTER CREATING AN INDEX:

```
SQL> CREATE INDEX cus_index
  2  ON customer(salary);

Index created.
```

QUERY AND OUTPUT:

```
SQL> SELECT * from customer
  2  WHERE salary > 10000;

        ID NAME              AGE ADDRESS        SALARY
---------- ---------- ---------- ---------- ----------
         3 John               20 Salem           16000
         5 Kavin              22 Bengal          18000
         2 Varun              19 Bhopal          19000
         4 Sam                21 Pune            21000
         1 Ram                26 Assam           25000


Execution Plan
----------------------------------------------------------
Plan hash value: 3555992055

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

| Id  | Operation                   | Name      | Rows  | Bytes | Cost (%CPU)| T
ime     |

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

|   0 | SELECT STATEMENT            |           |     5 |   265 |     1   (0)| 0
0:00:01 |

|   1 |  TABLE ACCESS BY INDEX ROWID| CUSTOMER  |     5 |   265 |     1   (0)| 0
0:00:01 |

|*  2 |   INDEX RANGE SCAN          | CUS_INDEX |     1 |       |     1   (0)| 0
0:00:01 |

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


Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("SALARY">10000)

Note
-----
   - dynamic sampling used for this statement (level=2)
```

RESULT:

Thus, the query to creating view and index for database tables with a large number of records was executed successfully.

EX NO: 10	CREATE AN XML DATABASE AND VALIDATE IT USING XML SCHEMA.
DATE:	

AIM:

To create a xml database and validate using xml schema.

PROCEDURE FOR VALIDATING USING XML SCHEMA:

Step 1: Open notepad++.

Step 2: Open new file.

Step 3: Write your XML file.

Step 4: Save file with .xml extension.

Step 5: Open another new file and write schema file.

Step 6: Save schema file with .xsd extension.

Step 7: In the Menu bar click on Plugins. Open Plugins menu open plugins admin.

Step 8: In Plugins Admin search and install XML Tools.

Step 9: Click Plugins and under XML Tools Click Validate now.

Step 10: Enter the location of the schema file.

Step 11: Click 'OK'.

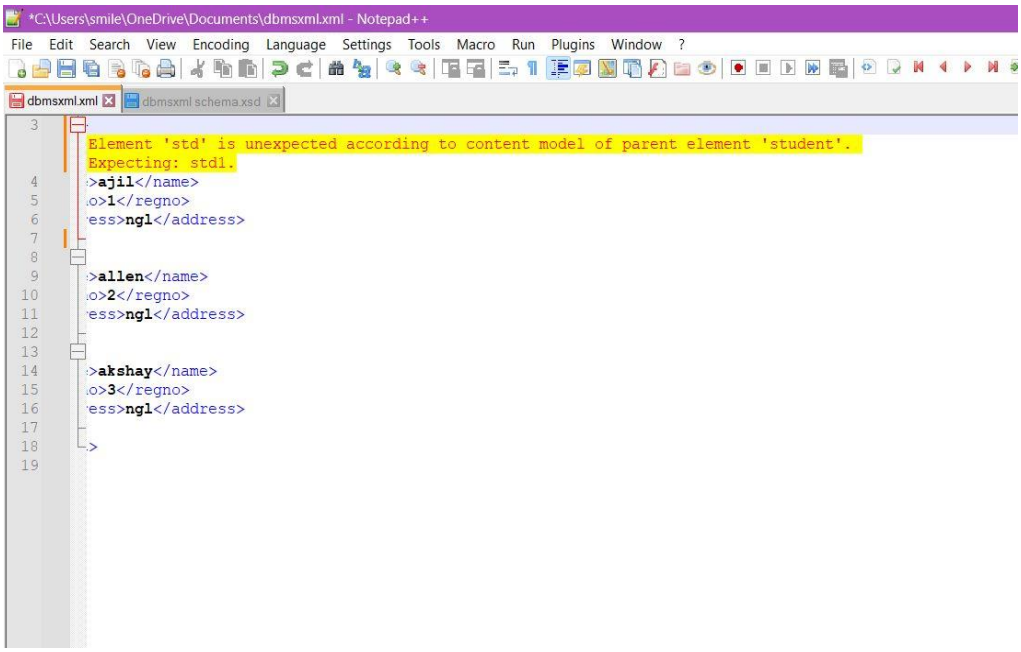
CREATING XML DATABASE:

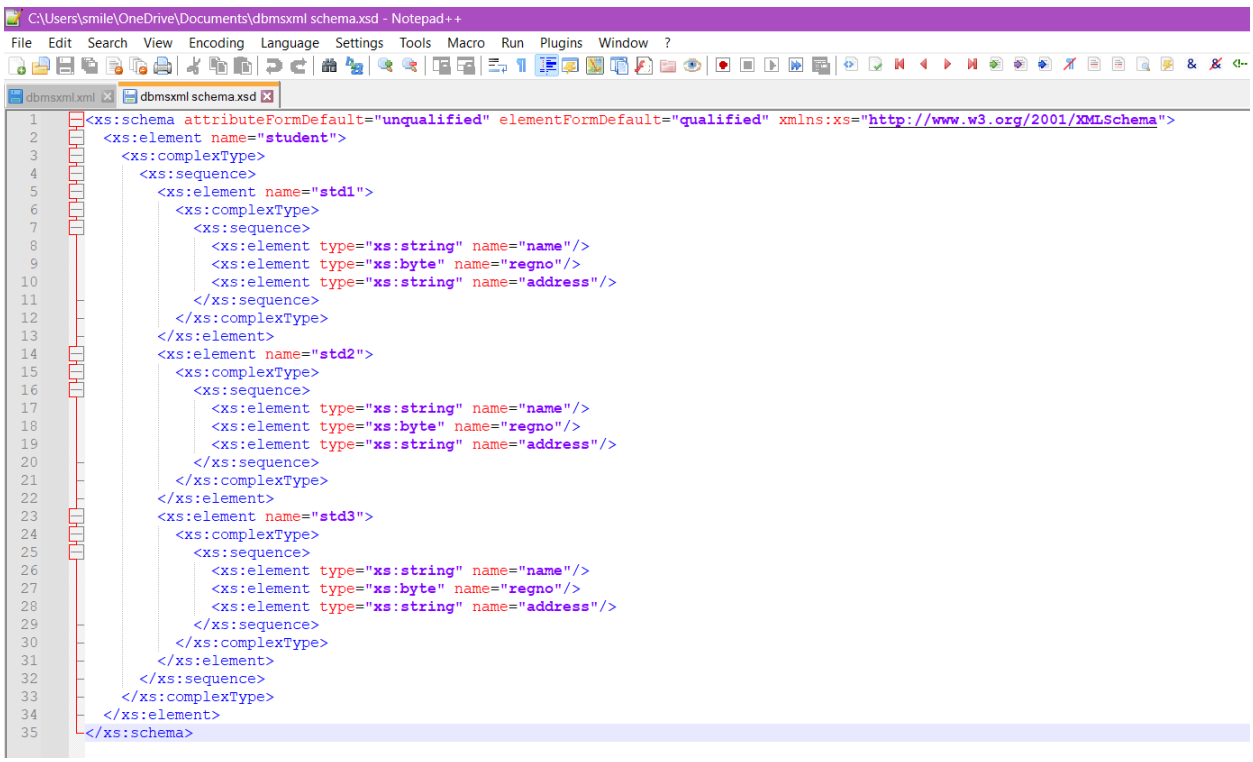
```
<?xml version="1.0" encoding="UTF-8"?>
<student>
        <std1>
            <name>ajil</name>
            <regno>1</regno>
            <address>ngl</address>
        </std1>
        <std2>
            <name>allen</name>
            <regno>2</regno>
            <address>konam</address>
        </std2>
        <std3>
            <name>abu</name>
            <regno>3</regno>
            <address>Tvm</address>
        </std3>
</student>
```

