A
Lab Journal
On

Database Engineering Lab Experiments

Submitted in partial fulfillment for

Database Engineering Experiments Lab (6IT351)

Submitted by

T1 Batch

Under the Guidance of

Mrs. M. S. Mane

Department of Information Technology,
Walchand College of Engineering, Sangli.
Maharashtra, India.
416415.

CERTIFICATE

This is to certify that the report entitled “Database Engineering Lab, Code : 6IT351” submitted by T1 Batch of T.Y I.T. a record of a batch’s own work carried out by them during the academic year 2024-2025, as per the curriculum/syllabus laid down for Lab at Third year B.Tech IT Sem-I. They have carried out respective experiments successfully.

Mrs. M. S. Mane
(Course Teacher)

Dr. R. R. Rathod
(I.T. Head of Dept.)

INDEX

Hemant Sharma(22610001)
TY IT (T1 batch)
Experiment No-1

Title: Construction of E-R Model and Schema for Employee and Company Database.

Aim: To design an E-R model for college and company databases, define schemas, create ER diagrams, and convert entities and relationships into relational tables.

Objective:

- To understand the process of translating an E-R diagram into relational tables.
- To apply relational database concepts to organize data effectively in tabular formats.
- To design an E-R model and develop a relational schema diagram that maintains relationships and ensures data integrity.

Theory:

Entity-Relationship (E-R) Model: The E-R model is a high-level data model used to represent the structure of a database. It provides a visual way to understand the relationships among various data entities. Key components include:

1. **Entities**: Objects that represent real-world concepts (e.g., Student, Employee).
2. **Attributes**: Properties that describe entities (e.g., USN, Name).
3. **Relationships**: Connections between entities (e.g., a student enrolling in a class).

Relational Model: Once the E-R diagram is constructed, it can be translated into a relational schema. The relational model organizes data into tables (relations) where each table consists of rows (records) and columns (attributes). It includes:

1. **Relation**: A set of tuples sharing the same attributes.
2. **Primary Key**: An attribute or a set of attributes that uniquely identifies a record in a table.
3. **Foreign Key**: An attribute in one table that links to the primary key of another table, establishing a relationship between the two.

Cardinality:

1. **One-to-One (1:1)**: Each entity in the relationship will have exactly one related entity (e.g., one department may have one manager).
2. **One-to-Many(1)**: An entity can be associated with multiple instances of another entity (e.g., one department can have many employees).
3. **Many-to-Many(M)**: Entities on both sides of the relationship can have multiple instances (e.g., many employees can work on many projects).

Relational Schema: After the E-R diagram is created, it is mapped to a relational schema. This involves converting entities and relationships into tables, defining primary keys and foreign keys to maintain referential integrity.

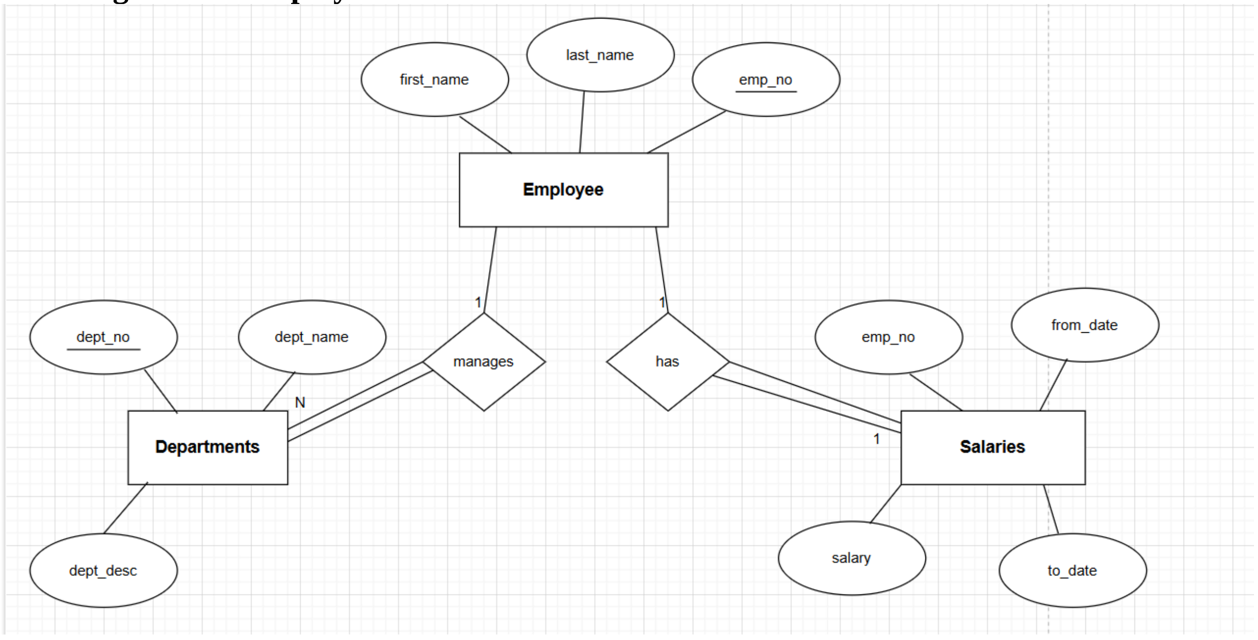
Normalization: Normalization is the process of organizing a database to reduce redundancy and improve data integrity. The E-R model ensures the data is organized in such a way that the schema is in at least 3rd normal form (3NF).

The **draw.io** provides a clear visualization of database structure and relationships:

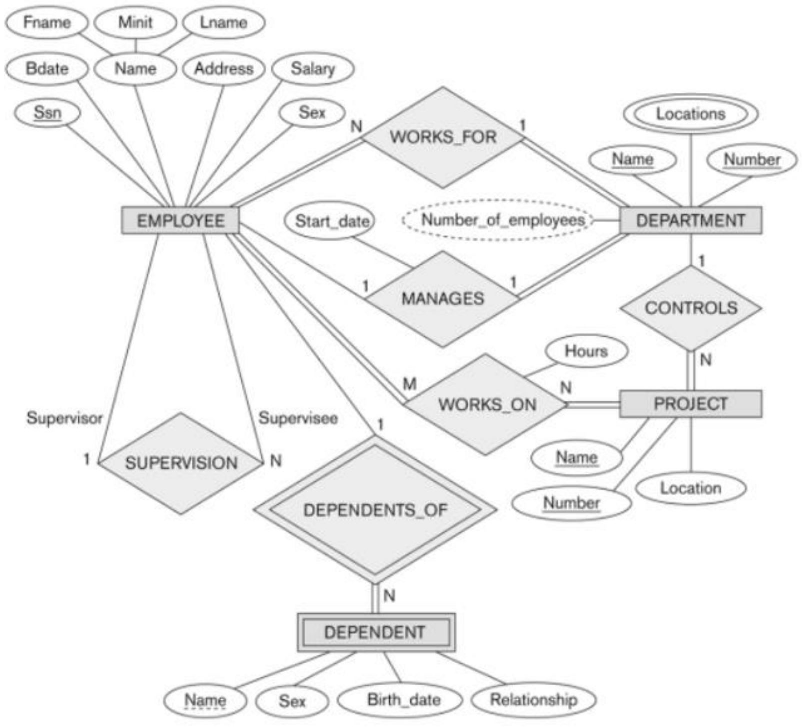
1. **E-R Diagram**: Use rectangles for entities, ellipses for attributes, and diamonds for relationships. Connect entities with lines and set line endings to indicate cardinality (1, N).
2. **Relational Table:** Use tables or rectangles to list attributes, marking primary (PK) and foreign keys (FK). Connect related tables with lines to show foreign key relationships.

