A

Lab Journal

On

Database Management Lab Experiments

Submitted in partial fulfillment for

Database Management Experiments Lab (6IT351)

Submitted by

T3 Batch

Under the Guidance of

Mrs. M. B. Shinde

Department of Information Technology,

Walchand College of Engineering, Sangli.

Maharashtra, India.

416415

INDEX

Experiment No.1

Title: To Construct ER Model for a given database. Consider the database of College and Student and define a schema for the same. For the same database draw ER diagram and convert entities and relationships to relation table for a given scenario.

Student DATABASE:

STUDENT (USN, SName, Address, Phone, Gender)

SEMSEC (SSID, Sem, Sec)

CLASS (USN, SSID) SUBJECT (Subcode, Title, Sem, Credits)

IAMARKS (USN, Subcode, SSID, Test1, Test2, Test3, FinalIA)

COMPANY DATABASE:

EMPLOYEE (SSN, Name, Address, Sex, Salary, SuperSSN, DNo)

DEPARTMENT (DNo, DName, MgrSSN, MgrStartDate)

DLOCATION (DNo,DLoc)

PROJECT (PNo, PName, PLocation, DNo)

WORKS_ON (SSN, PNo, Hours)

Lab Assignment: Schema Diagram

Objectives:

1. Understand Database Schema
2. Identify Entities and Relationships
3. Learn ER Diagram Construction
4. Apply SQL Skills

Description:

1.ER Diagram:

An ER Diagram (Entity-Relationship Diagram) is a visual representation of a database. It shows entities (tables), their attributes (fields), and the relationships between them. It helps in designing the structure of a database.

2.Schema Definition:

Schema Definition is the blueprint of the database, defined using SQL CREATE TABLE statements. It specifies the structure of tables, attributes, and the relationships between them, including primary and foreign keys.

3.Relational Table:

A Relational Table is a table in a database that stores data based on the schema. It consists of rows and columns, with primary keys identifying unique records and foreign keys linking related tables.

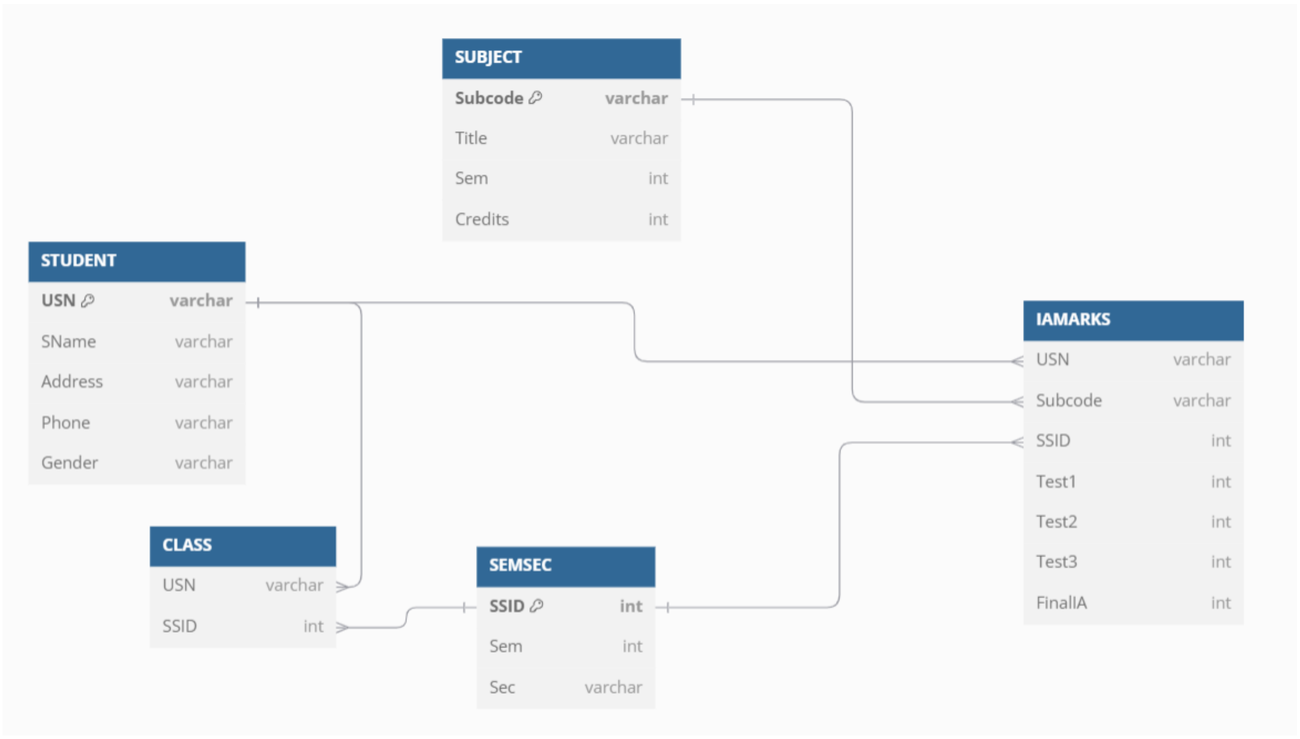
Procedure:

1. **Construct ER Diagram:**
   - **Student Database:** Entities like Student, SemSec, Class, Subject, and IAMarks, with relationships such as students enrolled in semesters and subjects.
   - **Company Database:** Entities like Employee, Department, Project, Works_On, and DLocation, defining relationships like employees working on projects and assigned departments.
2. **Define Schema:**
   - Use SQL CREATE TABLE statements to define tables with primary and foreign keys.
   - Capture attributes like USN, subject marks, employee salaries, and project hours.
3. **Convert Entities to Tables:**
   - Insert sample data into both databases using SQL INSERT INTO commands.
   - Validate the data with SELECT queries to ensure proper relationships.

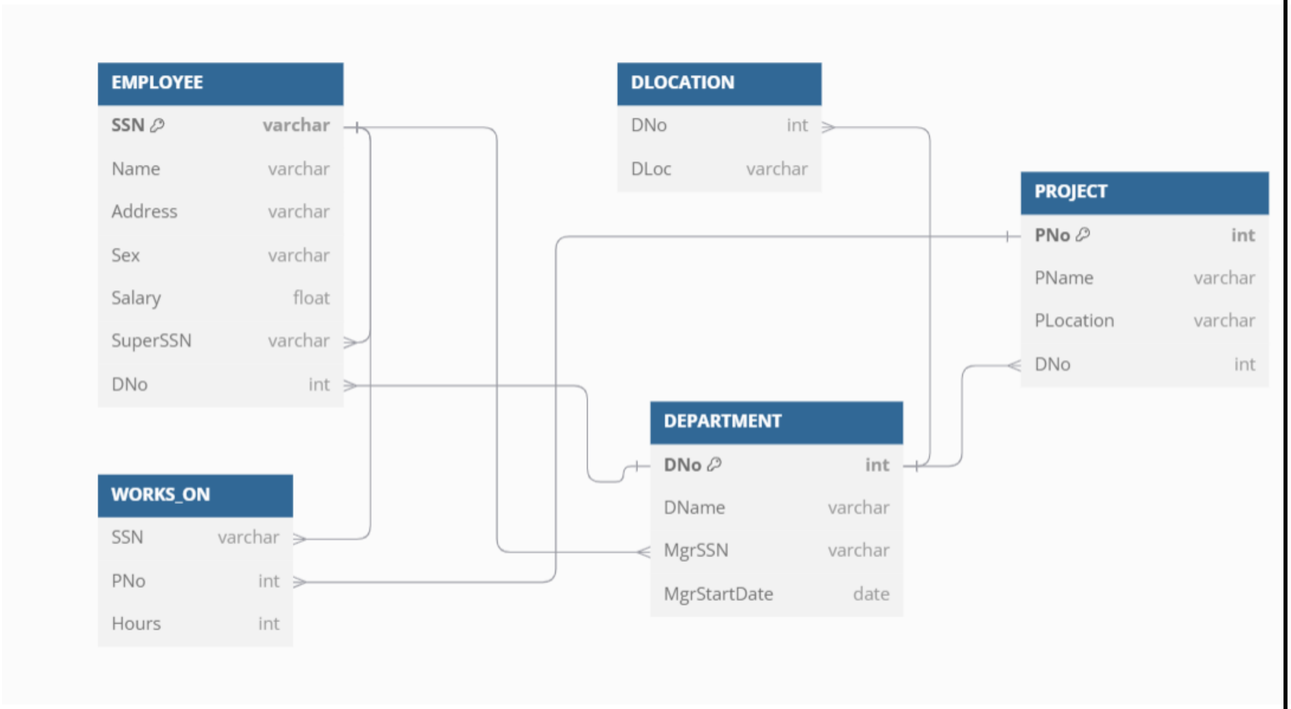
Output:

1.Construct ER Diagram

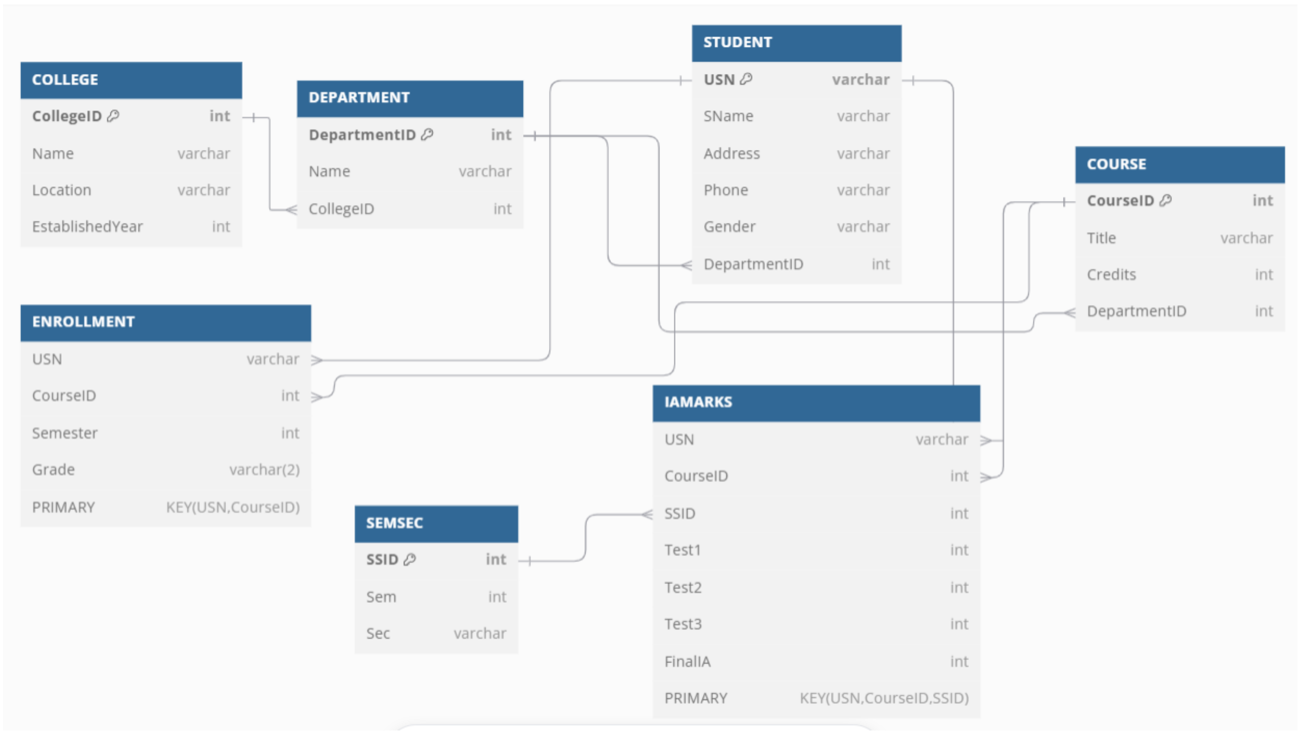
Student Database:



Company Database:



Student and College Database:



2.Define schema

- **Schema Definition of Student database :**

```
CREATE DATABASE Student_DB;
USE Student_DB;

CREATE TABLE STUDENT (
    USN INT PRIMARY KEY,
    SName VARCHAR(100),
    Address VARCHAR(255),
    Phone VARCHAR(15),
    Gender CHAR(1)
);

CREATE TABLE SEMSEC (
    SSID INT PRIMARY KEY,
    Sem INT,
    Sec CHAR(1)
);
```

```
CREATE TABLE SUBJECT (
   Subcode INT PRIMARY KEY,
   Title VARCHAR(100),
   Sem INT,
   Credits INT
);


CREATE TABLE CLASS (
   USN INT,
   SSID INT,
   PRIMARY KEY (USN, SSID),
   FOREIGN KEY (USN) REFERENCES STUDENT(USN),
   FOREIGN KEY (SSID) REFERENCES SEMSEC(SSID)
);

CREATE TABLE IAMARKS (
   USN INT,
   Subcode INT,
   SSID INT,
   Test1 INT,
   Test2 INT,
   Test3 INT,
   FinalIA INT,
   PRIMARY KEY (USN, Subcode, SSID),
   FOREIGN KEY (USN) REFERENCES STUDENT(USN),
   FOREIGN KEY (Subcode) REFERENCES SUBJECT(Subcode),
   FOREIGN KEY (SSID) REFERENCES SEMSEC(SSID)
);
```

Schema Definition of Company Database:

```
CREATE DATABASE Company_DB;
USE Company_DB;
CREATE TABLE DEPARTMENT (
    DNo INT PRIMARY KEY,
    DName VARCHAR(100),
    MgrSSN INT,
    MgrStartDate DATE
);

CREATE TABLE EMPLOYEE (
    SSN INT PRIMARY KEY,
    Name VARCHAR(100),
    Address VARCHAR(255),
    Sex CHAR(1),
    Salary DECIMAL(10, 2),
    SuperSSN INT,
    DNo INT,
    FOREIGN KEY (SuperSSN) REFERENCES EMPLOYEE(SSN),
    FOREIGN KEY (DNo) REFERENCES DEPARTMENT(DNo)
);

CREATE TABLE DLOCATION (
    DNo INT,
    DLoc VARCHAR(100),
    PRIMARY KEY (DNo, DLoc),
    FOREIGN KEY (DNo) REFERENCES DEPARTMENT(DNo)
);

CREATE TABLE PROJECT (
    PNo INT PRIMARY KEY,
    PName VARCHAR(100),
    PLocation VARCHAR(100),
    DNo INT,
    FOREIGN KEY (DNo) REFERENCES DEPARTMENT(DNo)
);

CREATE TABLE WORKS_ON (
    SSN INT,
    PNo INT,
    Hours INT,
    PRIMARY KEY (SSN, PNo),
    FOREIGN KEY (SSN) REFERENCES EMPLOYEE(SSN),
    FOREIGN KEY (PNo) REFERENCES PROJECT(PNo)
        );
```

Schema Definition of Student and College database:

```
CREATE TABLE COLLEGE (
    CollegeID INT PRIMARY KEY,
    Name VARCHAR(100),
    Location VARCHAR(100),
    EstablishedYear INT
);

CREATE TABLE DEPARTMENT (
    DepartmentID INT PRIMARY KEY,
    Name VARCHAR(100),
    CollegeID INT,
    FOREIGN KEY (CollegeID) REFERENCES COLLEGE(CollegeID)
);

CREATE TABLE STUDENT (
    USN VARCHAR(10) PRIMARY KEY,
    SName VARCHAR(50),
    Address VARCHAR(100),
    Phone VARCHAR(15),
    Gender VARCHAR(10),
    DepartmentID INT,
    FOREIGN KEY (DepartmentID) REFERENCES DEPARTMENT(DepartmentID)
```

```
);

CREATE TABLE COURSE (
    CourseID INT PRIMARY KEY,
    Title VARCHAR(100),
    Credits INT,
    DepartmentID INT,
    FOREIGN KEY (DepartmentID) REFERENCES DEPARTMENT(DepartmentID)
);

CREATE TABLE ENROLLMENT (
    USN VARCHAR(10),
    CourseID INT,
    Semester INT,
    Grade VARCHAR(2),
    PRIMARY KEY (USN, CourseID),
    FOREIGN KEY (USN) REFERENCES STUDENT(USN),
    FOREIGN KEY (CourseID) REFERENCES COURSE(CourseID)
);
CREATE TABLE SEMSEC (
    SSID INT PRIMARY KEY,
    Sem INT,
    Sec VARCHAR(5)
);

CREATE TABLE IAMARKS (
    USN VARCHAR(10),
    CourseID INT,
    SSID INT,
    Test1 INT,
    Test2 INT,
    Test3 INT,
    FinalIA INT,
    PRIMARY KEY (USN, CourseID, SSID),
    FOREIGN KEY (USN) REFERENCES STUDENT(USN),
    FOREIGN KEY (CourseID) REFERENCES COURSE(CourseID),
    FOREIGN KEY (SSID) REFERENCES SEMSEC(SSID)
);
```

3.Convert entities and relationships to relation table

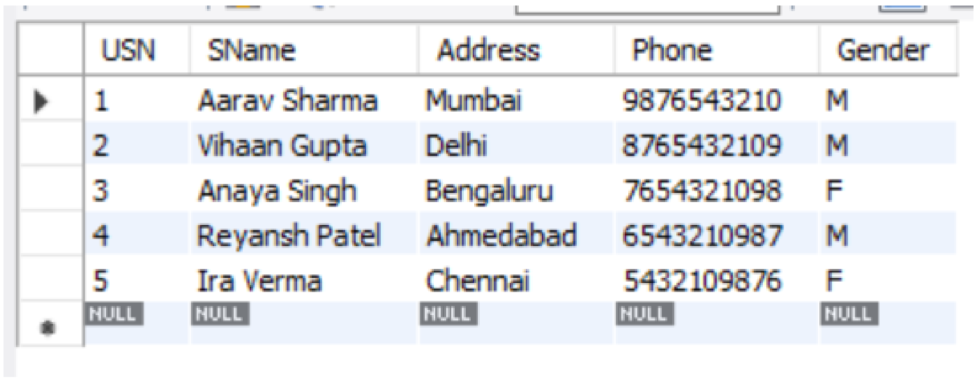
Student Database:

```
INSERT INTO STUDENT (USN, SName, Address, Phone, Gender) VALUES
(1, 'Aarav Sharma', 'Mumbai', '9876543210', 'M'),
(2, 'Vihaan Gupta', 'Delhi', '8765432109', 'M'),
(3, 'Anaya Singh', 'Bengaluru', '7654321098', 'F'),
(4, 'Reyansh Patel', 'Ahmedabad', '6543210987', 'M'),
(5, 'Ira Verma', 'Chennai', '5432109876', 'F');

INSERT INTO SEMSEC (SSID, Sem, Sec) VALUES
(1, 1, 'A'),
(2, 1, 'B'),
(3, 2, 'A');

INSERT INTO SUBJECT (Subcode, Title, Sem, Credits) VALUES
(101, 'Database Management', 1, 4),
(102, 'Data Structures', 1, 4),
(201, 'Operating Systems', 2, 3);
INSERT INTO CLASS (USN, SSID) VALUES
(1, 1),
(2, 2),
(3, 1),
(4, 3),
(5, 1);

INSERT INTO IAMARKS (USN, Subcode, SSID, Test1, Test2, Test3, FinalIA)
VALUES
(1, 101, 1, 18, 22, 25, 10),
(1, 102, 1, 20, 25, 30, 15),
(2, 101, 2, 25, 20, 22, 18),
(3, 102, 1, 30, 28, 32, 25),
```

(4, 201, 3, 15, 20, 18, 10),

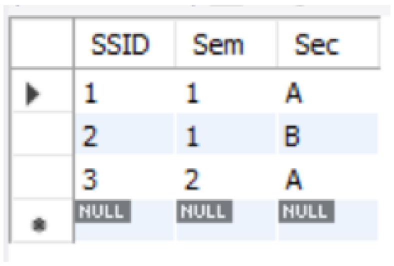
(5, 101, 1, 22, 24, 26, 12);

SELECT * FROM STUDENT;



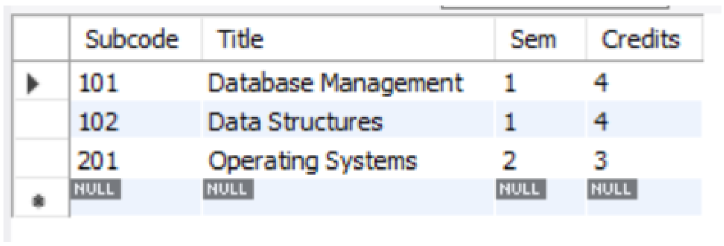
USN	SName	Address	Phone	Gender
1	Aarav Sharma	Mumbai	9876543210	M
2	Vihaan Gupta	Delhi	8765432109	M
3	Anaya Singh	Bengaluru	7654321098	F
4	Reyansh Patel	Ahmedabad	6543210987	M
5	Ira Verma	Chennai	5432109876	F
NULL	NULL	NULL	NULL	NULL

SELECT * FROM SEMSEC;



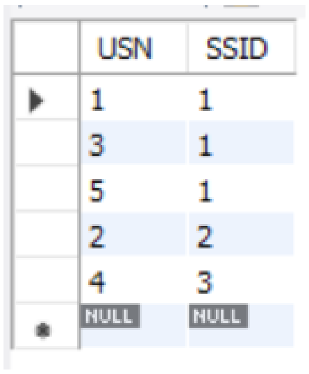
SSID	Sem	Sec
1	1	A
2	1	B
3	2	A
NULL	NULL	NULL

