**EX NO.: 01 DDL COMMANDS**

**AIM:**

To work with DDL commands

**DDL (Data Definition Language):**

DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. DDL is a set of SQL commands used to create, modify, and delete database structures but not data.

**PROCEDURE:**

**Step 1**: Open Run SQL on Command line and connect to SQL

**Step 2:** Then work with database using SQL queries.

**CREATE:**

This command is used to create the database or its objects (like table, index, function, views, store procedure, and triggers).

**SYNTAX:**

CREATE TABLE table\_name

(

column\_Name1 data\_type ( size of the column ) ,

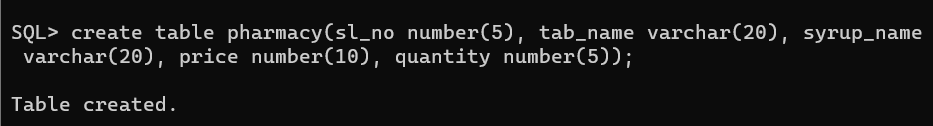
column\_Name2 data\_type ( size of the column) ,

...

column\_NameN data\_type ( size of the column )

) ;

**OUTPUT:**



**ALTER:**

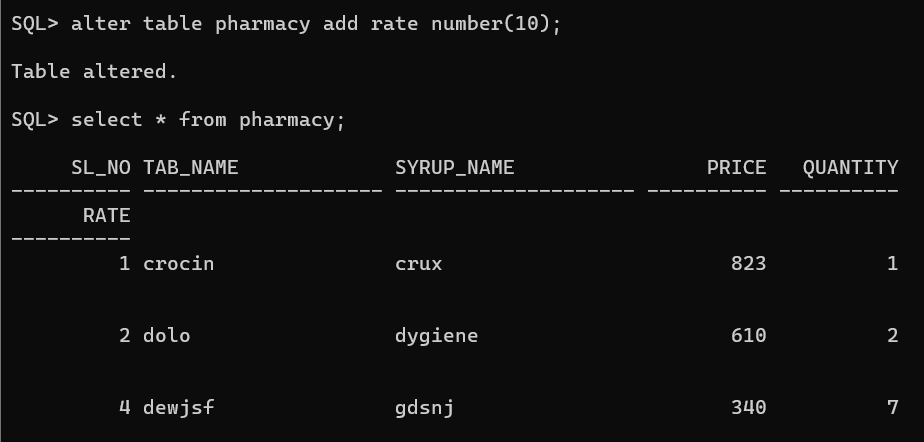
This is used to alter the structure of the database.

**ALTER ADD:**

**Syntax to add a new field in the table:**

ALTER TABLE name\_of\_table ADD column\_name column\_definition;

**OUTPUT:**

****

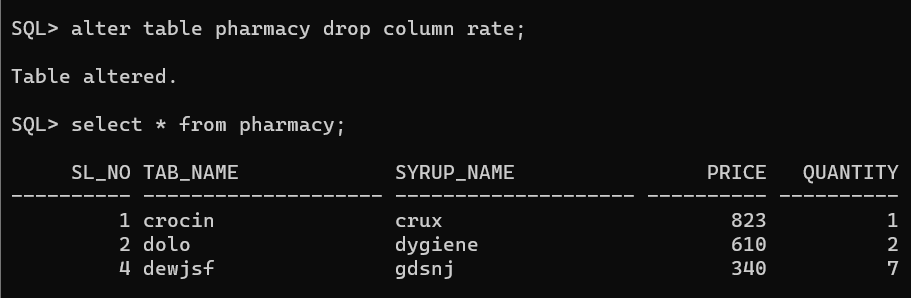
**ALTER DROP:**

**Syntax to remove a column from the table:**

ALTER TABLE name\_of\_table DROP

Column\_Name\_1 , column\_Name\_2 , ….., column\_Name\_N;

**OUTPUT:**

****

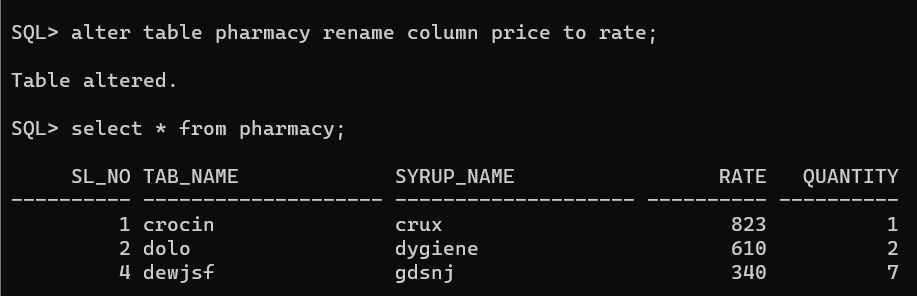
**ALTER RENAME:**

**Syntax to Rename a column from the table:**

ALTER TABLE name\_of\_table RENAME

Old Column\_Name to New Column Name;

**OUTPUT:**

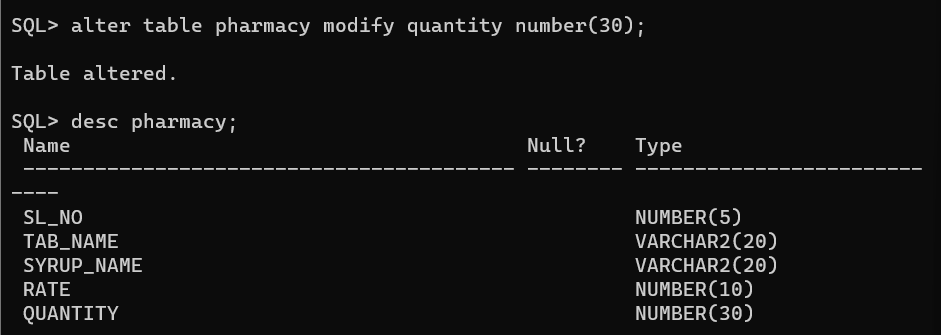
****

**ALTER MODIFY:**

**Syntax to modify the column of the table:**

ALTER TABLE table\_name MODIFY ( column\_name column\_datatype(size));

**OUTPUT:**

****

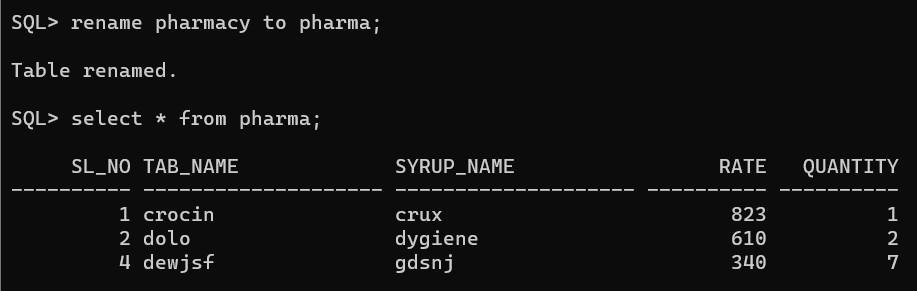
**RENAME:**

This is used to rename an object existing in the database.

**Syntax of rename command:**

RENAME TABLE Old\_Table\_Name TO New\_Table\_Name;

**OUTPUT:**

****

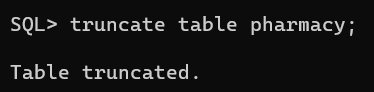
**TRUNCATE:**

This is used to remove all records from a table, including all spaces allocated for the records are removed.

