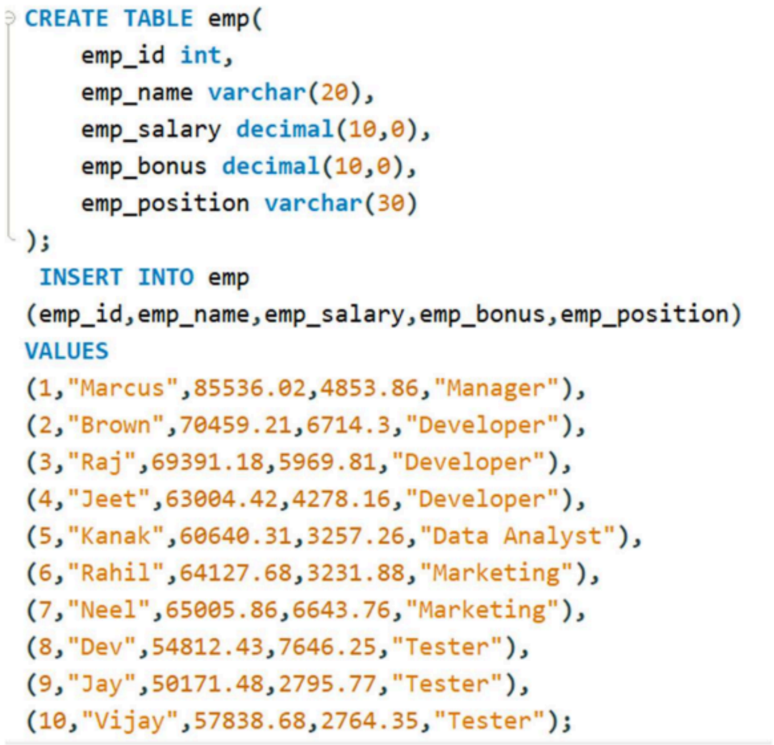
Practical 4

Aim :- To implement the logical , concatenation and like operators in SQL

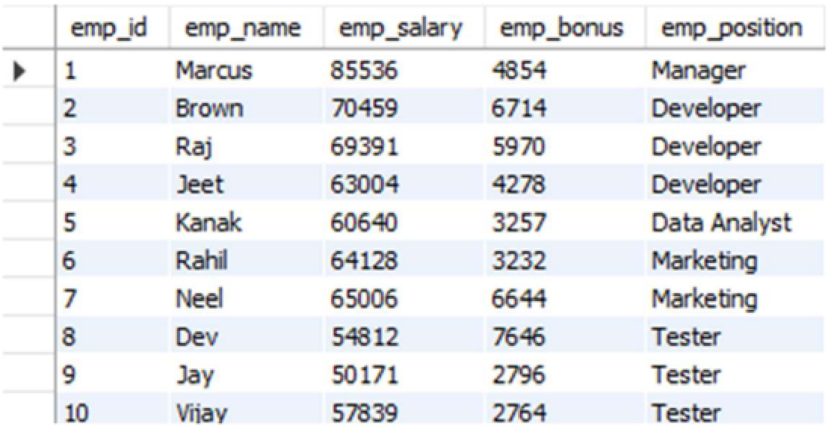
Theory :- Logical Operator

In SQL logical operator are used to combine multiple condition in where clause to filter data based on more complex criteria the three main logical operator in SQL are

- AND
- OR
- NOR



```
CREATE TABLE emp(
    emp_id int,
    emp_name varchar(20),
    emp_salary decimal(10,0),
    emp_bonus decimal(10,0),
    emp_position varchar(30)
);
 INSERT INTO emp
(emp_id,emp_name,emp_salary,emp_bonus,emp_position)
VALUES
(1,"Marcus",85536.02,4853.86,"Manager"),
(2,"Brown",70459.21,6714.3,"Developer"),
(3,"Raj",69391.18,5969.81,"Developer"),
(4,"Jeet",63004.42,4278.16,"Developer"),
(5,"Kanak",60640.31,3257.26,"Data Analyst"),
(6,"Rahil",64127.68,3231.88,"Marketing"),
(7,"Neel",65005.86,6643.76,"Marketing"),
(8,"Dev",54812.43,7646.25,"Tester"),
(9,"Jay",50171.48,2795.77,"Tester"),
(10,"Vijay",57838.68,2764.35,"Tester");
```

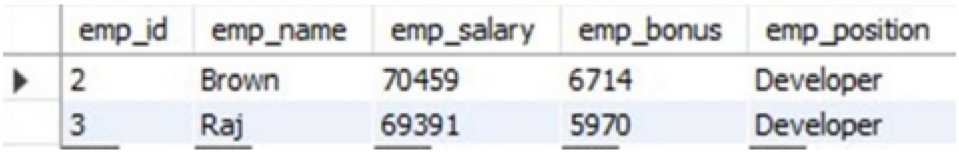


emp_id	emp_name	emp_salary	emp_bonus	emp_position
1	Marcus	85536	4854	Manager
2	Brown	70459	6714	Developer
3	Raj	69391	5970	Developer
4	Jeet	63004	4278	Developer
5	Kanak	60640	3257	Data Analyst
6	Rahil	64128	3232	Marketing
7	Neel	65006	6644	Marketing
8	Dev	54812	7646	Tester
9	Jay	50171	2796	Tester
10	Vijay	57839	2764	Tester

1. AND

Select * from emp where emp_position ='Developer' and

emp_bonus > 5000 ;

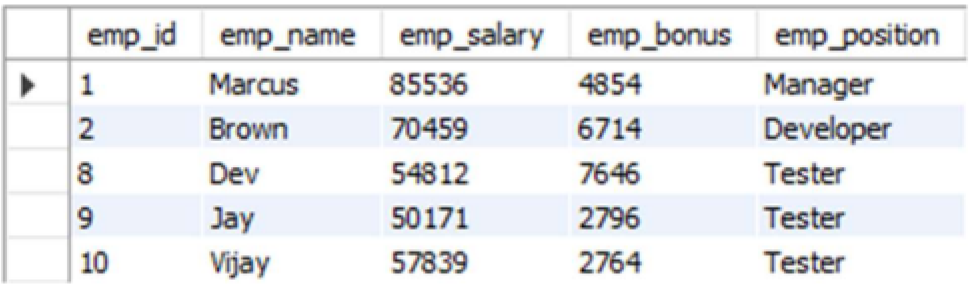


emp_id	emp_name	emp_salary	emp_bonus	emp_position
2	Brown	70459	6714	Developer
3	Raj	69391	5970	Developer