SELECT * FROM SUBJECT;



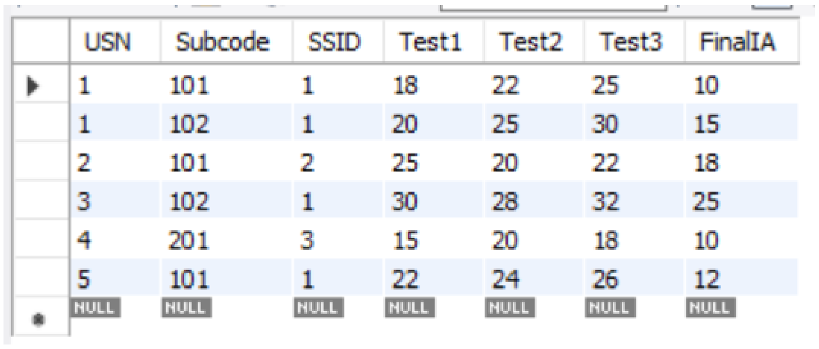
Subcode	Title	Sem	Credits
101	Database Management	1	4
102	Data Structures	1	4
201	Operating Systems	2	3
NULL	NULL	NULL	NULL

SELECT * FROM CLASS;



USN	SSID
1	1
3	1
5	1
2	2
4	3
NULL	NULL

SELECT * FROM IAMARKS;



USN	Subcode	SSID	Test1	Test2	Test3	FinalIA
1	101	1	18	22	25	10
1	102	1	20	25	30	15
2	101	2	25	20	22	18
3	102	1	30	28	32	25
4	201	3	15	20	18	10
5	101	1	22	24	26	12
NULL	NULL	NULL	NULL	NULL	NULL	NULL

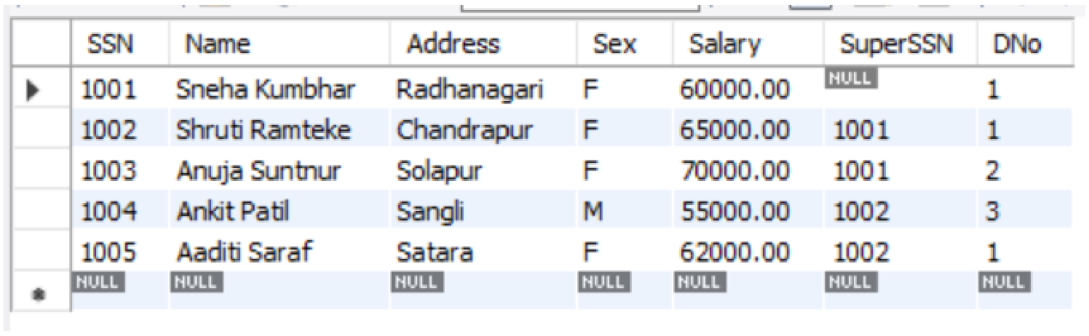
- **Company Database**

INSERT INTO EMPLOYEE (SSN, Name, Address, Sex, Salary, SuperSSN, DNo)
VALUES
(1001, 'Sneha Kumbhar', 'Radhanagari', 'F', 60000, NULL, 1),
(1002, 'Shruti Ramteke', 'Chandrapur', 'F', 65000, 1001, 1),
(1003, 'Anuja Suntnur', 'Solapur', 'F', 70000, 1001, 2),
(1004, 'Ankit Patil', 'Sangli', 'M', 55000, 1002, 3),
(1005, 'Aaditi Saraf', 'Satara', 'F', 62000, 1002, 1);

INSERT INTO DEPARTMENT (DNo, DName, MgrSSN, MgrStartDate) VALUES
(1, 'Human Resources', 1001, '2018-01-15'),
(2, 'Engineering', 1001, '2019-02-10'),
(3, 'Marketing', 1002, '2020-03-05');

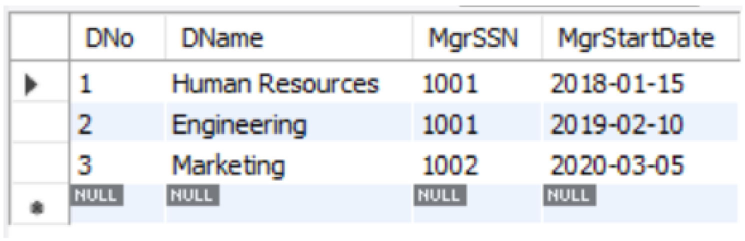
INSERT INTO DLOCATION (DNo, DLoc) VALUES
(1, 'Mumbai'),
(2, 'Japan'),

```
(3, 'Pune');

INSERT INTO PROJECT (PNo, PName, PLocation, DNo) VALUES
(1, 'Project Alpha', 'Mumbai', 1),
(2, 'Project Beta', 'Japan', 2),
(3, 'Project Gamma', 'Pune', 3);

INSERT INTO WORKS_ON (SSN, PNo, Hours) VALUES
(1001, 1, 40),
(1002, 1, 35),
(1003, 2, 45),
(1004, 3, 30),
(1005, 1, 20);

SELECT * FROM EMPLOYEE;
```



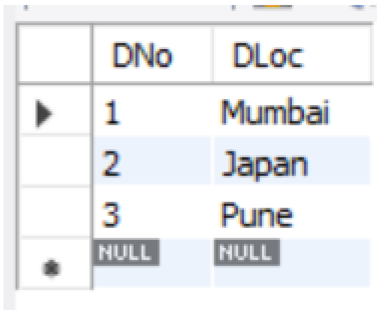
SSN	Name	Address	Sex	Salary	SuperSSN	DNo
1001	Sneha Kumbhar	Radhanagari	F	60000.00	NULL	1
1002	Shruti Ramteke	Chandrapur	F	65000.00	1001	1
1003	Anuja Suntnur	Solapur	F	70000.00	1001	2
1004	Ankit Patil	Sangli	M	55000.00	1002	3
1005	Aaditi Saraf	Satara	F	62000.00	1002	1
NULL	NULL	NULL	NULL	NULL	NULL	NULL

```
SELECT * FROM DEPARTMENT;
```



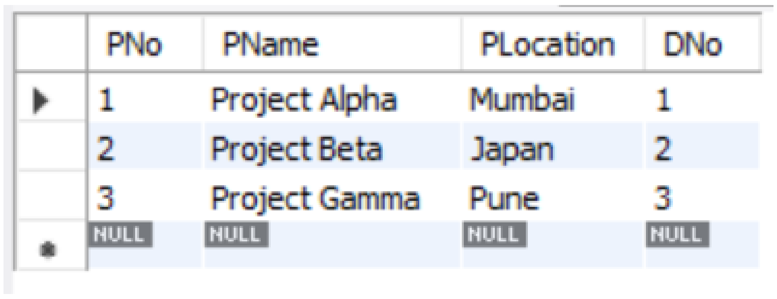
DNo	DName	MgrSSN	MgrStartDate
1	Human Resources	1001	2018-01-15
2	Engineering	1001	2019-02-10
3	Marketing	1002	2020-03-05
NULL	NULL	NULL	NULL

```
SELECT * FROM DLOCATION;
```



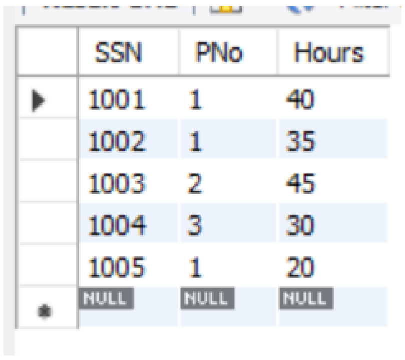
DNo	DLoc
1	Mumbai
2	Japan
3	Pune
NULL	NULL

SELECT * FROM PROJECT;



PNo	PName	PLocation	DNo
1	Project Alpha	Mumbai	1
2	Project Beta	Japan	2
3	Project Gamma	Pune	3
NULL	NULL	NULL	NULL

SELECT * FROM WORKS_ON;



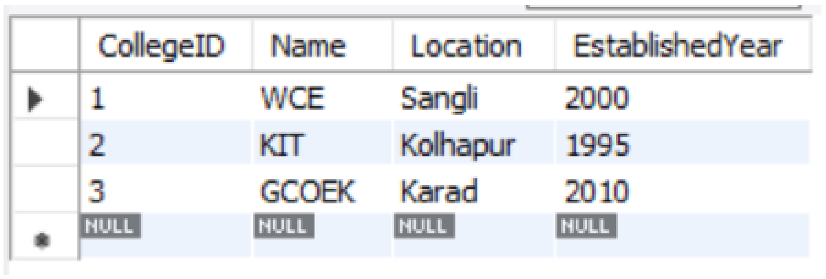
SSN	PNo	Hours
1001	1	40
1002	1	35
1003	2	45
1004	3	30
1005	1	20
NULL	NULL	NULL

- **College and Student Database:**

INSERT INTO COLLEGE (CollegeID, Name, Location, EstablishedYear) VALUES
(1, 'WCE', 'Sangli', 2000),
(2, 'KIT', 'Kolhapur', 1995),
(3, 'GCOEK', 'Karad', 2010);

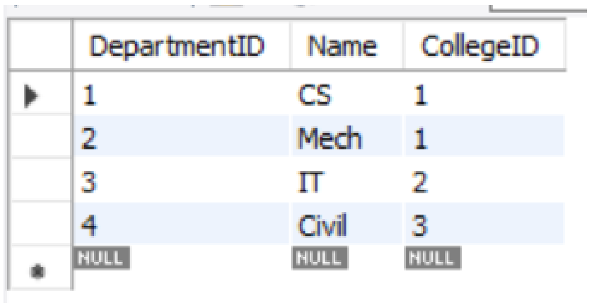
INSERT INTO DEPARTMENT (DepartmentID, Name, CollegeID) VALUES
(1, 'CS', 1),
(2, 'Mech', 1),
(3, 'IT', 2),
(4, 'Civil', 3);

```
INSERT INTO STUDENT (USN, SName, Address, Phone, Gender, DepartmentID)
VALUES
('101', 'Sneha', '123 Main St', '1234567890', 'F', 1),
('102', 'Anuja', '456 Oak St', '0987654321', 'M', 2),
('103', 'Shruti', '789 Pine St', '1122334455', 'M', 1),
('104', 'Aaditi', '321 Elm St', '1231231234', 'F', 3);

INSERT INTO COURSE (CourseID, Title, Credits, DepartmentID) VALUES
(201, 'Data Structures', 4, 1),
(202, 'Thermodynamics', 3, 2),
(203, 'Marketing Principles', 3, 3),
(204, 'Structural Analysis', 4, 4);

INSERT INTO ENROLLMENT (USN, CourseID, Semester, Grade) VALUES
('101', 201, 1, 'A'),
('101', 203, 1, 'B'),
('102', 202, 1, 'B'),
('103', 201, 1, 'A'),
('104', 203, 1, 'A');

INSERT INTO SEMSEC (SSID, Sem, Sec) VALUES
(1, 1, 'A'),
(2, 1, 'B');

INSERT INTO IAMARKS (USN, CourseID, SSID, Test1, Test2, Test3, FinalIA)
VALUES
('101', 201, 1, 85, 90, 88, 80),
('101', 203, 1, 78, 82, 80, 75),
('102', 202, 1, 75, 80, 78, 76),
('103', 201, 1, 90, 92, 88, 85),
('104', 203, 1, 88, 85, 80, 90);
SELECT * FROM COLLEGE;
```



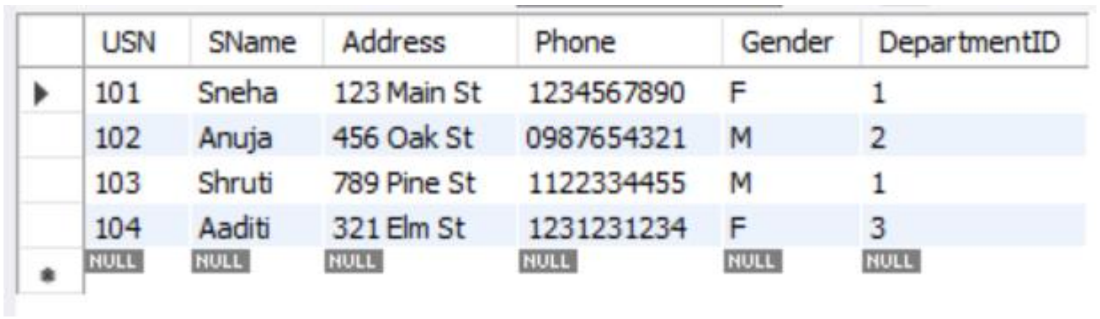
CollegeID	Name	Location	EstablishedYear
1	WCE	Sangli	2000
2	KIT	Kolhapur	1995
3	GCOEK	Karad	2010
NULL	NULL	NULL	NULL

SELECT * FROM DEPARTMENT;



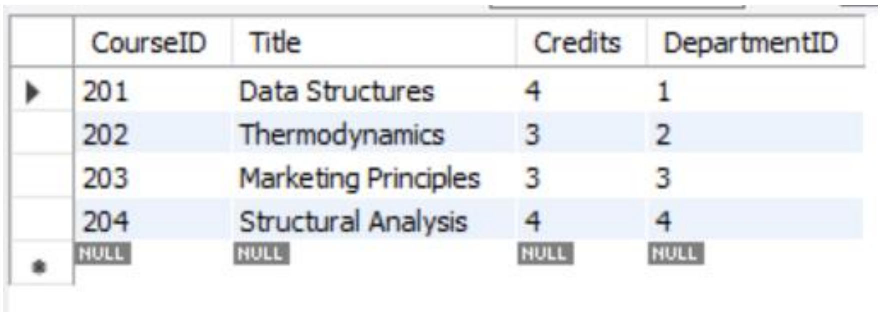
DepartmentID	Name	CollegeID
1	CS	1
2	Mech	1
3	IT	2
4	Civil	3
NULL	NULL	NULL

SELECT * FROM STUDENT;



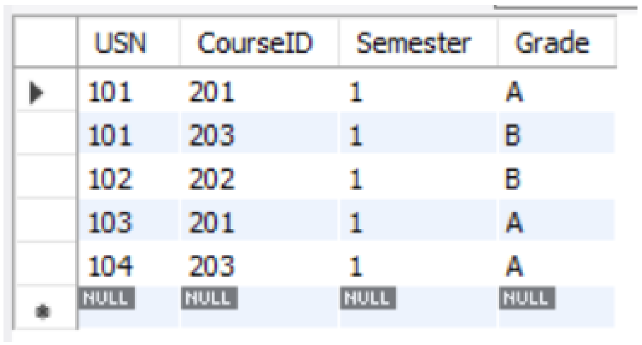
USN	SName	Address	Phone	Gender	DepartmentID
101	Sneha	123 Main St	1234567890	F	1
102	Anuja	456 Oak St	0987654321	M	2
103	Shruti	789 Pine St	1122334455	M	1
104	Aaditi	321 Elm St	1231231234	F	3
NULL	NULL	NULL	NULL	NULL	NULL

SELECT * FROM COURSE;



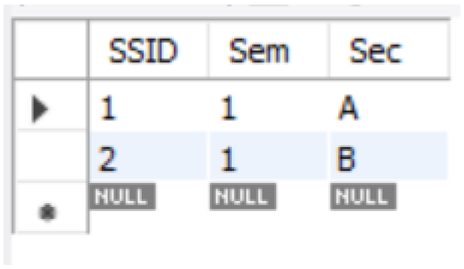
CourseID	Title	Credits	DepartmentID
201	Data Structures	4	1
202	Thermodynamics	3	2
203	Marketing Principles	3	3
204	Structural Analysis	4	4
NULL	NULL	NULL	NULL

SELECT * FROM ENROLLMENT;



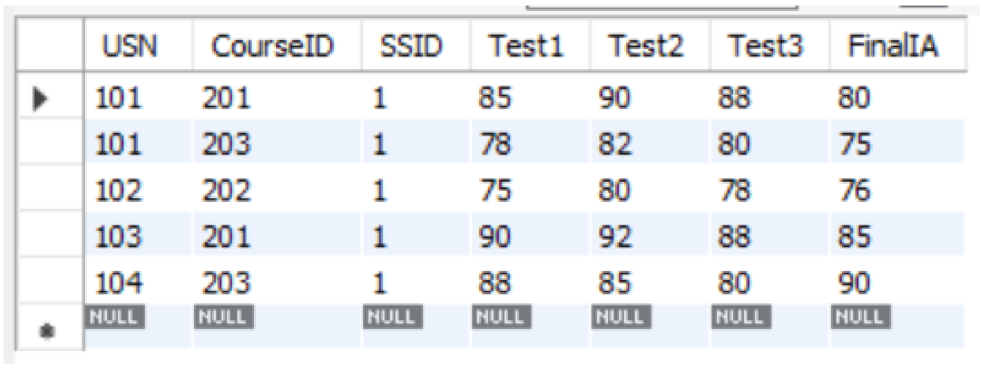
USN	CourseID	Semester	Grade
101	201	1	A
101	203	1	B
102	202	1	B
103	201	1	A
104	203	1	A
NULL	NULL	NULL	NULL

SELECT * FROM SEMSEC;



SSID	Sem	Sec
1	1	A
2	1	B
NULL	NULL	NULL

SELECT * FROM IAMARKS;



USN	CourseID	SSID	Test1	Test2	Test3	FinalIA
101	201	1	85	90	88	80
101	203	1	78	82	80	75
102	202	1	75	80	78	76
103	201	1	90	92	88	85
104	203	1	88	85	80	90
NULL	NULL	NULL	NULL	NULL	NULL	NULL

Conclusion

This exercise demonstrated how to design and implement databases using ER models and SQL. The ER diagrams provided a visual representation of entities and relationships, while SQL created and populated the database with integrity constraints. This helped in understanding how to manage real-world databases effectively.

Experiment No.2

Title:

Implementation of Relational Algebra Operations: Select, Project, Intersect, Minus, and Cartesian Operations on a Database

Aim:

The aim of this experiment is to implement various relational algebra operations, including **Selection**, **Projection**, **Intersection**, **Minus**, and **Cartesian Product**, on a given EMPLOYEE database. The EMPLOYEE database contains employee details such as Social Security Number (SSN), First Name (FNAME), Last Name (LNAME), Address, Sex, and Salary. By performing these relational operations, we aim to efficiently retrieve and manipulate data according to specific conditions, ensuring a practical understanding of relational database management concepts.

Objective:

To implement and understand the basic relational algebra operations such as select, project, intersect, minus, and cartesian product on a given EMPLOYEE database.

Description:

Relational algebra is a procedural query language that operates on relations, which are sets of tuples. It provides operations to manipulate the data stored in relations. In this experiment, we will implement the following relational algebra operations:

1. **Select**: Retrieves rows from a relation that satisfy a given condition.
2. **Project**: Retrieves specific columns from a relation.
3. **Intersect**: Returns the common tuples between two relations.
4. **Minus**: Returns the tuples that are in one relation but not in another.
5. **Cartesian Product**: Combines two relations to form pairs of tuples.

We will perform these operations on an EMPLOYEE database with attributes such as SSN, FNAME, LNAME, ADDRESS, SEX, and SALARY.

Dataset:

EMPLOYEE Relation:

SSN	**FNAME**	**LNAME**	**ADDRESS**	**SEX**	**SALARY**
123-45-6789	Pavan	Navalkar	100 St	M	50000
987-65-4321	sandesh	manvsr	400 St	M	60000
111-22-3333	Aniket	Gadge	300 St	M	55000
222-33-4444	Akshay	Powar	250 St	M	45000

Procedure:

1. **Select Operation**:
   - Retrieve all male employees.

```
SELECT * FROM EMPLOYEE
WHERE SEX = 'M';
```

2. **Project Operation**:
   - Retrieve the first names and salaries of all employees.

```
SELECT FNAME, SALARY
FROM EMPLOYEE;
```

3. **Intersect Operation**:
   - Find employees with the same SSN in another table, say EMPLOYEE_OLD.

```
SELECT * FROM EMPLOYEE
INTERSECT
```

SELECT * FROM EMPLOYEE_OLD;

4. **Minus Operation**:
   - Find employees who are no longer in the current EMPLOYEE table but exist in EMPLOYEE_OLD.