**Syntax of TRUNCATE command:**

TRUNCATE TABLE Table\_Name;

**OUTPUT:**

****

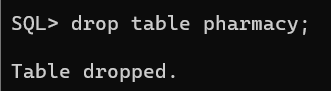
**DROP:**

This command is used to delete objects from the database.

**Syntax to remove a table:**

DROP TABLE Table\_Name;

**OUTPUT:**

****

**RESULT:**

The queries for DDL commands were successfully executed and the output is noted.

**EX NO.: 02 DML COMMANDS**

**AIM:**

To work with DML commands

**DML (Data Manipulation Language**):

The DML commands in Structured Query Language change the data present in the SQL database. We can easily access, store, modify, update and delete the existing records from the database using DML commands

**PROCEDURE:**

**Step 1**: Open Run SQL on Command line and connect to SQL

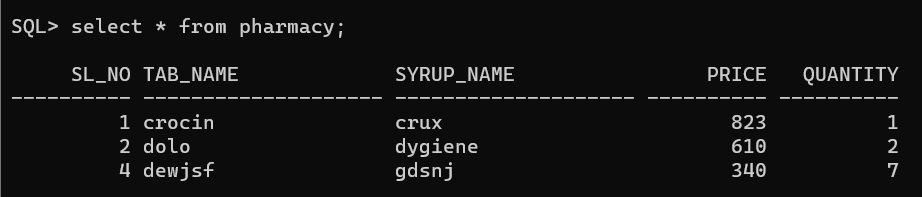
**Step 2:** Then work with database using SQL queries.

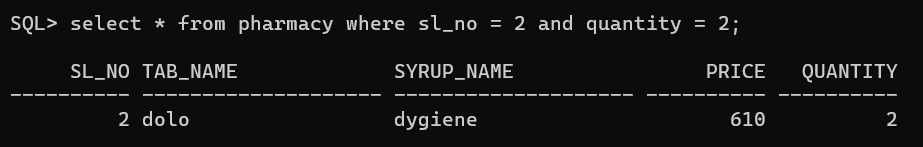
**SELECT:**

SELECT is the most important data manipulation command in Structured Query Language. The SELECT command shows the records of the specified table. It also shows the particular record of a particular column by using the WHERE clause.

**Syntax of SELECT DML command**

SELECT column\_Name\_1, column\_Name\_2, ….., column\_Name\_N FROM Name\_of\_table;

**OUTPUT:**

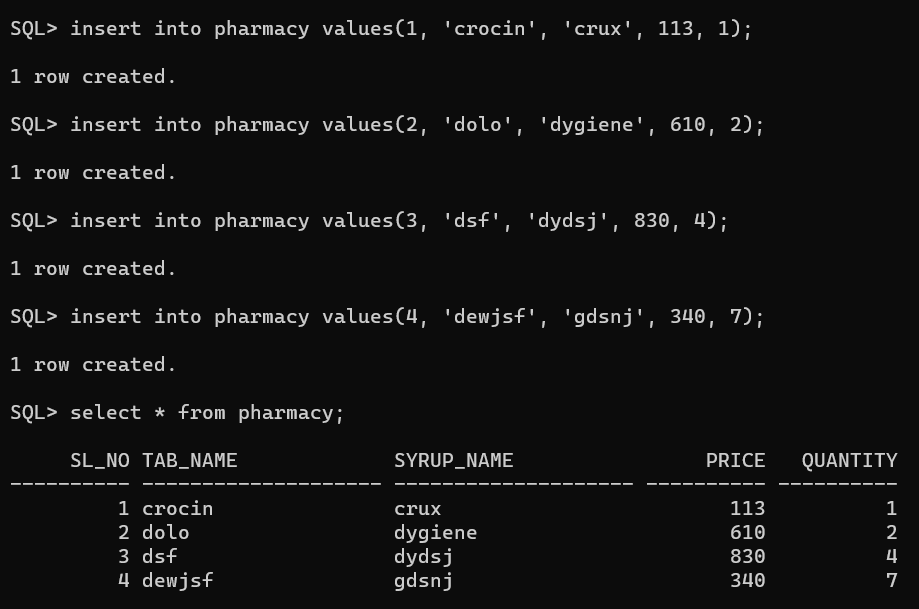
**OUTPUT for SELECT using WHERE:**

**INSERT:**

INSERT is another most important data manipulation command in Structured Query Language, which allows users to insert data in database tables.

**Syntax of INSERT Command**

**INSERT** **INTO** TABLE\_NAME ( column\_Name1 , column\_Name2 , column\_Name3 , .... column\_ NameN )  **VALUES** (value\_1, value\_2, value\_3, .... value\_N ) ;

**OUTPUT:**

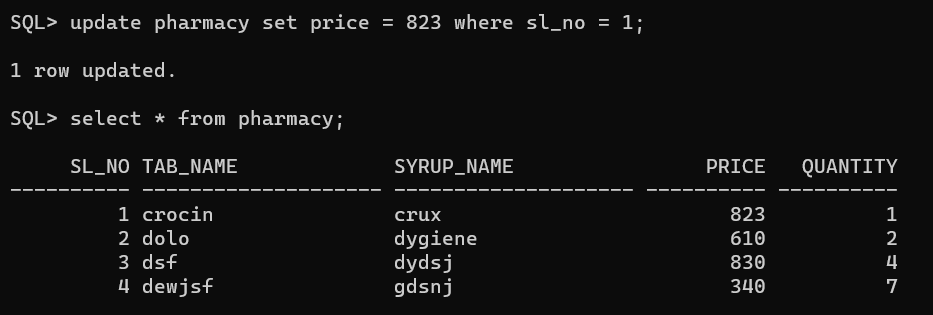
**UPDATE:**

UPDATE is another most important data manipulation command in Structured Query Language, which allows users to update or modify the existing data in database tables.

**Syntax of UPDATE Command**

**UPDATE** Table\_name **SET** [column\_name1= value\_1, ….., column\_nameN = value\_N]

**WHERE** CONDITION;

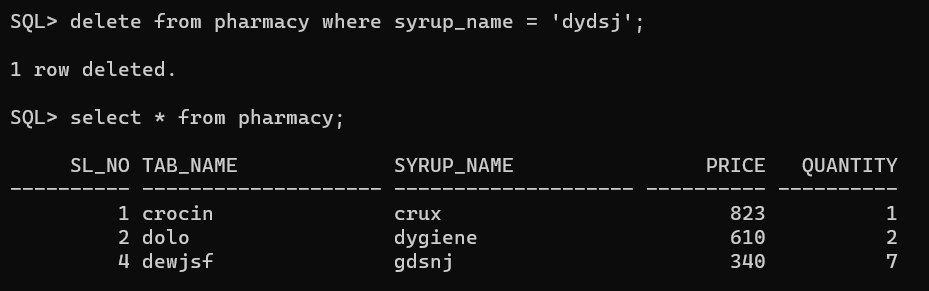
**OUTPUT:**

**DELETE:**

DELETE is a DML command which allows SQL users to remove single or multiple existing records from the database tables.