Implementation:

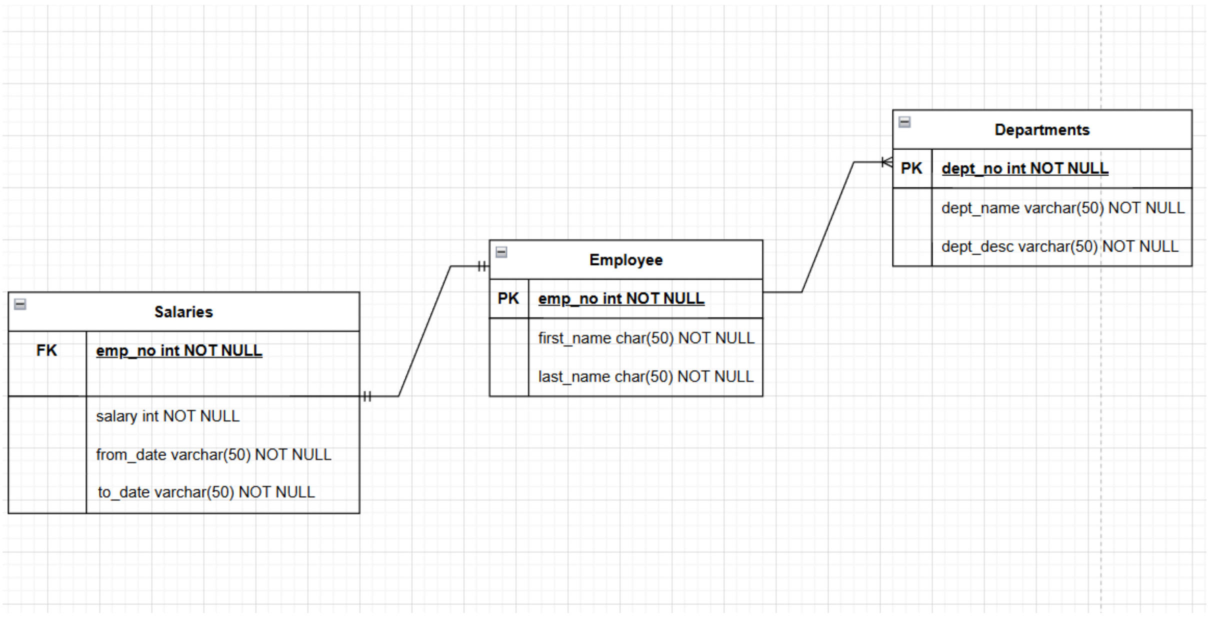
E-R Diagram for Employee Database:



ER Diagram for Company Database:



Relation Table:


Departments
PK
dept_no int NOT NULL
dept_name varchar(50) NOT NULL
dept_desc varchar(50) NOT NULL
Employee
PK
emp_no int NOT NULL
first_name char(50) NOT NULL
last_name char(50) NOT NULL
Salaries
FK
emp_no int NOT NULL
salary int NOT NULL
from_date varchar(50) NOT NULL
to_date varchar(50) NOT NULL

Conclusion: The ER models and schemas for the Employee and Company Databases provide structured frameworks for organizing data on student enrollments, courses, assessments, and employee information, departments, projects, and hours. These models ensure data integrity and support efficient database operations.

Saloni Dhurve (22610004)
TY IT (T1 Batch)
Experiment No. 2

Title: Implementation of Relational Algebra Operations in SQL

Aim: To understand and implement various relational algebra operations like Select (σ), Project (π), Intersect (∩), Minus (−), and Cartesian Product (×) using SQL on a college database.

Objectives:

- To apply the theoretical concepts of relational algebra to a real-world database.
- To translate relational algebra operations into equivalent SQL queries.
- To demonstrate how SQL supports or simulates relational algebra operations.

Theory: Relational algebra is a formal language used to describe queries on relational databases. It includes a set of operations that take one or two relations (tables) as input and return a new relation as output.

1. **Select (σ)**: Retrieves rows that satisfy a given predicate.
   Example: **σ SEX = 'M' (STUDENT)**
   SQL Equivalent:
   ```
   SELECT * FROM STUDENT WHERE SEX = 'M
   ```

2. **Project (π)**: Retrieves specific columns from a table.
   Example: **π FNAME, AGE (STUDENT)**
   SQL Equivalent:
   ```
   SELECT FNAME, AGE FROM STUDENT;
   ```

3. **Intersect** (∩): Retrieves rows common to two tables or result sets.
   SQL Equivalent:
   ```
   SELECT * FROM STUDENT
   WHERE DEPARTMENT = 'CSE'
   AND ID IN (SELECT ID FROM STUDENT
   WHERE DEPARTMENT = 'ECE');
   ```

4. **Minus (−)**: Retrieves rows from one table that are not present in another.
   SQL Equivalent:
   ```
   SELECT * FROM STUDENT WHERE DEPARTMENT = 'CSE'
   AND ID NOT IN (SELECT ID FROM STUDENT
   WHERE DEPARTMENT = 'ECE');
   ```