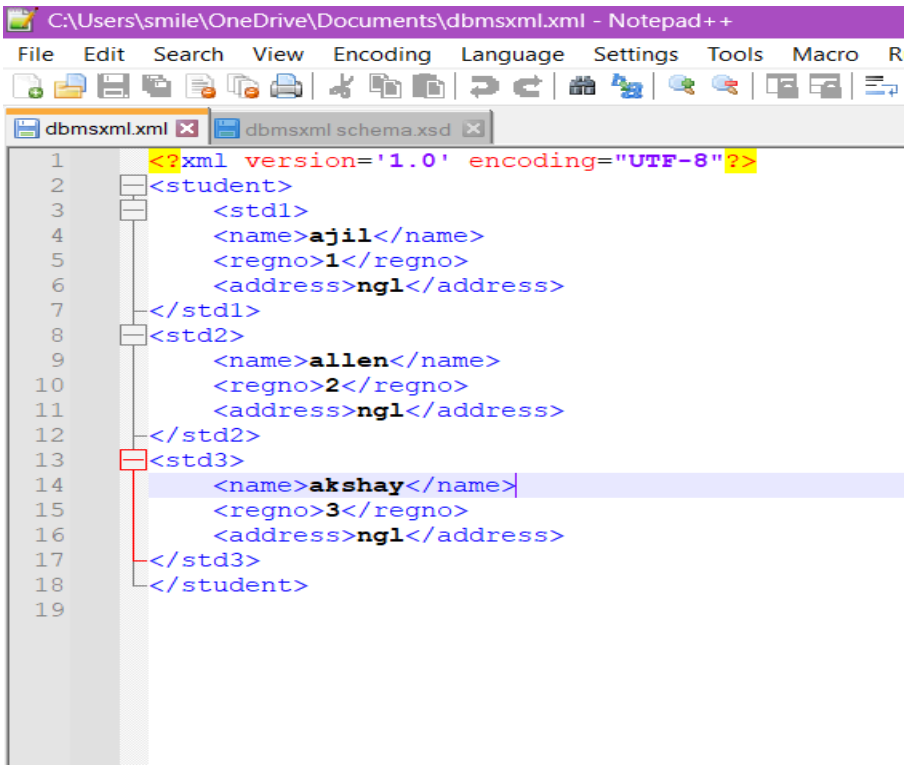
XML SCHEMA:

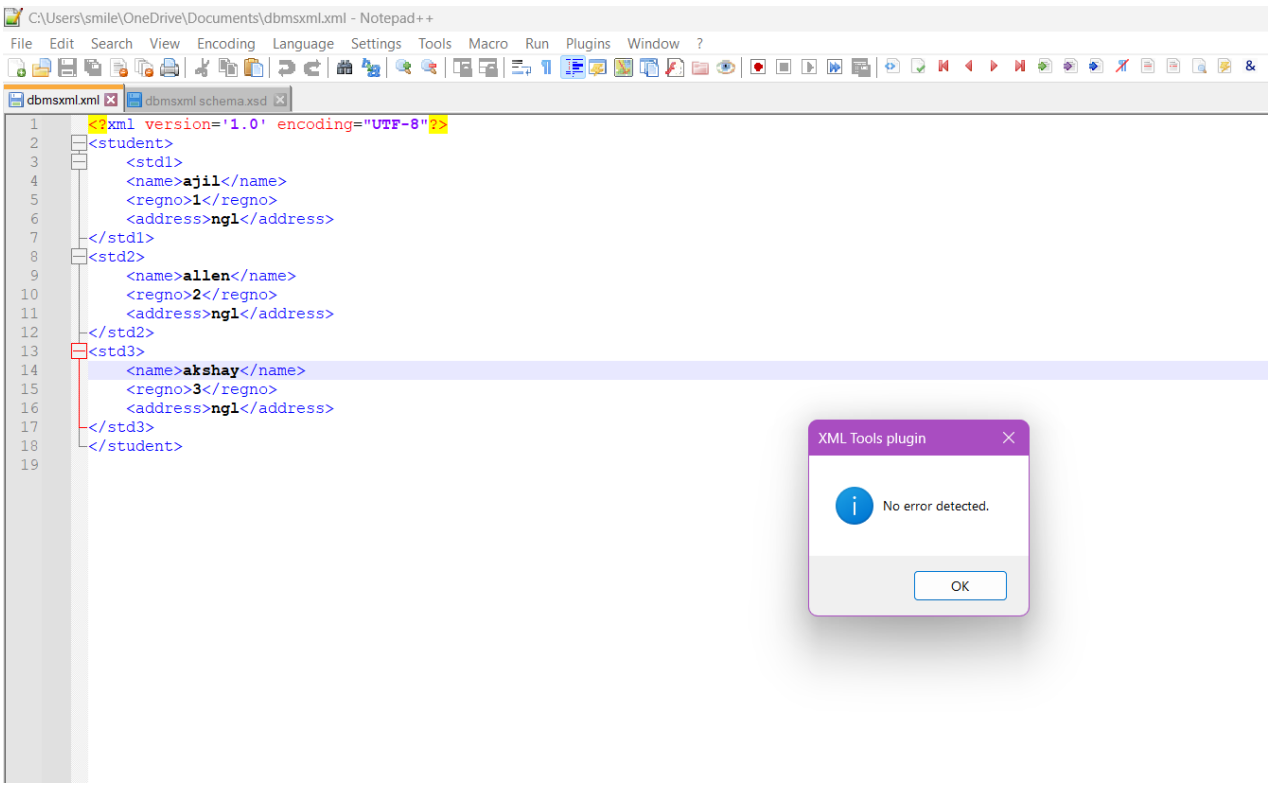
```
<xs:schemaattributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="student">
<xs:complexType>
<xs:sequence>
<xs:element name="std1">
<xs:complexType>
<xs:sequence>
<xs:element name="name" type="xs:string" />
<xs:element name="regno" type="xs:unsignedByte" />
<xs:element name="address" type="xs:string" />
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="std2">
<xs:complexType>
<xs:sequence>
<xs:element name="name" type="xs:string" />
<xs:element name="regno" type="xs:unsignedByte" />
<xs:element name="address" type="xs:string" />
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="std3">
<xs:complexType>
<xs:sequence>
<xs:element name="name" type="xs:string" />
<xs:element name="regno" type="xs:unsignedByte" />
<xs:element name="address" type="xs:string" />
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

QUERY AND OUTPUT:






C:\Users\smile\OneDrive\Documents\dbmsxml.xml - Notepad++
File Edit Search View Encoding Language Settings Tools Macro R
dbmsxml.xml
dbmsxml schema.xsd
<?xml version='1.0' encoding="UTF-8"?>
<student>
<std1>
<name>ajil</name>
<regno>1</regno>
<address>ngl</address>
</std1>
<std2>
<name>allen</name>
<regno>2</regno>
<address>ngl</address>
</std2>
<std3>
<name>akshay</name>
<regno>3</regno>
<address>ngl</address>
</std3>
</student>


C:\Users\smile\OneDrive\Documents\dbmsxml.xml - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
dbmsxml.xml
dbmsxml schema.xsd
<?xml version='1.0' encoding="UTF-8"?>
<student>
<std1>
<name>ajil</name>
<regno>1</regno>
<address>ngl</address>
</std1>
<std2>
<name>allen</name>
<regno>2</regno>
<address>ngl</address>
</std2>
<std3>
<name>akshay</name>
<regno>3</regno>
<address>ngl</address>
</std3>
</student>
XML Tools plugin
No error detected.
OK

RESULT:

Thus, the SQL query to create a set of tables, add foreign key constraints and incorporate referential integrity is verified and executed successfully.

EX NO:11	CREATE DOCUMENT, COLUMN AND GRAPH BASED DATA USING NOSQL DATABASE TOOLS
DATE:	

AIM:

To create a document-based NoSQL database using MongoDB, with columns and graphs, and populate it with sample data.

NOSQL:

NOSQL (Not Only SQL) is a type of database that does not use the traditional relational model that has been used in traditional SQL databases. Instead, NoSQL databases use a non-relational approach for data storage and retrieval. They are designed to handle large amounts of unstructured, semi-structured, and structured data, including text, images, videos, and social media data.

NoSQL databases are often used in big data and real-time web applications, where high performance and scalability are essential. Some popular examples of NoSQL databases include MongoDB, Cassandra, Couchbase, and Apache HBase.

VIEW DATABASE:

In MongoDB, a view is a virtual collection that presents the results of a pre-defined aggregation pipeline as if it were a collection of documents. Views do not store data themselves but rather provide a read-only representation of the data in one or more underlying collections.

QUERY AND OUTPUT:

```
test> show dbs
admin   40.00 KiB
config  60.00 KiB
local   72.00 KiB
test>
```

CREATING DATABASE:

In the mongo shell, you can create a database with the help of the following command. This command actually switches you to the new database if the given name does not exist and if the given name exists, then it will switch you to the existing database. Now at this stage, if you use the show command to see the database list where you will find that your new database is not present in that database list because, in MongoDB, the database is actually created when you start entering data in that database.

SYNTAX:

```
use database_name
```

QUERY AND OUTPUT:

```
test> show dbs
admin   40.00 KiB
config  60.00 KiB
local   72.00 KiB
test> use CSE
switched to db CSE
CSE> show dbs
admin   56.00 KiB
config  72.00 KiB
local   72.00 KiB
CSE>
```

CREATING COLLECTION:

In MongoDB, a collection is a grouping of MongoDB documents, similar to a table in a relational database. To create a collection in MongoDB, you can use the db.createCollection() method.

SYNTAX:

```
db.createCollection(name, options)
```

QUERY AND OUTPUT:

```
mongosh> use CSE
switched to db CSE
CSE> db.createCollection("Student")
{ ok: 1 }
CSE>
```

INSERTMANY COMMAND:

insertMany is a method in MongoDB that allows you to insert multiple documents into a collection at once. The method takes an array of documents as its argument, and inserts each document as a separate document in the collection.

SYNTAX:

```
db.collection.insertMany(
  [ <document 1> , <document 2>, ... ],
  {
    writeConcern: <document>,
    ordered: <boolean>
  }
)
```

QUERY AND OUTPUT:

DOCUMENT VIEW:

```
CSE> db.Student.insertMany([
... {name:"John",age:25,city:"New York"},
... {name:"Mary",age:30,city:"Los Angeles"},
... {name:"Bob",age:35,city:"Chicago"},
... {name:"Jane",age:40,city:"Houston"}
... ])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("6458acce4653642c945d4f93"),
    '1': ObjectId("6458acce4653642c945d4f94"),
    '2': ObjectId("6458acce4653642c945d4f95"),
    '3': ObjectId("6458acce4653642c945d4f96")
  }
}
```

```
CSE> db.Student.find()
[
  {
    _id: ObjectId("6458acce4653642c945d4f93"),
    name: 'John',
    age: 25,
    city: 'New York'
  },
  {
    _id: ObjectId("6458acce4653642c945d4f94"),
    name: 'Mary',
    age: 30,
    city: 'Los Angeles'
  },
  {
    _id: ObjectId("6458acce4653642c945d4f95"),
    name: 'Bob',
    age: 35,
    city: 'Chicago'
  },
  {
    _id: ObjectId("6458acce4653642c945d4f96"),
    name: 'Jane',
    age: 40,
    city: 'Houston'
  }
]
CSE>
```

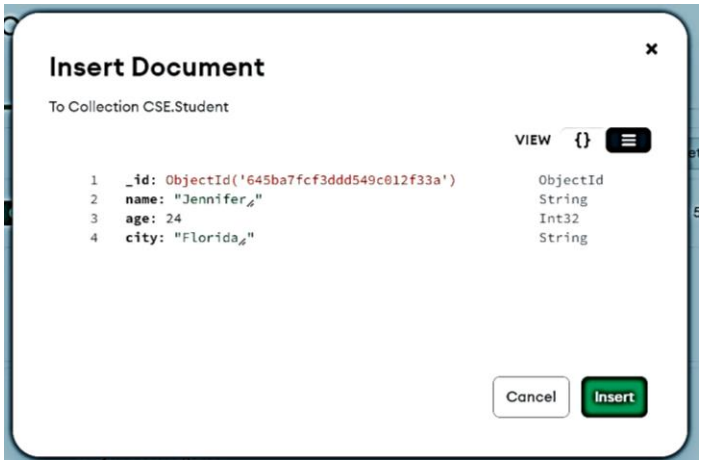
GRAPHICAL VIEW:

Step 1: Open MongoDB Compass Application.

Step 2: Click Connect on the displayed window.

Step 3: From the Databases select the database which has the collection you want to insert document.

Step 4: Click the collection -> ADD DATA -> Insert document. Add fields to the document and click Insert.



FINDING A SINGLE DOCUMENT:

In MongoDB, we can find a single document using the findOne() method, This method returns the first document that matches the given filter query expression.

SYNTAX:

```
db.collection_name.findOne ()
```

QUERY AND OUTPUT:

```
mangod> use CSE
switched to db CSE
CSE> db.Student.findOne()
{
  _id: ObjectId("6458acce4653642c945d4f93"),
  name: 'John',
  age: 25,
  city: 'New York'
}
CSE>
```

DISPLAYING DOCUMENTS IN A FORMATTED WAY:

In MongoDB, we can display documents of the specified collection in a well-formatted way using the pretty() method.

SYNTAX:

db.collection_name.find().pretty()

QUERY AND OUTPUT:

```
CSE> db.Student.find().pretty()
[
  {
    _id: ObjectId("6458acce4653642c945d4f93"),
    name: 'John',
    age: 25,
    city: 'New York'
  },
  {
    _id: ObjectId("6458acce4653642c945d4f94"),
    name: 'Mary',
    age: 30,
    city: 'Los Angeles'
  },
  {
    _id: ObjectId("6458acce4653642c945d4f95"),
    name: 'Bob',
    age: 35,
    city: 'Chicago'
  },
  {
    _id: ObjectId("6458acce4653642c945d4f96"),
    name: 'Jane',
    age: 40,
    city: 'Houston'
  },
  {
    _id: ObjectId("6458b71ac0822e0f680e6d77"),
    name: 'Aravind',
    age: '37',
    city: 'Mumbai'
  }
]
```

GREATER THAN FILTER QUERY:

To get the specific numeric data using conditions like greater than equal or less than equal use the $gte or $lte operator in the find() method.

SYNTAX:

db.collection_name.find({< key > : {$gte : < value >}})
or
db.collection_name.find({< key > : {$lte : < value >}})

QUERY AND OUTPUT:

```
CSE> db.Student.find({age:{$gte:25}})
[
  {
    _id: ObjectId("6458acce4653642c945d4f93"),
    name: 'John',
    age: 25,
    city: 'New York'
  },
  {
    _id: ObjectId("6458acce4653642c945d4f94"),
    name: 'Mary',
    age: 30,
    city: 'Los Angeles'
  },
  {
    _id: ObjectId("6458acce4653642c945d4f95"),
    name: 'Bob',
    age: 35,
    city: 'Chicago'
  },
  {
    _id: ObjectId("6458acce4653642c945d4f96"),
    name: 'Jane',
    age: 40,
    city: 'Houston'
  }
```

DELETING THE FIRST DOCUMENT:

In this example, we are deleting the first document from the Student collection by passing an empty document in the db.collection.deleteOne() method.

SYNTAX:

```
db.collection_name.deleteOne({})
```

QUERY AND OUTPUT:

```
CSE> db.Student.deleteOne({})
{ acknowledged: true, deletedCount: 1 }
CSE> db.Student.find()
[
  {
    _id: ObjectId("6458acce4653642c945d4f95"),
    name: 'Bob',
    age: 35,
    city: 'Chicago'
  },
  {
    _id: ObjectId("6458acce4653642c945d4f96"),
    name: 'Jane',
    age: 40,
    city: 'Houston'
  },
  {
    _id: ObjectId("645ba7fcf3ddd549c012f33a"),
    name: 'Jennifer',
    age: 24,
    city: 'Florida'
  }
]
CSE>
```

DROPPING A COLLECTION:

In MongoDB, a collection is a group of MongoDB documents that are stored together. Dropping a collection in MongoDB means permanently deleting the entire collection and all of its contents from the database.

SYNTAX:

```
db.collection_name.drop()
```

QUERY AND OUTPUT:

```
CSE> db.Student.drop()
true
CSE>
```

DROP A DATABASE:

In MongoDB, databases hold collections of documents. On a single MongoDB server, we can run multiple databases. when you install MongoDB some databases are automatically generated to use. many times, you need to delete some database when the database is no longer used.

db.dropDatabase() the command is used to drop an existing database. This command will delete the currently selected database. If you have not selected any database, then it will delete the default 'test' database.

SYNTAX:

```
db.dropDatabase()
```

QUERY AND OUTPUT:

```
CSE> db.dropDatabase()
{ ok: 1, dropped: 'CSE' }
CSE>
```

RESULT:

Thus create a document-based NoSQL database using MongoDB, with columns and graphs, and populate it with sample data.

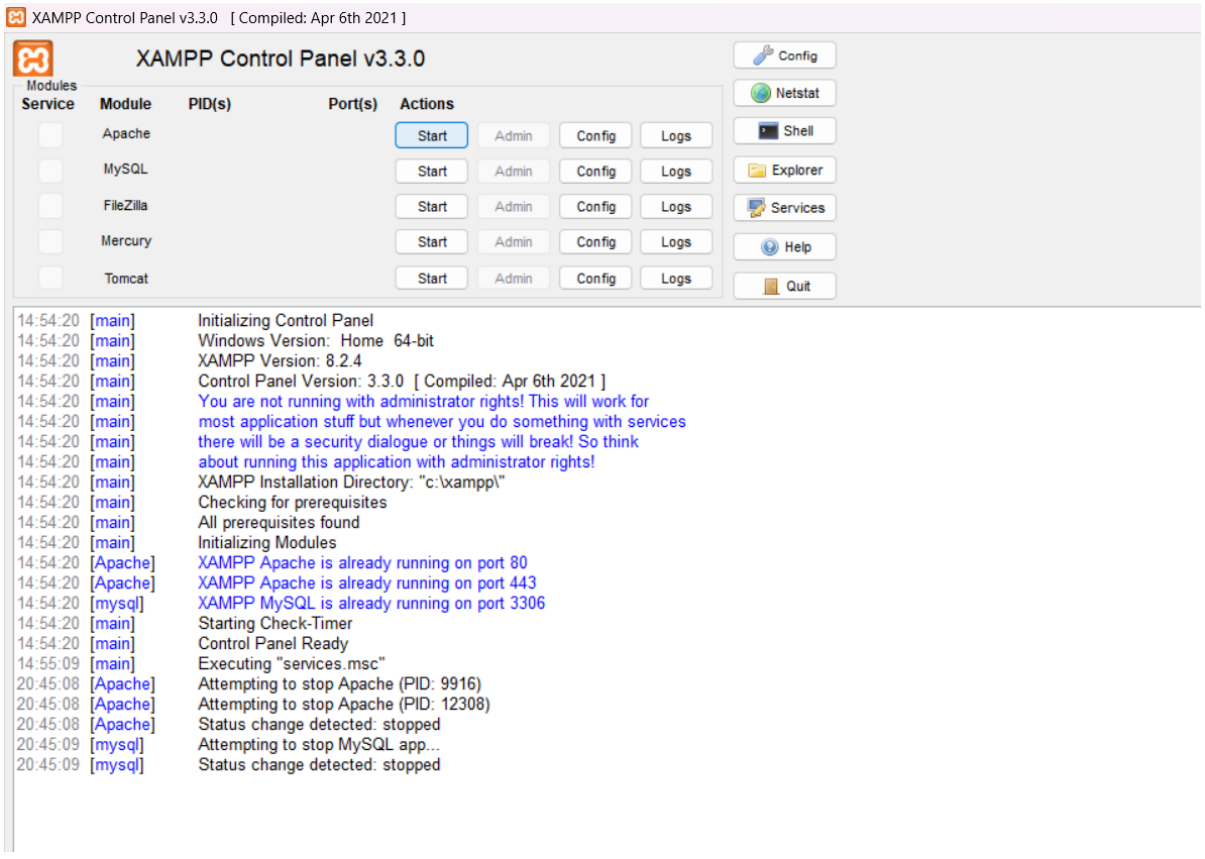
EX NO:12	DEVELOP A SIMPLE GUI BASED DATABASE APPLICATION AND INCORPORATE ALL THE ABOVE-MENTIONED FEATURES
DATE:	

AIM:

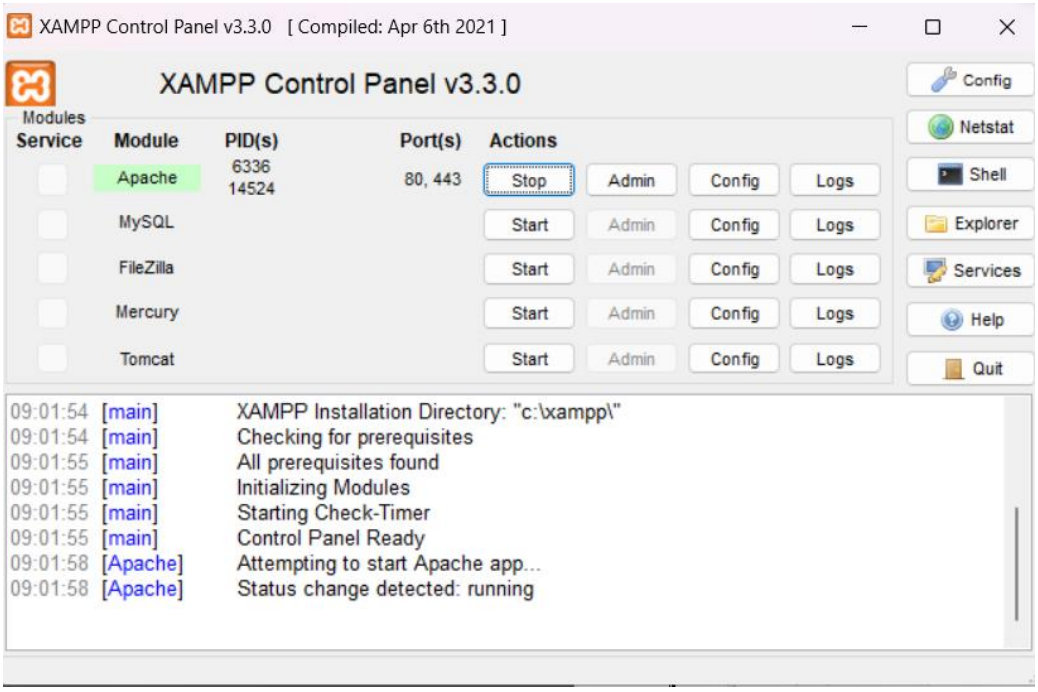
To implement mysql connectivity with frontend tool PHP and HTML and to develop Employees database.

DATABASE CONNECTIVITY:

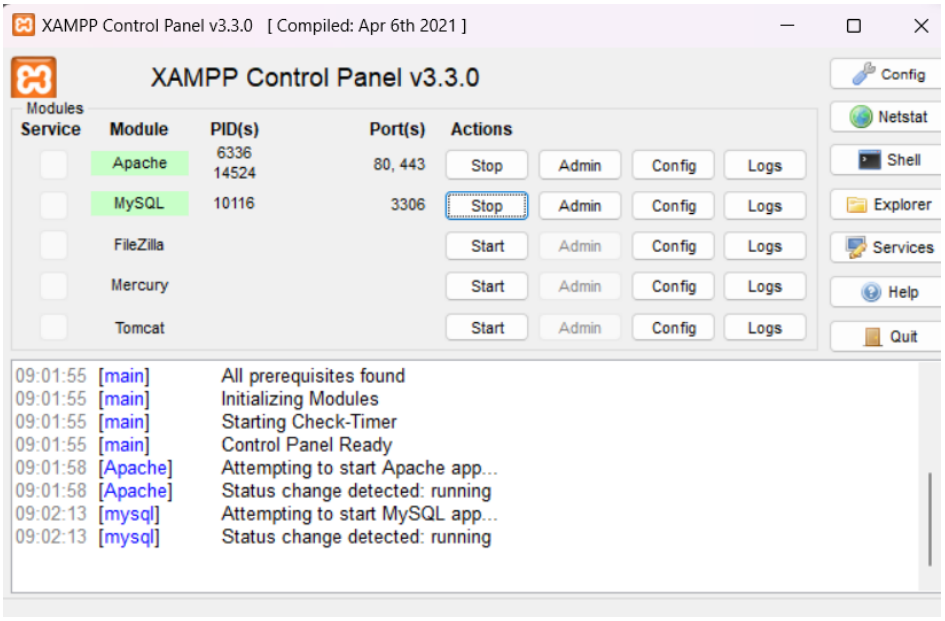
Step 1: Open Xampp Control Panel.



Step 2: Start Apache server



Step 3: Start mysql server.



SOURCE CODE FOR DATABASE CONNECTIVITY:

Step 1: go to xampp server installed directory and click xampp folder.

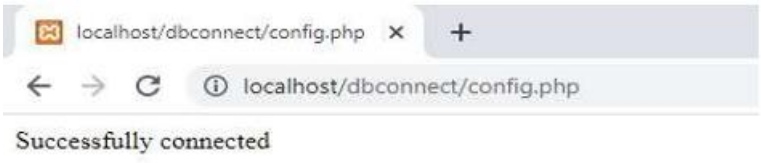
Step 2: click htdocs .

Step 3: Create PHP file for database connectivity with filename (config.php).

PROGRAM:

```
<?php

define('DB_SERVER', 'localhost');
define('DB_USERName', 'root');
define('DB_PASSWORD', '');
define('DB_Name', 'test');

/* Attempt to connect to MySQL database */
$link = mysqli_connect(DB_SERVER, DB_USERName, DB_PASSWORD, DB_Name);

// Check connection
if($link === false){
   die("ERROR: Could not connect. " . mysqli_connect_error());
}
?>
```

Step 4: Run config.php in the browser .



IMPLEMENTATION OF STUDENT DATABASE:

Create.php:

PROGRAM:

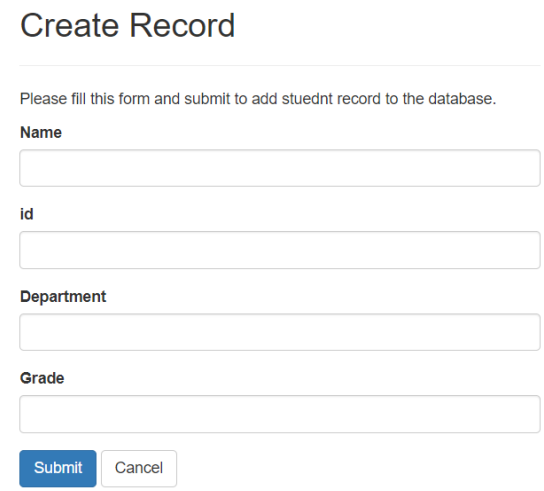
```
<?php
// Include config file
require_once "config.php";

// Define variables and initialize with empty values
$Name =$id= $Department = $Grade = "";
$Name_err =$id_err= $Department_err = $Grade = "";

// Processing form data when form is submitted
if($_SERVER["REQUEST_METHOD"] == "POST"){
    // Validate Name
    $input_Name = trim($_POST["Name"]);
    if(empty($input_Name)){
        $Name_err = "Please enter a Name.";
    } elseif(!filter_var($input_Name, FILTER_VALIDATE_REGEXP,
array("options"=>array("regexp"=>"/^[a-zA-Z\s]+$/")))){
        $Name_err = "Please enter a valid Name.";
    } else{
        $Name = $input_Name;
    }

    // Validate register number
    $input_id = trim($_POST["id"]);
    if(empty($input_id)){
        $id_err = "Please enter your register number";
    } else{
        $id = $input_id;
    }
    //validate Department
    $input_Department = trim($_POST["Department"]);
    if(empty($input_Department)){
        $Department_err = "Please enter your Department";
    } elseif(!filter_var($input_Department, FILTER_VALIDATE_REGEXP,
array("options"=>array("regexp"=>"/^[a-zA-Z\s]+$/")))){
        $Department_err = "Please enter a valid Name.";
    } else{
        $Department = $input_Department;
    }
```

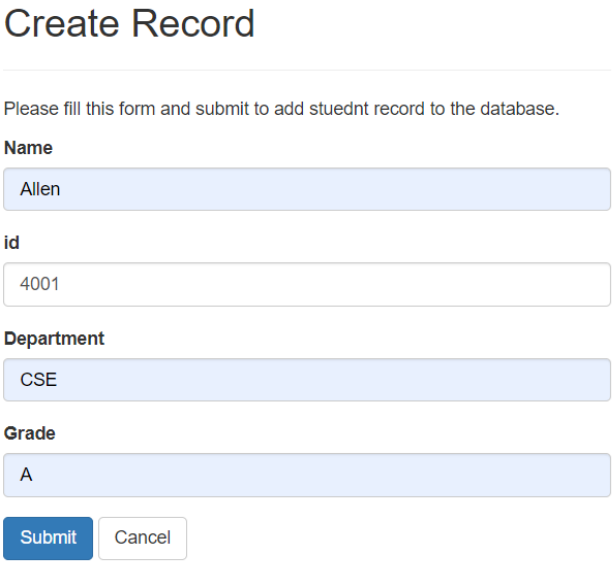
```
    // Validate Grade
    $input_Grade = trim($_POST["Grade"]);
    if(empty($input_Grade)){
        $Grade_err = "Please enter your Grade";
    }
     else{
        $Grade = $input_Grade;
    }

    // Check input errors before inserting in database
    if(empty($Name_err) && empty($id_err) && empty($Department_err) &&
empty($Grade_err)){
        // Prepare an insert statement
        $sql = "INSERT INTO student (Name,id,Department,Grade) VALUES (?, ?, ?,?)";

        if($stmt = mysqli_prepare($link, $sql)){
            // Bind variables to the prepared statement as parameters
            mysqli_stmt_bind_param($stmt, "ssss", $param_Name,$param_id,
$param_Department, $param_Grade);