2. Or

select * from emp where emp_posistion='Tester' or

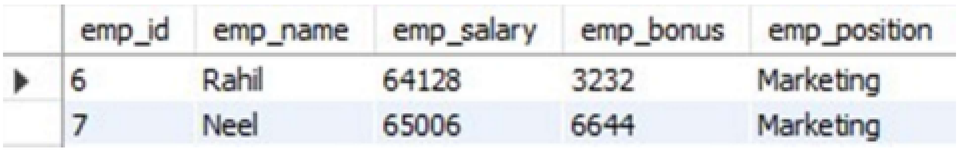
emp_salary > 70000 ;



emp_id	emp_name	emp_salary	emp_bonus	emp_position
1	Marcus	85536	4854	Manager
2	Brown	70459	6714	Developer
8	Dev	54812	7646	Tester
9	Jay	50171	2796	Tester
10	Vijay	57839	2764	Tester

3. Not

Select * From emp where not emp_position <> "Marketing" ;



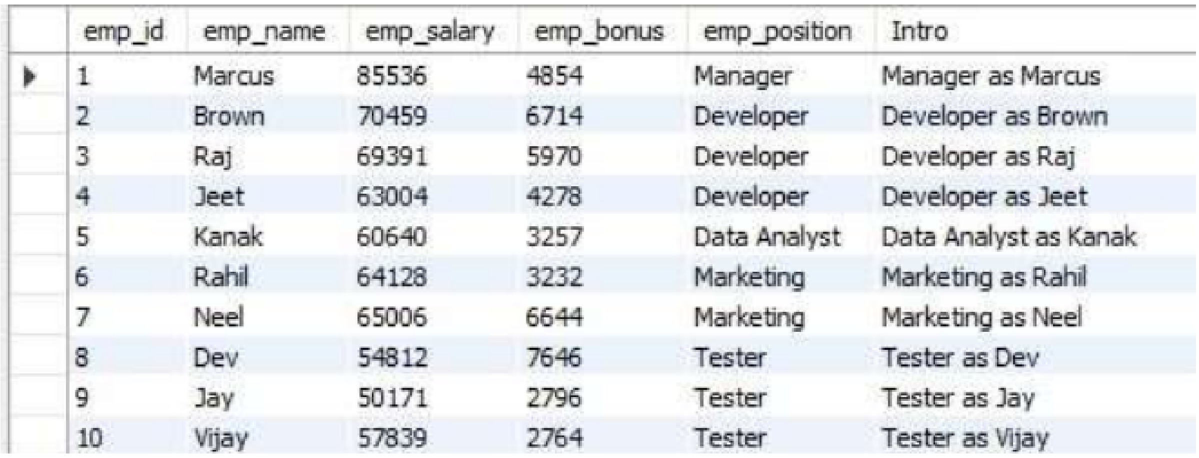
emp_id	emp_name	emp_salary	emp_bonus	emp_position
6	Rahil	64128	3232	Marketing
7	Neel	65006	6644	Marketing

Concatenation -

In SQL concatenation refers to the process of combining two or more string into a single string that are commonly used for string concatenation . Here 's how you can use it : -

Select * , Concat (emp_position,' as ', emp_name)

As “Intro” From emp;



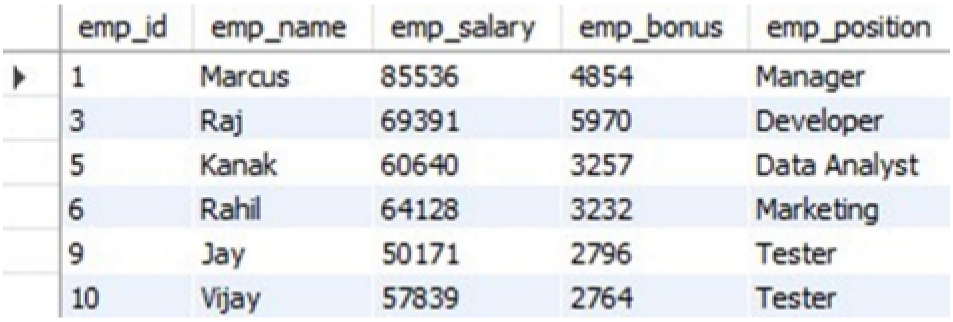
emp_id	emp_name	emp_salary	emp_bonus	emp_position	Intro
1	Marcus	85536	4854	Manager	Manager as Marcus
2	Brown	70459	6714	Developer	Developer as Brown
3	Raj	69391	5970	Developer	Developer as Raj
4	Jeet	63004	4278	Developer	Developer as Jeet
5	Kanak	60640	3257	Data Analyst	Data Analyst as Kanak
6	Rahil	64128	3232	Marketing	Marketing as Rahil
7	Neel	65006	6644	Marketing	Marketing as Neel
8	Dev	54812	7646	Tester	Tester as Dev
9	Jay	50171	2796	Tester	Tester as Jay
10	Vijay	57839	2764	Tester	Tester as Vijay

Like -

In SQL the ‘LIKE’ operator is used in a ‘where’ clause to search for a specific pattern in a column it is often used with wild card character to make pattern Here are some example are

- “R%” start with R
- %RA% specific pattern RA
- %a end with a

Select * from where emp_name like ‘%a%’ ;



emp_id	emp_name	emp_salary	emp_bonus	emp_position
1	Marcus	85536	4854	Manager
3	Raj	69391	5970	Developer
5	Kanak	60640	3257	Data Analyst
6	Rahil	64128	3232	Marketing
9	Jay	50171	2796	Tester
10	Vijay	57839	2764	Tester

Conclusion : Hence we performed this practical and fire the Query and manipulate the table data using logical , concatenation like operator successfully

Practical 5

Aim :- To implement the SQL Constraints in SQL commands

Theory :- Constraints are the rules that we can apply on the type of data in a table. That's we can specify the limit on the type of data that in a particle column in the table using constraints.

The available SQL constraints are :

NOT NULL :- This constraint tells that we cannot store a null value in a column. That's, If a column is specified as NOT NULL then we will not be able to store null in this particular column anymore.

UNIQUE : This constraint when specified with a column, tells that all the values in the column must be unique. That is used values in any row of a column must not be repeated.

PRIMARY KEY : Primary Key is field which can uniquely identify each row in a table. And this constraint is used to specify a field in a table as primary key.

FOREIGN KEY : Foreign Key is a field which can uniquely identify each row in a another table. And this constant is used to specify a field as foreign key.

CHECK : This constraint helps to validate the values of the column to meet a particular condition. That is, it helps to ensure that the value stored in a column meets a specific condition.