**Syntax of DELETE Command**

**DELETE** **FROM** Table\_Name **WHERE** condition;

**OUTPUT:**

**RESULT:**

The queries for DML commands were successfully executed and the output is noted.

**EX NO: 03 DCL COMMANDS**

**AIM:**

To work with DCL commands

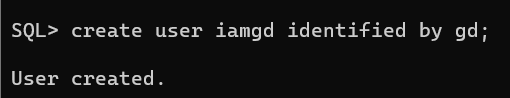
**PROCEDURE:**

**Step 1**: Open Run SQL on Command line and connect to SQL

**Step 2:** Then work with database using SQL queries.

**(DCL)Data control language:**

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

**USER CREATION:**

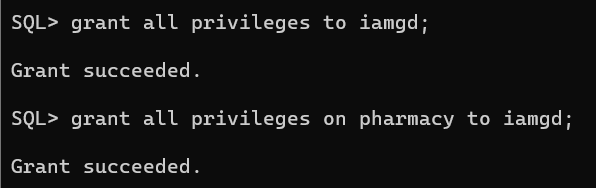
**1. GRANT:**

SQL Grant command is specifically used to provide privileges to database objects for a user. This command also allows users to grant permissions to other users too.

**Syntax:**

grant privilege\_name on object\_name

to {user\_name | public | role\_name}

**OUTPUT:**

**2. REVOKE:**

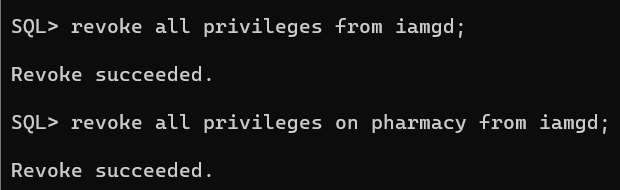
Revoke command withdraw user privileges on database objects if any granted. It does operations opposite to the Grant command. When a privilege is revoked from a particular user U, then the privileges granted to all other users by user U will be revoked.

**Syntax:**

revoke privilege\_name on object\_name

from {user\_name | public | role\_name}

**OUTPUT:**

****

**RESULT:**

The queries for DCL commands were successfully executed and the output is noted.

**EX NO.: 04 SUB QUERIES AND JOINS**

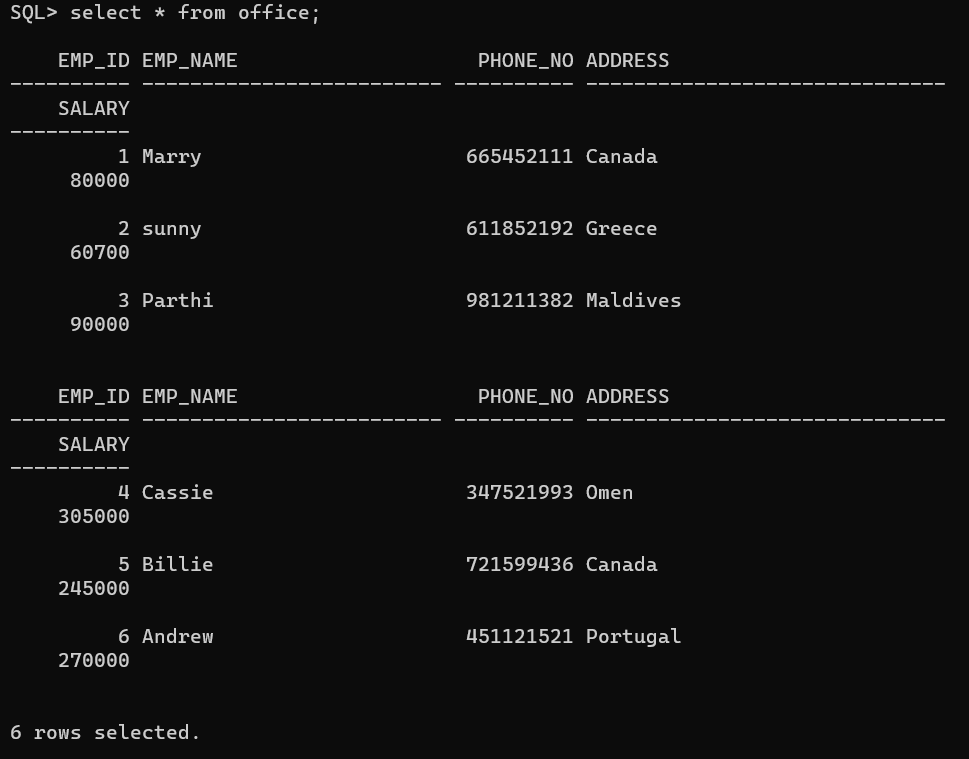
**AIM:**

To work with Sub queries and joins

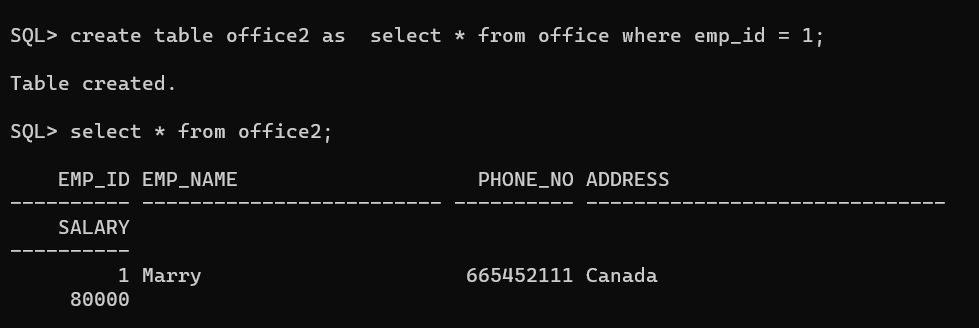
**SUB QUERIES:**

In SQL a Subquery can be simply defined as a query within another query. In other words we can say that a Subquery is a query that is embedded in WHERE clause of another SQL query.

**TABLE 1 – OFFICE:**



**TABLE 2 – OFFICE 2:**



**SUB QUERIES WITH SELECT STATEMENT:**

Subqueries are most frequently used with the SELECT statement.

**The basic syntax is as follows −**

SELECT column\_name [, column\_name ]

FROM table1 [, table2 ]

WHERE column\_name OPERATOR

(SELECT column\_name [, column\_name ]

FROM table1 [, table2 ]

[WHERE])

**OUTPUT:**



**SUB QUERIES WITH INSERT STATEMENT:**

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions.