            // Set parameters
            $param_Name = $Name;
            $param_id=$id;
            $param_Department = $Department;
            $param_Grade = $Grade;

            // Attempt to execute the prepared statement
            if(mysqli_stmt_execute($stmt)){
                // Records created successfully. Redirect to landing page
                header("location: index.php");
                exit();
            } else{
                echo "Something went wrong. Please try again later.";
            }
        }

        // Close statement
        mysqli_stmt_close($stmt);
    }
        // Close connection
    mysqli_close($link);
}
?>
 <!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
```

```
<title>Create Record</title>
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.css">
<style type="text/css">
.wrapper{
width: 500px;
margin: 0 auto;
}
</style>
</head>
<body>
<div class="wrapper">
<div class="container-fluid">
<div class="row">
<div class="col-md-12">
<div class="page-header">
<h2>Create Record</h2>
</div>
<p>Please fill this form and submit to add stuednt record to the database.</p>
<form action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]); ?>"
method="post">
<div class="form-group <?php echo (!empty($Name_err)) ? 'has-error' : ''; ?>">
<label>Name</label>
<input type="text" Name="Name" class="form-control" value="<?php echo $Name; ?>">
<span class="help-block"><?php echo $Name_err;?></span>
</div>
<div class="form-group <?php echo (!empty($Grade_err)) ? 'has-error' : ''; ?>">
<label>id</label>
<input type="text" Name="id" class="form-control" value="<?php echo $id; ?>">
<span class="help-block"><?php echo $id_err;?></span>
</div>
<div class="form-group <?php echo (!empty($Department_err)) ? 'has-error' : ''; ?>">
<label>Department</label>
<input type="text" Name="Department" class="form-control" value="<?php echo
$Department; ?>">
<span class="help-block"><?php echo $Department_err;?></span>
</div>
<div class="form-group <?php echo (!empty($Grade_err)) ? 'has-error' : ''; ?>">
<label>Grade</label>
<input type="text" Name="Grade" class="form-control" value="<?php echo $Grade; ?>">
<span class="help-block"><?php echo $Grade;?></span>
</div>

<input type="submit" class="btn btn-primary" value="Submit">
<a href="index.php" class="btn btn-default">Cancel</a>
</form>
</div>
</div>
</div>
</div>
```

```
</body>
</html>
```

OUTPUT:





Delete.php

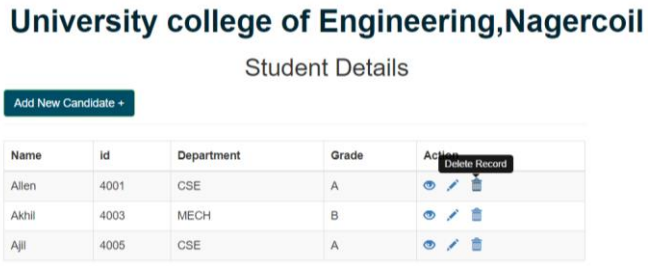
PROGRAM:

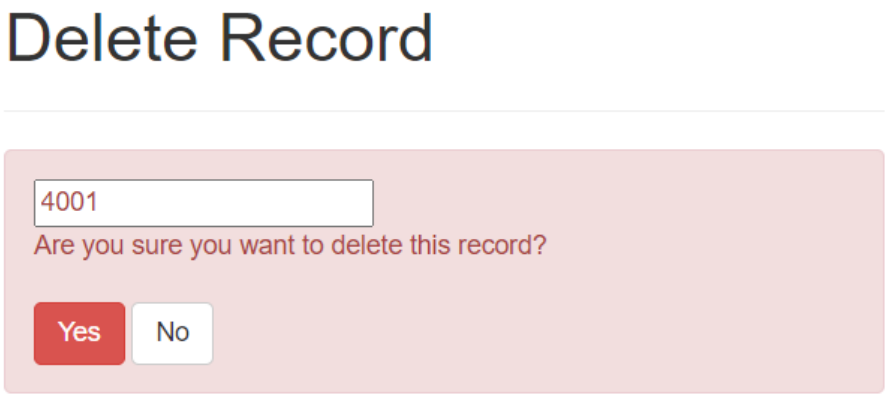
```
<?php
// Process delete operation after confirmation
if(isset($_POST["id"]) && !empty($_POST["id"])){
    // Include config file
    require_once "config.php";

    // Prepare a delete statement
    $sql = "DELETE FROM student WHERE id = ?";
```

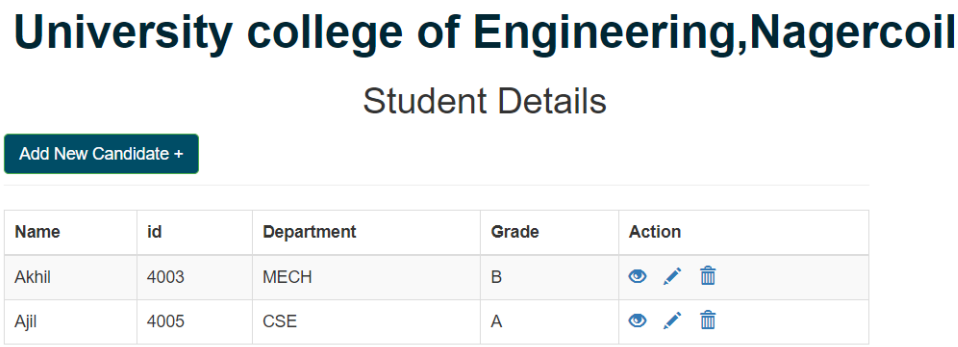
```
    if($stmt = mysqli_prepare($link, $sql)){
        // Bind variables to the prepared statement as parameters
        mysqli_stmt_bind_param($stmt, "i", $param_id);

        // Set parameters
        $param_id = trim($_POST["id"]);

        // Attempt to execute the prepared statement
        if(mysqli_stmt_execute($stmt)){
            // Records deleted successfully. Redirect to landing page
            header("location: index.php");
            exit();
        } else{
            echo "Oops! Something went wrong. Please try again later.";
        }
    }
         // Close statement
    mysqli_stmt_close($stmt);
        // Close connection
    mysqli_close($link);
} else{
    // Check existence of id parameter
    if(empty(trim($_GET["id"]))){
        // URL doesn't contain id parameter. Redirect to error page
        header("location: error.php");
        exit();
    }
}
?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>View Record</title>
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.css">
    <style type="text/css">
        .wrapper{
            width: 500px;
            margin: 0 auto;
        }
    </style>
</head>
<body>
    <div class="wrapper">
```

```
    <div class="container-fluid">
        <div class="row">
            <div class="col-md-12">
                <div class="page-header">
                    <h1>Delete Record</h1>
                </div>
                <form action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]); ?>"
method="post">
                    <div class="alert alert-danger fade in">
                        <input type="text" Name="id" value="<?php echo trim($_GET["id"]);
?>"/>
                        <p>Are you sure you want to delete this record?</p><br>
                        <p>
                            <input type="submit" value="Yes" class="btn btn-danger">
                            <a href="index.php" class="btn btn-default">No</a>
                        </p>
                    </div>
                </form>
            </div>
        </div>
    </div>
</body>
</html>
```

OUTPUT:







Update,php:

PROGRAM:

```
<?php
// Include config file
require_once "config.php";

// Define variables and initialize with empty values
$Name =$id= $Department = $Grade = "";
$Name_err =$id_err= $Department_err = $Grade_err = "";

// Processing form data when form is submitted
if(isset($_POST["id"]) && !empty($_POST["id"])){
    // Get hidden input value
    $id = $_POST["id"];