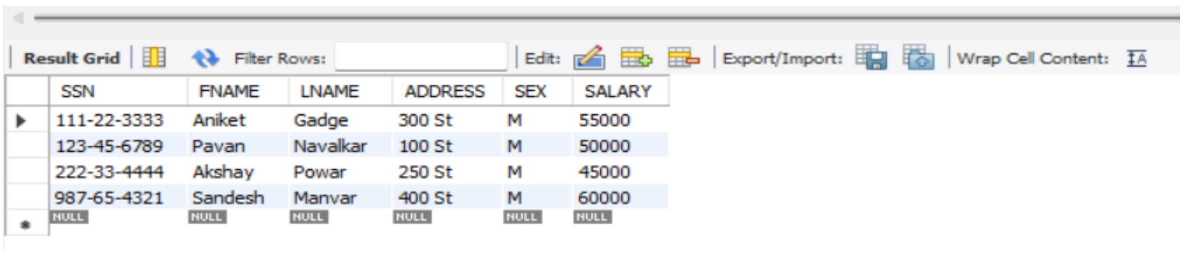
```
SELECT * FROM EMPLOYEE_OLD
MINUS
SELECT * FROM EMPLOYEE;
```

5. **Cartesian Product**:
   - Combine the EMPLOYEE table with a DEPARTMENT table (assuming it exists) to generate all possible pairs of employees and departments.

```
SELECT * FROM EMPLOYEE, DEPARTMENT;
```

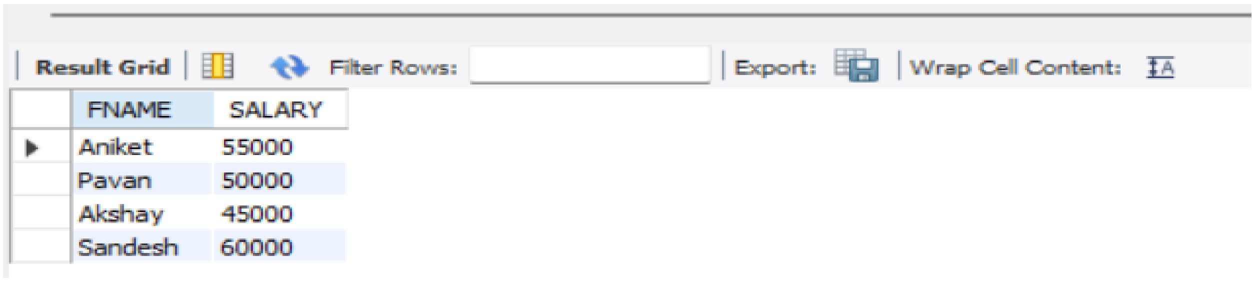
Output:

1. **Select Operation**:



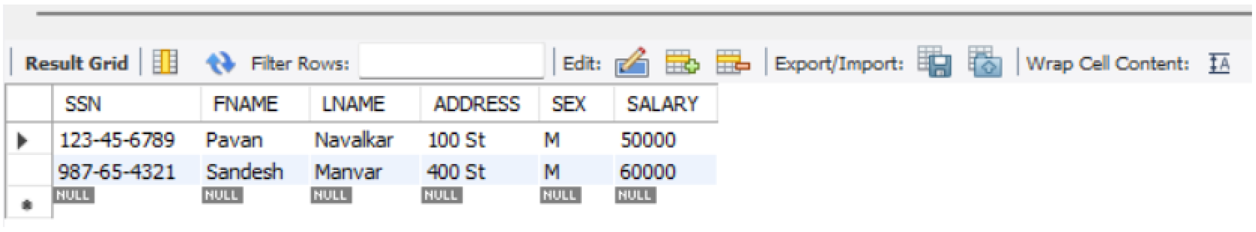
SSN	FNAME	LNAME	ADDRESS	SEX	SALARY
111-22-3333	Aniket	Gadge	300 St	M	55000
123-45-6789	Pavan	Navalkar	100 St	M	50000
222-33-4444	Akshay	Powar	250 St	M	45000
987-65-4321	Sandesh	Manvar	400 St	M	60000
NULL	NULL	NULL	NULL	NULL	NULL

2. **Project Operation**:



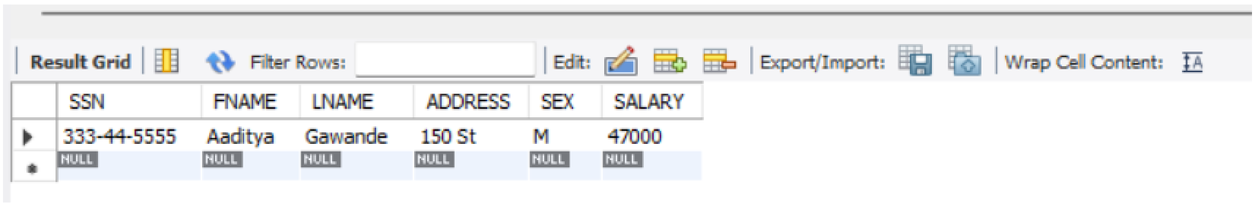
FNAME	SALARY
Aniket	55000
Pavan	50000
Akshay	45000
Sandesh	60000

3. **Intersect Operation**:



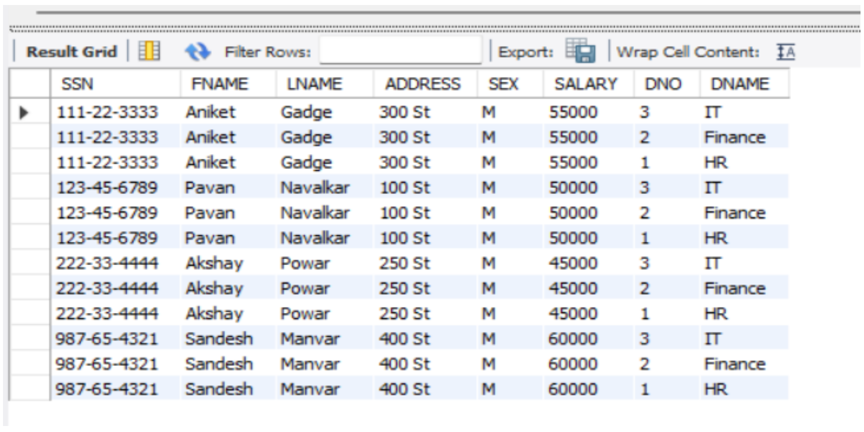
SSN	FNAME	LNAME	ADDRESS	SEX	SALARY
123-45-6789	Pavan	Navalkar	100 St	M	50000
987-65-4321	Sandesh	Manvar	400 St	M	60000
NULL	NULL	NULL	NULL	NULL	NULL

4. **Minus Operation**:



SSN	FNAME	LNAME	ADDRESS	SEX	SALARY
333-44-5555	Aaditya	Gawande	150 St	M	47000
NULL	NULL	NULL	NULL	NULL	NULL

5. **Cartesian Product**:



SSN	FNAME	LNAME	ADDRESS	SEX	SALARY	DNO	DNAME
111-22-3333	Aniket	Gadge	300 St	M	55000	3	IT
111-22-3333	Aniket	Gadge	300 St	M	55000	2	Finance
111-22-3333	Aniket	Gadge	300 St	M	55000	1	HR
123-45-6789	Pavan	Navalkar	100 St	M	50000	3	IT
123-45-6789	Pavan	Navalkar	100 St	M	50000	2	Finance
123-45-6789	Pavan	Navalkar	100 St	M	50000	1	HR
222-33-4444	Akshay	Powar	250 St	M	45000	3	IT
222-33-4444	Akshay	Powar	250 St	M	45000	2	Finance
222-33-4444	Akshay	Powar	250 St	M	45000	1	HR
987-65-4321	Sandesh	Manvar	400 St	M	60000	3	IT
987-65-4321	Sandesh	Manvar	400 St	M	60000	2	Finance
987-65-4321	Sandesh	Manvar	400 St	M	60000	1	HR

Conclusion:

This experiment demonstrates how relational algebra operations like select, project, intersect, minus, and cartesian product can be used to query and manipulate data in a relational database. These operations are essential for understanding how to retrieve and manage data effectively in SQL and other relational database systems.

Experiment No. 3

Title:

Study and Implementation of DDL and DML Commands of SQL with Suitable Examples

Aim: The aim of this experiment is to study, understand, and implement fundamental SQL commands, specifically focusing on Data Definition Language (DDL) and Data Manipulation Language (DML) commands. This includes learning how to define, modify, and manage database structures (DDL) and how to manipulate data within databases (DML) through practical examples.

Objectives:

1. **Understand DDL**: Learn and implement commands like CREATE, ALTER, DROP, and TRUNCATE to manage database structures.
2. **Understand DML**: Learn and implement commands like INSERT, UPDATE, DELETE, and SELECT to manipulate data in databases.
3. **Differentiate DDL and DML**: Identify how DDL affects the database structure while DML affects the data within that structure.
4. **Explore Real-world Scenarios**: Apply SQL commands to practical use cases, such as managing employee or customer records.

Theory:

Data Definition Language (DDL):

- **Definition**: DDL is a subset of SQL used to define, alter, and manage the structure of database objects like tables, indexes, and views. It primarily deals with the database schema (structure).

- **Common DDL Commands**:

CREATE:

Used to create new database objects, such as tables, indexes, or views

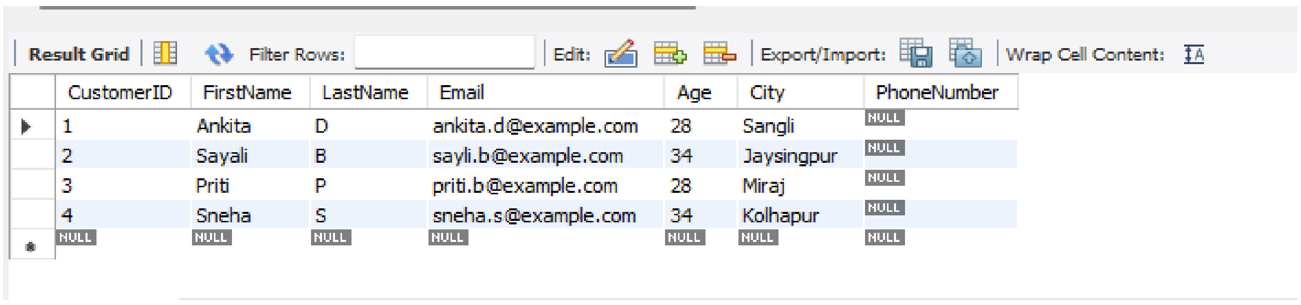
.

```sql
CREATE DATABASE DB;
USE DB;

CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Email VARCHAR(100),
    Age INT,
    City VARCHAR(50)
);
```

ALTER:

Used to modify an existing database object, such as adding or deleting columns in a table.

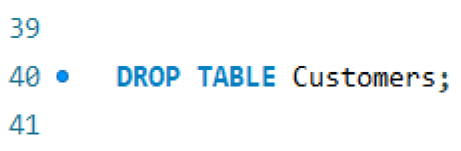
```sql
ALTER TABLE Customers ADD PhoneNumber VARCHAR(15);
```



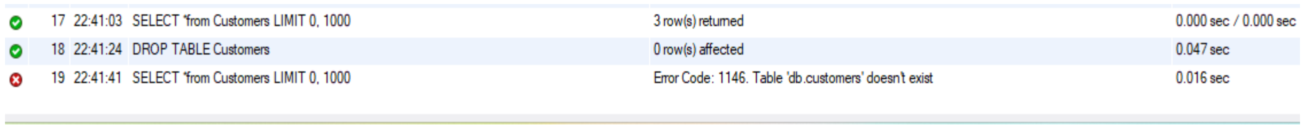
CustomerID	FirstName	LastName	Email	Age	City	PhoneNumber
1	Ankita	D	ankita.d@example.com	28	Sangli	NULL
2	Sayali	B	sayli.b@example.com	34	Jaysingpur	NULL
3	Priti	P	priti.b@example.com	28	Miraj	NULL
4	Sneha	S	sneha.s@example.com	34	Kolhapur	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL

DROP:

Used to delete existing database objects, such as tables or indexes. Once dropped, the data and structure are lost.



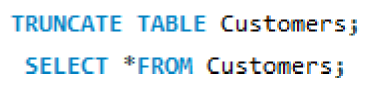
```sql
DROP TABLE Customers;
```



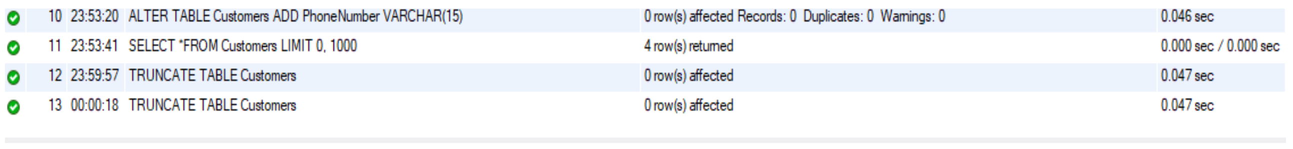
```
✓   17  22:41:03   SELECT *from Customers LIMIT 0, 1000      3 row(s) returned                                    0.000 sec / 0.000 sec
✓   18  22:41:24   DROP TABLE Customers                      0 row(s) affected                                    0.047 sec
✗   19  22:41:41   SELECT *from Customers LIMIT 0, 1000      Error Code: 1146. Table 'db.customers' doesn't exist  0.016 sec
```

TRUNCATE:

Used to remove all rows from a table but retain the structure for future use. It is faster than DELETE since it does not log individual row deletions.



```sql
TRUNCATE TABLE Customers;
 SELECT *FROM Customers;
```



```
✓   10  23:53:20   ALTER TABLE Customers ADD PhoneNumber VARCHAR(15)   0 row(s) affected Records: 0  Duplicates: 0  Warnings: 0   0.046 sec
✓   11  23:53:41   SELECT *FROM Customers LIMIT 0, 1000                4 row(s) returned                                           0.000 sec / 0.000 sec
✓   12  23:59:57   TRUNCATE TABLE Customers                            0 row(s) affected                                           0.047 sec
✓   13  00:00:18   TRUNCATE TABLE Customers                            0 row(s) affected                                           0.047 sec
```

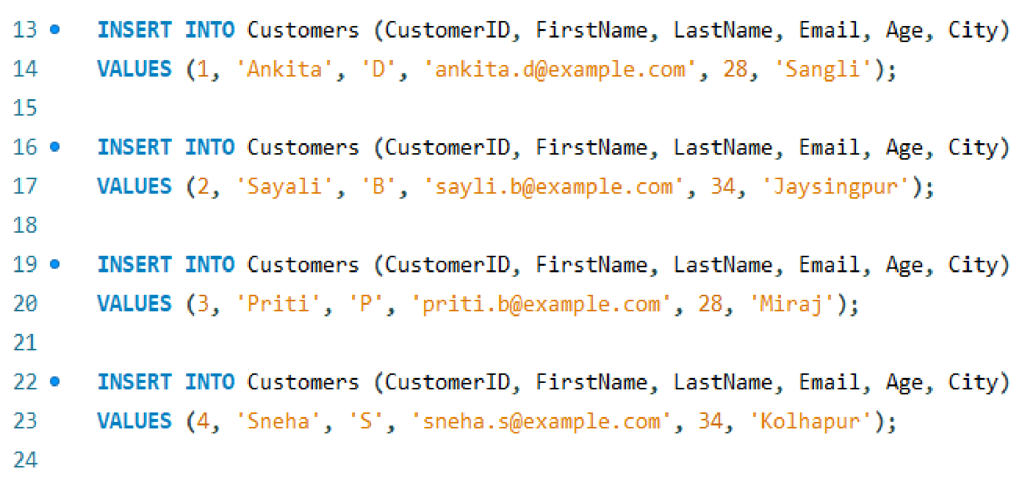
Data Manipulation Language (DML):

- **Definition:** DML is a subset of SQL used to manipulate data stored within the database objects. It focuses on the data itself, enabling operations like inserting, updating, deleting, or retrieving data from the tables.

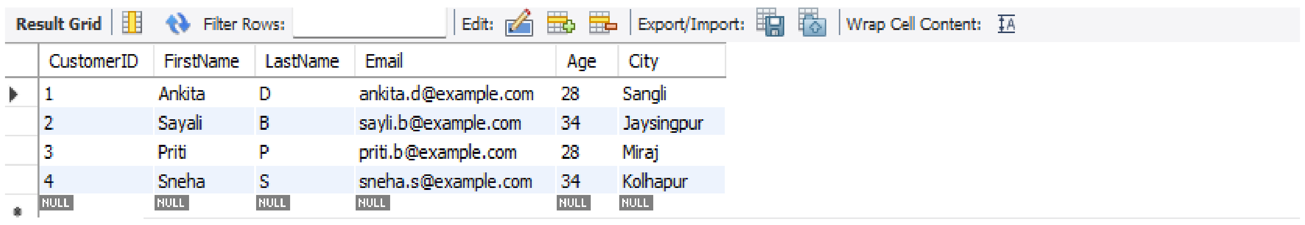
- **Common DML Commands**:

INSERT:

Used to add new records to a table.



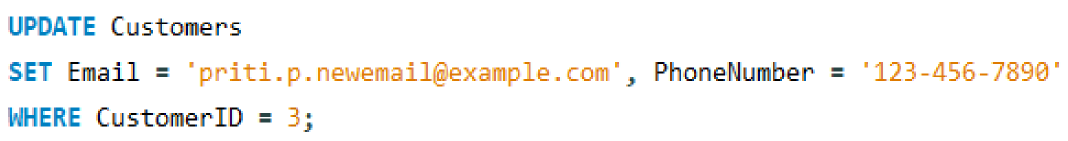
```sql
INSERT INTO Customers (CustomerID, FirstName, LastName, Email, Age, City)
VALUES (1, 'Ankita', 'D', 'ankita.d@example.com', 28, 'Sangli');

INSERT INTO Customers (CustomerID, FirstName, LastName, Email, Age, City)
VALUES (2, 'Sayali', 'B', 'sayli.b@example.com', 34, 'Jaysingpur');

INSERT INTO Customers (CustomerID, FirstName, LastName, Email, Age, City)
VALUES (3, 'Priti', 'P', 'priti.b@example.com', 28, 'Miraj');

INSERT INTO Customers (CustomerID, FirstName, LastName, Email, Age, City)
VALUES (4, 'Sneha', 'S', 'sneha.s@example.com', 34, 'Kolhapur');
```



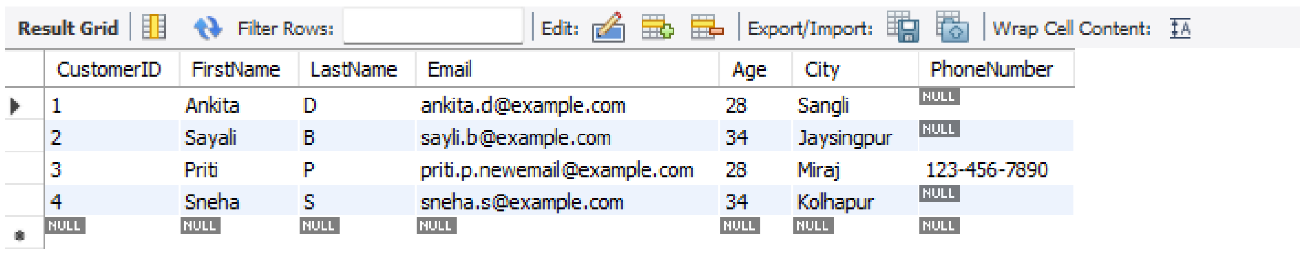
CustomerID	FirstName	LastName	Email	Age	City
1	Ankita	D	ankita.d@example.com	28	Sangli
2	Sayali	B	sayli.b@example.com	34	Jaysingpur
3	Priti	P	priti.b@example.com	28	Miraj
4	Sneha	S	sneha.s@example.com	34	Kolhapur
NULL	NULL	NULL	NULL	NULL	NULL

UPDATE:

Used to modify existing records in a table.



```sql
UPDATE Customers
SET Email = 'priti.p.newemail@example.com', PhoneNumber = '123-456-7890'
WHERE CustomerID = 3;
```

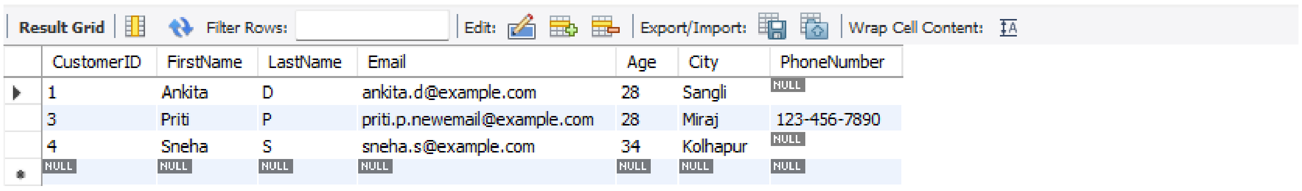


CustomerID	FirstName	LastName	Email	Age	City	PhoneNumber
1	Ankita	D	ankita.d@example.com	28	Sangli	NULL
2	Sayali	B	sayli.b@example.com	34	Jaysingpur	NULL
3	Priti	P	priti.p.newemail@example.com	28	Miraj	123-456-7890
4	Sneha	S	sneha.s@example.com	34	Kolhapur	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL

DELETE:

Used to remove records from a table based on a condition.

```
DELETE FROM Customers
WHERE CustomerID = 2;
```

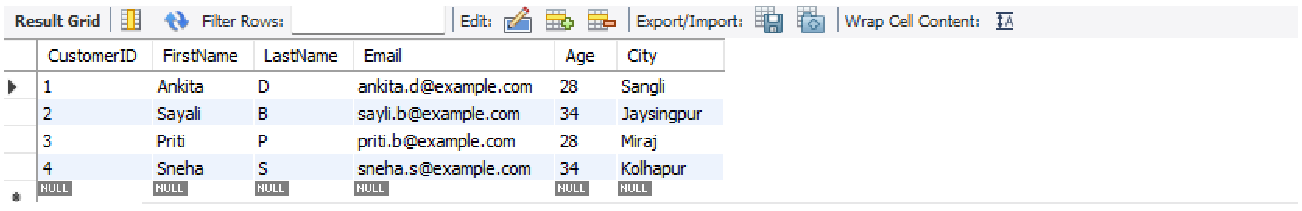


CustomerID	FirstName	LastName	Email	Age	City	PhoneNumber
1	Ankita	D	ankita.d@example.com	28	Sangli	NULL
3	Priti	P	priti.p.newemail@example.com	28	Miraj	123-456-7890
4	Sneha	S	sneha.s@example.com	34	Kolhapur	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL

SELECT:

Used to retrieve data from one or more tables. It can include filtering, sorting, and aggregating data.

```
SELECT *FROM Customers;
```



CustomerID	FirstName	LastName	Email	Age	City
1	Ankita	D	ankita.d@example.com	28	Sangli
2	Sayali	B	sayli.b@example.com	34	Jaysingpur
3	Priti	P	priti.b@example.com	28	Miraj
4	Sneha	S	sneha.s@example.com	34	Kolhapur
NULL	NULL	NULL	NULL	NULL	NULL

Differences Between DDL and DML:

Feature	DDL	DML
Focus	Defines and manages the structure of the database (schema)	Manipulates the data within the database
Common Commands	CREATE, ALTER, DROP, TRUNCATE	INSERT, UPDATE, DELETE, SELECT
Effect	Affects the structure or schema of the database	Affects the data stored in the database
Transactional	Non-transactional; changes take effect immediately and cannot be rolled back	Often transactional; changes can be rolled back
Example	Creating tables, defining relationships, altering structures	Adding, updating, or deleting records, querying data

Conclusion:

Understanding DDL and DML commands is crucial for effective database management. DDL commands provide the foundation for database structure, while DML commands allow interaction with and manipulation of data stored within that structure. Together, these SQL commands form the backbone of database operations, supporting tasks from creating tables to managing and querying data in real-world scenarios.