**The basic syntax is as follows.**

INSERT INTO table\_name [ (column1 [, column2 ]) ]

SELECT [ \*|column1 [, column2 ]

FROM table1 [, table2 ]

[ WHERE VALUE OPERATOR ]

**OUTPUT:**



**SUB QUERIES WITH UPDATE STATEMENT:**

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

**The basic syntax is as follows**

UPDATE table

SET column\_name = new\_value

[ WHERE OPERATOR [ VALUE ]

(SELECT COLUMN\_NAME

FROM TABLE\_NAME)

[ WHERE) ]

**OUTPUT:**



**SUB QUERIES WITH DELETE STATEMENT:**

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

**The basic syntax is as follows.**

DELETE FROM TABLE\_NAME

[ WHERE OPERATOR [ VALUE ]

(SELECT COLUMN\_NAME

FROM TABLE\_NAME)

[ WHERE) ]

**OUTPUT:**



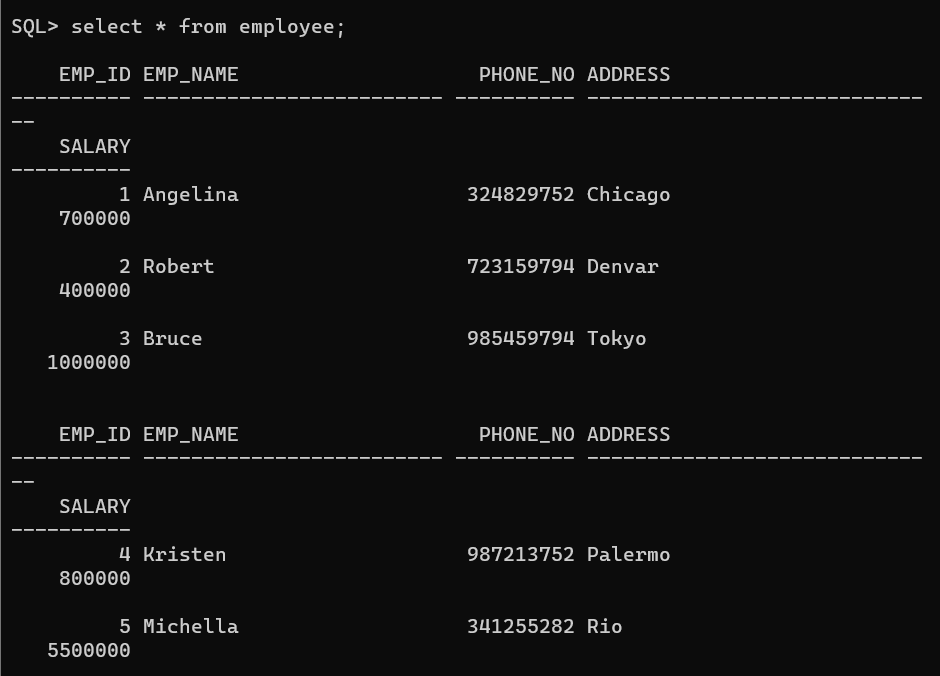
**JOINS:**

Different types of Joins are as follows:

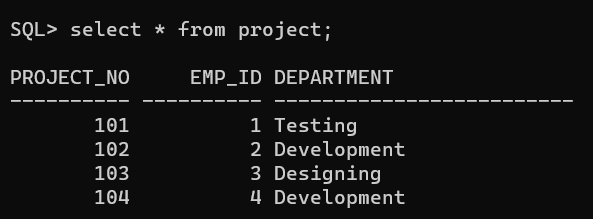
* INNER JOIN
* LEFT JOIN
* RIGHT JOIN
* FULL JOIN

Consider the two tables below:

**EMPLOYEE TABLE:**



**PROJECT TABLE:**



**A. INNER JOIN**

The INNER JOIN keyword selects all rows from both the tables as long as the condition is satisfied. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.

**Syntax**:

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

INNER JOIN table2

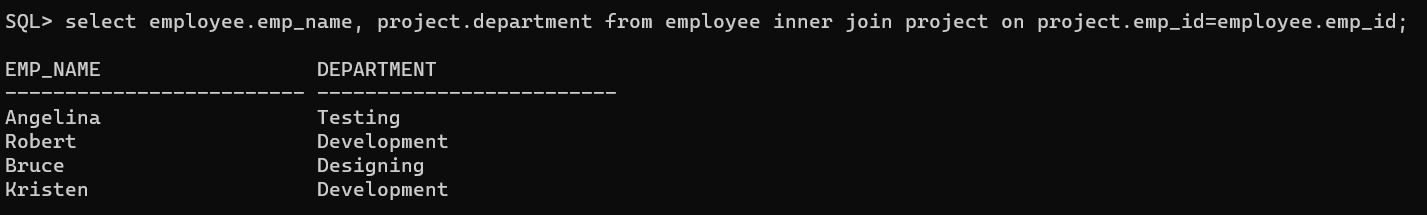
ON table1.matching\_column = table2.matching\_column;

**table1**: First table.

**table2**: Second table

**matching\_column**: Column common to both the tables.

**OUTPUT:**



**B. LEFT JOIN**

This join returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join. For the rows for which there is no matching row on the right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

LEFT JOIN table2

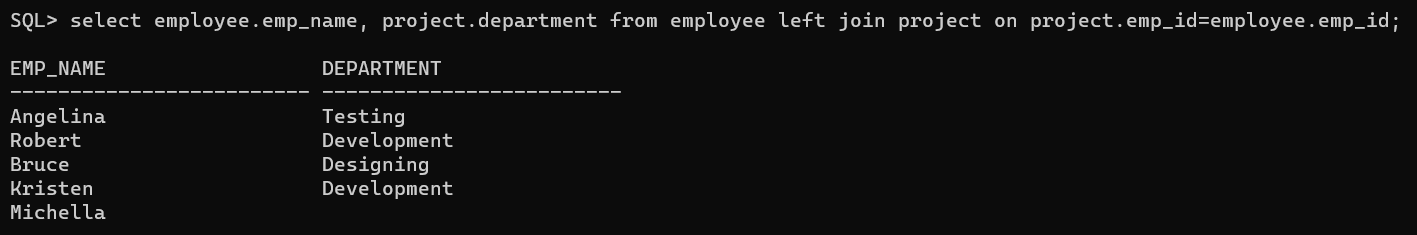
ON table1.matching\_column = table2.matching\_column;

**table1:** First table.

**table2:** Second table

**matching\_column:** Column common to both the tables.

**OUTPUT:**



**C. RIGHT JOIN**

RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join. For the rows for which there is no matching row on the left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

RIGHT JOIN table2

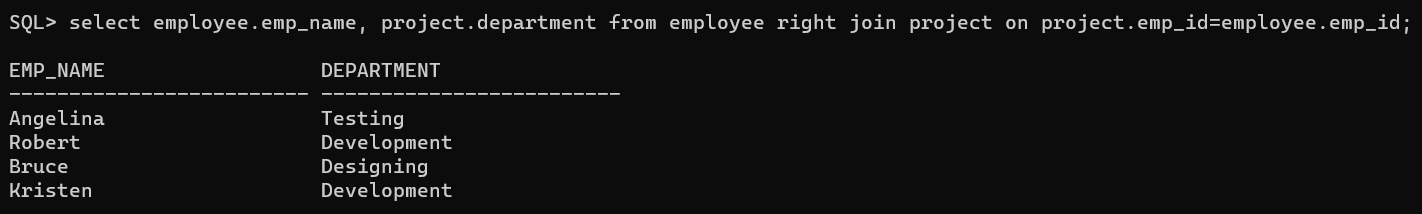
ON table1.matching\_column = table2.matching\_column;

**table1:** First table.

**table2:** Second table

**matching\_column:** Column common to both the tables.

**OUTPUT:**



**D. FULL JOIN**

FULL JOIN creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both tables. For the rows for which there is no matching, the result-set will contain *NULL* values.