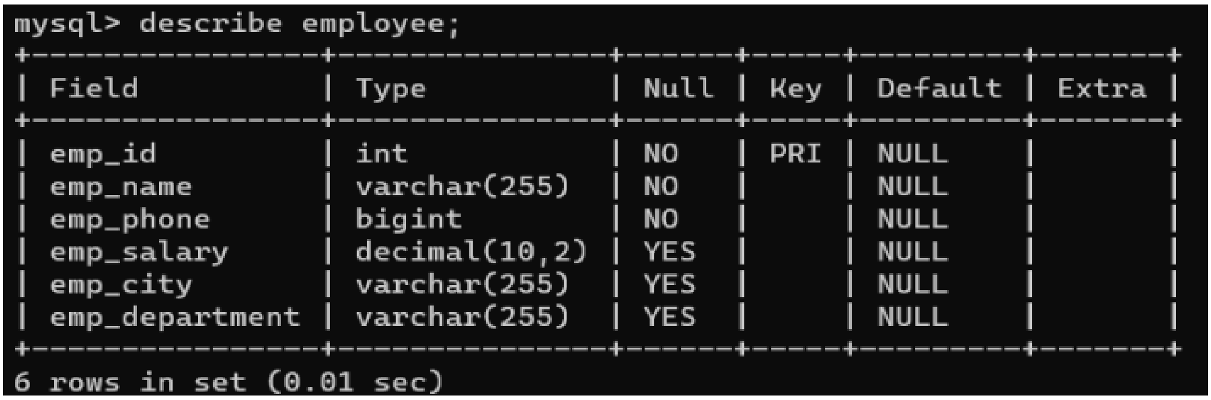
DEFAULT : This constraints specifies a default value for the column when no value is specified by the user.

Queries:-

Create Table using Primary Key, Not Null and Unique constraints.

Query: CREATE TABLE Employee (

```
  emp_id INT NOT NULL PRIMARY KEY,

  emp_name VARCHAR(255) NOT NULL,

  emp_phone BIGINT NOT NULL,

  emp_salary DECIMAL(10,2),

  emp_city VARCHAR(265),

  emp_department VARCHAR(255),

  CONSTRAINT UC_Employee UNIQUE (emp_id, emp_phone)

);
```



```
mysql> describe employee;
+----------------+---------------+------+-----+---------+-------+
| Field          | Type          | Null | Key | Default | Extra |
+----------------+---------------+------+-----+---------+-------+
| emp_id         | int           | NO   | PRI | NULL    |       |
| emp_name       | varchar(255)  | NO   |     | NULL    |       |
| emp_phone      | bigint        | NO   |     | NULL    |       |
| emp_salary     | decimal(10,2) | YES  |     | NULL    |       |
| emp_city       | varchar(255)  | YES  |     | NULL    |       |
| emp_department | varchar(255)  | YES  |     | NULL    |       |
+----------------+---------------+------+-----+---------+-------+
6 rows in set (0.01 sec)
```

Insert Data into Employe Table :

```
Query: INSERT INTO Employee (emp_id, emp_name, emp_phone,
emp_salary, emp_city, emp_department)

VALUES

  (101, 'Pronay', 1111122222, 90000.00, 'Bonglose', 'IT'),

  (102, 'Gaurday', 2222233333, 80000.00, 'Pune', 'Finances'),

  (103, 'Whomedeci', 3333344444, 85000.00, 'Mumbai', 'HR'),

  (104, 'Shubhankar', 4444455555, 75000.00, 'Delhi', 'Marketing'),
```

(105, 'Rohit', 5555566666, 65000.00, 'Chennai', 'Sales'),

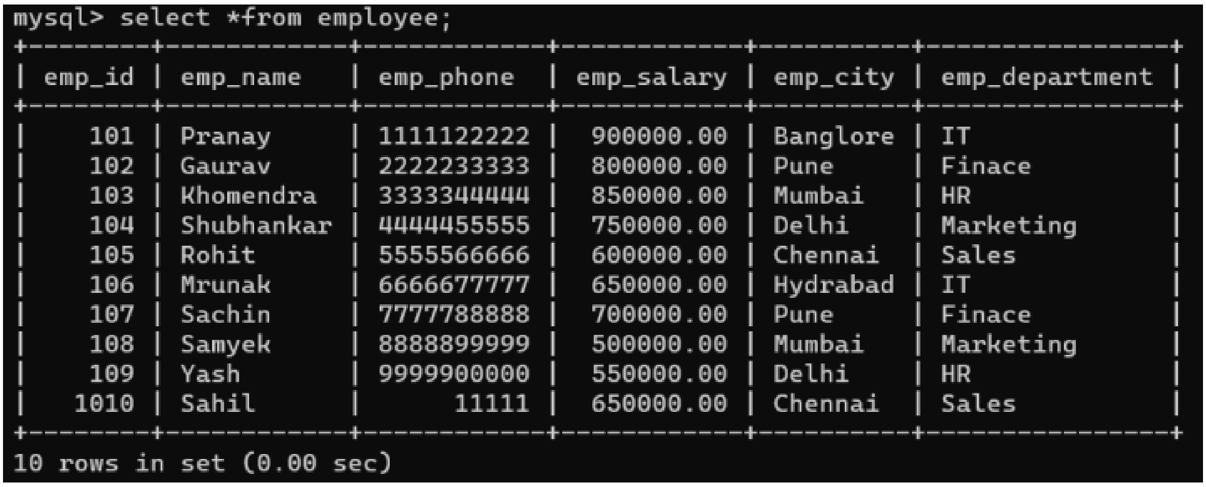
(106, 'Mounak', 6666677777, 65000.00, 'Hyderabad', 'IT'),

(107, 'Sachin', 7777788888, 70000.00, 'Pune', 'Finance'),

(108, 'Sonyek', 8888899999, 50000.00, 'Mumbai', 'Marketing'),

(109, 'Yash', 9999900000, 55000.00, 'Delhi', 'HR'),

(110, 'Sahil', 0000011111, 65000.00, 'Chennai', 'Sales');