Experiment No. 4

Title:

Create a database as constumer_information and apply DQL and TCL SQL commands.

Aim:

To create a database named customer_information and apply Data Query Language (DQL) and Transaction Control Language (TCL) SQL commands to manage and retrieve customer data efficiently.

Objectives:

1. **Create a database**: Set up a database named customer_information with appropriate tables for storing customer details.
2. **Insert customer data**: Populate the customer_information database with sample customer records.
3. **Apply DQL commands**: Use SQL queries to retrieve data from the database.
4. **Apply TCL commands**: Implement transaction management using TCL commands to ensure data integrity during multi-step operations.
5. **Analyze query results**: Interpret the data retrieved from the database using various DQL queries.
6. **Ensure consistency**: Maintain the atomicity, consistency, isolation, and durability (ACID) properties using TCL commands.

Theory

1. Database Creation:

A database is a structured collection of data, stored and managed by a database management system (DBMS). SQL (Structured Query Language) is used to interact with relational databases. A typical database may consist of multiple tables that store related data.

Customer Information Database: The customer_information database is designed to store customer details such as customer_id, name, email, address, and phone_number. The SQL command to create this database is:

```
CREATE DATABASE customer_information;
```

To create customer table:

```
CREATE TABLE customers (
customer_id INT PRIMARY KEY,
 name VARCHAR(50),
email VARCHAR(50),
address VARCHAR(100),
phone_number VARCHAR(15)
);
```

2. Data Query Language (DQL):

DQL is used to retrieve data from the database. The most commonly used DQL command is the SELECT statement.

Example DQL commands for the customer_information database:

- Retrieve all customer details:

  ```
  SELECT * FROM customers;
  ```

- Retrieve specific customer data based on conditions:

  ```
  SELECT name, email FROM customers WHERE customer_id = 1;
  ```

3. Transaction Control Language (TCL):

TCL commands are used to manage transactions in a database. A **transaction** is a sequence of one or more SQL operations that are executed as a single unit. TCL ensures that transactions are processed in a way that maintains the integrity of the database.

Key TCL commands:

- **COMMIT**: Saves the changes made by the transaction to the database.
- **ROLLBACK**: Reverts the changes made by the transaction in case of errors.
- **SAVEPOINT**: Sets a point within a transaction to which you can later roll back.
- **RELEASE SAVEPOINT**: Removes the savepoint created earlier.

5)Solve Lab Practice:

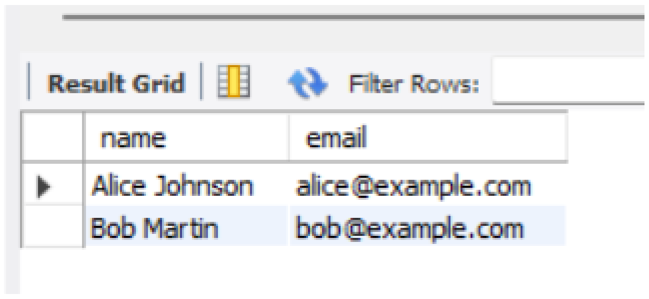
a) Write a query to implement the save point.

Query:

```
CREATE DATABASE customer_information;

USE customer_information;


CREATE TABLE customers (

    customer_id INT PRIMARY KEY,

    name VARCHAR(50),

    email VARCHAR(50),

    address VARCHAR(100),

    phone_number VARCHAR(15)

);

BEGIN;


INSERT INTO customers (customer_id, name, email, address, phone_number)

VALUES (1, 'Alice Johnson', 'alice@example.com', '12 Oak St', '555-7890');


SAVEPOINT savepoint1;


INSERT INTO customers (customer_id, name, email, address, phone_number)

VALUES (2, 'Bob Martin', 'bob@example.com', '34 Pine St', '555-1239');


INSERT INTO customers (customer_id, name, email, address, phone_number)
```

```
VALUES (3, 'Charlie White', 'charlie@example.com', '56 Cedar St', '555-4567');


-- Optionally rollback to the savepoint if needed

-- ROLLBACK TO savepoint1;


COMMIT;


-- Example DQL query that returns at least two names

SELECT name, email FROM customers WHERE customer_id <= 2;
```

Output:



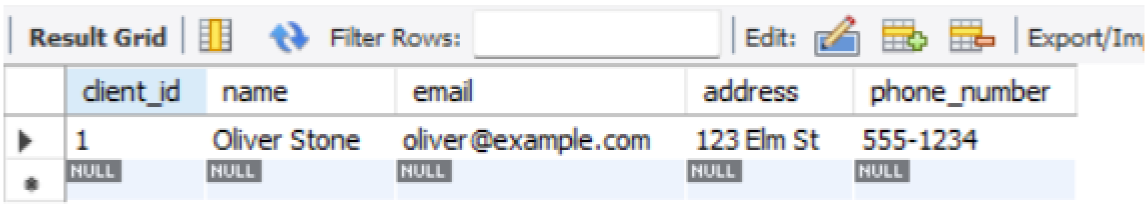
```
name            email
Alice Johnson   alice@example.com
Bob Martin      bob@example.com
```

b) Write a query to implement the rollback.

Query:

```
CREATE DATABASE customer_info;

USE customer_info;


CREATE TABLE clients (

    client_id INT PRIMARY KEY,

    name VARCHAR(50),

    email VARCHAR(50),

    address VARCHAR(100),
    phone_number VARCHAR(15)
);

BEGIN;

INSERT INTO clients (client_id, name, email, address, phone_number)
VALUES (1, 'Oliver Stone', 'oliver@example.com', '123 Elm St', '555-1234');

SAVEPOINT savepoint1;

INSERT INTO clients (client_id, name, email, address, phone_number)
VALUES (2, 'Sophia Brown', 'sophia@example.com', '456 Maple St', '555-5678');

-- Rollback to the savepoint, undoing the second insert
ROLLBACK TO savepoint1;

-- Commit the transaction (only Oliver Stone will be saved)
COMMIT;

-- Example DQL query to view the result
SELECT * FROM clients;
```

Output:



client_id	name	email	address	phone_number
1	Oliver Stone	oliver@example.com	123 Elm St	555-1234
NULL	NULL	NULL	NULL	NULL

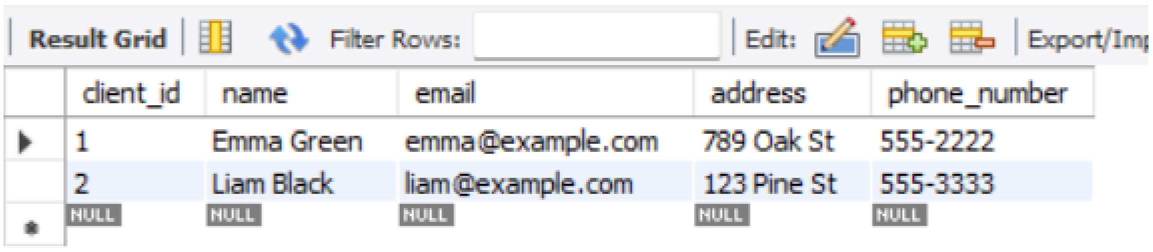
c) Write a query to implement the commit.

Query:

```
CREATE DATABASE client_data;

USE client_data;


CREATE TABLE clients1 (

    client_id INT PRIMARY KEY,

    name VARCHAR(50),

    email VARCHAR(50),

    address VARCHAR(100),

    phone_number VARCHAR(15)

);


BEGIN;


INSERT INTO clients1 (client_id, name, email, address, phone_number)

VALUES (1, 'Emma Green', 'emma@example.com', '789 Oak St', '555-2222');


SAVEPOINT savepoint1;


INSERT INTO clients1 (client_id, name, email, address, phone_number)
```

```
VALUES (2, 'Liam Black', 'liam@example.com', '123 Pine St', '555-3333');


-- Commit the transaction to save all the changes

COMMIT;


-- Example DQL query to view all inserted data

SELECT * FROM clients1;
```

Output:

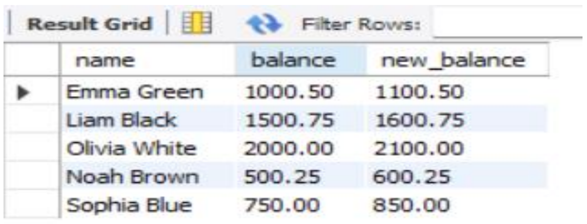


```
client_id  name         email              address      phone_number
1          Emma Green   emma@example.com   789 Oak St   555-2222
2          Liam Black   liam@example.com   123 Pine St  555-3333
NULL       NULL         NULL               NULL         NULL
```

d) Use of all SQL Operators on database.

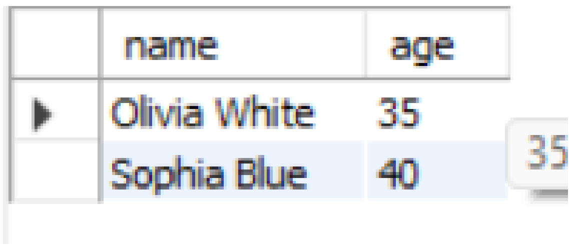
Query:

```
CREATE DATABASE client_data;

USE client_data;


CREATE TABLE clients2 (

   client_id INT PRIMARY KEY,

   name VARCHAR(50),

   email VARCHAR(50),

   address VARCHAR(100),

   phone_number VARCHAR(15),

   age INT,
    balance DECIMAL(10, 2)
);

INSERT INTO clients2 (client_id, name, email, address, phone_number, age, balance)
VALUES
(1, 'Emma Green', 'emma@example.com', '789 Oak St', '555-2222', 30, 1000.50),
(2, 'Liam Black', 'liam@example.com', '123 Pine St', '555-3333', 25, 1500.75),
(3, 'Olivia White', 'olivia@example.com', '456 Maple St', '555-4444', 35, 2000.00),
(4, 'Noah Brown', 'noah@example.com', '789 Birch St', '555-7777', 28, 500.25),
(5, 'Sophia Blue', 'sophia@example.com', '321 Cedar St', '555-8888', 40, 750.00);

SELECT name, balance, balance + 100 AS new_balance FROM clients2;
```



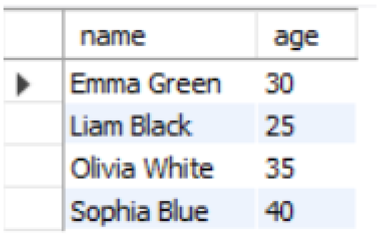
name	balance	new_balance
Emma Green	1000.50	1100.50
Liam Black	1500.75	1600.75
Olivia White	2000.00	2100.00
Noah Brown	500.25	600.25
Sophia Blue	750.00	850.00

```
SELECT name, age FROM clients2 WHERE age > 30;
```



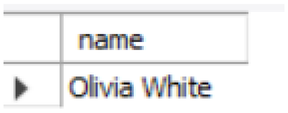
name	age
Olivia White	35
Sophia Blue	40

```
SELECT name, age FROM clients2 WHERE age > 30 OR balance > 1000;
```



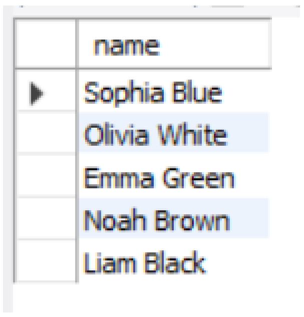
name	age
Emma Green	30
Liam Black	25
Olivia White	35
Sophia Blue	40

SELECT name FROM clients2 WHERE name LIKE 'O%';



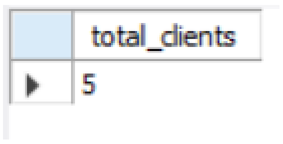
name
Olivia White

SELECT name FROM clients2 ORDER BY age DESC;



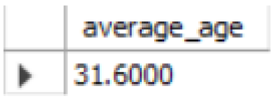
name
Sophia Blue
Olivia White
Emma Green
Noah Brown
Liam Black

SELECT COUNT(*) AS total_clients FROM clients2;



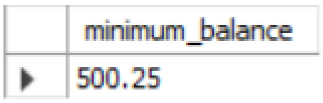
total_clients
5

SELECT AVG(age) AS average_age FROM clients2;



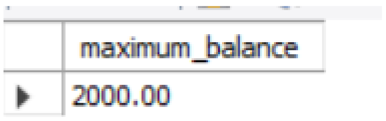
average_age
31.6000

SELECT MIN(balance) AS minimum_balance FROM clients2;



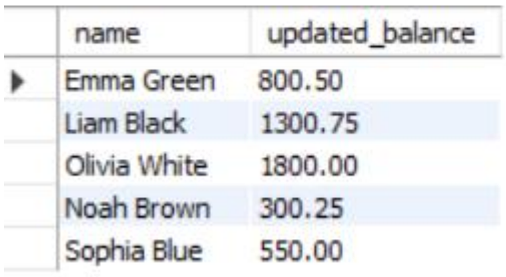
minimum_balance
500.25

SELECT MAX(balance) AS maximum_balance FROM clients2;



maximum_balance
2000.00

SELECT name, balance - 200 AS updated_balance FROM clients2;



name	updated_balance
Emma Green	800.50
Liam Black	1300.75
Olivia White	1800.00
Noah Brown	300.25
Sophia Blue	550.00

Conclusion:

By creating a customer_information database and applying DQL and TCL commands, we can efficiently manage and retrieve customer data while ensuring transaction integrity. DQL allows for precise querying of data, and TCL ensures safe and controlled transaction handling, thus maintaining the consistency and reliability of the database.

Experiment No 5

Title:

Perform Various Types of Functions in SQL

Objective:

To understand and implement different types of SQL functions, including string functions, aggregate functions, mathematical functions, and date functions, to perform data manipulation and retrieval.

Description:

SQL functions play a key role in data handling, enabling users to modify, summarize, and extract insights from the stored data. In this experiment, we will explore four categories of functions—String Functions, Aggregate Functions, Mathematical Functions, and Date Functions—to perform various tasks on a dataset.

Data Set:

Assume we have the following **employees** table for performing the functions:

Employee_id	**Name**	**Salary**	**Hire_date**
1	Annuja Suntnur	50,000	2020-01-10
2	Sneha Kumbhar	45000	2019-05-24
3	Shruti Ramteke	80000	2018-08-15
4	Aaditi Digraje	70000	2017-03-12

Procedure:

1. **Understanding String Functions**:
   - Perform operations like converting to uppercase, extracting substrings, and calculating string length.
2. **Using Aggregate Functions**:
   - Apply functions like COUNT(), SUM(), AVG(), MAX(), and MIN() to retrieve summarized data.
3. **Implementing Mathematical Functions**:
   - Use functions like ABS(), CEIL(), FLOOR(), and ROUND() to perform calculations on numeric data.
4. **Applying Date Functions**:
   - Manipulate date values using functions like CURRENT_DATE, DATE_ADD(), DATEDIFF(), and CURRENT_TIMESTAMP.

Code:

1. **String Functions**:

```
#Convert name to uppercase
SELECT UPPER(name) AS upper_case_name FROM employees;
-#Extract substring from employee name
SELECT SUBSTRING(name, 1, 4) AS substring_name FROM employees;
#Find the length of employee name
SELECT LENGTH(name) AS name_length FROM employees;
```

2. **Aggregate Functions**:

-#Count the number of employees

SELECT COUNT(*) AS total_employees FROM employees;

#Sum of all employee salaries

SELECT SUM(salary) AS total_salary FROM employees;

# Average employee salary

SELECT AVG(salary) AS average_salary FROM employees;

#Maximum and Minimum salary

SELECT MAX(salary) AS highest_salary, MIN(salary) AS lowest_salary FROM employees;

5.Date Functions:

#Get the current date
SELECT CURRENT_DATE AS today_date;

#Add 10 days to the hire date of each employee
SELECT name, DATE_ADD(hire_date, INTERVAL 10 DAY) AS new_hire_date FROM employees;

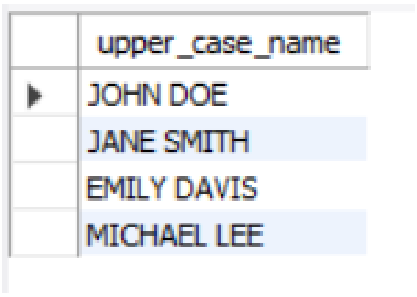
#Difference between two dates (in days)
SELECT name, DATEDIFF(CURRENT_DATE, hire_date) AS days_employed FROM employees;

.

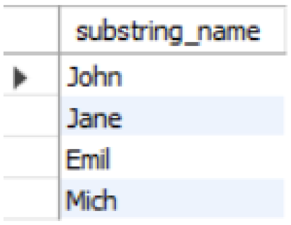
#Get the current timestamp
SELECT CURRENT_TIMESTAMP AS current_time;

Output:

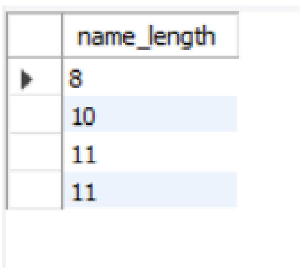
1.String Functions Output:



upper_case_name
JOHN DOE
JANE SMITH
EMILY DAVIS
MICHAEL LEE

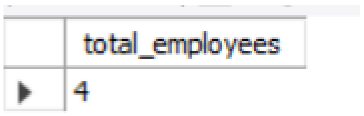


substring_name
John
Jane
Emil
Mich

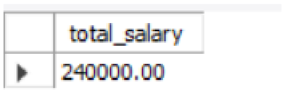


name_length
8
10
11
11

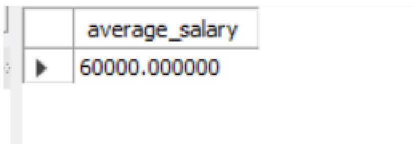
2. Aggregate Functions Output:



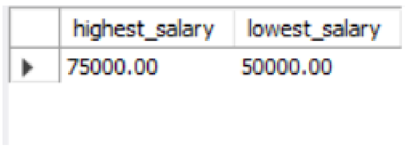
total_employees
4



total_salary
240000.00

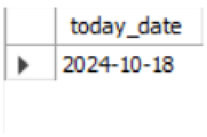


average_salary
60000.000000

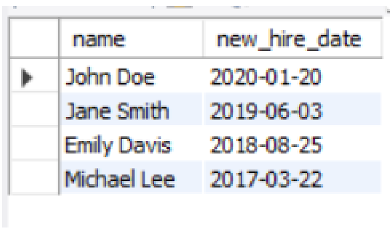


highest_salary	lowest_salary
75000.00	50000.00

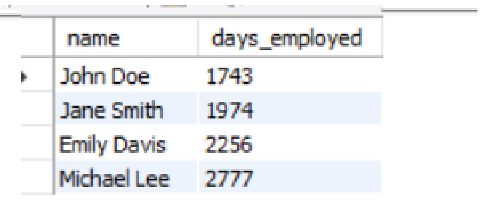
3. Mathematical Functions Output:



today_date
2024-10-18



name	new_hire_date
John Doe	2020-01-20
Jane Smith	2019-06-03
Emily Davis	2018-08-25
Michael Lee	2017-03-22



name	days_employed
John Doe	1743
Jane Smith	1974
Emily Davis	2256
Michael Lee	2777

Conclusion:

This experiment demonstrated the importance of SQL functions in data manipulation and retrieval using a sample employees table. We explored string functions for formatting text, aggregate functions for summarizing data, mathematical functions for precise calculations, and date functions for managing timelines. Mastering these functions is crucial for effective data analysis and decision-making. Overall, this knowledge enhances our ability to derive meaningful insights from databases in various industries.