**Syntax:**

SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

FULL JOIN table2

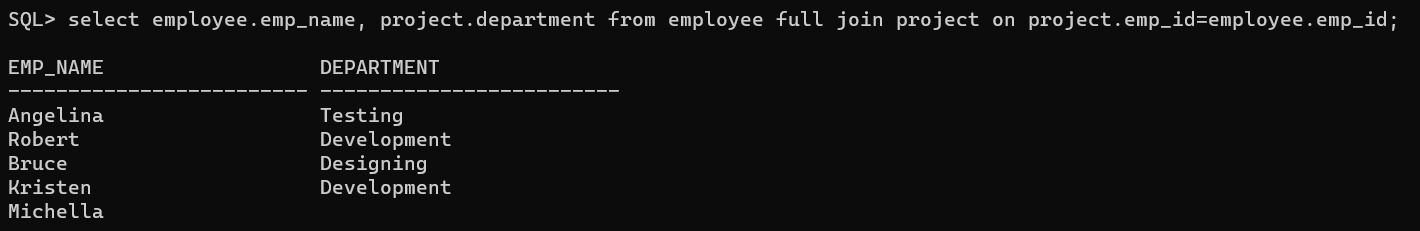
ON table1.matching\_column = table2.matching\_column;

**table1:** First table.

**table2:** Second table

**matching\_column:** Column common to both the tables.

**OUTPUT:**



**RESULT:**

The queries for Sub queries and Joins were successfully executed and the output is noted.

**EX NO.: 05 PL/SQL**

**AIM:**

To work with PL/SQL commands

**PROCEDURE:**

Step1: Open Run SQL on Command line and connect to SQL .

Step 2: Then work with database using PL/SQL Block commands

1. Declare
2. Begin
3. Exception
4. End

**SYNTAX:**

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling>

END;

**EXAMPLE:**

1. **ADDITION OF TWO NUMBERS:**

**PROGRAM CODE:**

SQL> SET SERVEROUTPUT ON;

SQL> declare

2 x number(5);

3 y number(5);

4 z number(5);

5 begin

6 x:=50;

7 y:=20;

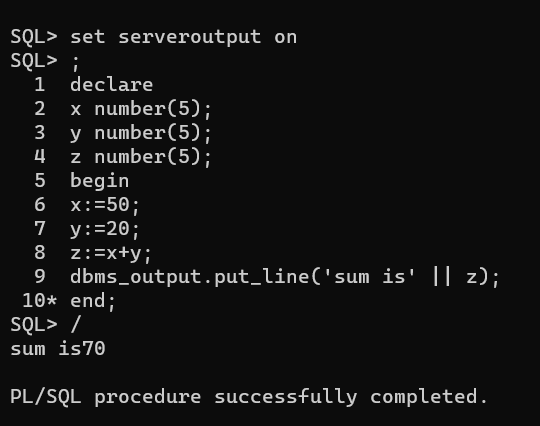
8 z:=x+y;

9 dbms\_output.put\_line('sum is'||z);

10 end;

11 /

**OUTPUT:**



1. **GENERATING SERIES:**

**PROGRAM CODE:**

SQL> SET SERVEROUTPUT ON;

SQL> declare

2 n number(5);

3 tempp number(5);

4 begin

5 n:=1; --1 for print first 10 numbers,2 for even number,3 for odd

6 for i in 1..10 loop

7 case n

8 when 1 then

9 dbms\_output.put\_line(i);

10 when 2 then

11 if mod(i,2)=0 then

12 dbms\_output.put\_line(i);

13 end if;

14 when 3 then

15 if mod(i,2)!=0 then

16 dbms\_output.put\_line(i);

17 end if;

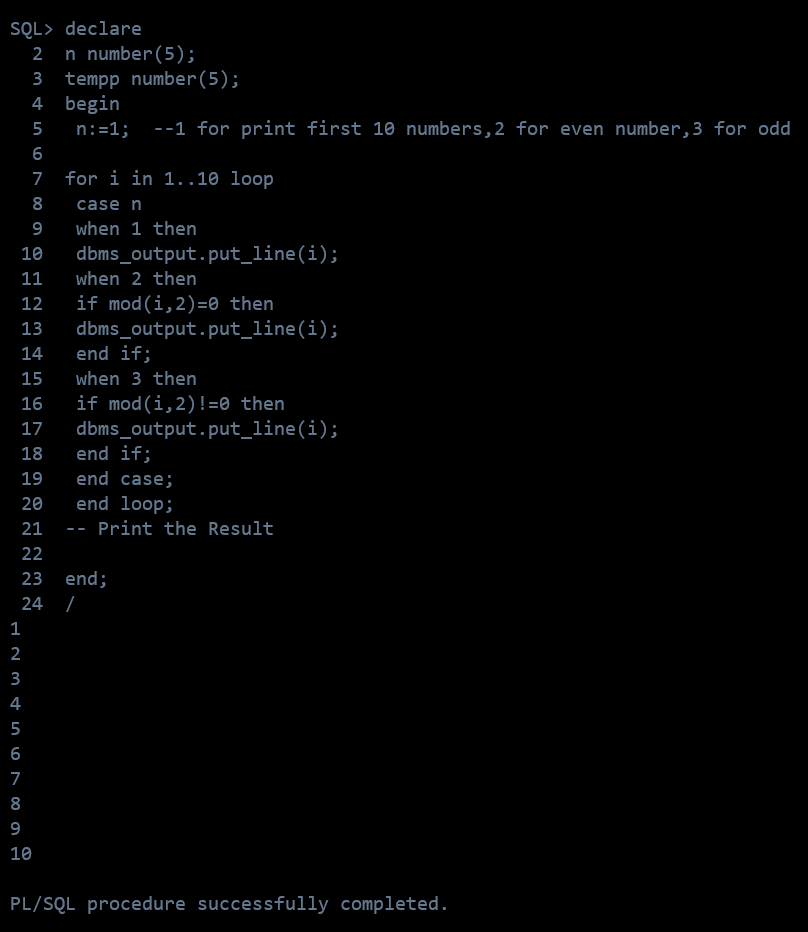
18 end case;

19 end loop;

20 end;

21 /

**OUTPUT:**



**RESULT:**

The PL/SQL queries were successfully executed and the output is noted.

**EX NO.: 06 CURSOR PROCEDURE FUNCTIONS**

**AIM:**

To write a SQL program to work with cursor, procedure and functions.

**PROCEDURE:**

**Step 1**: Open Run SQL on Command line and connect to SQL

**Step 2:** Then work with database using SQL queries.

**PL/SQL PROCEDURE:**

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.

The procedure contains a header and a body.

* **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.
* **Body:** The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

**Syntax for creating procedure:**

CREATE [OR REPLACE] PROCEDURE procedure\_name

[ (parameter [,parameter]) ]

IS

[declaration\_section]

BEGIN

executable\_section

[EXCEPTION

exception\_section]

END [procedure\_name];

**TABLE QUERY:**

create table employee(emp\_id number(5)primary key, emp\_name varchar2(20), city varchar2(20), salary number(7), age number(5));

insert into employee values (1, 'Raju', 'Pdy', 800000, 20);

insert into employee values (2, 'Niteesh', 'Pdy', 790000, 21);

insert into employee values (3, 'Punith', 'AP', 750000, 20);

insert into employee values (4, 'Sidharth', 'MP', 650000, 21);

insert into employee values (5, 'Mantu', 'Delhi', 900000, 22);

**PROGRAM CODE:**

DECLARE

PROCEDURE pro

AS

BEGIN

dbms\_output.put\_line('It is working perfectly!');

END;

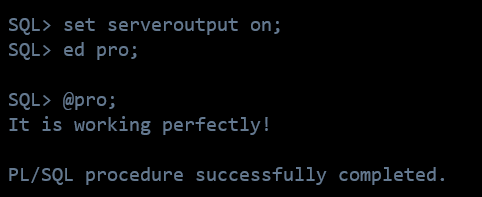
BEGIN

pro();

END;

/

**OUTPUT:**



**PL/SQL – CURSORS:**

A cursor is used to referred to a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

* Implicit Cursors
* Explicit Cursors

**IMPLICIT CURSOR:**

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement.