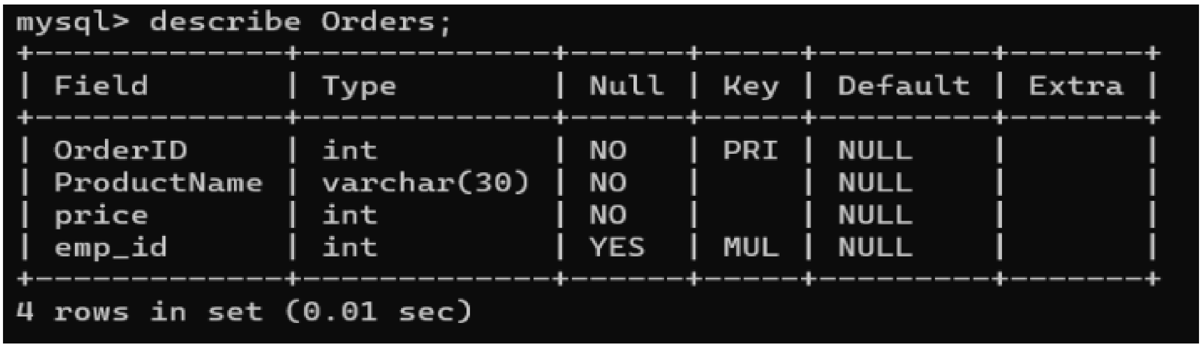
```
mysql> select *from employee;
+--------+------------+------------+------------+----------+----------------+
| emp_id | emp_name   | emp_phone  | emp_salary | emp_city | emp_department |
+--------+------------+------------+------------+----------+----------------+
|    101 | Pranay     | 1111122222 |  900000.00 | Banglore | IT             |
|    102 | Gaurav     | 2222233333 |  800000.00 | Pune     | Finace         |
|    103 | Khomendra  | 3333344444 |  850000.00 | Mumbai   | HR             |
|    104 | Shubhankar | 4444455555 |  750000.00 | Delhi    | Marketing      |
|    105 | Rohit      | 5555566666 |  600000.00 | Chennai  | Sales          |
|    106 | Mrunak     | 6666677777 |  650000.00 | Hydrabad | IT             |
|    107 | Sachin     | 7777788888 |  700000.00 | Pune     | Finace         |
|    108 | Samyek     | 8888899999 |  500000.00 | Mumbai   | Marketing      |
|    109 | Yash       | 9999900000 |  550000.00 | Delhi    | HR             |
|   1010 | Sahil      |      11111 |  650000.00 | Chennai  | Sales          |
+--------+------------+------------+------------+----------+----------------+
10 rows in set (0.00 sec)
```

Create table using Primary and Foreign key constraints.

Query: CREATE TABLE Orders (

OrderID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,

ProductName VARCHAR(30) NOT NULL, Price INT NOT NULL, emp_id INT NOT NULL, FOREIGN KEY (emp_id) REFERENCES Employee(emp_id));



```
mysql> describe Orders;
+-------------+-------------+------+-----+---------+-------+
| Field       | Type        | Null | Key | Default | Extra |
+-------------+-------------+------+-----+---------+-------+
| OrderID     | int         | NO   | PRI | NULL    |       |
| ProductName | varchar(30) | NO   |     | NULL    |       |
| price       | int         | NO   |     | NULL    |       |
| emp_id      | int         | YES  | MUL | NULL    |       |
+-------------+-------------+------+-----+---------+-------+
4 rows in set (0.01 sec)
```

Insert Data into orders Table.

Query: INSERT INTO Orders (OrderID, ProductName, Price, emp_id)

VALUES

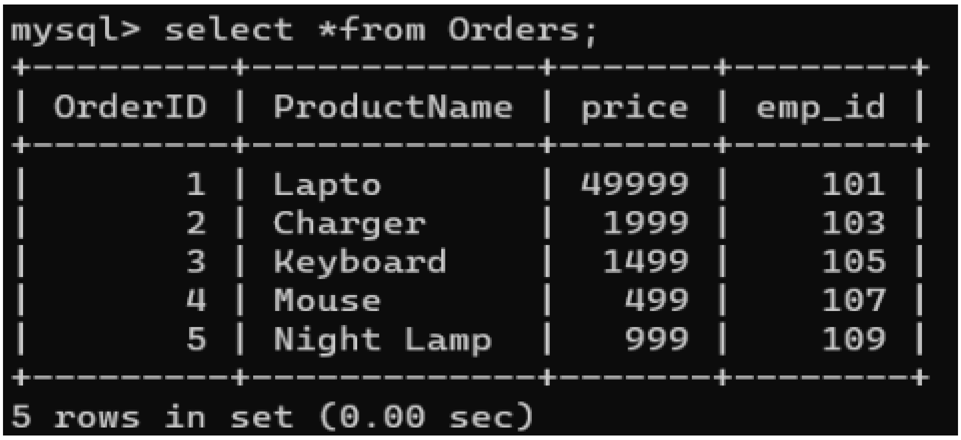
(1, 'Laptop', 49999, 101),

(12, 'Charger', 1999, 103),

(3, 'Keyboard', 1499, 105),

(4, 'Mouse', 499, 107),

(5, 'Night Lamp', 999, 109);



```
mysql> select *from Orders;
+---------+-------------+-------+--------+
| OrderID | ProductName | price | emp_id |
+---------+-------------+-------+--------+
|       1 | Lapto       | 49999 |    101 |
|       2 | Charger     |  1999 |    103 |
|       3 | Keyboard    |  1499 |    105 |
|       4 | Mouse       |   499 |    107 |
|       5 | Night Lamp  |   999 |    109 |
+---------+-------------+-------+--------+
5 rows in set (0.00 sec)
```

Conclusion: We have successfully implement the SQL constraints Commands in MySQL databases

Practical 6

Aim :- To implement the aggregate function in SQL commands

Theory :-

An SQL aggregate function calculates on a set of values and returns a single value. For example, the average function (AVG) takes a list of values and returns the average.

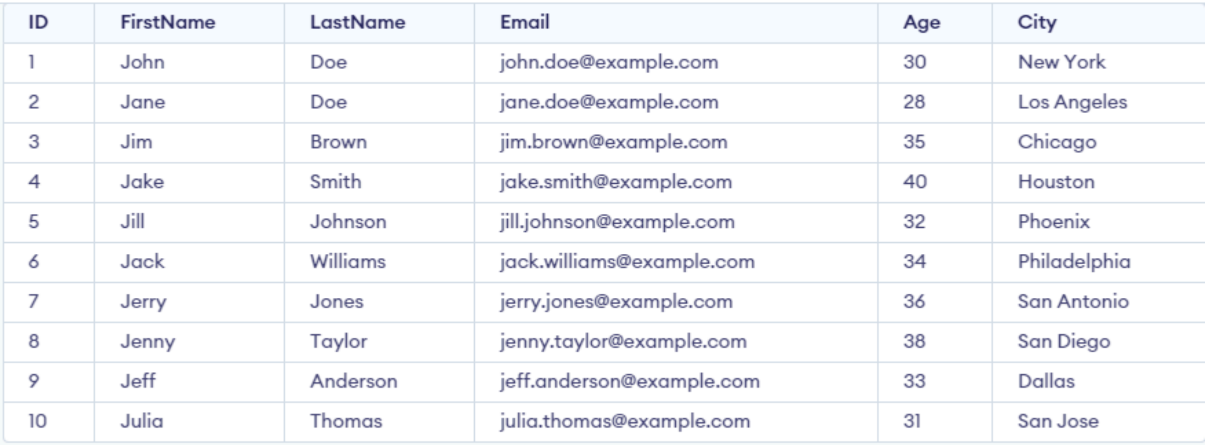
Because an aggregate function operates on a set of values, it is often used with the GROUP BY clause of the SELECT statement. The GROUP BY clause divides the result set into groups of values and the aggregate function returns a single value for each group.