Experiment No. 6

Title - Study and Implementation of various types of constraint in SQL

Objective -

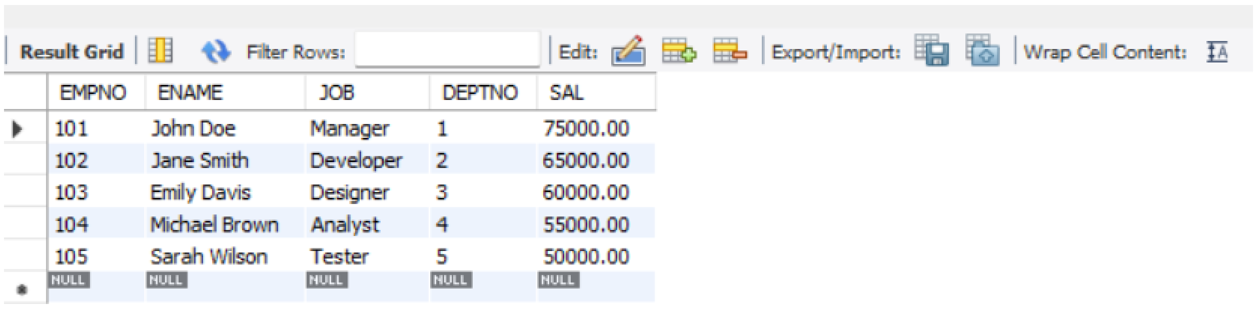
The objective of this practical is to implement various SQL constraints while creating an EMP table for employee management. It focuses on ensuring data integrity through constraints like NOT NULL for essential fields, CHECK for business rules (e.g., EMPNO > 100), UNIQUE for DEPTNO, and PRIMARY KEY on EMPNO. Participants will gain hands-on experience in table creation, data insertion, and understanding how constraints maintain data quality in real-world applications.

Code -

```
-- Create the EMP table with specified constraints
-- Create a new database (if needed)
CREATE DATABASE my_database;

-- Select the database to use
USE my_database;



CREATE TABLE EMP (
   EMPNO INT(6) NOT NULL CHECK (EMPNO > 100),  -- EMPNO must be greater than 100
   ENAME VARCHAR(20) NOT NULL,                  -- ENAME cannot be NULL
   JOB VARCHAR(10) NOT NULL,                    -- JOB cannot be NULL
   DEPTNO INT(3) UNIQUE,                        -- DEPTNO must be unique
   SAL DECIMAL(7, 2),                           -- Salary with 7 digits, 2 decimal places
   PRIMARY KEY (EMPNO)                          -- Primary key on EMPNO
);

-- Insert sample data into the EMP table
```

```
INSERT INTO EMP (EMPNO, ENAME, JOB, DEPTNO, SAL) VALUES (101, 'John Doe',
'Manager', 1, 75000.00);
INSERT INTO EMP (EMPNO, ENAME, JOB, DEPTNO, SAL) VALUES (102, 'Jane Smith',
'Developer', 2, 65000.00);
INSERT INTO EMP (EMPNO, ENAME, JOB, DEPTNO, SAL) VALUES (103, 'Emily Davis',
'Designer', 3, 60000.00);
INSERT INTO EMP (EMPNO, ENAME, JOB, DEPTNO, SAL) VALUES (104, 'Michael Brown',
'Analyst', 4, 55000.00);
INSERT INTO EMP (EMPNO, ENAME, JOB, DEPTNO, SAL) VALUES (105, 'Sarah Wilson',
'Tester', 5, 50000.00);

-- Query to display all records from the EMP table
SELECT * FROM EMP;
```

Output -



```
EMPNO  ENAME          JOB        DEPTNO  SAL
101    John Doe       Manager    1       75000.00
102    Jane Smith     Developer  2       65000.00
103    Emily Davis    Designer   3       60000.00
104    Michael Brown  Analyst    4       55000.00
105    Sarah Wilson   Tester     5       50000.00
NULL   NULL           NULL       NULL    NULL
```

Conclusion :

this practical exercise demonstrates the crucial role of SQL constraints in maintaining data integrity and enforcing business rules in database management. By implementing constraints such as NOT NULL, CHECK, UNIQUE, and PRIMARY KEY on the EMP table, participants ensure that the data is accurate, consistent, and reliable. Through hands-on experience in table creation and data insertion, participants learn how these constraints prevent errors, enforce valid data, and maintain the quality of information in real-world employee management systems. This exercise highlights the importance of well-designed constraints in building robust and error-resistant databases.

Experiment No - 7

Title:

Implementation of Different Types of Joins:

- Inner Join
- Outer Join
- Natural Join, etc.

Aim:

To understand and implement various types of SQL joins, such as Inner Join, Outer Join, and Natural Join, using a relational database schema.

Objective:

1. To create a relational database and perform SQL queries using different types of joins.

2. To retrieve data from multiple tables based on specific conditions.

3. To apply real-world scenarios using the Sailors, Boats, and Reserves tables.

Theory:

SQL (Structured Query Language) is the standard language for managing relational databases. Joins are essential in SQL for combining records from multiple tables. The following types of joins will be implemented in this experiment:

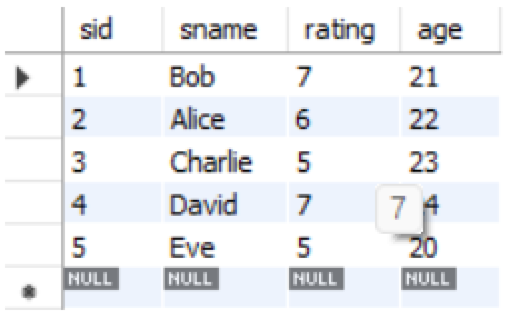
- Inner Join: Retrieves records with matching values in both tables.

- Outer Join: Includes rows from one or both tables even if no match exists.

- Natural Join: Automatically joins tables based on columns with the same name and data type.

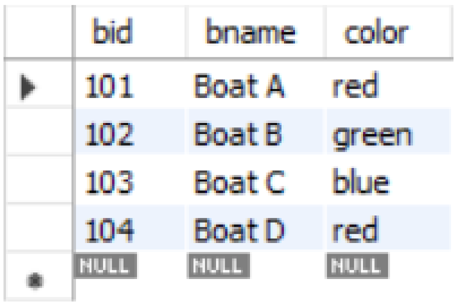
Initially All Three Tables Data:

- **Sailor's Table**



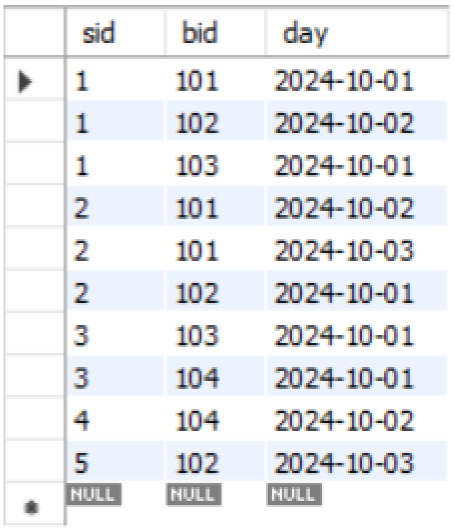
sid	sname	rating	age
1	Bob	7	21
2	Alice	6	22
3	Charlie	5	23
4	David	7	4
5	Eve	5	20
NULL	NULL	NULL	NULL

- **Boat Table**



bid	bname	color
101	Boat A	red
102	Boat B	green
103	Boat C	blue
104	Boat D	red
NULL	NULL	NULL

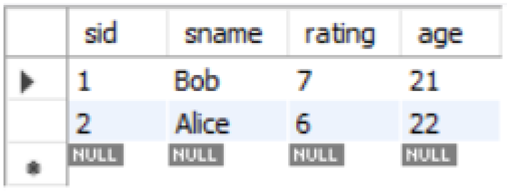
- **Reserves Table**



sid	bid	day
1	101	2024-10-01
1	102	2024-10-02
1	103	2024-10-01
2	101	2024-10-02
2	101	2024-10-03
2	102	2024-10-01
3	103	2024-10-01
3	104	2024-10-01
4	104	2024-10-02
5	102	2024-10-03
NULL	NULL	NULL

1. Find all information of sailors who have reserved boat number 101.

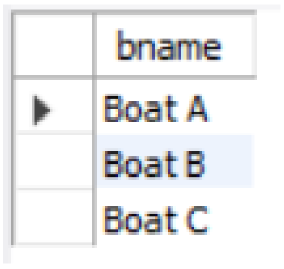
```
SELECT *
FROM Sailors
WHERE sid IN (SELECT sid FROM Reserves WHERE bid = 101);
```



sid	sname	rating	age
1	Bob	7	21
2	Alice	6	22
NULL	NULL	NULL	NULL

2. Find the name of boat reserved by Bob.

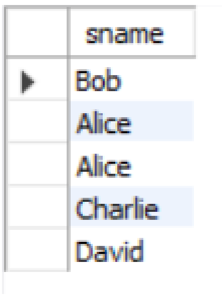
```
SELECT b.bname
FROM Boats b
JOIN Reserves r ON b.bid = r.bid
JOIN Sailors s ON r.sid = s.sid
WHERE s.sname = 'Bob';
```



```
bname
Boat A
Boat B
Boat C
```

3. Find the names of sailors who have reserved a red boat, and list in the order of age.

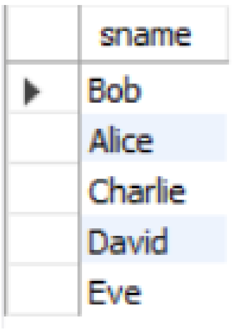
```
SELECT s.sname
FROM Sailors s
JOIN Reserves r ON s.sid = r.sid
JOIN Boats b ON r.bid = b.bid
WHERE b.color = 'red'
ORDER BY s.age;
```



```
sname
Bob
Alice
Alice
Charlie
David
```

4. Find the names of sailors who have reserved at least one boat.

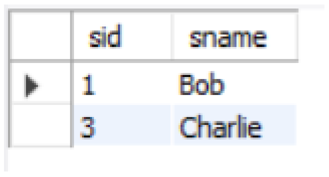
```
SELECT DISTINCT s.sname
FROM Sailors s
JOIN Reserves r ON s.sid = r.sid;
```



```
sname
Bob
Alice
Charlie
David
Eve
```

5. Find the ids and names of sailors who have reserved two different boats on the same day.

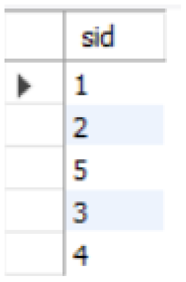
```
SELECT r1.sid, s.sname
FROM Reserves r1
JOIN Reserves r2 ON r1.sid = r2.sid AND r1.day = r2.day AND r1.bid <> r2.bid
JOIN Sailors s ON r1.sid = s.sid
GROUP BY r1.sid, s.sname;
```



	sid	sname
▶	1	Bob
	3	Charlie

6. Find the ids of sailors who have reserved a red boat or a green boat.

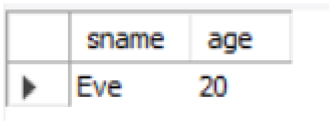
```
SELECT DISTINCT r.sid
FROM Reserves r
JOIN Boats b ON r.bid = b.bid
WHERE b.color IN ('red', 'green');
```



	sid
▶	1
	2
	5
	3
	4

7. Find the name and the age of the youngest sailor.

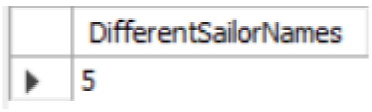
```
SELECT s.sname, s.age
FROM Sailors s
WHERE s.age = (SELECT MIN(age) FROM Sailors);
```



	sname	age
▶	Eve	20

8. Count the number of different sailor names.

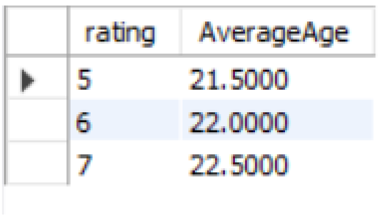
```
SELECT COUNT(DISTINCT sname) AS DifferentSailorNames
FROM Sailors;
```



	DifferentSailorNames
▶	5

9. Find the average age of sailors for each rating level.

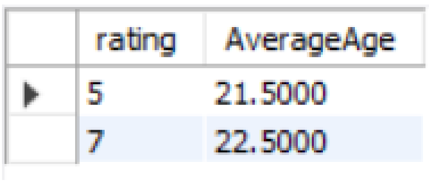
```
SELECT rating, AVG(age) AS AverageAge
FROM Sailors
GROUP BY rating;
```



	rating	AverageAge
▶	5	21.5000
	6	22.0000
	7	22.5000

10. Find the average age of sailors for each rating level that has at least two sailors.

```
SELECT rating, AVG(age) AS AverageAge
FROM Sailors
GROUP BY rating
HAVING COUNT(sid) >= 2;
```



	rating	AverageAge
▶	5	21.5000
	7	22.5000

Conclusion:

The implementation of SQL joins using the **Sailors**, **Boats**, and **Reserves** schema demonstrates how to efficiently manage and query relational data. By executing various queries, we showcased the utility of **inner joins** to filter related information and the potential of **outer joins** and **natural joins** for broader data retrieval.

Experiment No. 8

Title:

Study & Implementation of Various Types of Clauses and Indexing.

Aim:

To understand and implement various SQL clauses such as GROUP BY, HAVING, ORDER BY, and WHERE, as well as the concepts of indexing (unique and clustered) in relational databases.

Objectives:

1. To create a relational database and implement various SQL queries.
2. To analyse employee data using aggregate functions and grouping.
3. To demonstrate the use of indexing to optimize data retrieval.
4. To practice SQL queries for real-world scenarios involving employee management.

Theory

SQL (Structured Query Language) is the standard language for managing and manipulating relational databases. This experiment focuses on several key SQL clauses, which are essential for querying and organizing data efficiently. The following sections will cover various SQL clauses as well as a comprehensive overview of indexing, its types, and its significance.

SQL Clauses

- **GROUP BY:**
  The GROUP BY clause is used to arrange identical data into groups. This is particularly useful when combined with aggregate functions such as SUM(), COUNT(), AVG(), etc. For instance, grouping employee data by department allows one to calculate the total salary per department.
- **HAVING:**
  The HAVING clause is applied to filter records that work with aggregate functions. Unlike the WHERE clause, which filters records before any groupings are made, HAVING filters

groups after the aggregation has occurred. For example, one can use HAVING to find departments with an average salary above a certain threshold.

- **ORDER BY:**
  The ORDER BY clause is used to sort the result set of a query by one or more columns, either in ascending (ASC) or descending (DESC) order. This can help in presenting the data in a more organized manner, such as listing employees by their salaries from highest to lowest.
- **WHERE:**

  The WHERE clause is used to filter records based on specific conditions before any aggregation occurs. It allows for the selection of records that meet certain criteria, such as finding all employees with a salary greater than a specified amount.

Indexing -

Indexing is a crucial aspect of database performance, serving as a data structure that improves the speed of data retrieval operations on a database table. An index allows the database management system (DBMS) to find and access the required rows quickly, rather than scanning the entire table. This is especially important for large datasets where performance can significantly impact user experience.

Types of Indexing:

- **Unique Index:**
  A unique index ensures that all values in a column are different. It prevents duplicate values in the indexed column, which is crucial for columns that serve as primary keys or unique identifiers. For instance, if an employee ID column is indexed as unique, the database will reject any attempt to insert another record with the same employee ID.

- **Clustered Index:**
  A clustered index determines the physical order of data rows in the table. This means that the data is stored on disk in the same order as the index. In a clustered index, there can only be one index per table because the data rows can only be sorted in one way. The primary key of a table is typically implemented as a clustered index.

- **Non-Clustered Index**:
  A non-clustered index creates a separate structure from the data rows, maintaining a pointer to the actual data. This allows multiple non-clustered indexes on a table. For example, a non-clustered index can be created on employee names to quickly find specific employees without having to scan the entire employee table.
  When designing a database, careful consideration of indexing strategies is essential. Over-indexing can lead to performance degradation during data modifications (inserts, updates, deletes) since the indexes must be updated as well. Therefore, it's important to balance the need for speed in data retrieval with the overhead that indexing introduces.

Outputs:

The outputs will consist of the results of the implemented SQL queries, which include:

Total salary spent for each job category.

Details of the lowest-paid employee under each manager.

Number of employees in each department along with department names.

Employee details sorted by salary.

Records of employees earning more than 16,000 in each department.

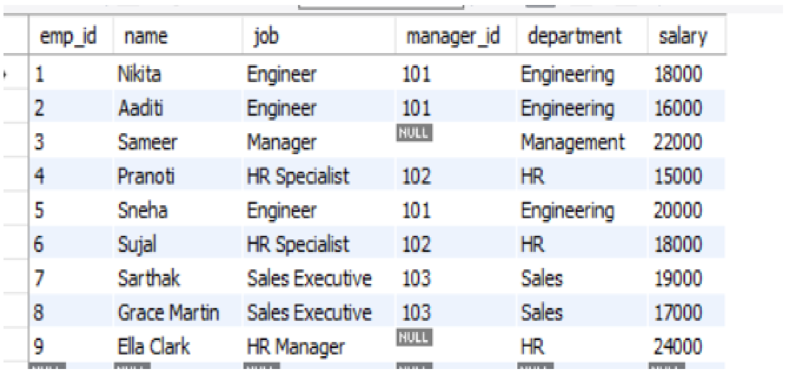
6) Solve Lab Practice:

a) Create a relation and implement the following queries:

Query:

```
CREATE TABLE employee (

emp_id INT PRIMARY KEY,

name VARCHAR(50),

job VARCHAR(50),

manager_id INT,

department VARCHAR(50),

salary INT
);

INSERT INTO employee (emp_id, name, job, manager_id, department, salary) VALUES

(1, ‘Nikita’, ‘Engineer’, 101, ‘Engineering’, 18000),

(2, ‘Aaditi’, ‘Engineer’, 101, ‘Engineering’, 16000),

(3, ‘Sameer’, ‘Manager’, NULL, ‘Management’, 22000),

(4, ‘Pranoti’, ‘HR Specialist’, 102, ‘HR’, 15000),

(5, ‘Sneha’, ‘Engineer’, 101, ‘Engineering’, 20000),

(6, ‘Sujal’, ‘HR Specialist’, 102, ‘HR’, 18000),

(7, ‘Sarthak’, ‘Sales Executive’, 103, ‘Sales’, 19000),

(8, ‘Grace Martin’, ‘Sales Executive’, 103, ‘Sales’, 17000),

(9, ‘Ella Clark’, ‘HR Manager’, NULL, ‘HR’, 24000);
```

Output:



emp_id	name	job	manager_id	department	salary
1	Nikita	Engineer	101	Engineering	18000
2	Aaditi	Engineer	101	Engineering	16000
3	Sameer	Manager	NULL	Management	22000
4	Pranoti	HR Specialist	102	HR	15000
5	Sneha	Engineer	101	Engineering	20000
6	Sujal	HR Specialist	102	HR	18000
7	Sarthak	Sales Executive	103	Sales	19000
8	Grace Martin	Sales Executive	103	Sales	17000
9	Ella Clark	HR Manager	NULL	HR	24000

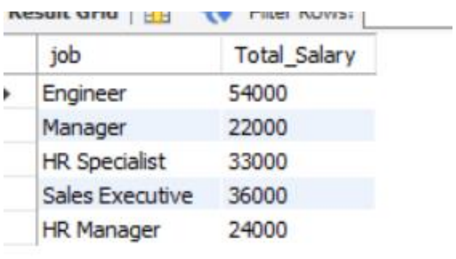
b) Queries Implementation:(input random data in table)

Total salary spent for each job category:

Query:

```
SELECT job, SUM(salary) AS Total_Salary

FROM employee

GROUP BY job;
```

Output:



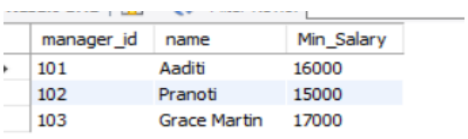
job	Total_Salary
Engineer	54000
Manager	22000
HR Specialist	33000
Sales Executive	36000
HR Manager	24000

c)Lowest paid employee details under each manager:

Query:

```
SELECT manager_id, name, MIN(salary) AS Min_Salary

FROM employee

GROUP BY manager_id

ORDER BY manager_id;
```

Output:



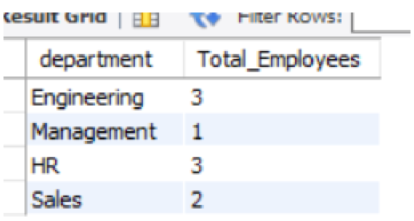
manager_id	name	Min_Salary
101	Aaditi	16000
102	Pranoti	15000
103	Grace Martin	17000

d)Number of employees working in each department and their department name:

Query:

```
SELECT department, COUNT(emp_id) AS Total_Employees

FROM employee

GROUP BY department;
```

Output:



department	Total_Employees
Engineering	3
Management	1
HR	3
Sales	2

e) Details of employees sorting the salary in increasing order:

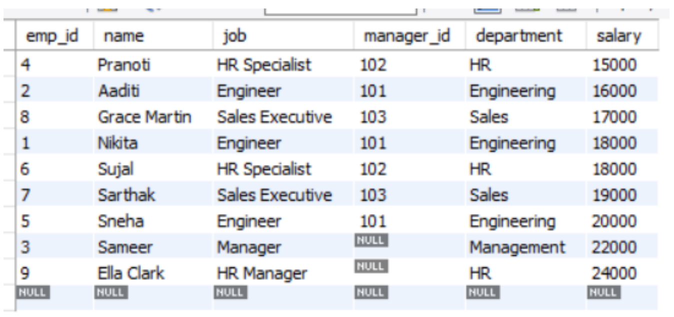
Query:

SELECT *

FROM employee

ORDER BY salary ASC;

Output:



emp_id	name	job	manager_id	department	salary
4	Pranoti	HR Specialist	102	HR	15000
2	Aaditi	Engineer	101	Engineering	16000
8	Grace Martin	Sales Executive	103	Sales	17000
1	Nikita	Engineer	101	Engineering	18000
6	Sujal	HR Specialist	102	HR	18000
7	Sarthak	Sales Executive	103	Sales	19000
5	Sneha	Engineer	101	Engineering	20000
3	Sameer	Manager	NULL	Management	22000
9	Ella Clark	HR Manager	NULL	HR	24000
NULL	NULL	NULL	NULL	NULL	NULL

f) Record of employees earning a salary greater than 16000 in each department:

Query:

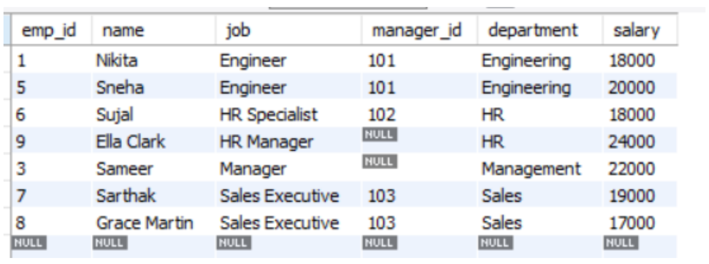
SELECT *

FROM employee

WHERE salary > 16000

ORDER BY department;

Output:



emp_id	name	job	manager_id	department	salary
1	Nikita	Engineer	101	Engineering	18000
5	Sneha	Engineer	101	Engineering	20000
6	Sujal	HR Specialist	102	HR	18000
9	Ella Clark	HR Manager	NULL	HR	24000
3	Sameer	Manager	NULL	Management	22000
7	Sarthak	Sales Executive	103	Sales	19000
8	Grace Martin	Sales Executive	103	Sales	17000
NULL	NULL	NULL	NULL	NULL	NULL

g) Create Unique and Clustered Index on the given Database:

-- Create a unique index on emp_id

Query:

CREATE UNIQUE INDEX idx_emp_id ON employee(emp_id);

SHOW INDEX FROM employee;

Output:

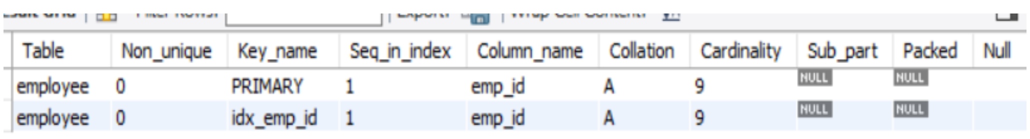


Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null
employee	0	PRIMARY	1	emp_id	A	9	NULL	NULL	
employee	0	idx_emp_id	1	emp_id	A	9	NULL	NULL	

Query:

-- Create a clustered index on salary

CREATE CLUSTERED INDEX idx_salary ON employee(salary);

SHOW INDEX FROM employee;

Output:

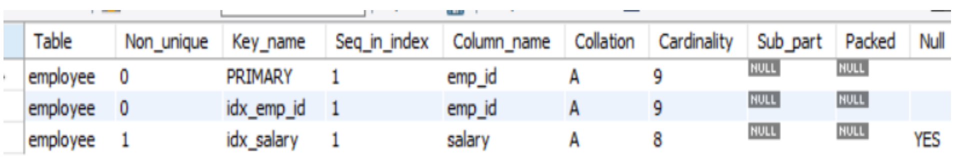


Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null
employee	0	PRIMARY	1	emp_id	A	9	NULL	NULL	
employee	0	idx_emp_id	1	emp_id	A	9	NULL	NULL	
employee	1	idx_salary	1	salary	A	8	NULL	NULL	YES

Conclusion:

The experiment successfully demonstrated the use of various SQL clauses for data manipulation and retrieval in a relational database. By creating a database of employee records, executing various queries, and implementing indexing techniques, the efficiency and effectiveness of data operations were enhanced. This practical exercise reinforced the theoretical concepts learned in class and highlighted the importance of structured querying and indexing in database management systems.

Experiment No. 9

Title:

Study and Implementation of Triggers and Views in SQL.

Aim:

To understand the concept of SQL triggers and views, and implement them using the Sailors and Boats tables.

Objectives:

1. Understand Triggers: To learn how to create triggers that automatically execute predefined actions when certain events (INSERT, UPDATE, DELETE) occur in a database table.
2. Implement Triggers on Sailors Table: To apply a trigger on the Sailors table to monitor changes such as updates or insertions, demonstrating the automated execution of tasks based on database events.
3. Understand Views: To understand the concept of views, which are virtual tables created by querying existing tables, allowing for data abstraction and query simplification.
4. Create and Use Views on Boats Table: To create a view on the Boats table that abstracts and simplifies access to the data, making it easier to query specific information like boat names and colours.
5. Analyze the Benefits of Triggers and Views: To explore the practical benefits of using triggers for automation and views for data abstraction and security, understanding how they improve the functionality and performance of database systems.

Theory

- **Trigger:**
  A trigger in SQL is a special type of stored procedure that is automatically executed when a specific event occurs in the database, such as an INSERT, UPDATE, or DELETE. Triggers help maintain the integrity of the database by automating actions. They can be defined to execute either:

BEFORE: The trigger executes before the operation.

AFTER: The trigger executes after the operation.

Triggers are particularly useful in ensuring that certain business rules or constraints are enforced automatically without manual intervention. For example, a trigger can automatically update a sailor's rating if their age is updated to exceed a certain threshold.

View:

A view in SQL is a virtual table based on the result of a SELECT query. Unlike actual tables, views do not store data physically but provide a window into existing tables. Views are helpful for:

· Simplifying complex queries: A view can encapsulate a complicated query, making it easier to reuse.

· Data security: By limiting the columns or rows exposed, views ensure that users see only the data they need.

Views are especially useful when we need to show only a subset of data, such as boats of a specific color.

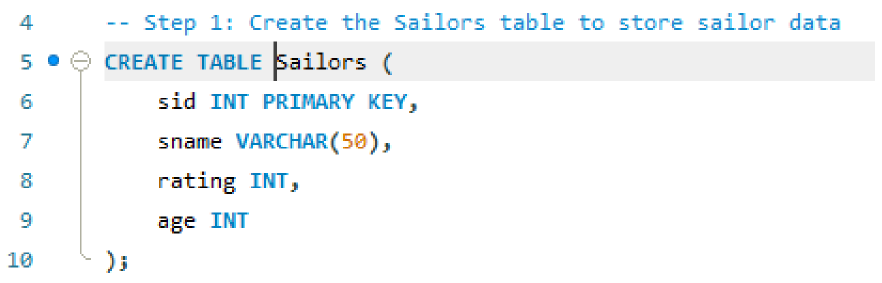
Sailors Table: This table contains details about sailors, including their unique ID (sid), name (sname), rating, and age.

Boats Table: This table contains details about boats, including their unique ID (bid), name (bname), and color.

- **Trigger for Sailors Table**: We create a trigger that automatically updates the rating of a sailor when their age is updated to be greater than 30.
- **View for Boats Table**: A view is created to show boats of a specific color, such as 'Red'.

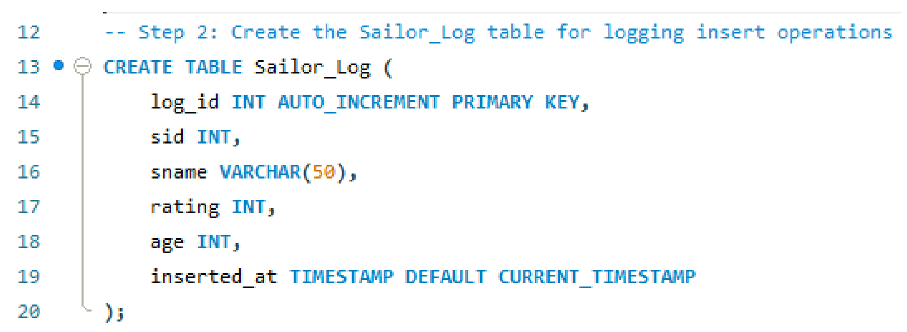
Trigger Implementation

Step 1: Create the Sailors table



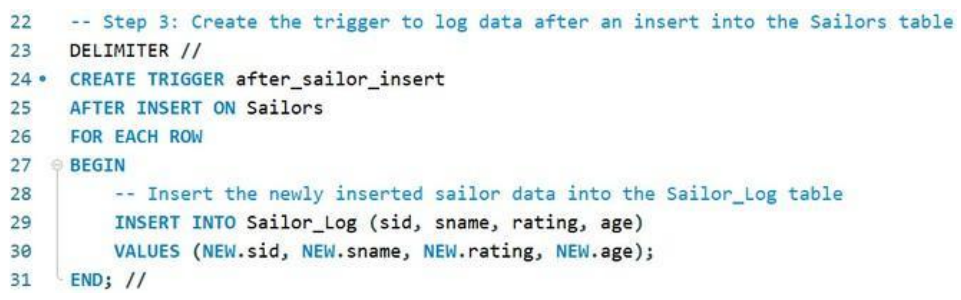
```sql
-- Step 1: Create the Sailors table to store sailor data
CREATE TABLE Sailors (
    sid INT PRIMARY KEY,
    sname VARCHAR(50),
    rating INT,
    age INT
);
```

Step 2: Create the Sailor_Log table for logging insert operations



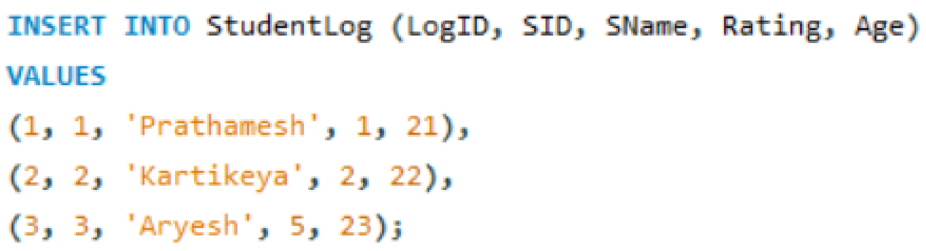
```sql
-- Step 2: Create the Sailor_Log table for logging insert operations
CREATE TABLE Sailor_Log (
    log_id INT AUTO_INCREMENT PRIMARY KEY,
    sid INT,
    sname VARCHAR(50),
    rating INT,
    age INT,
    inserted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 3: Create the Trigger to log data after an insert operation on the Sailors table



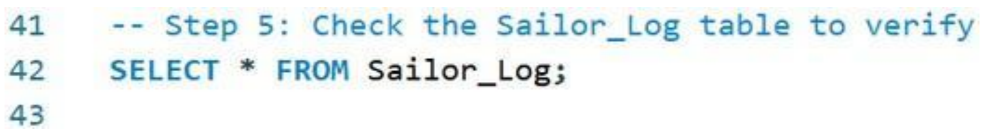
```sql
-- Step 3: Create the trigger to log data after an insert into the Sailors table
DELIMITER //
CREATE TRIGGER after_sailor_insert
AFTER INSERT ON Sailors
FOR EACH ROW
BEGIN
    -- Insert the newly inserted sailor data into the Sailor_Log table
    INSERT INTO Sailor_Log (sid, sname, rating, age)
    VALUES (NEW.sid, NEW.sname, NEW.rating, NEW.age);
END; //
```

Step 4: Insert sample data into the Sailors table to test the trigger



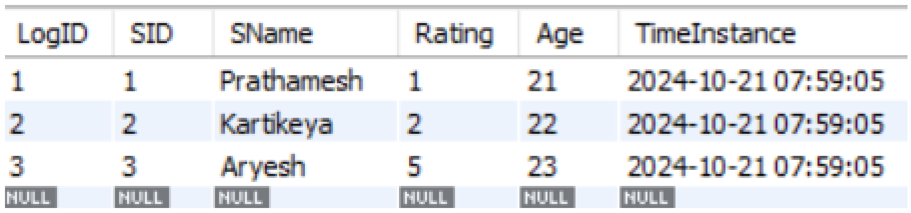
```sql
INSERT INTO StudentLog (LogID, SID, SName, Rating, Age)
VALUES
(1, 1, 'Prathamesh', 1, 21),
(2, 2, 'Kartikeya', 2, 22),
(3, 3, 'Aryesh', 5, 23);
```

Step 5: Check the Sailor_Log table to verify the trigger worked



```
-- Step 5: Check the Sailor_Log table to verify
SELECT * FROM Sailor_Log;

```

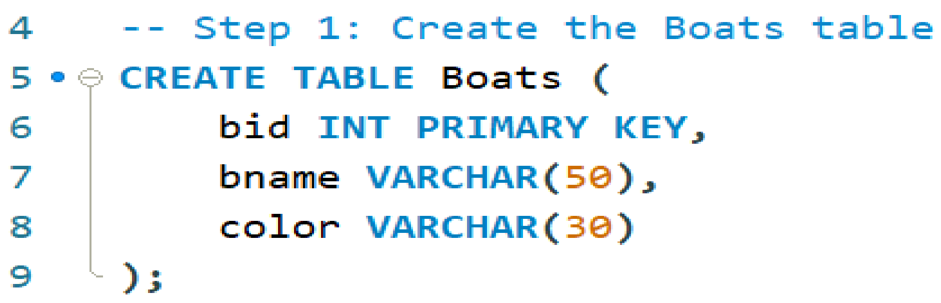
Output:



LogID	SID	SName	Rating	Age	TimeInstance
1	1	Prathamesh	1	21	2024-10-21 07:59:05
2	2	Kartikeya	2	22	2024-10-21 07:59:05
3	3	Aryesh	5	23	2024-10-21 07:59:05
NULL	NULL	NULL	NULL	NULL	NULL

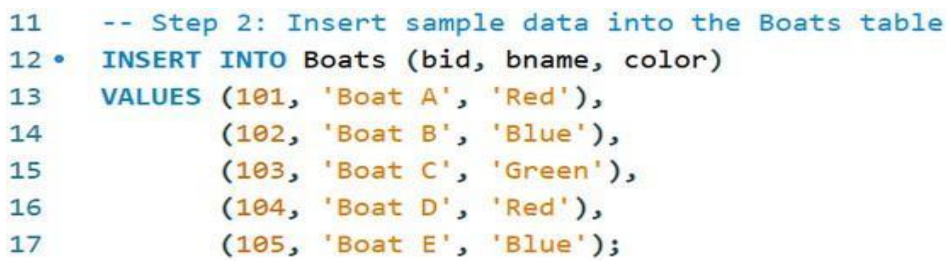
View Implementation

Step 1: Create the Boats table



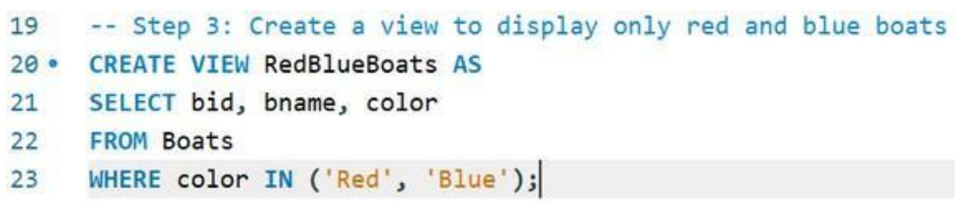
```
-- Step 1: Create the Boats table
CREATE TABLE Boats (
    bid INT PRIMARY KEY,
    bname VARCHAR(50),
    color VARCHAR(30)
);
```

Step 2: Insert sample data into the Boats table



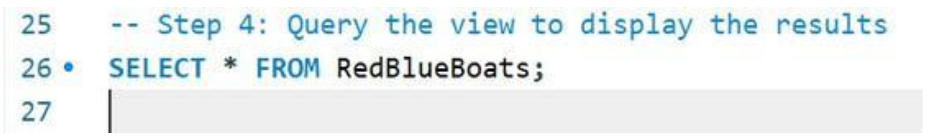
```
-- Step 2: Insert sample data into the Boats table
INSERT INTO Boats (bid, bname, color)
VALUES (101, 'Boat A', 'Red'),
       (102, 'Boat B', 'Blue'),
       (103, 'Boat C', 'Green'),
       (104, 'Boat D', 'Red'),
       (105, 'Boat E', 'Blue');
```

Step 3: Create a view to display only boats with color 'Red' or 'Blue'



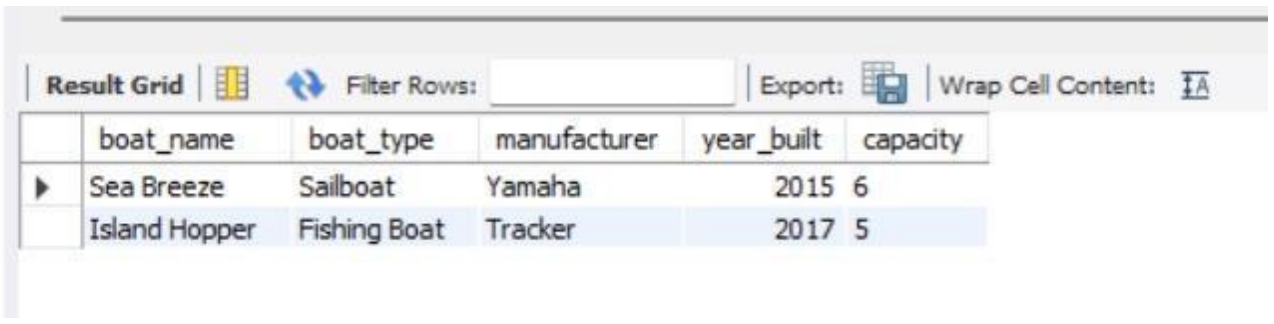
```sql
-- Step 3: Create a view to display only red and blue boats
CREATE VIEW RedBlueBoats AS
SELECT bid, bname, color
FROM Boats
WHERE color IN ('Red', 'Blue');
```

Step 4: Query the RedBlueBoats view to see the results



```sql
-- Step 4: Query the view to display the results
SELECT * FROM RedBlueBoats;
```

Output :



boat_name	boat_type	manufacturer	year_built	capacity
Sea Breeze	Sailboat	Yamaha	2015	6
Island Hopper	Fishing Boat	Tracker	2017	5

Conclusion:

By creating a customer_information database and applying DQL and TCL commands, we can efficiently manage and retrieve customer data while ensuring transaction integrity. DQL allows for precise querying of data, and TCL ensures safe and controlled transaction handling, thus maintaining the consistency and reliability of the database.

Experiment No - 10

Title - Study and implement the CRUD operations on Mongo DB Database.

Objective -

The primary objective of CRUD (Create, Read, Update, Delete) operations is to enable efficient interaction with a database by providing a standardized way to manage and manipulate data. Specifically, CRUD operations serve the following purposes:

1. **Create**: To allow users to **add new records** (documents) to the database, ensuring that new data can be stored.

2. **Read**: To enable users to **retrieve and view existing data** from the database, often with filters and queries to display specific information.

3. **Update**: To provide the capability to **modify existing data** in the database, ensuring that records can be edited and kept up to date without creating new records.

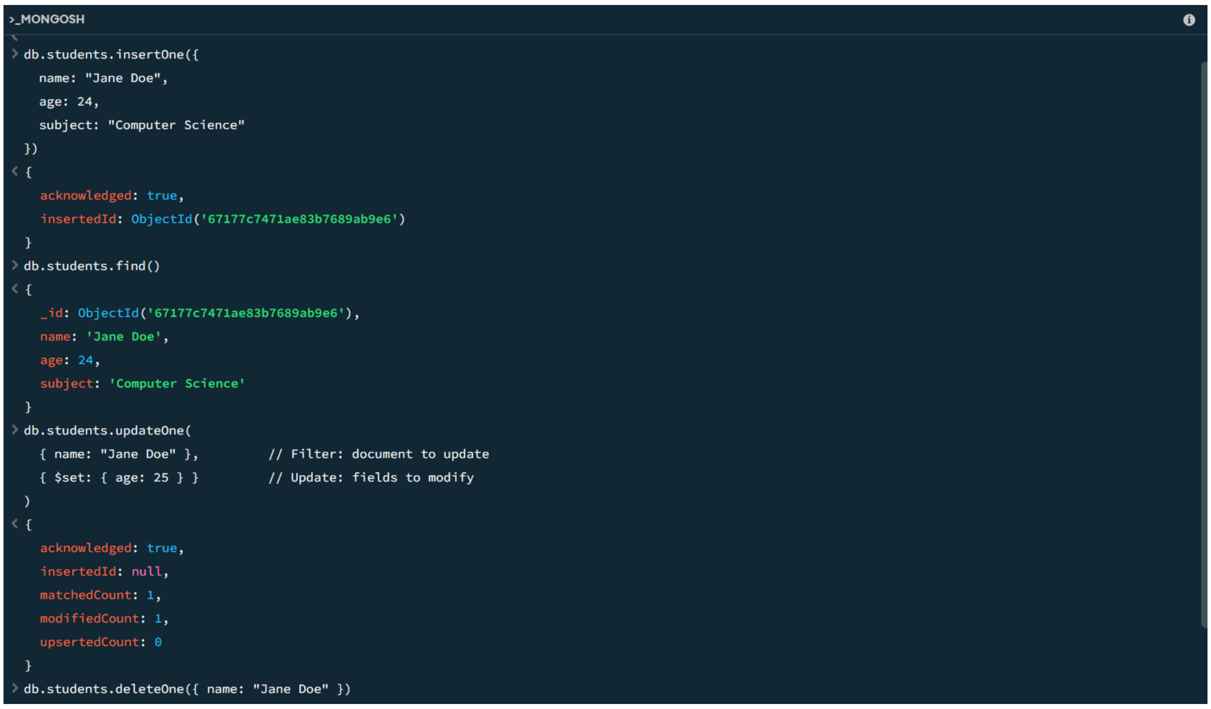
4. **Delete**: To enable the removal of **unecessary or outdated data** from the database, ensuring that the database only contains relevant information..

Code -

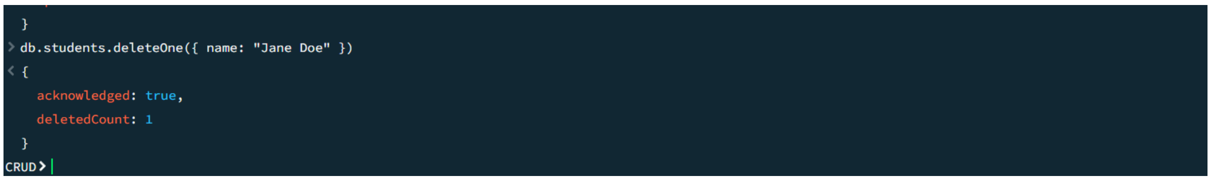
```
// Step 1: Switch to the CRUD database (will be created if it doesn't exist)
use CRUD;