    // Validate Name
    $input_Name = trim($_POST["Name"]);
    if(empty($input_Name)){
        $Name_err = "Please enter a Name.";
    } elseif(!filter_var($input_Name, FILTER_VALIDATE_REGEXP,
array("options"=>array("regexp"=>"/^[a-zA-Z\s]+$/")))){
        $Name_err = "Please enter a valid Name.";
    } else{
        $Name = $input_Name;
    }

     // Validate register number
     $input_id = trim($_POST["id"]);
     if(empty($input_id)){
         $id_err = "Please enter your register number";
     } else{
```

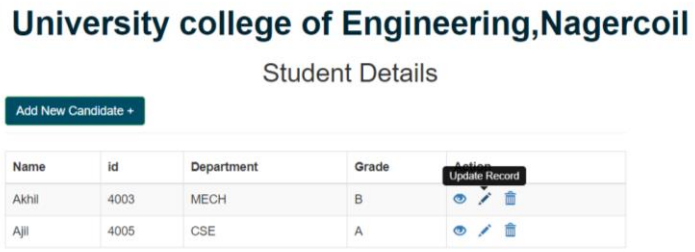
```
        $id = $input_id;
    }
    // Validate Department
    $input_Department = trim($_POST["Department"]);
    if(empty($input_Department)){
        $Department_err = "Please enter an Department.";
    } else{
        $Department = $input_Department;
    }

    // Validate Grade
    $input_Grade = trim($_POST["Grade"]);
    if(empty($input_Grade)){
        $Grade_err = "Please enter your Grade.";
    } else{
        $Grade = $input_Grade;
    }

    // Check input errors before inserting in database
    if(empty($Name_err) && empty($Department_err) && empty($Grade_err)&&
empty($id_err)){
        // Prepare an update statement
        $sql = "UPDATE student SET Name=?, Department=?, Grade=? WHERE id=?";

        if($stmt = mysqli_prepare($link, $sql)){
            // Bind variables to the prepared statement as parameters
            mysqli_stmt_bind_param($stmt, "sssi", $param_Name, $param_Department,
$param_Grade, $param_id);

            // Set parameters
            $param_Name = $Name;

            $param_Department = $Department;
            $param_Grade = $Grade;
            $param_id = $id;
                    // Attempt to execute the prepared statement
            if(mysqli_stmt_execute($stmt)){
                // Records updated successfully. Redirect to landing page
                header("location: index.php");
                exit();
            } else{
                echo "Something went wrong. Please try again later.";
            }
        }
             // Close statement
        mysqli_stmt_close($stmt);
```

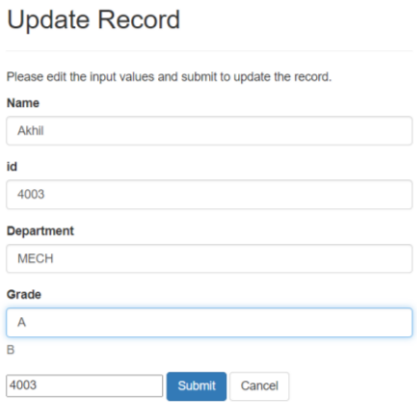
```
    }
        // Close connection
    mysqli_close($link);
} else{
    // Check existence of id parameter before processing further
    if(isset($_GET["id"]) && !empty(trim($_GET["id"]))){
        // Get URL parameter
        $id =  trim($_GET["id"]);

        // Prepare a select statement
        $sql = "SELECT * FROM student WHERE id =?";
        if($stmt = mysqli_prepare($link, $sql)){
            // Bind variables to the prepared statement as parameters
            mysqli_stmt_bind_param($stmt, "i", $param_id);

            // Set parameters
            $param_id = $id;

            // Attempt to execute the prepared statement
            if(mysqli_stmt_execute($stmt)){
                $result = mysqli_stmt_get_result($stmt);

                if(mysqli_num_rows($result) == 1){
                    /* Fetch result row as an associative array. Since the result set
                    contains only one row, we don't need to use while loop */
                    $row = mysqli_fetch_array($result, MYSQLI_ASSOC);

                    // Retrieve individual field value
                    $Name = $row["Name"];
                    $id = $row["id"];
                    $Department=$row["Department"];
                    $Grade = $row["Grade"];
                } else{
                    // URL doesn't contain valid id. Redirect to error page
                    header("location: error.php");
                    exit();
                }

            } else{
                echo "Oops! Something went wrong. Please try again later.";
            }
        }

        // Close statement
        mysqli_stmt_close($stmt);
```

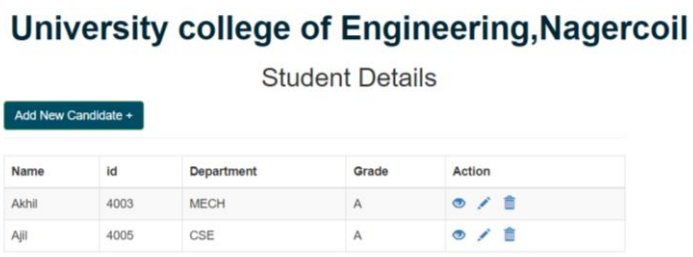
```
        // Close connection
        mysqli_close($link);
    } else{
        // URL doesn't contain id parameter. Redirect to error page
        header("location: error.php");
        exit();
    }
}
?>
 <!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Update Record</title>
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.css">
<style type="text/css">
.wrapper{
width: 500px;
margin: 0 auto;
}
</style>
</head>
<body>
<div class="wrapper">
<div class="container-fluid">
<div class="row">
<div class="col-md-12">
<div class="page-header">
<h2>Update Record</h2>
</div>
<p>Please edit the input values and submit to update the record.</p>
<form action="<?php echo htmlspecialchars(baseName($_SERVER['REQUEST_URI']));
?>" method="post">
<div class="form-group <?php echo (!empty($Name_err)) ? 'has-error' : ''; ?>">
<label>Name</label>
<input type="text" Name="Name" class="form-control" value="<?php echo $Name; ?>">
<span class="help-block"><?php echo $Name_err;?></span>
</div>
<div class="form-group <?php echo (!empty($Grade_err)) ? 'has-error' : ''; ?>">
<label>id</label>
<input type="number" Name="id" class="form-control" value="<?php echo $id; ?>">
<span class="help-block"><?php echo $id_err;?></span>
</div>
<div class="form-group <?php echo (!empty($Department_err)) ? 'has-error' : ''; ?>">
<label>Department</label>
```