Aggregate Function in SQL are : -

- AVG() – returns the average of a set.
- COUNT() – returns the number of items in a set.
- MAX() – returns the maximum value in a set.
- MIN() – returns the minimum value in a set
- SUM() – returns the sum of all or distinct values in a set

Query : -

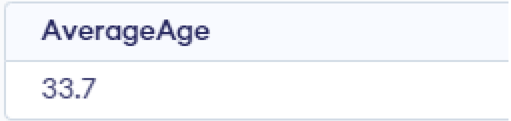
```
CREATE TABLE Employees (

   ID INT PRIMARY KEY,

   FirstName VARCHAR(50),

   LastName VARCHAR(50),

   Email VARCHAR(100),

   Age INT,

   City VARCHAR(50)

);

INSERT INTO Employees (ID, FirstName, LastName, Email, Age, City)

VALUES

(1, 'John', 'Doe', 'john.doe@example.com', 30, 'New York'),
(2, 'Jane', 'Doe', 'jane.doe@example.com', 28, 'Los Angeles'),

(3, 'Jim', 'Brown', 'jim.brown@example.com', 35, 'Chicago'),

(4, 'Jake', 'Smith', 'jake.smith@example.com', 40, 'Houston'),

(5, 'Jill', 'Johnson', 'jill.johnson@example.com', 32, 'Phoenix'),

(6, 'Jack', 'Williams', 'jack.williams@example.com', 34, 'Philadelphia'),

(7, 'Jerry', 'Jones', 'jerry.jones@example.com', 36, 'San Antonio'),

(8, 'Jenny', 'Taylor', 'jenny.taylor@example.com', 38, 'San Diego'),

(9, 'Jeff', 'Anderson', 'jeff.anderson@example.com', 33, 'Dallas'),

(10, 'Julia', 'Thomas', 'julia.thomas@example.com', 31, 'San Jose');
```



ID	FirstName	LastName	Email	Age	City
1	John	Doe	john.doe@example.com	30	New York
2	Jane	Doe	jane.doe@example.com	28	Los Angeles
3	Jim	Brown	jim.brown@example.com	35	Chicago
4	Jake	Smith	jake.smith@example.com	40	Houston
5	Jill	Johnson	jill.johnson@example.com	32	Phoenix
6	Jack	Williams	jack.williams@example.com	34	Philadelphia
7	Jerry	Jones	jerry.jones@example.com	36	San Antonio
8	Jenny	Taylor	jenny.taylor@example.com	38	San Diego
9	Jeff	Anderson	jeff.anderson@example.com	33	Dallas
10	Julia	Thomas	julia.thomas@example.com	31	San Jose

1. AVG -

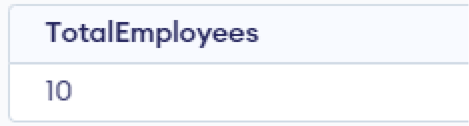
   ```
   SELECT AVG(Age) AS AverageAge

   FROM Employees;
   ```

AverageAge
33.7

2. COUNT

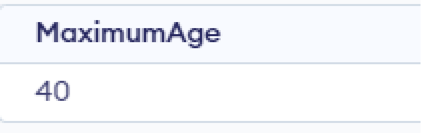
   ```
   SELECT COUNT(*) AS TotalEmployees

   FROM Employees;
   ```



TotalEmployees
10

3. MAX

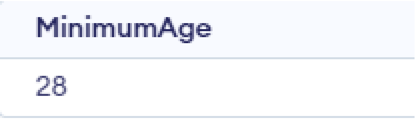
```
SELECT MAX(Age) AS MaximumAge

FROM Employees;
```



MaximumAge
40

4. MIN

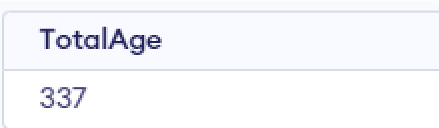
```
SELECT MIN(Age) AS MinimumAge

FROM Employees;
```



MinimumAge
28

5. SUM

```
 SELECT SUM(Age) AS TotalAge

FROM Employees;
```



TotalAge
337

Conclusion : We'd successfully learned about aggregate function in SQL commands and implement them in SQL database

Practical 7

Aim :- To implement the SQL clauses in SQL commands

Theory :-

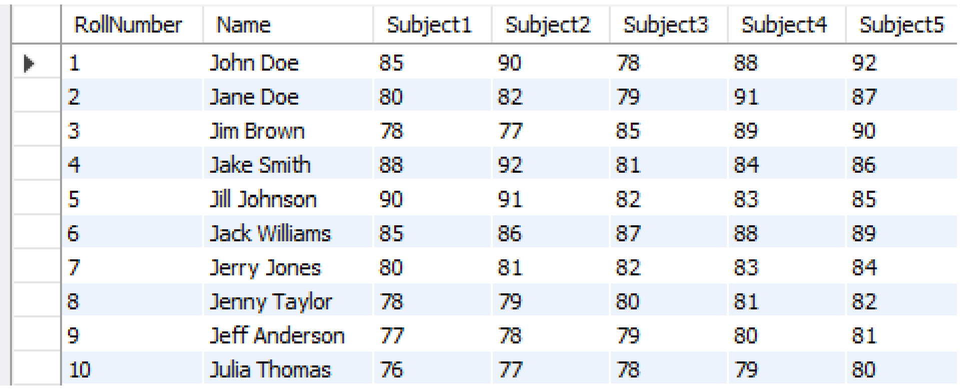
What are Clauses in SQL?

Clauses are in-built functions available to us in SQL. With the help of clauses, we can deal with data easily stored in the table.

Clauses help us filter and analyse data quickly. When we have large amounts of data stored in the database, we use Clauses to query and get data required by the user.