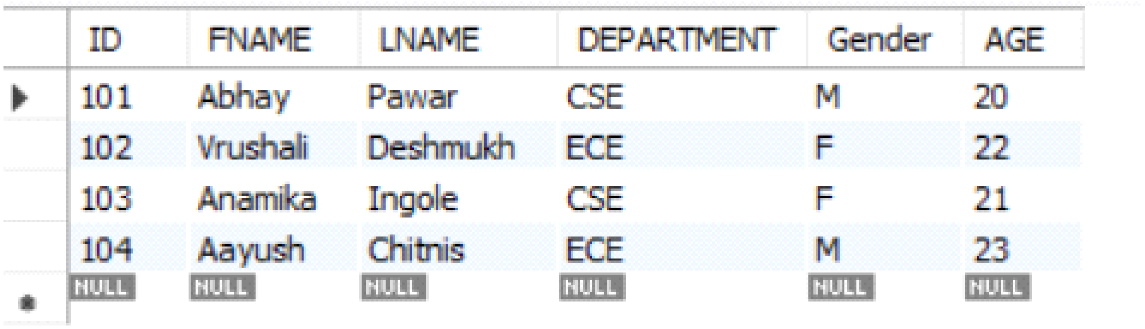
5. **Cartesian Product (×)**: Combines all rows from two tables, resulting in every possible combination of rows.
   Example: STUDENT × DEPARTMENT
   SQL Equivalent:
   ```
   SELECT * FROM STUDENT CROSS JOIN DEPARTMENT;
   ```

Implementation:

Query:

```
CREATE DATABASE College;
USE College;

CREATE TABLE STUDENT (
   ID INT PRIMARY KEY,
   FNAME VARCHAR(50),
   LNAME VARCHAR(50),
   DEPARTMENT VARCHAR(50),
   Gender CHAR(1),
   AGE INT
);

INSERT INTO STUDENT (ID, FNAME, LNAME, DEPARTMENT, Gender, AGE)
VALUES
(101, 'Abhay', 'Pawar', 'CSE', 'M', 20),
(102, 'Vrushali', 'Deshmukh', 'ECE', 'F', 22),
(103, 'Anamika', 'Ingole', 'CSE', 'F', 21),
(104, 'Aayush', 'Chitnis', 'ECE', 'M', 23);

SELECT * FROM STUDENT;
```

Table 1:

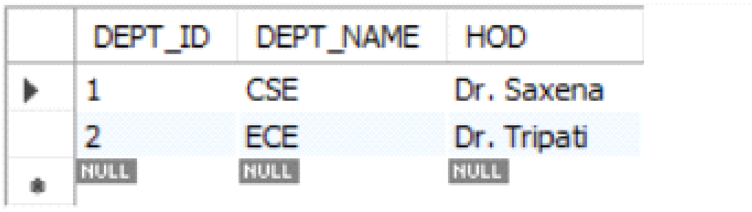


ID	FNAME	LNAME	DEPARTMENT	Gender	AGE
101	Abhay	Pawar	CSE	M	20
102	Vrushali	Deshmukh	ECE	F	22
103	Anamika	Ingole	CSE	F	21
104	Aayush	Chitnis	ECE	M	23
NULL	NULL	NULL	NULL	NULL	NULL

Query:

```
CREATE TABLE DEPARTMENT (
   DEPT_ID INT PRIMARY KEY,
   DEPT_NAME VARCHAR(50),
   HOD VARCHAR(50)
);

INSERT INTO DEPARTMENT (DEPT_ID, DEPT_NAME, HOD)
VALUES
(1, 'CSE', 'Dr. Saxena'),
(2, 'ECE', 'Dr. Tripati');

SELECT * FROM DEPARTMENT;
```

Table 2:

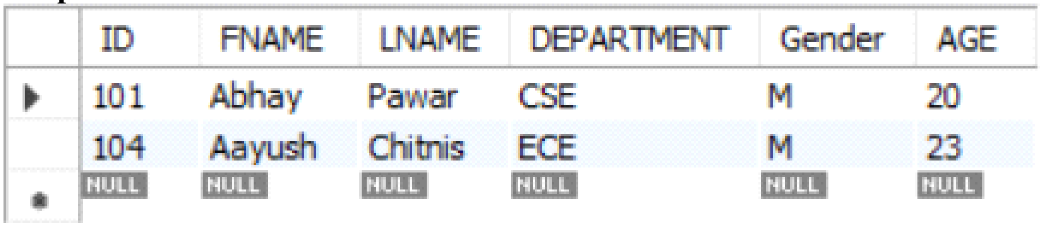


	DEPT_ID	DEPT_NAME	HOD
▶	1	CSE	Dr. Saxena
	2	ECE	Dr. Tripati
*	NULL	NULL	NULL

1. Select Operation (σ SEX = 'M'):

Query:
SELECT * FROM STUDENT WHERE Gender = 'M';

Output:

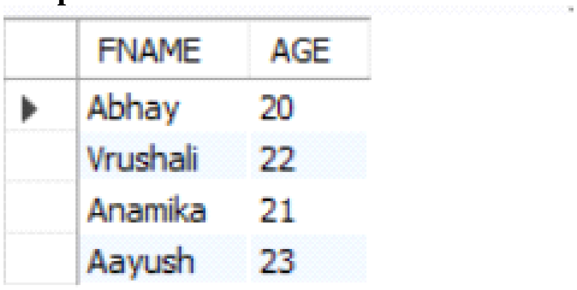


	ID	FNAME	LNAME	DEPARTMENT	Gender	AGE
▶	101	Abhay	Pawar	CSE	M	20
	104	Aayush	Chitnis	ECE	M	23
*	NULL	NULL	NULL	NULL	NULL	NULL

2. Project Operation (π FNAME, AGE):

Query:
SELECT FNAME, AGE FROM STUDENT;

Output:

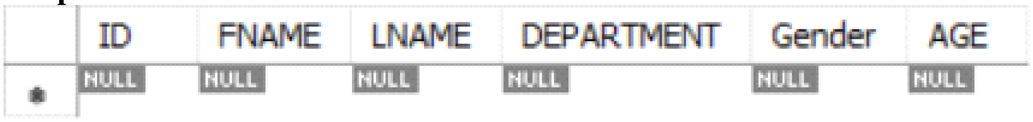


	FNAME	AGE
▶	Abhay	20
	Vrushali	22
	Anamika	21
	Aayush	23

3. Intersect Operation:

Query:
SELECT * FROM STUDENT WHERE DEPARTMENT = 'CSE'
AND ID IN (SELECT ID FROM STUDENT WHERE DEPARTMENT = 'ECE');

Output:

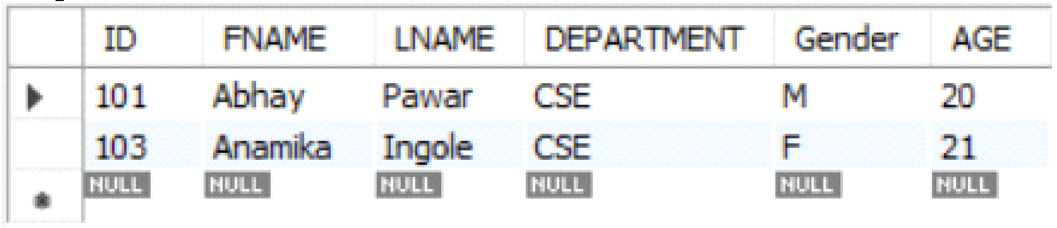


	ID	FNAME	LNAME	DEPARTMENT	Gender	AGE
*	NULL	NULL	NULL	NULL	NULL	NULL

4. Minus Operation:

Query:
SELECT * FROM STUDENT WHERE DEPARTMENT = 'CSE'
AND ID NOT IN (SELECT ID FROM STUDENT WHERE DEPARTMENT = 'ECE');