// Step 2: Create (Insert) a Document into the 'students' collection
db.students.insertOne({
  name: "Jane Doe",
  age: 24,
  subject: "Computer Science"
});

// Step 3: Read (Query) all Documents in the 'students' collection
```

```
db.students.find();

// Step 4: Query (Read) specific documents, filtering by 'name'
db.students.find({ name: "Jane Doe" });

// Step 5: Update a specific Document (e.g., change Jane Doe's age to 25)
db.students.updateOne(
  { name: "Jane Doe" },        // Filter: document to update
  { $set: { age: 25 } }        // Update: fields to modify
);

// Step 6: Delete a specific Document (e.g., remove Jane Doe)
db.students.deleteOne({ name: "Jane Doe" });
```

Output -



```
>_MONGOSH
> db.students.insertOne({
    name: "Jane Doe",
    age: 24,
    subject: "Computer Science"
  })
< {
    acknowledged: true,
    insertedId: ObjectId('67177c7471ae83b7689ab9e6')
  }
> db.students.find()
< {
    _id: ObjectId('67177c7471ae83b7689ab9e6'),
    name: 'Jane Doe',
    age: 24,
    subject: 'Computer Science'
  }
> db.students.updateOne(
    { name: "Jane Doe" },         // Filter: document to update
    { $set: { age: 25 } }         // Update: fields to modify
  )
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 1,
    modifiedCount: 1,
    upsertedCount: 0
  }
> db.students.deleteOne({ name: "Jane Doe" })
```



```
  }
> db.students.deleteOne({ name: "Jane Doe" })
< {
    acknowledged: true,
    deletedCount: 1
  }
CRUD>
```

Conclusion:

CRUD operations are fundamental for interacting with databases, offering a standardized framework to manage data effectively. Through Create, Read, Update, and Delete functionalities, users can seamlessly add, view, modify, and remove records, ensuring data remains accurate, relevant, and up-to-date. These operations form the core of database management, enabling efficient data handling, improving system performance, and maintaining data integrity in real-world applications. Mastery of CRUD operations is essential for any developer working with databases.

Experiment No. 11

Title:

Filtering Data Efficiently on MongoDB Database

Aim:

To perform efficient data filtering operations on a MongoDB database and explore querying techniques to retrieve relevant data without duplication.

Objectives:

- Understand the MongoDB query language (MQL) for efficient data filtering.
- Learn how to use operators like $match, $and, $or, and others for advanced querying.
- Avoid data duplication during filtering operations.
- Implement and verify efficient data filtering in MongoDB using real datasets.

Theory:

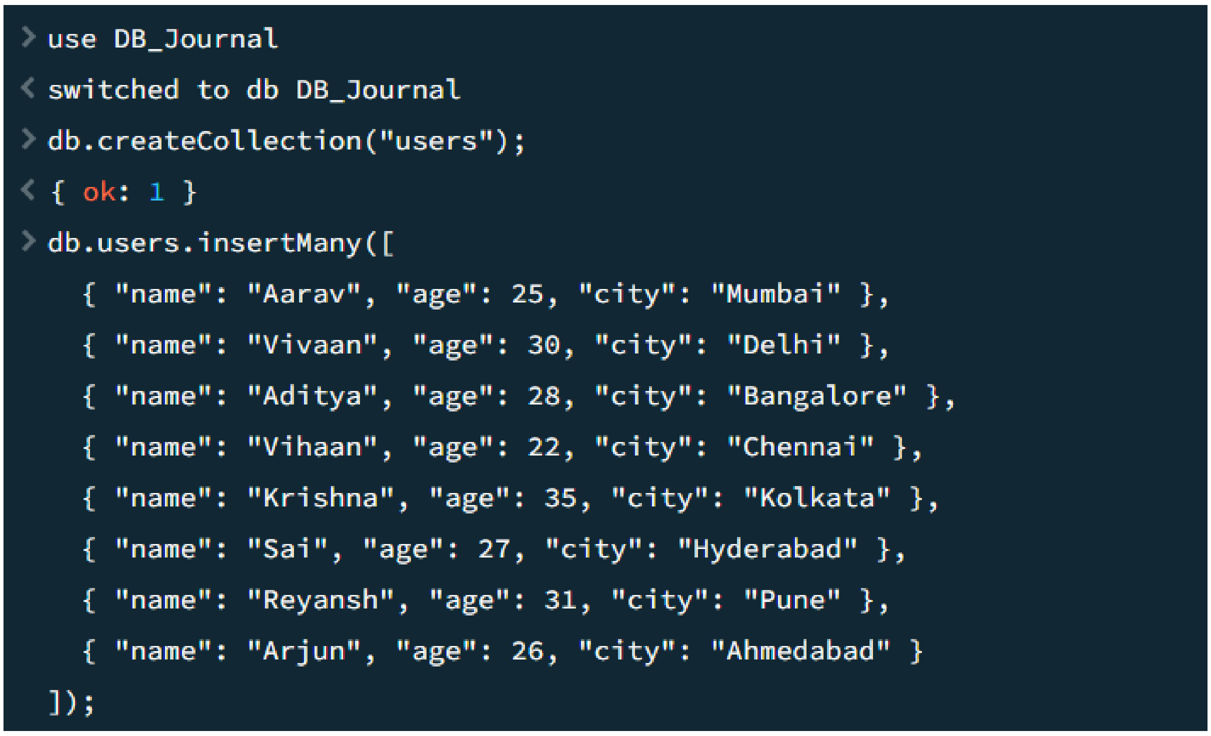
MongoDB is a NoSQL database that stores data in flexible, JSON-like documents. One of its key advantages is the ability to filter and retrieve data efficiently using a powerful query language known as MongoDB Query Language (MQL).

- MongoDB Documents: MongoDB stores data in BSON (Binary JSON), which is a binary representation of JSON-like documents.
- Queries in MongoDB: MongoDB provides the $match operator for filtering documents in collections. Other operators include $and, $or, $in, $gt, $lt, etc., which help in refining data filtering.
- Indexes: Indexes improve the efficiency of query execution. MongoDB supports various types of indexes, such as single-field, compound, and multi-key indexes.

By mastering MongoDB's querying capabilities, you can avoid retrieving unnecessary or duplicate data, thus ensuring that your operations are as efficient as possible.

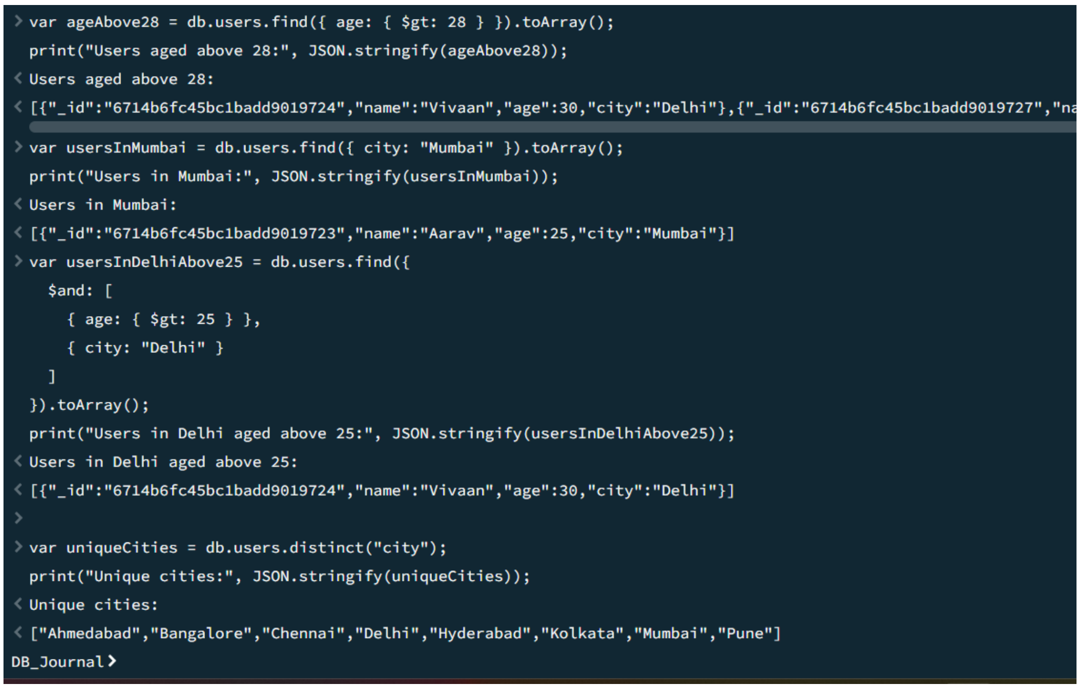
Outputs:

1] Data:



```
> use DB_Journal
< switched to db DB_Journal
> db.createCollection("users");
< { ok: 1 }
> db.users.insertMany([
    { "name": "Aarav", "age": 25, "city": "Mumbai" },
    { "name": "Vivaan", "age": 30, "city": "Delhi" },
    { "name": "Aditya", "age": 28, "city": "Bangalore" },
    { "name": "Vihaan", "age": 22, "city": "Chennai" },
    { "name": "Krishna", "age": 35, "city": "Kolkata" },
    { "name": "Sai", "age": 27, "city": "Hyderabad" },
    { "name": "Reyansh", "age": 31, "city": "Pune" },
    { "name": "Arjun", "age": 26, "city": "Ahmedabad" }
  ]);
```

Data filtering queries:



```
> var ageAbove28 = db.users.find({ age: { $gt: 28 } }).toArray();
  print("Users aged above 28:", JSON.stringify(ageAbove28));
< Users aged above 28:
< [{"_id":"6714b6fc45bc1badd9019724","name":"Vivaan","age":30,"city":"Delhi"},{"_id":"6714b6fc45bc1badd9019727","na
> var usersInMumbai = db.users.find({ city: "Mumbai" }).toArray();
  print("Users in Mumbai:", JSON.stringify(usersInMumbai));
< Users in Mumbai:
< [{"_id":"6714b6fc45bc1badd9019723","name":"Aarav","age":25,"city":"Mumbai"}]
> var usersInDelhiAbove25 = db.users.find({
    $and: [
      { age: { $gt: 25 } },
      { city: "Delhi" }
    ]
  }).toArray();
  print("Users in Delhi aged above 25:", JSON.stringify(usersInDelhiAbove25));
< Users in Delhi aged above 25:
< [{"_id":"6714b6fc45bc1badd9019724","name":"Vivaan","age":30,"city":"Delhi"}]
>
> var uniqueCities = db.users.distinct("city");
  print("Unique cities:", JSON.stringify(uniqueCities));
< Unique cities:
< ["Ahmedabad","Bangalore","Chennai","Delhi","Hyderabad","Kolkata","Mumbai","Pune"]
DB_Journal>
```

1. Retrieve specific records based on certain field criteria.
2. Use aggregation and filtering to extract and display non-duplicate data.
3. Optimize data retrieval by using MongoDB indexes.

---

6) Lab Practice - Example Queries

```
db.users.insertMany([
  { "name": "Aarav", "age": 25, "city": "Mumbai" },
  { "name": "Vivaan", "age": 30, "city": "Delhi" },
  { "name": "Aditya", "age": 28, "city": "Bangalore" },
  { "name": "Vihaan", "age": 22, "city": "Chennai" },
  { "name": "Krishna", "age": 35, "city": "Kolkata" },
  { "name": "Sai", "age": 27, "city": "Hyderabad" },
  { "name": "Reyansh", "age": 31, "city": "Pune" },
  { "name": "Arjun", "age": 26, "city": "Ahmedabad" }
]);



var ageAbove28 = db.users.find({ age: { $gt: 28 } }).toArray();
print("Users aged above 28:", JSON.stringify(ageAbove28));

var usersInMumbai = db.users.find({ city: "Mumbai" }).toArray();
print("Users in Mumbai:", JSON.stringify(usersInMumbai));

var usersInMumbai = db.users.find({ city: "Mumbai" }).toArray();
print("Users in Mumbai:", JSON.stringify(usersInMumbai));

var usersInDelhiAbove25 = db.users.find({
  $and: [
    { age: { $gt: 25 } },
    { city: "Delhi" }
  ]
}).toArray();
```

```
print("Users in Delhi aged above 25:", JSON.stringify(usersInDelhiAbove25));

var uniqueCities = db.users.distinct("city");
print("Unique cities:", JSON.stringify(uniqueCities));
```

7) Solve Lab Practice

1. Apply queries to filter data based on age, location, and other attributes.
2. Avoid duplicates using MongoDB's distinct function and appropriate filtering techniques.

Conclusion:

Efficient filtering in MongoDB allows for precise retrieval of data without unnecessary overhead or duplication. By utilizing the available querying and indexing features, we can optimize data retrieval processes, ensuring faster and more relevant results from the database.

Experiment No - 12

Title:

Working with Command Prompts and Creating a Student Database Table in MariaDB

Aim:

To install and configure MariaDB using the Ubuntu terminal, create a student database table, and perform SQL operations like inserting and retrieving data through command prompts, demonstrating basic database management skills.

Objective:

To demonstrate how to use the Ubuntu terminal to install MariaDB, set it up, and manage databases

through the command line. This involves creating a new database, adding a table with structured

data, inserting sample records, and retrieving data using SQL queries.

Description:

MariaDB is an open-source relational database system and a popular alternative to MySQL. In this

task, you will use the Ubuntu terminal to install and manage MariaDB. The steps include setting up

MariaDB (if it’s not already installed), starting the service, creating a database, and defining a table structure. Additionally, you will insert sample student data and execute SQL queries to validate the operations.

Data Set:

The database will contain a students table with fields like student_id, student_name, course, and grade.

Procedure (Code):

1. Install MariaDB (if not already installed):

Update the system and install MariaDB using:

```
sudo apt update
```

```
sudo apt install mariadb-server
```

2. Start MariaDB Service:

Ensure the MariaDB service is active:

```
sudo systemctl start mariadb
```

Follow the prompts to configure the root password and remove unnecessary users or services.

4. Log into MariaDB:

Access the MariaDB shell as the root user:

```
sudo mariadb
```

5. Create a Database:

Create a new database named school_db:

```
CREATE DATABASE school_db;
```

6. Select the Database:

Switch to the new database:

```
USE school_db;
```

7. Create a Table:

Define the students table with the following columns:

* student_id: Primary key, auto-incremented

* student_name: Name of the student

* course: Course the student is enrolled in

* grade: Student's grade as a decimal value

```
CREATE TABLE students (
student_id INT PRIMARY KEY AUTO_INCREMENT,
student_name VARCHAR(100),
course VARCHAR(50),
grade DECIMAL(4, 2)
);
```

8. Insert Data into the Table:

Insert sample student records:

```
INSERT INTO students (student_name, course, grade) VALUES
('Vedant Lonkar', 'Computer Science', 88.50),
('Parth Lonkar', 'Mechanical Engineering', 82.00),
('Ananya Iyer', 'Electrical Engineering', 76.25);
```

9. View the Data:

Verify the inserted records by running:

```
SELECT * FROM students;
```

10. Exit MariaDB:

Once done, exit the MariaDB shell:

```
EXIT;
```

Output:

If everything is executed correctly, the SELECT query will return:

```
+------------+----------------+-----------------------+--------+
| student_id | student_name | course | grade |
+------------+----------------+-----------------------+--------+
| 1 | Vedant Lonkar | Computer Science | 88.50 |
| 2 | Parth Lonkar | Mechanical Engineering | 82.00 |
| 3 | Ananya Iyer | Electrical Engineering | 76.25 |
+------------+----------------+-----------------------+--------+
```

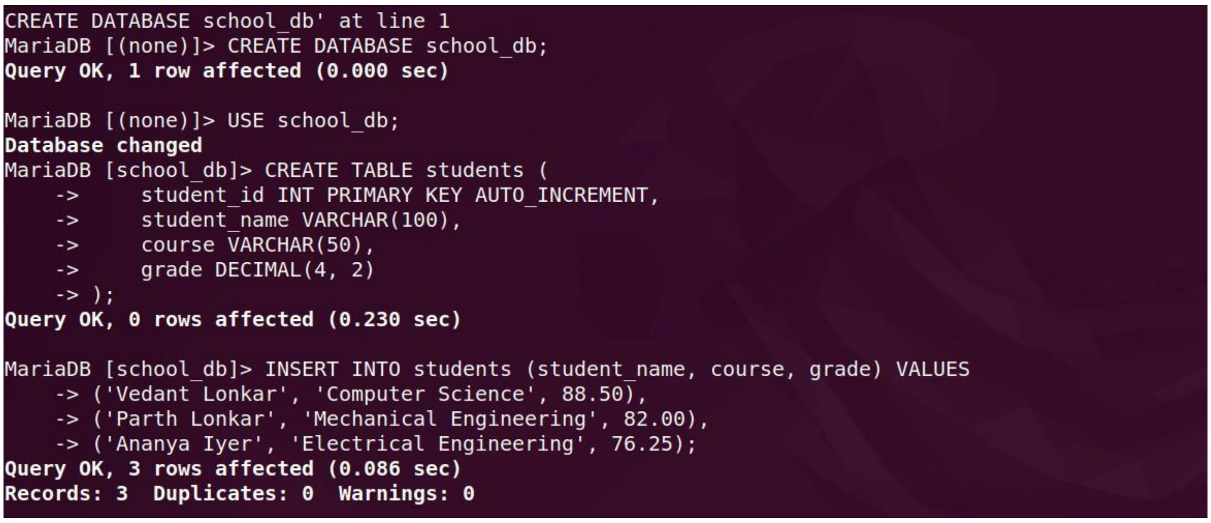
Conclusion:

Using the Ubuntu terminal and MariaDB, you successfully set up a database named school_db,

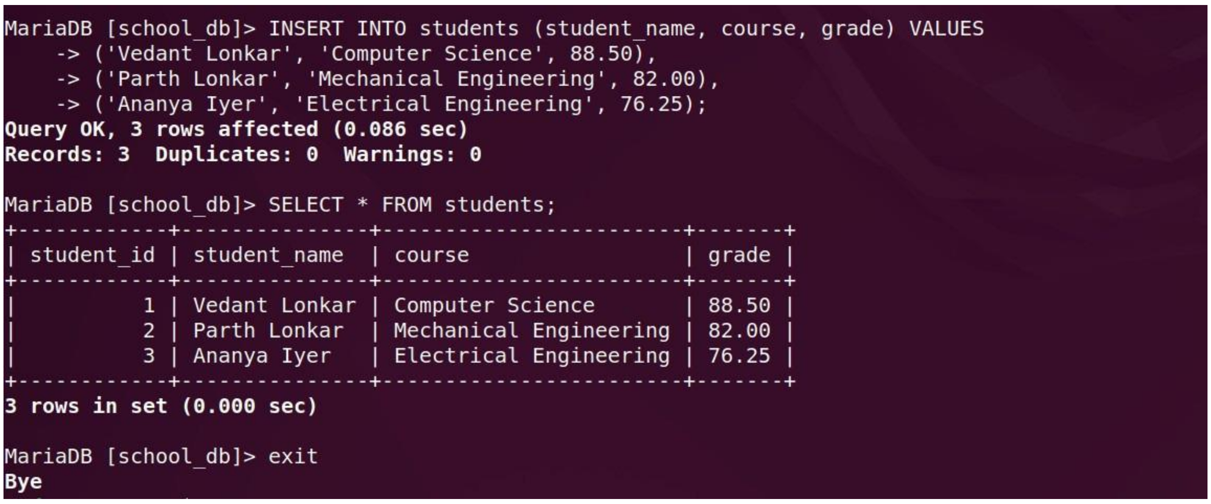
created a students table, and inserted sample records. You also queried the data to confirm your

operations, demonstrating basic skills in SQL database management via the command line. This

exercise provides a solid foundation for managing relational databases.