Query : -

```
CREATE TABLE Students (

   RollNumber INT PRIMARY KEY,

   Name VARCHAR(50),

   Subject1 INT,

   Subject2 INT,

   Subject3 INT,

   Subject4 INT,

   Subject5 INT

);

INSERT INTO Students (RollNumber, Name, Subject1, Subject2, Subject3, Subject4, Subject5)

VALUES

(1, 'John Doe', 85, 90, 78, 88, 92),

(2, 'Jane Doe', 80, 82, 79, 91, 87),
(3, 'Jim Brown', 78, 77, 85, 89, 90),

(4, 'Jake Smith', 88, 92, 81, 84, 86),

(5, 'Jill Johnson', 90, 91, 82, 83, 85),

(6, 'Jack Williams', 85, 86, 87, 88, 89),

(7, 'Jerry Jones', 80, 81, 82, 83, 84),

(8, 'Jenny Taylor', 78, 79, 80, 81, 82),

(9, 'Jeff Anderson', 77, 78, 79, 80, 81),

(10, 'Julia Thomas', 76, 77, 78, 79, 80);
```

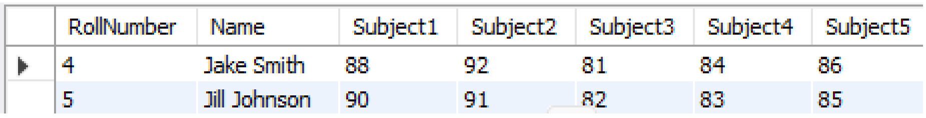


RollNumber	Name	Subject1	Subject2	Subject3	Subject4	Subject5
1	John Doe	85	90	78	88	92
2	Jane Doe	80	82	79	91	87
3	Jim Brown	78	77	85	89	90
4	Jake Smith	88	92	81	84	86
5	Jill Johnson	90	91	82	83	85
6	Jack Williams	85	86	87	88	89
7	Jerry Jones	80	81	82	83	84
8	Jenny Taylor	78	79	80	81	82
9	Jeff Anderson	77	78	79	80	81
10	Julia Thomas	76	77	78	79	80

1. **Where Clause in SQL**

We use the WHERE clause to specify conditionals in our SQL query. Where clauses can be used in the update and delete statements as well as to perform operations on the desired data.

```
SELECT *

FROM Students

WHERE Subject1 > 85;
```



RollNumber	Name	Subject1	Subject2	Subject3	Subject4	Subject5
4	Jake Smith	88	92	81	84	86
5	Jill Johnson	90	91	82	83	85

2. **Having Clause in SQL**

We use Having use for filtering of query results based on aggregate functions and groupings, which cannot be achieved using the WHERE clause that is used to filter individual

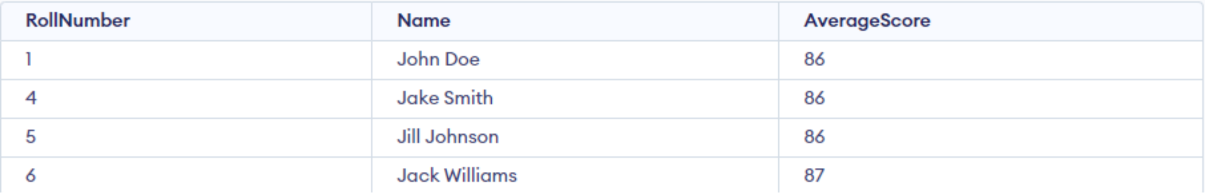
SELECT RollNumber, Name,

(Subject1 + Subject2 + Subject3 + Subject4 + Subject5)/5 as AverageScore

FROM Students

GROUP BY RollNumber, Name

HAVING (Subject1 + Subject2 + Subject3 + Subject4 + Subject5)/5 > 85;



RollNumber	Name	AverageScore
1	John Doe	86
4	Jake Smith	86
5	Jill Johnson	86
6	Jack Williams	87

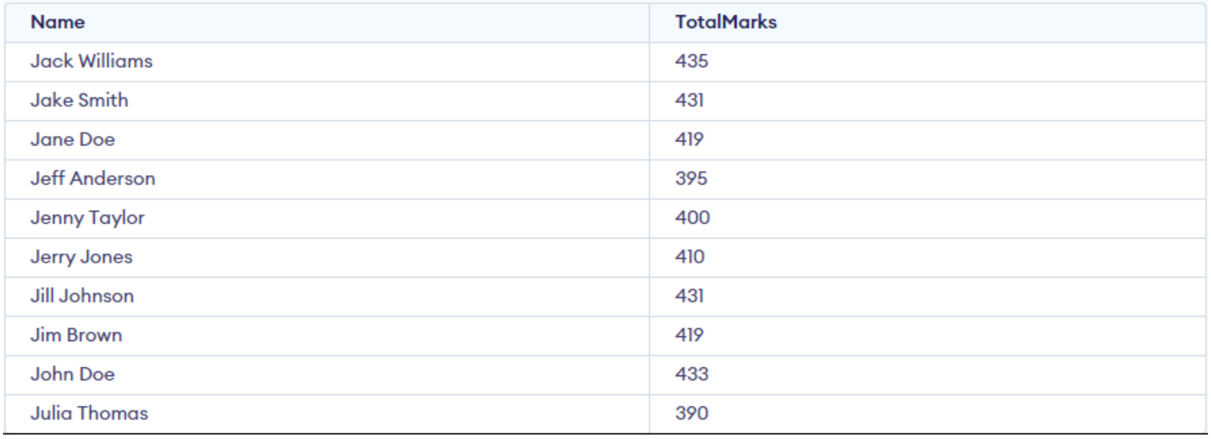
3. **Group By Clause in SQL**

We use order by clause to get the summary of data in rows and is mostly taken in usage with the aggregate functions like Count, Sum, etc.

SELECT

Name, SUM(Subject1 + Subject2 + Subject3 + Subject4 + Subject5)

AS TotalMarks FROM Students GROUP BY Name;

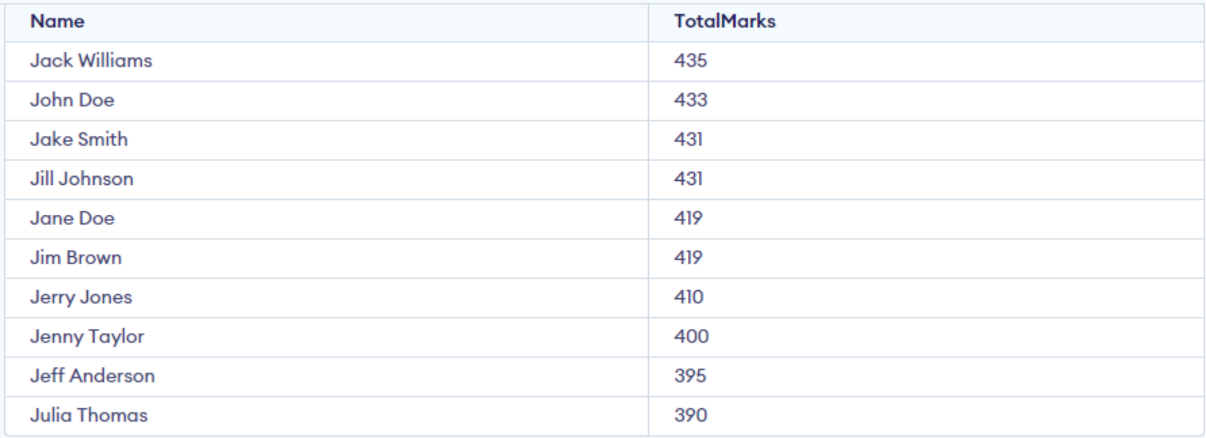


Name	TotalMarks
Jack Williams	435
Jake Smith	431
Jane Doe	419
Jeff Anderson	395
Jenny Taylor	400
Jerry Jones	410
Jill Johnson	431
Jim Brown	419
John Doe	433
Julia Thomas	390