```
<input type="text" Name="Department" class="form-control" value="<?php echo
$Department; ?>">
<span class="help-block"><?php echo $Department_err;?></span>
</div>
<div class="form-group <?php echo (!empty($Grade_err)) ? 'has-error' : ''; ?>">
<label>Grade</label>
<input type="text" Name="Grade" class="form-control" value="<?php echo $Grade; ?>">
<span class="help-block"><?php echo $Grade;?></span>
</div>
<input type="number" Name="id" value="<?php echo $id; ?>"/>
<input type="submit" class="btn btn-primary" value="Submit">
<a href="index.php" class="btn btn-default">Cancel</a>
</form>
</div>
</div>
</div>
</div>
</body>
</html>
```

OUTPUT:







Read.php:

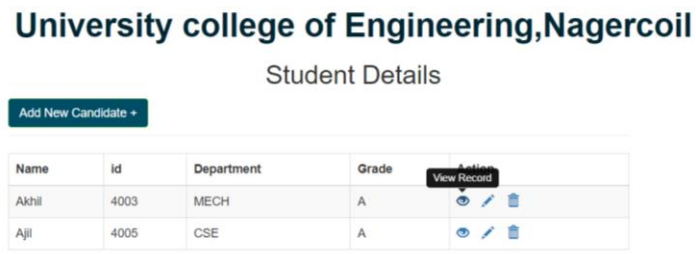
PROGRAM:

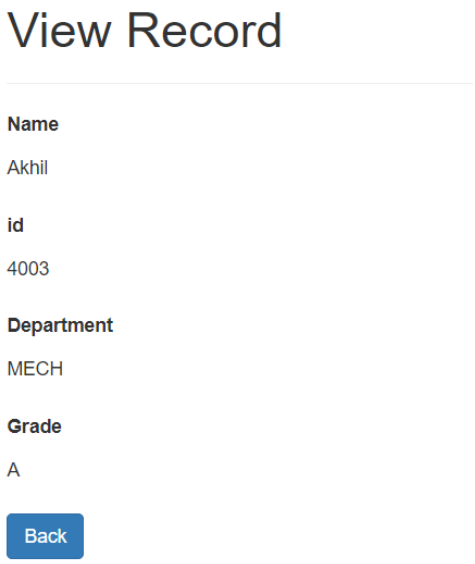
```
<?php
// Check existence of id parameter before processing further
if(isset($_GET["id"]) && !empty(trim($_GET["id"]))){
    // Include config file
    require_once "config.php";

    // Prepare a select statement
    $sql = "SELECT * FROM student WHERE id = ?";

    if($stmt = mysqli_prepare($link, $sql)){
        // Bind variables to the prepared statement as parameters
        mysqli_stmt_bind_param($stmt, "i", $param_id);

        // Set parameters
        $param_id = trim($_GET["id"]);

        // Attempt to execute the prepared statement
        if(mysqli_stmt_execute($stmt)){
            $result = mysqli_stmt_get_result($stmt);

            if(mysqli_num_rows($result) == 1){
                /* Fetch result row as an associative array. Since the result set
                contains only one row, we don't need to use while loop */
                $row = mysqli_fetch_array($result, MYSQLI_ASSOC);

                // Retrieve individual field value
                $Name = $row["Name"];
                $id = $row["id"];
                $Department = $row["Department"];
                $Grade = $row["Grade"];
            } else{
                // URL doesn't contain valid id parameter. Redirect to error page
```

```
            header("location: error.php");
            exit();
        }

    } else{
        echo "Oops! Something went wrong. Please try again later.";
    }
}

// Close statement
mysqli_stmt_close($stmt);

// Close connection
mysqli_close($link);
} else{
// URL doesn't contain id parameter. Redirect to error page
header("location: error.php");
exit();
}
?>
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>View Record</title>
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.css">
<style type="text/css">
    .wrapper{
        width: 500px;
        margin: 0 auto;
    }
</style>
</head>
<body>
<div class="wrapper">
    <div class="container-fluid">
        <div class="row">
            <div class="col-md-12">
                <div class="page-header">
                    <h1>View Record</h1>
                </div>
                <div class="form-group">
                    <label>Name</label>
                    <p class="form-control-static"><?php echo $row["Name"]; ?></p>
                </div>
```

```
                <div class="form-group">
                    <label>id</label>
                    <p class="form-control-static"><?php echo $row["id"]; ?></p>
                </div>
                <div class="form-group">
                    <label>Department</label>
                    <p class="form-control-static"><?php echo $row["Department"]; ?></p>
                </div>
                <div class="form-group">
                    <label>Grade</label>
                    <p class="form-control-static"><?php echo $row["Grade"]; ?></p>
                </div>
                <p><a href="index.php" class="btn btn-primary">Back</a></p>
            </div>
        </div>
    </div>
</div>
</body>
</html>
```

OUTPUT:





Index.php:

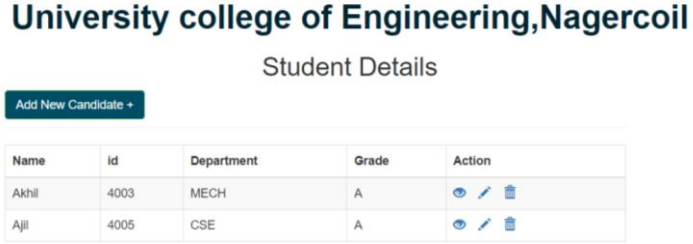
PROGRAM:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Dashboard</title>
   <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.css">
   <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"></script>
   <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.js"></script>
   <style type="text/css">
       body {
    background-color:white;
   }
   .btn {
     background-color: #004d66;
     color:white ;

   }
   h1 {
  color: #002633;
  font-size: 38px;
  width: 800px;
}
h2{

   width:800px;
   align-items: center;
}
tr{
   background-color: white;
}
       .wrapper{
           width: 750px;
           margin: 0 auto;
       }
       .wrapper-h1{

           margin: 0 auto;
           width: 800;
       }
       .page-header h1{
           margin-top: 20;
```

```
            text-align: center;
                }
        .page-header h2{
            margin-top: 10;
        }
        table tr td:last-child a{
            margin-right: 15px;
                }
    </style>
        <script type="text/javascript">
        $(document).ready(function(){
            $('[data-toggle="tooltip"]').tooltip();
        });
    </script>
</head>
<body>
    <div class="wrapper">
        <div class="container-fluid">
            <div class="row">
                <div class="col-md-12">
                    <div class="page-header clearfix">
                                <h1 class="text-center"> <b>University college of
Engineering,Nagercoil</b></h1>
                        <h2 class="text-center">Student Details</h2>
                        <a href="create.php" class="btn btn-success  ">Add New Candidate
                        </a>
                    </div>
                    <?php
                    // Include config file
                    require_once "config.php";

                    // Attempt select query execution
                    $sql = "SELECT * FROM student";
                    if($result = mysqli_query($link, $sql)){
                        if(mysqli_num_rows($result) > 0){
                            echo "<table class='table table-bordered table-striped'>";
                                echo "<thead>";
                                    echo "<tr>";
                                        echo "<th>Name</th>";
                                        echo "<th>id</th>";
                                        echo "<th>Department</th>";
                                        echo "<th>Grade</th>";
                                        echo "<th>Action</th>";
                                    echo "</tr>";
                                echo "</thead>";
                                echo "<tbody>";
```

```
                    while($row = mysqli_fetch_array($result)){
                        echo "<tr>";
                            echo "<td>" . $row['Name'] . "</td>";
                            echo "<td>" . $row['id'] . "</td>";
                            echo "<td>" . $row['Department'] . "</td>";
                            echo "<td>" . $row['Grade'] . "</td>";
                            echo "<td>";
                                echo "<a href='read.php?id=". $row['id'] ."' title='View Record'
data-toggle='tooltip'><span class='glyphicon glyphicon-eye-open'></span></a>";
                                echo "<a href='update.php?id=". $row['id'] ."' title='Update
Record' data-toggle='tooltip'><span class='glyphicon glyphicon-pencil'></span></a>";
                                echo "<a href='delete.php?id=". $row['id'] ."' title='Delete Record'
data-toggle='tooltip'><span class='glyphicon glyphicon-trash'></span></a>";
                            echo "</td>";
                        echo "</tr>";
                    }
                    echo "</tbody>";
                echo "</table>";
                // Free result set
                mysqli_free_result($result);
            } else{
                echo "<p class='lead'><em>No records were found.</em></p>";
            }
        } else{
            echo "ERROR: Could not able to execute $sql. " . mysqli_error($link);
        }
         // Close connection
        mysqli_close($link);
        ?>
      </div>
    </div>
  </div>
</body>
</html>
```

OUTPUT:



RESULT:

Thus, the implementation of mysql connectivity with frontend tool PHP and HTML and the Employees database is developed and executed successfully.