Output:

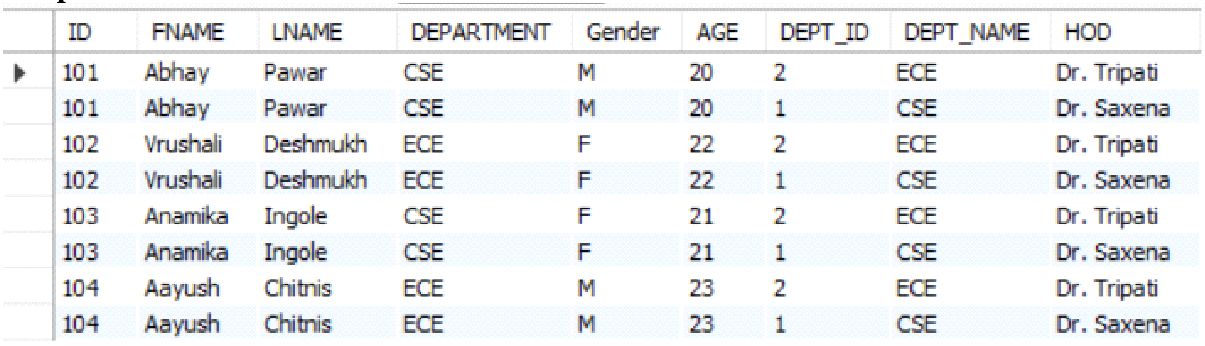


	ID	FNAME	LNAME	DEPARTMENT	Gender	AGE
▶	101	Abhay	Pawar	CSE	M	20
	103	Anamika	Ingole	CSE	F	21
*	NULL	NULL	NULL	NULL	NULL	NULL

5. Cartesian Product:

Query:
SELECT * FROM STUDENT, DEPARTMENT;

Output:



	ID	FNAME	LNAME	DEPARTMENT	Gender	AGE	DEPT_ID	DEPT_NAME	HOD
▶	101	Abhay	Pawar	CSE	M	20	2	ECE	Dr. Tripati
	101	Abhay	Pawar	CSE	M	20	1	CSE	Dr. Saxena
	102	Vrushali	Deshmukh	ECE	F	22	2	ECE	Dr. Tripati
	102	Vrushali	Deshmukh	ECE	F	22	1	CSE	Dr. Saxena
	103	Anamika	Ingole	CSE	F	21	2	ECE	Dr. Tripati
	103	Anamika	Ingole	CSE	F	21	1	CSE	Dr. Saxena
	104	Aayush	Chitnis	ECE	M	23	2	ECE	Dr. Tripati
	104	Aayush	Chitnis	ECE	M	23	1	CSE	Dr. Saxena

Conclusion: This assignment demonstrated relational algebra operations using SQL. Despite MySQL's lack of direct support for operations like INTERSECT and MINUS, we simulated them with JOIN, IN, and NOT IN, deepening our understanding of relational algebra as the foundation for SQL operations.

Srushti Karadi (22610005)
TY IT (T1 Batch)
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 the fundamental commands of SQL, specifically focusing on Data Definition Language (DDL) and Data Manipulation Language (DML) commands. This will include learning how to define, modify, and manage the structure of databases (DDL) and how to manipulate data within databases (DML), using real-life examples.

Objectives:

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

Theory:

Data Definition Language (DDL): DDL is a subset of SQL used to define, alter, and manage the structure of database objects. It primarily deals with the schema of the database.

Common DDL Commands:

1) **CREATE:** Used to create new database objects, such as tables, indexes, and views.
   Syntax:
   CREATE DATABASE databasename;
   CREATE TABLE tablename (column1 datatype, column2 datatype, ...);

2) **ALTER:** Used to modify an existing database object, such as adding or deleting columns in a table.
   Syntax:
   ALTER TABLE tablename MODIFY COLUMN column_name new_datatype;

3) **DROP:** Used to delete existing database objects, such as tables or indexes.
   Syntax:
   DROP DATABASE databasename;

4) **TRUNCATE:** Used to delete all rows from a table without removing the table structure.
   Syntax:
   TRUNCATE TABLE tablename;

Data Manipulation Language (DML): DML is a subset of SQL used to manipulate data stored in database objects. It focuses on the data itself, allowing users to perform operations such as inserting, updating, or deleting data.

Common DML Commands:

1) **INSERT:** Used to add new records to a table.
   Syntax:
   ```
   INSERT INTO tablename (column1, column2, ...)
   VALUES (value1, value2, ...);
   ```

2) **UPDATE:** Used to modify existing records in a table.
   Syntax:
   ```
   UPDATE tablename
   SET column1 = value1, column2 = value2, ...
   WHERE condition;
   ```

3) **DELETE:** Used to remove records from a table.
   Syntax:
   ```
   DELETE FROM tablename WHERE condition;
   ```

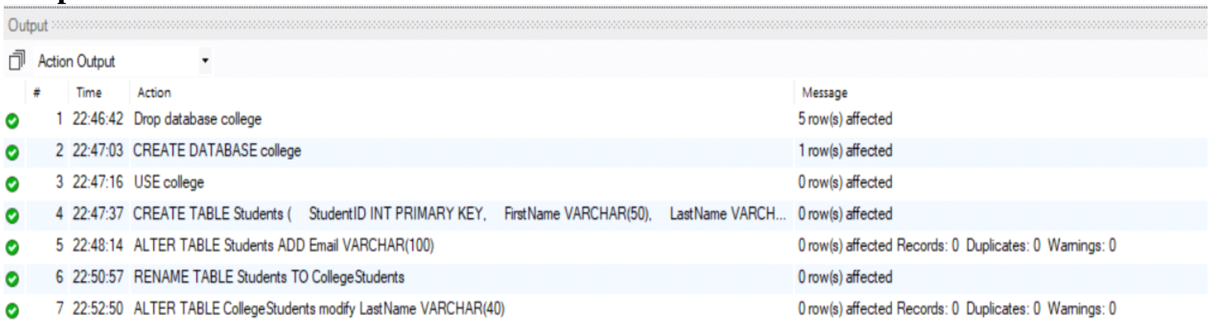
4) **SELECT:** Used to retrieve data from one or more tables. It can include filtering, sorting and aggregating of data.
   Syntax:
   ```
   SELECT column1, column2, ... FROM tablename;
   ```