4. **Order By Clause in SQL**

We use order by clause to sort data in ascending or descending order as required by the user. By default, the data is sorted in ascending order.

```
SELECT

Name , (Subject1 + Subject2 + Subject3 + Subject4 + Subject5) AS TotalMarks

FROM Students

ORDER BY TotalMarks DESC;
```



Name	TotalMarks
Jack Williams	435
John Doe	433
Jake Smith	431
Jill Johnson	431
Jane Doe	419
Jim Brown	419
Jerry Jones	410
Jenny Taylor	400
Jeff Anderson	395
Julia Thomas	390

Conclusion :- We successfully studied and implement the SQL clauses in SQL commands

Practical 8

Aim :- To implement SQL subquery in SQL commands

Theory :-

In SQL, a subquery is a query nested within another query. It simplifies building intricate queries to retrieve data that meets specific conditions from various tables. Subqueries are often challenging for beginners, but with practice, they become an essential tool for more complex data analysis

Use Cases

Here are some common use cases for SQL subqueries1:

Filtering data: Use subqueries in the WHERE clause to filter data based on specific conditions, making your queries more dynamic.

Nested aggregations: Employ subqueries to perform aggregations within aggregations, allowing for more complex calculations.

Checking existence: Determine whether a specific value exists in another table using subqueries with the EXISTS or IN operator.

Correlated subqueries: Create subqueries that reference columns from the outer query, enabling context-aware filtering.

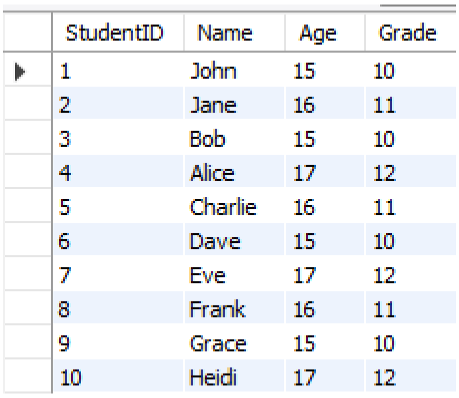
Subquery in SELECT clause: Include a subquery in the SELECT clause to retrieve a single value or set of values that can be used in the main query.

Subquery in FROM clause: Use a subquery in the FROM clause to create a temporary table, allowing for more complex joins.

Query : -

Table

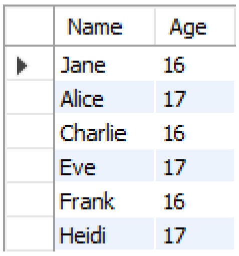
```
CREATE TABLE Students (

   StudentID INT PRIMARY KEY,

   Name VARCHAR(50),

   Age INT,
   Grade INT
);
INSERT INTO Students (StudentID, Name, Age, Grade)
VALUES (1, 'John', 15, 10), (2, 'Jane', 16, 11), (3, 'Bob', 15, 10),
       (4, 'Alice', 17, 12), (5, 'Charlie', 16, 11), (6, 'Dave', 15, 10),
       (7, 'Eve', 17, 12), (8, 'Frank', 16, 11), (9, 'Grace', 15, 10),
       (10, 'Heidi', 17, 12);
```



	StudentID	Name	Age	Grade
▶	1	John	15	10
	2	Jane	16	11
	3	Bob	15	10
	4	Alice	17	12
	5	Charlie	16	11
	6	Dave	15	10
	7	Eve	17	12
	8	Frank	16	11
	9	Grace	15	10
	10	Heidi	17	12

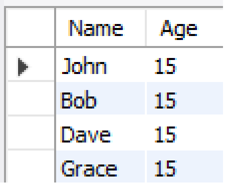
Subqueries ;-

1.
   ```
   SELECT Name, Age
    FROM Students
   WHERE Age > (SELECT AVG(Age) FROM Students);
   ```



	Name	Age
▶	Jane	16
	Alice	17
	Charlie	16
	Eve	17
	Frank	16
	Heidi	17

2.
   ```
   SELECT Name, Age
   FROM Students
   WHERE Age = (SELECT MIN(Age) FROM Students);
   ```

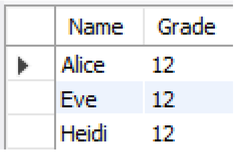


Name	Age
John	15
Bob	15
Dave	15
Grace	15

3. SELECT Name, Grade

   FROM Students

   WHERE Grade = (SELECT MAX(Grade) FROM Students);



Name	Grade
Alice	12
Eve	12
Heidi	12

Conclusion :- we successfully studied about subquery in SQL and Implemented on a database

Practical 9

Aim :- to implement SQL joins in SQL commands

Theory:-

SQL joins are used to combine rows from two or more tables based on a related column between them12. There are several types of SQL joins: INNER JOIN, LEFT JOIN, RIGHT JOIN

1. INNER JOIN: This is the most common type of join. It returns records that have matching values in both tables
2. LEFT JOIN (or LEFT OUTER JOIN): This join returns all the rows from the left table and the matched rows from the right table. If there is no match, the result is NULL on the right side
3. RIGHT JOIN (or RIGHT OUTER JOIN): This join returns all the rows from the right table and the matched rows from the left table. If there is no match, the result is NULL on the left side

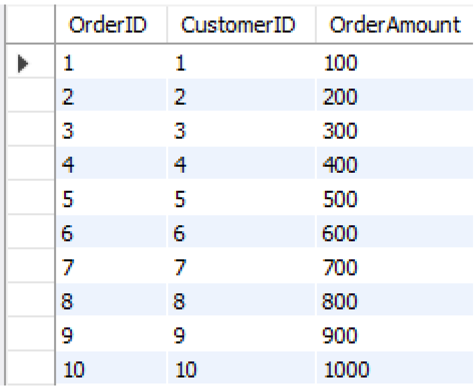
Here are some common use cases for SQL joins:

- Combining data: Use joins to combine data from two or more tables.
- Data analysis: Employ joins to analyse data from different tables together.
- Data integrity: Determine whether there are any orphan records in your database (i.e., records that reference other records that no longer exist).
- Data cleaning: Use joins to identify and clean duplicate records, incorrect references, and other common data issues.