**1 %FOUND**

Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.

**2 %NOTFOUND**

The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

**3 %ISOPEN**

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

**4 %ROWCOUNT**

Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

**EXPLICIT CURSOR:**

Explicit cursors are programmer-defined cursors for gaining more control over the context area.

**The syntax for creating an explicit cursor is −**

CURSOR cursor\_name IS select\_statement;

**Working with an explicit cursor includes the following steps −**

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

**PROGRAM CODE:**

DECLARE

e\_id employee.emp\_id%type;

e\_name employee.emp\_name%type;

e\_city employee.city%type;

cursor e\_employee is

select emp\_id, emp\_name, city from employee;

begin

open e\_employee;

loop

fetch e\_employee into e\_id, e\_name, e\_city;

exit when e\_employee%notfound;

dbms\_output.put\_line(e\_id || ' ' || e\_name || ' ' || e\_city);

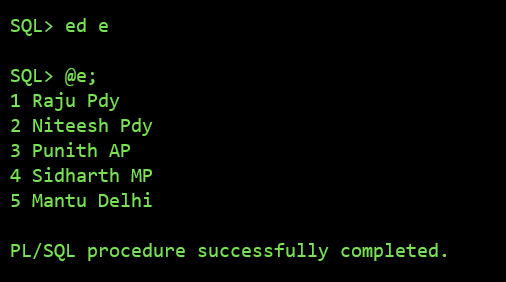
end loop;

close e\_employee;

end;

/

**OUTPUT:**



**PL/SQL FUNCTION:**

The PL/SQL Function is very similar to PL/SQL Procedure. The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value.

**Syntax to create a function:**

CREATE [OR REPLACE] FUNCTION function\_name [parameters]

[(parameter\_name [IN | OUT | IN OUT] type [, ...])]

RETURN return\_datatype

{IS | AS}

BEGIN

< function\_body >

END [function\_name];

**PROGRAM CODE:**

DECLARE

n number;

t number;

FUNCTION func

RETURN number IS

total number(2) := 0;

BEGIN

SELECT count(\*) into total

FROM employee;

RETURN total;

END;

BEGIN

n:=2;

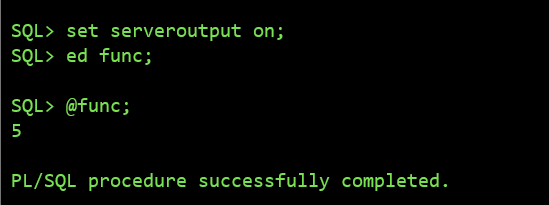
t:=func();

dbms\_output.put\_line(t);

END;

/

**OUTPUT:**



**RESULT:**

The queries for Procedure, Cursors and Functions were successfully executed and the output is noted.

**EX NO.: 07 TRIGGERS**

**AIM:**

To create and work with triggers.

**PROCEDURE:**

**Step 1**: Open Run SQL on Command line and connect to SQL

**Step 2:** Then work with database using SQL queries.

**Trigger:** A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

**Syntax:**

create trigger [trigger\_name]

[before | after]

{insert | update | delete}

on [table\_name]

[for each row]

[trigger\_body]

**PROGRAM CODE:**

SQL> CREATE OR REPLACE TRIGGER set\_default\_salary

2 BEFORE INSERT ON employee

3 FOR EACH ROW

4 BEGIN

5 :NEW.salary := 50000;

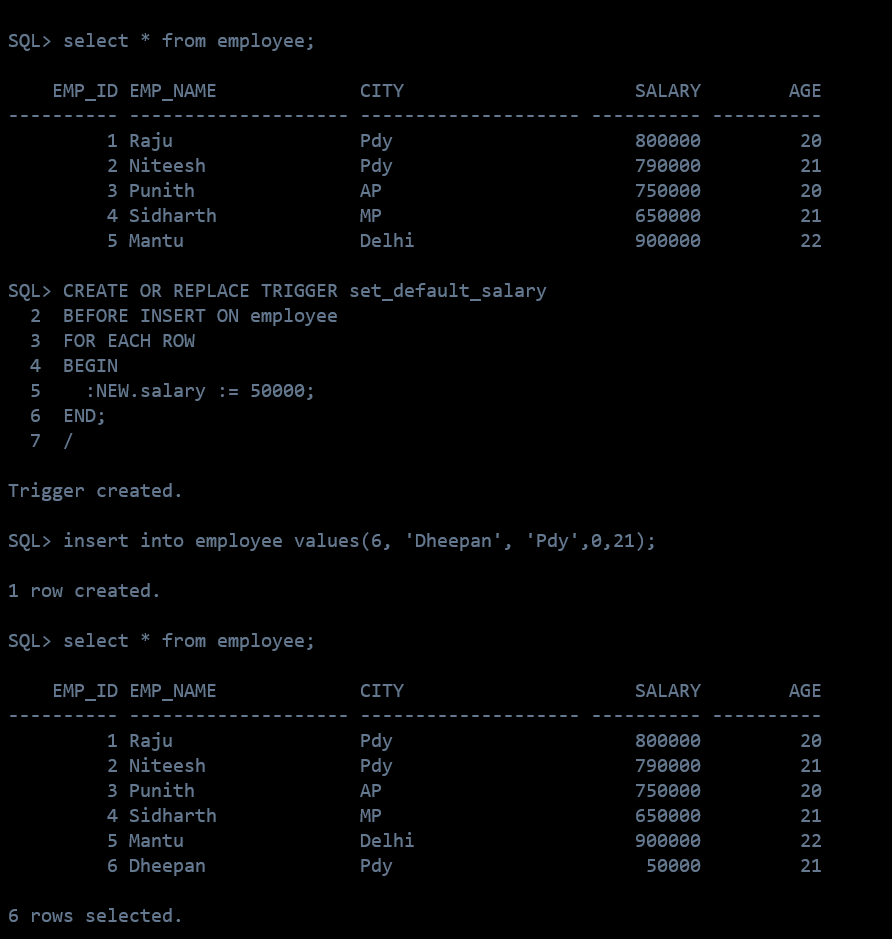
6 END;

7 /

This trigger program creates a trigger called **set\_default\_salary** that fires before an insert operation is performed on the **employee** table. The trigger sets the **salary** value of the new employee being inserted to 50000.

Note that the **BEFORE INSERT** clause specifies that the trigger should fire before the insert operation is performed, and the **FOR EACH ROW** clause specifies that the trigger should execute once for each row that is inserted. Finally, the **:NEW** keyword is used to reference the values being inserted into the table.

**OUTPUT:**



**RESULT:**

The queries for Triggers were successfully executed and the output is noted.