```
CREATE DATABASE school_db' at line 1
MariaDB [(none)]> CREATE DATABASE school_db;
Query OK, 1 row affected (0.000 sec)

MariaDB [(none)]> USE school_db;
Database changed
MariaDB [school_db]> CREATE TABLE students (
    ->     student_id INT PRIMARY KEY AUTO_INCREMENT,
    ->     student_name VARCHAR(100),
    ->     course VARCHAR(50),
    ->     grade DECIMAL(4, 2)
    -> );
Query OK, 0 rows affected (0.230 sec)

MariaDB [school_db]> INSERT INTO students (student_name, course, grade) VALUES
    -> ('Vedant Lonkar', 'Computer Science', 88.50),
    -> ('Parth Lonkar', 'Mechanical Engineering', 82.00),
    -> ('Ananya Iyer', 'Electrical Engineering', 76.25);
Query OK, 3 rows affected (0.086 sec)
Records: 3  Duplicates: 0  Warnings: 0
```



```
MariaDB [school_db]> INSERT INTO students (student_name, course, grade) VALUES
    -> ('Vedant Lonkar', 'Computer Science', 88.50),
    -> ('Parth Lonkar', 'Mechanical Engineering', 82.00),
    -> ('Ananya Iyer', 'Electrical Engineering', 76.25);
Query OK, 3 rows affected (0.086 sec)
Records: 3  Duplicates: 0  Warnings: 0

MariaDB [school_db]> SELECT * FROM students;
+------------+---------------+------------------------+-------+
| student_id | student_name  | course                 | grade |
+------------+---------------+------------------------+-------+
|          1 | Vedant Lonkar | Computer Science       | 88.50 |
|          2 | Parth Lonkar  | Mechanical Engineering | 82.00 |
|          3 | Ananya Iyer   | Electrical Engineering | 76.25 |
+------------+---------------+------------------------+-------+
3 rows in set (0.000 sec)

MariaDB [school_db]> exit
Bye
```

Conclusion :

The process of installing and configuring MariaDB on Ubuntu, followed by creating a student database and performing basic SQL operations like inserting and retrieving data, provides a solid foundation in database management skills. This hands-on approach helps users understand how to set up and interact with databases via command prompts, ensuring data can be effectively stored, manipulated, and retrieved. Mastering these essential operations builds a strong basis for managing more complex databases in real-world scenarios.

Experiment No. 13

Title:

Performing CRUD Operations in MariaDB

Aim:

To perform CRUD (Create, Read, Update, Delete) operations on a MariaDB database using SQL commands.

Objectives:

- To understand the basic CRUD operations in a database context.
- To learn how to create records, read data, update existing records, and delete data in a MariaDB table.
- To practice SQL queries and commands required to manage and manipulate data.
- To ensure data integrity and accuracy while performing these operations.

Theory:

CRUD operations are the four fundamental actions performed on databases to manage data. In MariaDB, these operations are carried out using SQL (Structured Query Language) commands:

- Create (C): Involves inserting new records into a table.
- Read (R): Involves querying data from the table.
- Update (U): Involves modifying existing data in the table.
- Delete (D): Involves removing data from the table.

SQL commands associated with these operations in MariaDB:

- **Create Table -**

Create table table_name(column1 datatype, column2 datatype,…);

- **Insert Into Table–**

INSERT INTO table_name (column1, column2, ...) VALUES (value1, value2, ...);

- **Read (Select Data)-**

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

- **Update (Modify Data)-**

UPDATE table_name SET column1 = value1, column2 = value2 WHERE condition;

- **Delete (Remove Data)-**

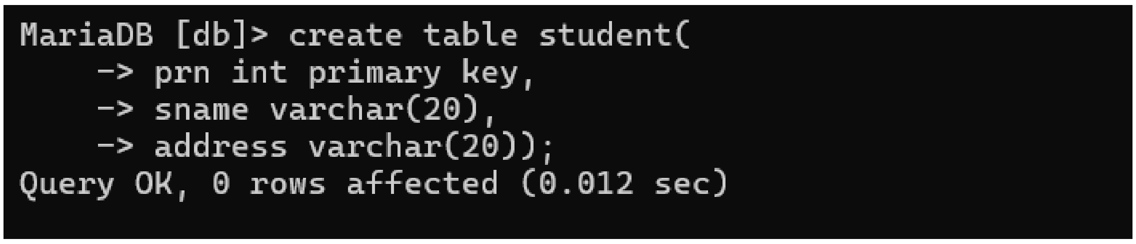
DELETE FROM table_name WHERE condition;

CRUD operations form the backbone of any relational database management system, allowing for dynamic and efficient data management. MariaDB facilitates these operations with standard SQL syntax.

Outputs:

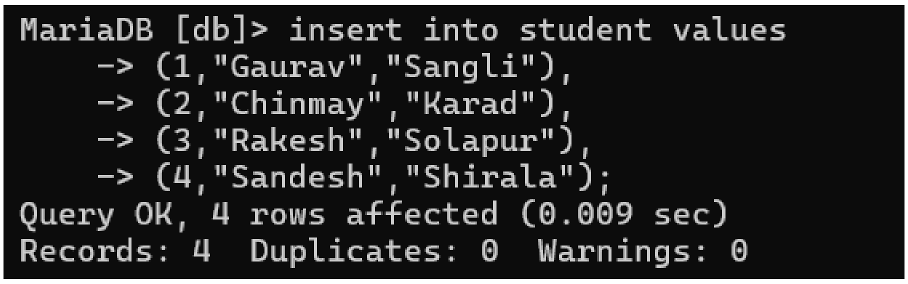
Create Operation (Insert Data)

- Create the Table-



```
MariaDB [db]> create table student(
    -> prn int primary key,
    -> sname varchar(20),
    -> address varchar(20));
Query OK, 0 rows affected (0.012 sec)
```

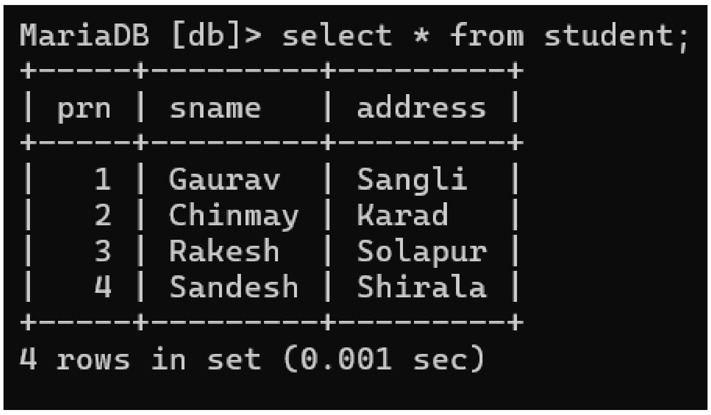
- Insert Records-



```
MariaDB [db]> insert into student values
    -> (1,"Gaurav","Sangli"),
    -> (2,"Chinmay","Karad"),
    -> (3,"Rakesh","Solapur"),
    -> (4,"Sandesh","Shirala");
Query OK, 4 rows affected (0.009 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

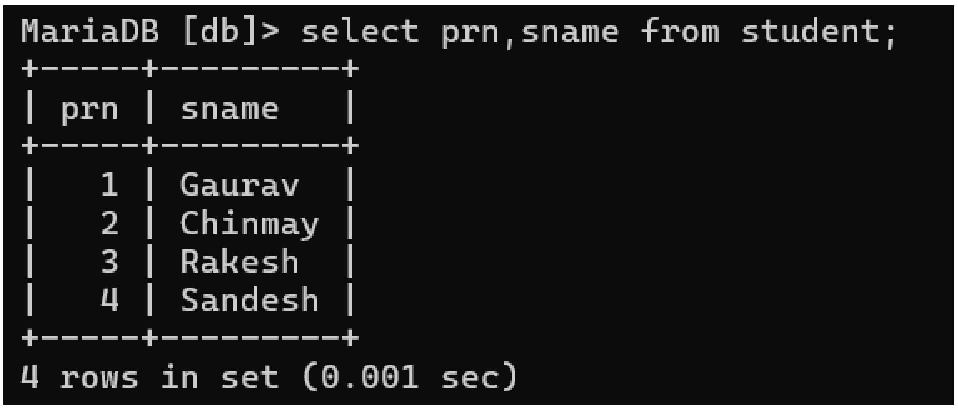
Read Operation (Retrieve Data)

- Select all records-



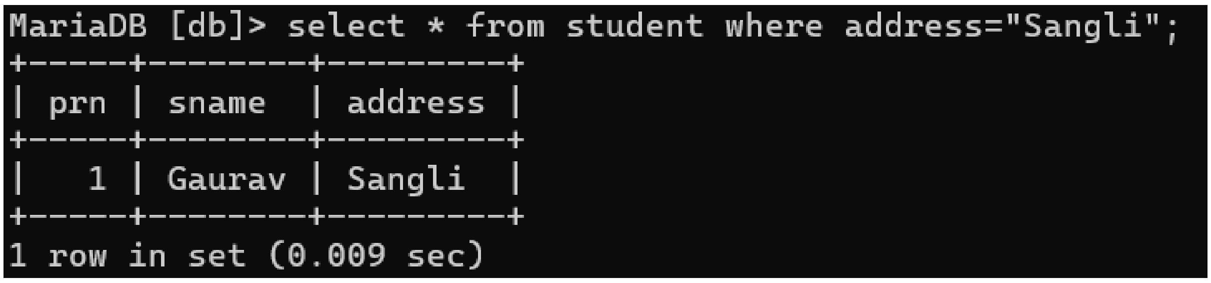
```
MariaDB [db]> select * from student;
+-----+---------+---------+
| prn | sname   | address |
+-----+---------+---------+
|   1 | Gaurav  | Sangli  |
|   2 | Chinmay | Karad   |
|   3 | Rakesh  | Solapur |
|   4 | Sandesh | Shirala |
+-----+---------+---------+
4 rows in set (0.001 sec)
```

- Select specific columns-



```
MariaDB [db]> select prn,sname from student;
+-----+---------+
| prn | sname   |
+-----+---------+
|   1 | Gaurav  |
|   2 | Chinmay |
|   3 | Rakesh  |
|   4 | Sandesh |
+-----+---------+
4 rows in set (0.001 sec)
```

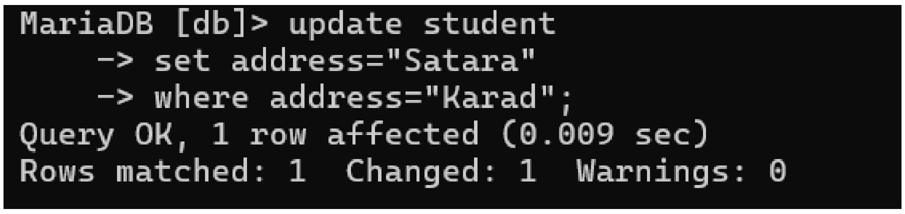
- Filter-



```
MariaDB [db]> select * from student where address="Sangli";
+-----+--------+---------+
| prn | sname  | address |
+-----+--------+---------+
|   1 | Gaurav | Sangli  |
+-----+--------+---------+
1 row in set (0.009 sec)
```

Update Operation (Modify Data)

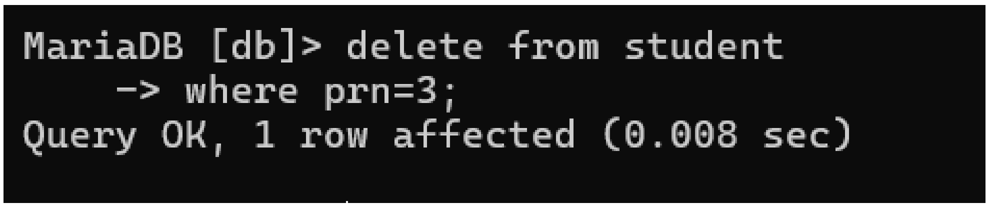
- Update Records-



```
MariaDB [db]> update student
    -> set address="Satara"
    -> where address="Karad";
Query OK, 1 row affected (0.009 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

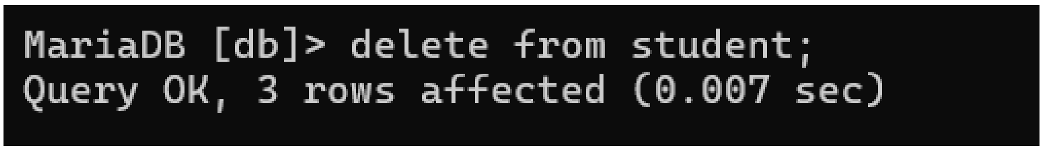
Delete Operation (Remove Data)

- Delete a specific record-



```
MariaDB [db]> delete from student
    -> where prn=3;
Query OK, 1 row affected (0.008 sec)
```

- Delete all records-



```
MariaDB [db]> delete from student;
Query OK, 3 rows affected (0.007 sec)
```

Conclusion:

In this experiment, we successfully performed CRUD operations using MariaDB. We learned how to create new records, read and retrieve data, update existing records, and delete unwanted data. These operations are essential for managing any relational database and form the core functionality of database systems

Experiment No. 14

Title:

Implement JDBC and ODBC connectivity

Objective:

To demonstrate how to use Java's JDBC (Java Database Connectivity) API to connect to a MySQL database, create a new database and table, insert sample records, and retrieve data using SQL queries through a Java program. This exercise showcases the basics of connecting a Java application to a relational database and performing standard CRUD operations.

Description:

JDBC is an API for the Java programming language that defines how a client may access a database. It provides methods for querying and updating data in a database. In this task, you will create a Java program to connect to a MySQL server, create a database named `school_db`, define a `students` table, insert some sample student records, and then retrieve and display the data from the table.

Data Set:

The database will contain a `students` table with fields like `student_id`, `student_name`, `course`, and `grade`.

Procedure (Code):

1. Set Up the MySQL JDBC Driver:

- Download the MySQL Connector/J JAR file (`mysql-connector-j-9.1.0.jar` or any compatible version) from the official MySQL website.
- Add the JAR file to the classpath of your Java project.

2. Create a Java Program for Database Operations:

- Use the following code to connect to the MySQL database, create the database and table, insert data, and retrieve records.

```
import java.sql.*;

public class DatabaseOperations {
    // Database connection parameters
    static final String JDBC_DRIVER = "com.mysql.cj.jdbc.Driver";
    static final String DB_URL = "jdbc:mysql://localhost:3306/mydb"; // Change mydb to your
database name
    static final String USER = "root";
    static final String PASS = "root";

    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;

        try {
            // Register JDBC driver
            Class.forName(JDBC_DRIVER);

            // Open a connection
            System.out.println("Connecting to database...");
            conn = DriverManager.getConnection(DB_URL, USER, PASS);

            // Create a statement
            stmt = conn.createStatement();

            // Create Students table
            String createTableSQL = "CREATE TABLE IF NOT EXISTS Students ("
                    + "Roll_No INT PRIMARY KEY,"
                    + "Name VARCHAR(255),"
                    + "City VARCHAR(255),"
                    + "Grade CHAR(1),"
                    + "Marks DECIMAL(5,2)"
                    + ")";
            stmt.executeUpdate(createTableSQL);

            // Insert data into the table
            String[] insertData = {
                    "INSERT INTO Students VALUES (1, 'Atul', 'Sangli', 'A', 90.50)",
                    "INSERT INTO Students VALUES (2, 'Sangram', 'Sangli', 'B', 70.25)",
                    "INSERT INTO Students VALUES (3, 'Satya', 'Mumbai', 'B', 61.36)",
                    "INSERT INTO Students VALUES (4, 'Jaydeep', 'Pune', 'B', 60.95)",
                    "INSERT INTO Students VALUES (5, 'Prashant', 'Sangli', 'C', 55.26)",
                    "INSERT INTO Students VALUES (6, 'Abhi', 'Pune', 'C', 55.84)"
            };
            for (String sql : insertData) {
              stmt.executeUpdate(sql);
            }
            System.out.println("Data inserted successfully.");

            // Delete record for Roll_No 5
            String deleteRecordSQL = "DELETE FROM Students WHERE Roll_No = 5";
            stmt.executeUpdate(deleteRecordSQL);
            System.out.println("Record with Roll_No 5 deleted successfully.");

            // Update city from Sangli to Pune
            String updateCitySQL = "UPDATE Students SET City = 'Pune' WHERE City =
'Sangli'";
            stmt.executeUpdate(updateCitySQL);
            System.out.println("City updated successfully.");

            // Display names of students having marks greater than 60
```

```
            String displayNamesSQL = "SELECT Name FROM Students WHERE Marks > 60";
            ResultSet rs = stmt.executeQuery(displayNamesSQL);
            System.out.println("Names of students with marks greater than 60:");
            while (rs.next()) {
                System.out.println(rs.getString("Name"));
            }

            // Display students according to their marks (Descending order)
            String displayByMarksSQL = "SELECT * FROM Students ORDER BY Marks
DESC";
            rs = stmt.executeQuery(displayByMarksSQL);
            System.out.println("\nStudents sorted by marks (Descending order):");
            while (rs.next()) {
                System.out.println(rs.getInt("Roll_No") + "\t" +
                                   rs.getString("Name") + "\t" +
                                   rs.getString("City") + "\t" +
                                   rs.getString("Grade") + "\t" +
                                   rs.getDouble("Marks"));
            }

            // Clean-up environment
            rs.close();
            stmt.close();
            conn.close();
        } catch (SQLException se) {
            // Handle errors for JDBC
            se.printStackTrace();
        } catch (Exception e) {
            // Handle errors for Class.forName
            e.printStackTrace();
        } finally {
            // Finally block used to close resources
            try {
                if (stmt != null) stmt.close();
            } catch (SQLException se2) {
            } // nothing we can do
            try {
                if (conn != null) conn.close();
            } catch (SQLException se) {
                se.printStackTrace();
            } // end finally try
        } // end try
    }
}
```

3. Compile and Run the Java Program:

- Open Command Prompt and navigate to the directory where the Java program and MySQL connector JAR file are located.

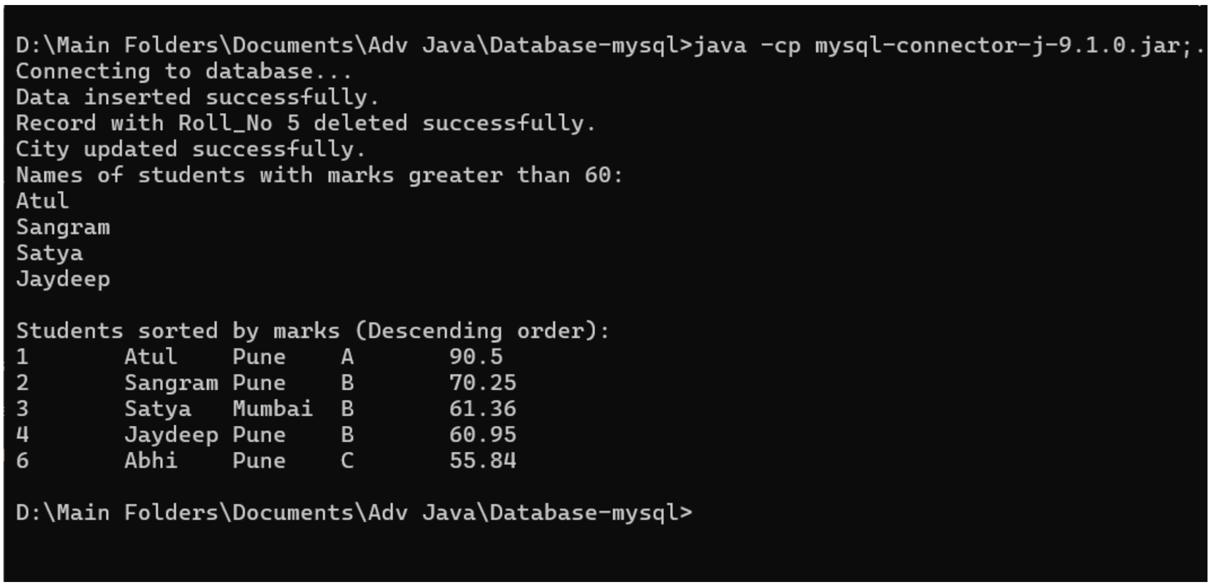
- Compile the program:

```
javac -cp mysql-connector-j-9.1.0.jar;. DatabaseOperations.java
```

- Run the program:

```
java -cp mysql-connector-j-9.1.0.jar;. DatabaseOperations
```

4. Expected Output:



```
D:\Main Folders\Documents\Adv Java\Database-mysql>java -cp mysql-connector-j-9.1.0.jar;.
Connecting to database...
Data inserted successfully.
Record with Roll_No 5 deleted successfully.
City updated successfully.
Names of students with marks greater than 60:
Atul
Sangram
Satya
Jaydeep

Students sorted by marks (Descending order):
1       Atul    Pune    A       90.5
2       Sangram Pune    B       70.25
3       Satya   Mumbai  B       61.36
4       Jaydeep Pune    B       60.95
6       Abhi    Pune    C       55.84

D:\Main Folders\Documents\Adv Java\Database-mysql>
```

5. Exit the Program:

The program will automatically close the connection once all the operations are completed.

Conclusion:

Using JDBC in Java, you successfully connected to a MySQL database, created a new database named `school_db`, defined a `students` table, inserted sample records, and retrieved the data. This exercise demonstrates basic database management using Java's JDBC API and provides a foundation for more advanced database programming.