Implementation:

1. Create, Alter:

Query:
```
// create database
CREATE DATABASE college;
USE college;

//create table
CREATE TABLE Students (
StudentID INT PRIMARY KEY,
FirstName VARCHAR(50),
LastName VARCHAR(50),
Age INT
);

//Alter table
ALTER TABLE Students
ADD Email VARCHAR(100);
RENAME TABLE Students to CollegeStudents;
ALTER TABLE CollegeStudents modify LastName VARCHAR (40);
```

Output:

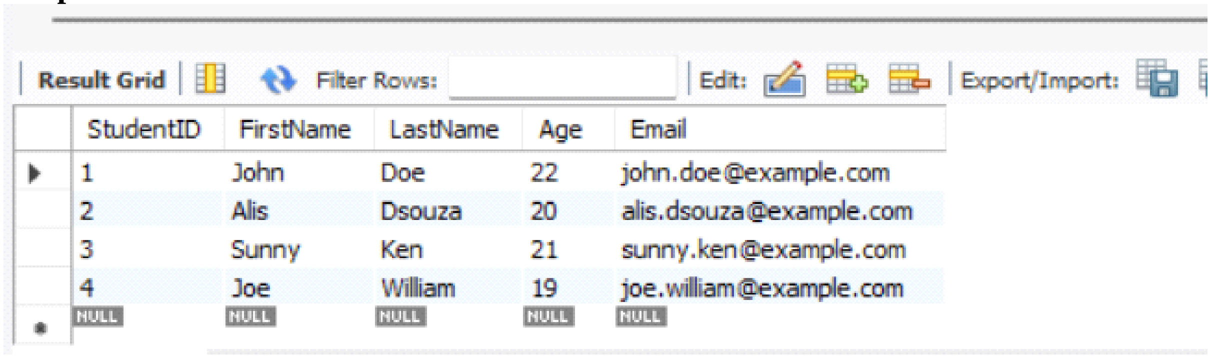


```
Output
Action Output
 #  Time      Action                                                                          Message
 1  22:46:42  Drop database college                                                           5 row(s) affected
 2  22:47:03  CREATE DATABASE college                                                         1 row(s) affected
 3  22:47:16  USE college                                                                     0 row(s) affected
 4  22:47:37  CREATE TABLE Students (   StudentID INT PRIMARY KEY,   FirstName VARCHAR(50),   LastName VARCH...   0 row(s) affected
 5  22:48:14  ALTER TABLE Students ADD Email VARCHAR(100)                                     0 row(s) affected Records: 0  Duplicates: 0  Warnings: 0
 6  22:50:57  RENAME TABLE Students TO CollegeStudents                                        0 row(s) affected
 7  22:52:50  ALTER TABLE CollegeStudents modify LastName VARCHAR(40)                         0 row(s) affected Records: 0  Duplicates: 0  Warnings: 0
```

2. Insert:

Query:
//Insert data into table
INSERT INTO Students (StudentID, FirstName, LastName, Age, Email)
VALUES (1, 'John', 'Doe', 22, 'john.doe@example.com'),
(2, 'Alis', 'Dsouza', 20, 'alis.dsouza@example.com'),
(3, 'Sunny', 'Ken', 21, 'sunny.ken@example.com'),
(4, 'Joe', 'William', 29, 'joe.william@example.com');
SELECT *FROM Students;

Output:

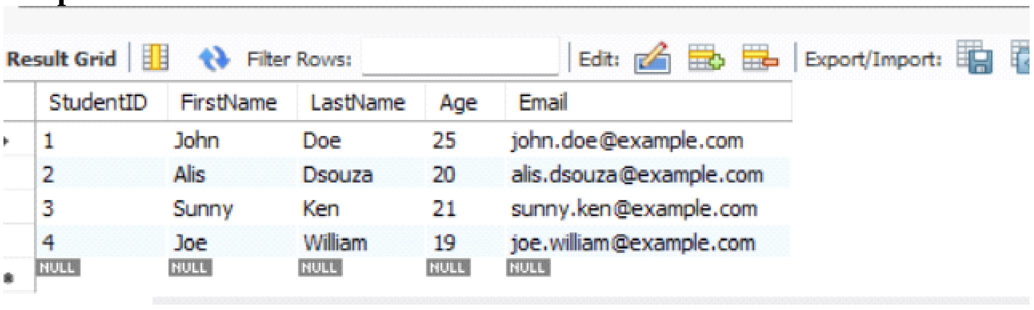


StudentID	FirstName	LastName	Age	Email
1	John	Doe	22	john.doe@example.com
2	Alis	Dsouza	20	alis.dsouza@example.com
3	Sunny	Ken	21	sunny.ken@example.com
4	Joe	William	19	joe.william@example.com
NULL	NULL	NULL	NULL	NULL

3. Update:

Query:
//Update table
UPDATE Students
SET Age = 25
WHERE StudentID = 1;

Output:

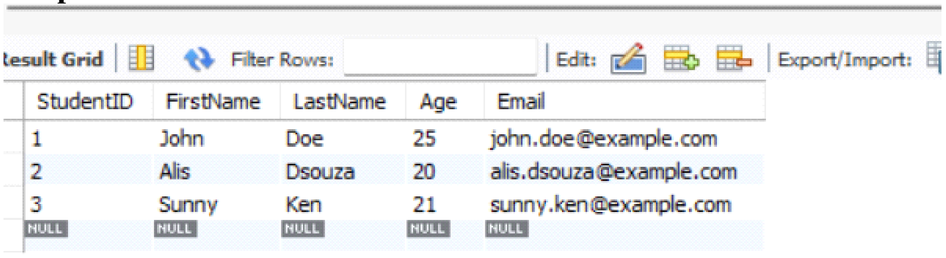


StudentID	FirstName	LastName	Age	Email
1	John	Doe	25	john.doe@example.com
2	Alis	Dsouza	20	alis.dsouza@example.com
3	Sunny	Ken	21	sunny.ken@example.com
4	Joe	William	19	joe.william@example.com
NULL	NULL	NULL	NULL	NULL

4. Delete:

Query:
//Delete data
DELETE FROM Students
WHERE LastName = ‘William’;

Output:

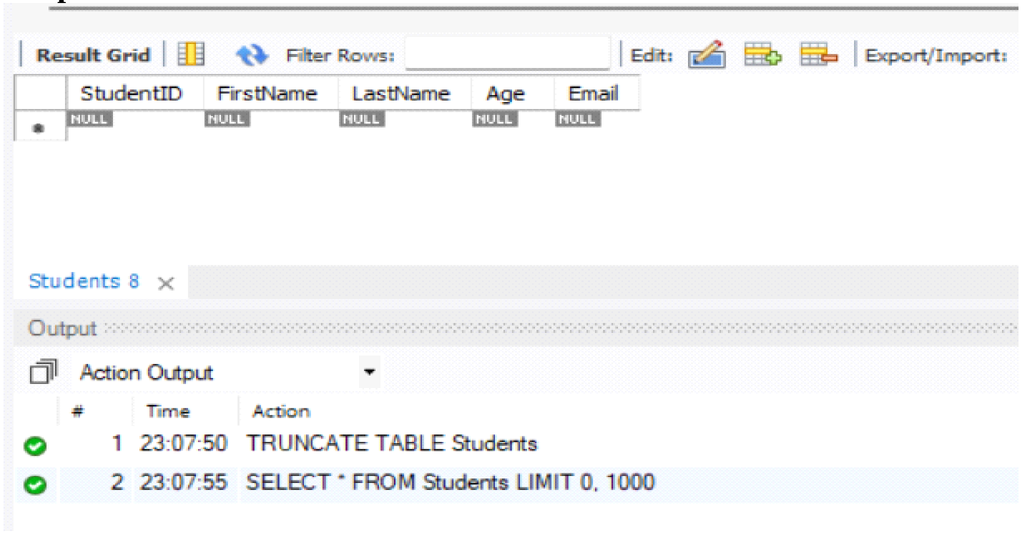


StudentID	FirstName	LastName	Age	Email
1	John	Doe	25	john.doe@example.com
2	Alis	Dsouza	20	alis.dsouza@example.com
3	Sunny	Ken	21	sunny.ken@example.com
NULL	NULL	NULL	NULL	NULL

5. Truncate:

Query:
//Truncate Table
TRUNCATE TABLE Students;

Output:



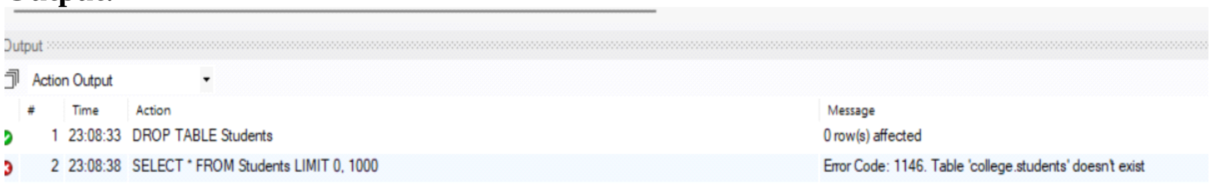
StudentID	FirstName	LastName	Age	Email
NULL	NULL	NULL	NULL	NULL

```
Students 8
Output
Action Output
#  Time      Action
1  23:07:50  TRUNCATE TABLE Students
2  23:07:55  SELECT * FROM Students LIMIT 0, 1000
```

6. Drop:

Query:
//Drop table
DROP TABLE Students;

Output:



```
Output
Action Output
#  Time      Action                          Message
1  23:08:33  DROP TABLE Students             0 row(s) affected
2  23:08:38  SELECT * FROM Students LIMIT 0, 1000   Error Code: 1146. Table 'college.students' doesn't exist
```

Conclusion: Understanding DDL and DML commands is crucial for effective database management. DDL provides the foundation for the database structure, while DML enables users to interact with and manipulate the data stored within that structure. Together, they form the backbone of SQL database operations.

Rachit Bhat (22610007)
TY IT (T1 Batch)
Experiment No. 4

Title: Create a database and apply DQL and TCL SQL commands.

Aim: To create a database and demonstrate the usage of Data Query Language (DQL) and Transaction Control Language (TCL) SQL commands for effective data manipulation and transaction management.

Objectives:

- **Design a Database:** Create a database with appropriate tables and relationships.
- **Insert Data:** Populate the data with sample records using DML commands.
- **Query Data:** Use DQL commands to retrieve and display data based on specific conditions.
- **Manage Transactions:** Implement TCL commands to control transactions and ensure data integrity.

Theory:

Database: A database is an organized collection of structured information, typically managed by a Database Management System (DBMS). Understanding relational databases, tables, primary and foreign keys, and normalization is crucial for effective design.

- **DQL (Data Query Language):** DQL is primarily used for querying data from the database. The main command in DQL is SELECT, which retrieves data based on specified conditions. Other commands include JOIN, WHERE, ORDER BY, and aggregate functions like SUM, COUNT, and AVG.

- **TCL (Transaction Control Language):** TCL is used to manage transactions in a database. Key commands include:
  1. **COMMIT:** Saves all changes made during the current transaction.
  2. **ROLLBACK:** Undoes all changes made during the current transaction if an error occurs.
  3. **SAVEPOINT:** Sets a point within a transaction to which you can later roll back.

Implementation:

Query:

```
CREATE DATABASE company;
USE company;

CREATE TABLE employees (
   id INT PRIMARY KEY AUTO_INCREMENT,
   name VARCHAR(100),
   position VARCHAR(100),
   salary DECIMAL(10, 2)
);
```

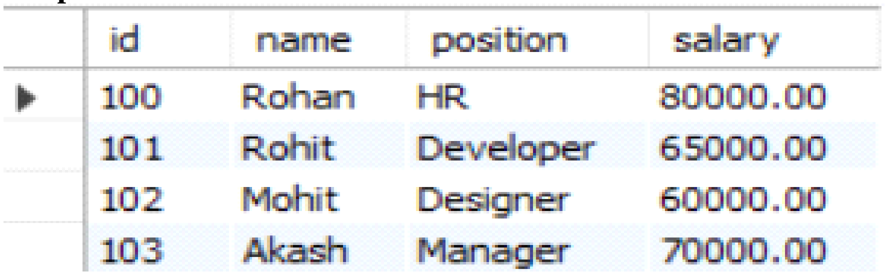
```
INSERT INTO employees VALUES
(100,"Rohan","HR",80000.00),
(101,"Rohit","Developer",65000.00),
(102,"Mohit","Designer",60000.00),
(103,"Akash","Manager",70000.00);
```

1. DQL Commands:

Query:

```
SELECT * FROM employees;
```

Output:

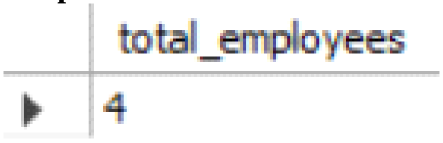


```
id    name    position    salary
100   Rohan   HR          80000.00
101   Rohit   Developer   65000.00
102   Mohit   Designer    60000.00
103   Akash   Manager     70000.00
```

Query:

```
SELECT COUNT (*) AS total_employees FROM employees;
```

Output:

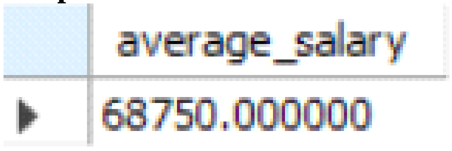


```
total_employees
4
```

Query:

```
SELECT AVG (salary) AS average_salary FROM employees;
```

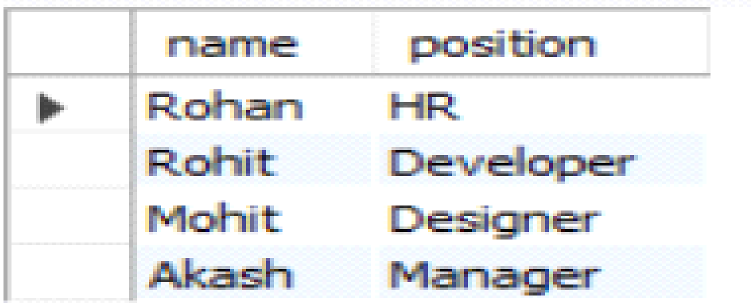
Output:



```
average_salary
68750.000000
```

Query:
SELECT name, position FROM employees;

Output:

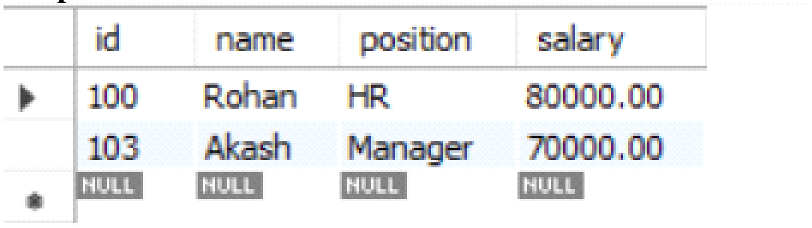


name	position
Rohan	HR
Rohit	Developer
Mohit	Designer
Akash	Manager

Use of all SQL Operators on database.

Query:
SELECT * FROM employees WHERE salary > 65000;

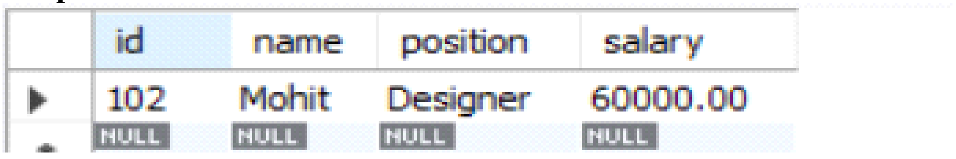
Output:



id	name	position	salary
100	Rohan	HR	80000.00
103	Akash	Manager	70000.00
NULL	NULL	NULL	NULL

Query:
SELECT * FROM employees WHERE position = "Designer";

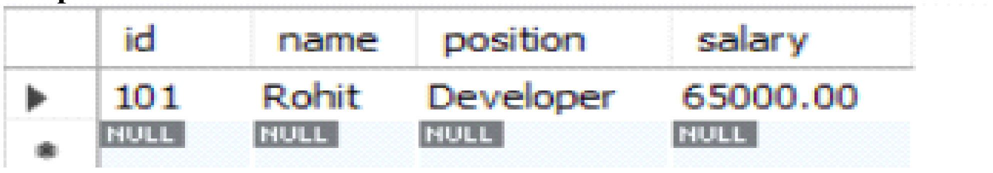
Output:



id	name	position	salary
102	Mohit	Designer	60000.00
NULL	NULL	NULL	NULL

Query:
SELECT * FROM employees WHERE salary < 80000 AND position = 'Developer';

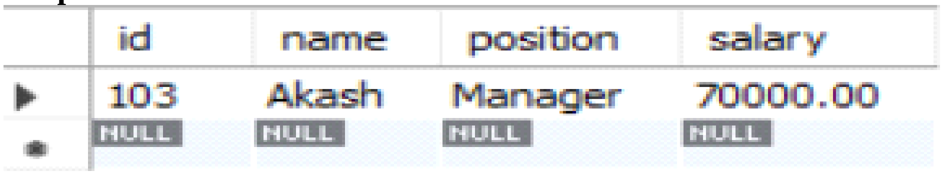
Output:



id	name	position	salary
101	Rohit	Developer	65000.00
NULL	NULL	NULL	NULL

Query:
SELECT * FROM employees WHERE salary < 60000 OR position = 'Manager';

Output:

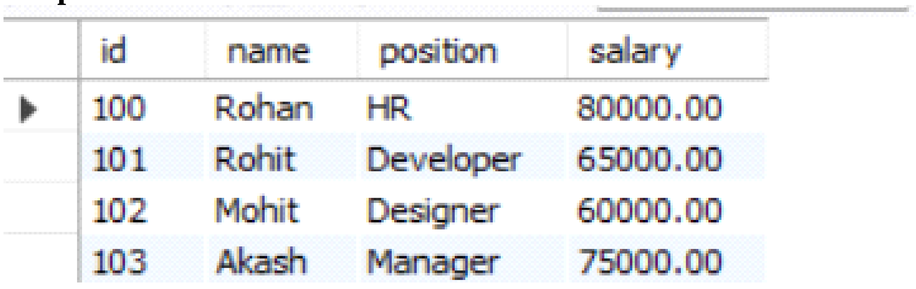


id	name	position	salary
103	Akash	Manager	70000.00
NULL	NULL	NULL	NULL

2. Save point, Rollback and Commit:

Query:
START TRANSACTION;
SAVEPOINT savepoint1;
UPDATE employees SET salary = 75000 WHERE id = 103;
ROLLBACK TO savepoint1;
COMMIT;

Output:



id	name	position	salary
100	Rohan	HR	80000.00
101	Rohit	Developer	65000.00
102	Mohit	Designer	60000.00
103	Akash	Manager	75000.00

Conclusion: Creating a database and applying DQL and TCL commands are key skills in database management, ensuring data integrity, consistency, and effective retrieval. These concepts are essential for working with relational databases in various applications. Future work could focus on query optimization and exploring advanced SQL features.

Ishwari Gharge (22610008)
TY IT (T1 Batch)
Experiment No. 5

Title: Perform Various Types of Functions in SQL.

Aim: To understand and demonstrate the usage of different types of functions in SQL, including String Functions, Aggregate Functions, Mathematical Functions, and Date Functions.

Objective:

- To perform and observe string manipulation using SQL String Functions.
- To execute and analyze SQL Aggregate Functions for summarizing data.
- To work with SQL Mathematical Functions for performing mathematical calculations.
- To use SQL Date Functions to manipulate and extract date-related data.