**APPLICATION PROJECT**

**BLOOD DONATION APP**

**ABSTRACT:**

* The blood donation website is a platform designed to encourage and facilitate blood donation.
* This website aims to educate the public about the importance of donating blood and to provide information about the donation process.
* It also serves as a means for individuals to register as blood donors and for blood banks to manage their inventory.
* The website features a user-friendly interface that allows donors to schedule appointments, track their donation history, and receive notifications about upcoming blood drives.
* Additionally, the website provides resources for individuals who may have questions or concerns about the donation process.
* Overall, the blood donation website is a valuable tool for promoting and supporting the lifesaving act of blood donation.

**MODULES:**

The project consists of 2 modules, which are

* Become a donor
* Need a blood donor

**SOFTWARE REQUIREMENTS:**

* Operating system – Windows 10
* Web Server : XAMPP [5.6.4]
* Database : MYSQL [5.0.21]
* Coding Language : Web Tech (HTML, CSS, Java Script, PHP)

**LIMITATION OF EXISTING SYSTEM:**

* Time - consuming
* Inefficient
* Limited reach
* Lack of privacy
* Safety concerns

**PROGRAM CODE:**

**//SAVE\_DATA**

<?php

$name=$\_POST['fullname'];

$number=$\_POST['mobileno'];

$email=$\_POST['emailid'];

$age=$\_POST['age'];

$gender=$\_POST['gender'];

$blood\_group=$\_POST['blood'];

$address=$\_POST['address'];

$conn=mysqli\_connect("localhost","root","","blood\_donation") or die("Connection error");

$sql= "INSERT INTO donor\_details(donor\_name,donor\_number,donor\_mail,donor\_age,donor\_gender,donor\_blood,donor\_address) values('{$name}','{$number}','{$email}','{$age}','{$gender}','{$blood\_group}','{$address}')";

$result=mysqli\_query($conn,$sql) or die("query unsuccessful.");

echo "<script>alert('Successfully Requested... ');</script>";

header("Location: http://localhost:7070/blood/index.php");

mysqli\_close($conn);

?>

**//INDEX.PHP**

<html>

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<meta name="description" content="">

<meta name="author" content="">

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.16.0/umd/popper.min.js"></script>

<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>

</head>

<body>

<div id="page-container" style="margin-top:50px; position: relative;min-height: 84vh;">

<div class="container">

<div id="content-wrap" style="padding-bottom:50px;">

<div class="row">

<div class="col-lg-6">

<h1 class="mt-4 mb-3">Need Blood</h1>

</div>

</div>

<form name="needblood" action="" method="post">

<div class="row">

<div class="col-lg-4 mb-4">

<div class="font-italic">Blood Group<span style="color:red">\*</span></div>

<div><select name="blood" class="form-control" required>

<option value=""selected disabled>Select</option>

<?php

include 'conn.php';

$sql= "select \* from blood";

$result=mysqli\_query($conn,$sql) or die("query unsuccessful.");

while($row=mysqli\_fetch\_assoc($result)){

?>

<option value=" <?php echo $row['blood\_id'] ?>"> <?php echo $row['blood\_group'] ?> </option>

<?php } ?>

</select>

</div>

</div>

<div class="col-lg-4 mb-4">

<div class="font-italic">Reason, why do you need blood?<span style="color:red">\*</span></div>

<div><textarea class="form-control" name="address" required></textarea></div></div>

</div>

<div class="row">

<div class="col-lg-1">

<div><input type="submit" name="search" class="btn btn-primary" value="Search" style="cursor:pointer"></div>

</div>

<div class="col-lg-4 mb-4">

Willing To DonateBlood,

<a href="donate\_blood.php">

click here to Donate

</a>

</div>

</div><div class="row">

<?php if(isset($\_POST['search'])){

$bg=$\_POST['blood'];

$sql= "select \* from donor\_details join blood where donor\_details.donor\_blood=blood.blood\_id AND donor\_blood='{$bg}' order by rand() limit 5";

$result=mysqli\_query($conn,$sql) or die("query unsuccessful.");

if(mysqli\_num\_rows($result)>0) {

while($row = mysqli\_fetch\_assoc($result)) {

?>

<div class="col-lg-4 col-sm-6 portfolio-item" ><br>

<div class="card" style="width:300px">

<img class="card-img-top" src="blood\_drop\_logo.jpg" alt="Card image" style="width:100%;height:300px">

<div class="card-body">

<h3 class="card-title"><?php echo $row['donor\_name']; ?></h3>

<p class="card-text">

<b>Blood Group : </b> <b><?php echo $row['blood\_group']; ?></b><br>

<b>Mobile No. : </b> <?php echo $row['donor\_number']; ?><br>

<b>Gender : </b><?php echo $row['donor\_gender']; ?><br>

<b>Age : </b> <?php echo $row['donor\_age']; ?><br>

<b>Address : </b> <?php echo $row['donor\_address']; ?><br>

</p>

</div>

</div>

</div>

<?php

}

}

else

{

echo '<div class="alert alert-danger">No Donor Found For your search Blood group </div>';

}

} ?>

</div>

</div>

</div>

</div>

</body>

</html>

**//BLOOD\_DONATION**

-- Database: `blood\_donation`

-- Table structure for table `blood`

CREATE TABLE `blood` (

`blood\_id` int(11) NOT NULL,

`blood\_group` varchar(10) NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1\_swedish\_ci;

-- Dumping data for table `blood`

INSERT INTO `blood` (`blood\_id`, `blood\_group`) VALUES

(1, 'B+'),

(2, 'B-'),

(3, 'A+'),

(4, 'O+'),

(5, 'O-'),

(6, 'A-'),

(7, 'AB+'),

(8, 'AB-');

-- Table structure for table `donor\_details`

CREATE TABLE `donor\_details` (

`donor\_id` int(11) NOT NULL,

`donor\_name` varchar(50) NOT NULL,

`donor\_number` varchar(10) NOT NULL,

`donor\_mail` varchar(50) DEFAULT NULL,

`donor\_age` int(60) NOT NULL,

`donor\_gender` varchar(10) NOT NULL,

`donor\_blood` varchar(10) NOT NULL,

`donor\_address` varchar(100) NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1\_swedish\_ci;

-- Dumping data for table `donor\_details`

INSERT INTO `donor\_details` (`donor\_id`, `donor\_name`, `donor\_number`, `donor\_mail`, `donor\_age`, `donor\_gender`, `donor\_blood`, `donor\_address`) VALUES

(7, 'gd', '9585718037', 'gd@gmail.com', 20, 'Male', ' 2', 'Pondy'),

(8, 'gs', '9585718037', 'gs@gmail.com', 18, 'Male', ' 2', 'Pdy'),

(9, 'gs', '9585718037', 'gs@gmail.com', 18, 'Male', ' 2', 'Pdy');

-- Indexes for dumped tables

-- Indexes for table `blood`

ALTER TABLE `blood`

ADD PRIMARY KEY (`blood\_id`);

-- Indexes for table `donor\_details`

ALTER TABLE `donor\_details`

ADD PRIMARY KEY (`donor\_id`);

-- AUTO\_INCREMENT for dumped tables

-- AUTO\_INCREMENT for table `blood`

ALTER TABLE `blood`

MODIFY `blood\_id` int(11) NOT NULL AUTO\_INCREMENT, AUTO\_INCREMENT=9;