Query :-

Table 1:-

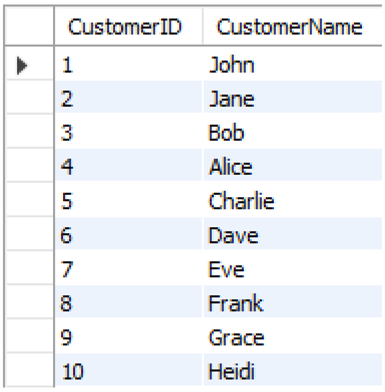
```
CREATE TABLE Orders (

   OrderID INT PRIMARY KEY,

   CustomerID INT,
    OrderAmount INT

);

INSERT INTO Orders (OrderID, CustomerID, OrderAmount)

VALUES (1, 1, 100), (2, 2, 200), (3, 3, 300), (4, 4, 400), (5, 5, 500),

       (6, 6, 600), (7, 7, 700), (8, 8, 800), (9, 9, 900), (10, 10, 1000);
```



	OrderID	CustomerID	OrderAmount
▶	1	1	100
	2	2	200
	3	3	300
	4	4	400
	5	5	500
	6	6	600
	7	7	700
	8	8	800
	9	9	900
	10	10	1000

Table 2

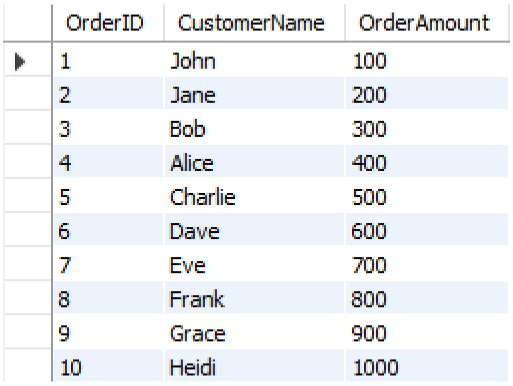
```
CREATE TABLE Customers (

    CustomerID INT PRIMARY KEY,

    CustomerName VARCHAR(50)

);

INSERT INTO Customers (CustomerID, CustomerName)

VALUES (1, 'John'), (2, 'Jane'), (3, 'Bob'), (4, 'Alice'), (5, 'Charlie'),

       (6, 'Dave'), (7, 'Eve'), (8, 'Frank'), (9, 'Grace'), (10, 'Heidi');
```



	CustomerID	CustomerName
▶	1	John
	2	Jane
	3	Bob
	4	Alice
	5	Charlie
	6	Dave
	7	Eve
	8	Frank
	9	Grace
	10	Heidi

1. Inner Join

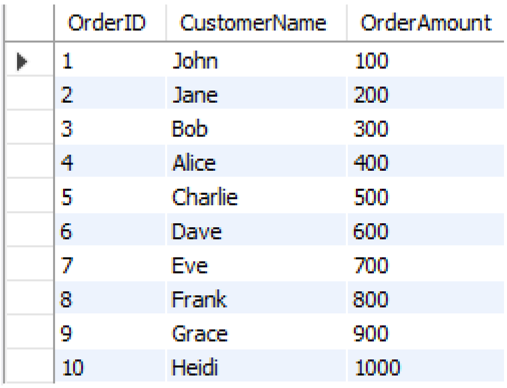
```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderAmount

FROM Orders

INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```



	OrderID	CustomerName	OrderAmount
▶	1	John	100
	2	Jane	200
	3	Bob	300
	4	Alice	400
	5	Charlie	500
	6	Dave	600
	7	Eve	700
	8	Frank	800
	9	Grace	900
	10	Heidi	1000

2. Left join

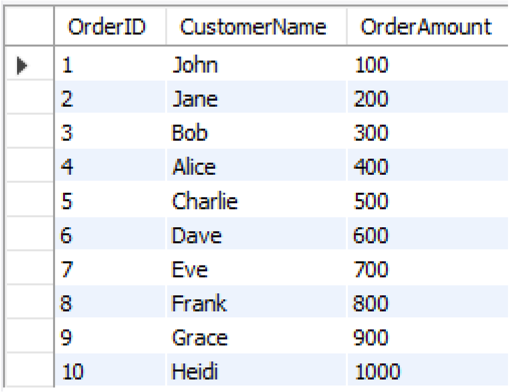
```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderAmount

FROM Orders

LEFT JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```



	OrderID	CustomerName	OrderAmount
▶	1	John	100
	2	Jane	200
	3	Bob	300
	4	Alice	400
	5	Charlie	500
	6	Dave	600
	7	Eve	700
	8	Frank	800
	9	Grace	900
	10	Heidi	1000

3. Right join

```
 SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderAmount  FROM Orders

RIGHT JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```



	OrderID	CustomerName	OrderAmount
▶	1	John	100
	2	Jane	200
	3	Bob	300
	4	Alice	400
	5	Charlie	500
	6	Dave	600
	7	Eve	700
	8	Frank	800
	9	Grace	900
	10	Heidi	1000

Conclusion :- We successfully studied about inner , left and right joins and implement it on a data base

Practical 10

Aim:- Study of normalisation in SQL

Theory

Normalisation in SQL is a technique used to organise data in a relational database to reduce redundancy and improve data integrity. It involves breaking down large tables into smaller, related tables to reduce data duplication.

The process of normalisation was first proposed by Edgar F. Codd as part of his relational model4. It's based on a set of guidelines known as normal forms1. These normal forms serve as the foundation for the normalisation process and help reduce or eliminate abnormalities, inconsistencies, and data duplication that might occur when data is stored in a single table.

Different stages of normalisation exist, each with its own requirements and standards. These include:

1NF (First Normal Form): A table is in 1NF if every cell contains a single value (atomicity), and each record is unique and distinct.

2NF (Second Normal Form): A table is in 2NF if it is in 1NF and every non-key column in the table depends on the complete primary key, not just a portion of it.

3NF (Third Normal Form): A table is in 3NF if it is in 2NF and there are no transitive functional dependencies.

BCNF (Boyce-Codd Normal Form): A table is in BCNF if it is in 3NF and for every non-trivial functional dependency X -> Y, X is a super key.

4NF (Fourth Normal Form): A table is in 4NF if it is in BCNF and there are no multivalued dependencies.

5NF (Fifth Normal Form or Project-Join Normal Form): A table is in 5NF if it is in 4NF and every join dependency in the table is implied by the candidate keys.

Normalisation is essential for eliminating redundant data, ensuring data dependencies make sense, and making the database structure more scalable and adaptable2. It also helps in keeping data consistent by storing the data in one table and referencing it everywhere else.

Conclusion :- we successfully studied about Normalisation in SQL