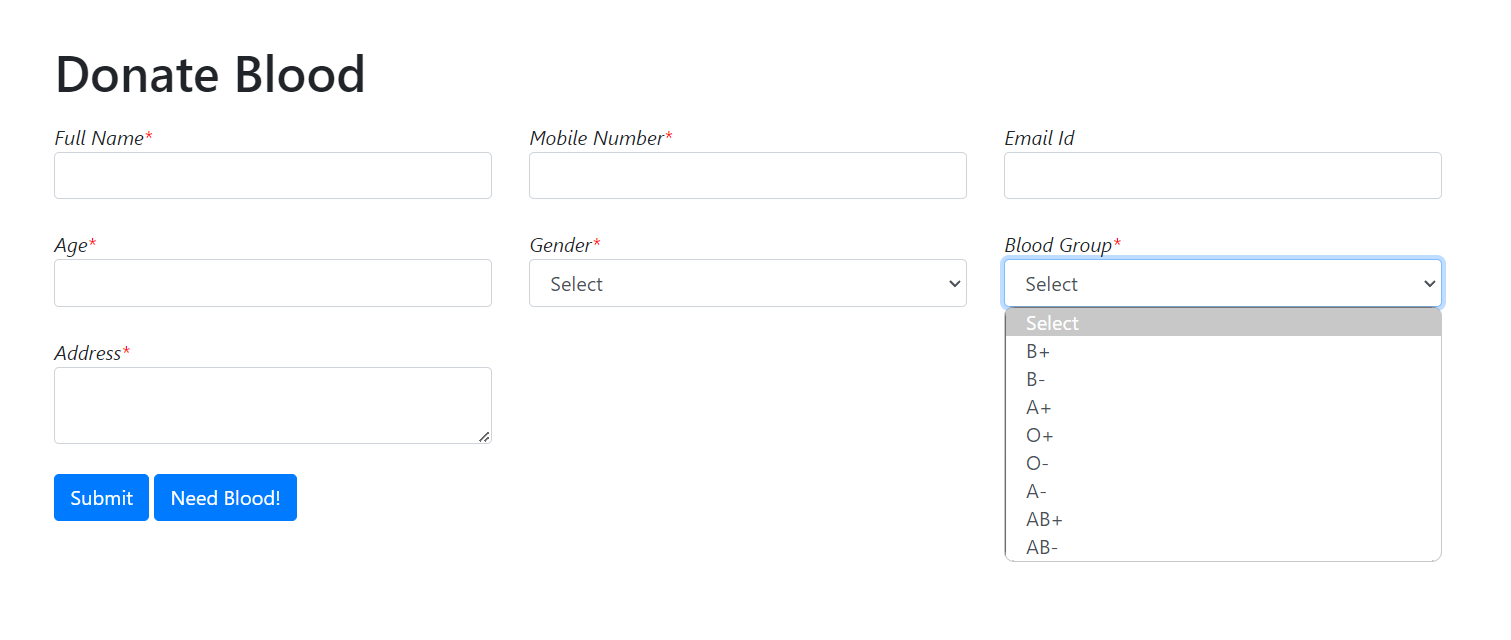
-- AUTO\_INCREMENT for table `donor\_details`

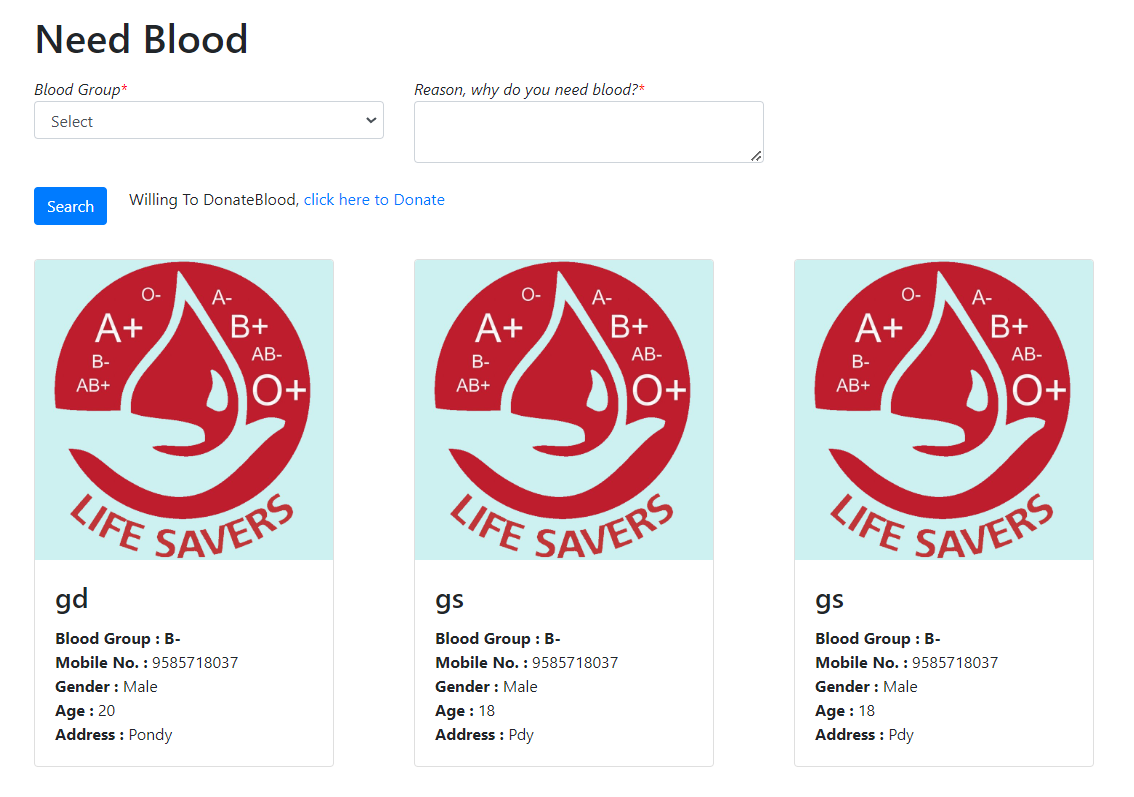
ALTER TABLE `donor\_details`

MODIFY `donor\_id` int(11) NOT NULL AUTO\_INCREMENT, AUTO\_INCREMENT=10;

COMMIT;

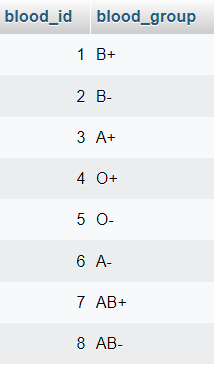
**OUTPUT:**

1. **DONATE BLOOD:**
2. **NEED BLOOD:**

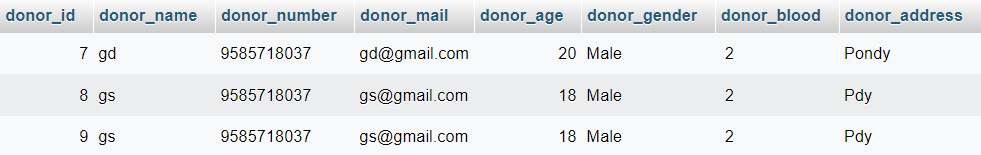


**DATABASE OUTPUT:**

**BLOOD GROUP TABLE:**

****

**BLOOD DONOR DETAILS:**

****