Theory: SQL functions are built-in operations provided by the database to manipulate and perform operations on data. They are categorized into several types based on the purpose they serve:

String Functions: These functions manipulate string data.
Examples:
1. UPPER(): Converts a string to uppercase.
2. LOWER(): Converts a string to lowercase.
3. CONCAT(): Concatenates two or more strings.
4. LENGTH(): Returns the length of a string.

Aggregate Functions: These are used to perform calculations on multiple values and return a single result.
Examples:
1. SUM(): Returns the sum of a column.
2. COUNT(): Counts the number of rows in a column.
3. AVG(): Returns the average value of a column.
4. MAX(): Returns the maximum value in a set of values.
5. MIN(): Returns the minimum value in a set of values.

Mathematical (Math) Functions: These functions perform mathematical operations on numeric data.
Examples:
1. ABS(): Returns the absolute value of a number.
2. ROUND(): Rounds a number to the specified number of decimal places.
3. POWER(): Returns the value of a number raised to a power.
4. SQRT(): Returns the square root of a number.

Date Functions: These functions allow manipulation of date values.
Examples:

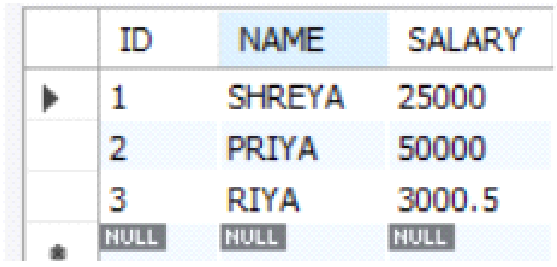
1. NOW(): Returns the current date and time.
2. DATEADD(): Adds a specified number of days to a date.
3. DATEDIFF(): Returns the difference between two dates.

Implementation:

Query:

```
CREATE DATABASE company;
USE company;

CREATE TABLE employee (
   id INT PRIMARY KEY,
   name VARCHAR(100),
   salary DOUBLE
);

INSERT INTO employee (id, name, salary) VALUES
(1, 'Shreya', 25000),
(2, 'Priya', 50000),
(3, 'Riya', 3000.5);
```

Table:



	ID	NAME	SALARY
▶	1	SHREYA	25000
	2	PRIYA	50000
	3	RIYA	3000.5
*	NULL	NULL	NULL

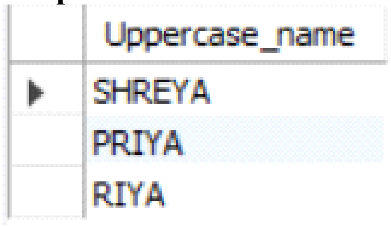
1. String Function:

UPPER():

Query:

```
-- Convert name to uppercase
SELECT UPPER(name) AS UpperCase_name FROM employee;
```

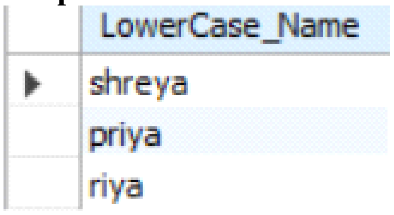
Output:



	Uppercase_name
▶	SHREYA
	PRIYA
	RIYA

LOWER():

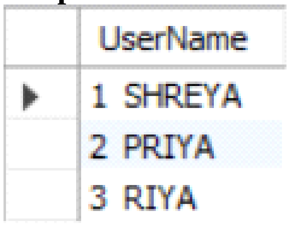
Query:
SELECT LOWER(name) AS LowerCase_name FROM employee;

Output:


```
LowerCase_Name
shreya
priya
riya
```

CONCAT():

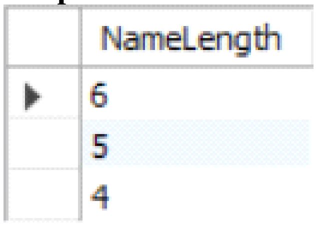
Query:
-- Concatenate name with salary
SELECT CONCAT(id, ‘ ‘,name) AS Username FROM employee;

Output:


```
UserName
1 SHREYA
2 PRIYA
3 RIYA
```

LENGTH():

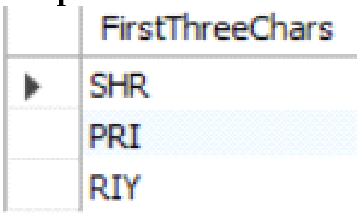
Query:
-- Get the length of each employee's name
SELECT name, LENGTH(name) AS NameLength FROM employee;

Output:


```
NameLength
6
5
4
```

SUBSTRING():

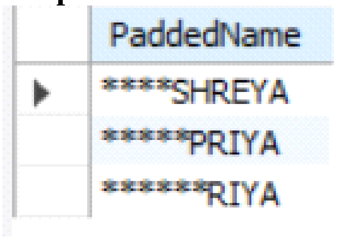
Query:
-- Get a substring of the name (first 3 characters)
SELECT name, SUBSTRING(name, 1, 3) AS FirstThreeChars FROM employee;

Output:


```
FirstThreeChars
SHR
PRI
RIY
```

LPAD():

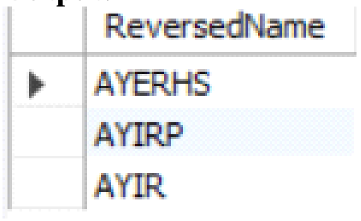
Query:
-- Left pad the name with spaces to a length of 15
SELECT name, LPAD(name, 10, '* ') AS PaddedName FROM employee;

Output:


```
PaddedName
****SHREYA
*****PRIYA
******RIYA
```

REVERSE():

Query:
-- Reverse the name
SELECT name, REVERSE(name) AS Reversedname FROM employee;

Output:


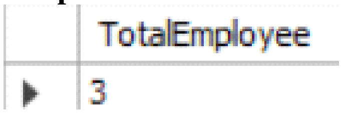
```
ReversedName
AYERHS
AYIRP
AYIR
```

2. Aggregate Functions:

COUNT():

Query:
```
-- Count the total number of employees
SELECT COUNT(*) AS TotalEmployee FROM employee;
```

Output:

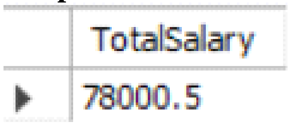


```
TotalEmployee
3
```

SUM():

Query:
```
-- Sum of all salaries
SELECT SUM(salary) AS TotalSalary FROM employee;
```

Output:

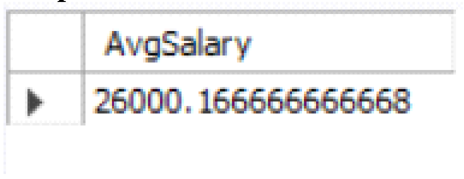


```
TotalSalary
78000.5
```

AVG():

Query:
```
-- Average salary
SELECT AVG(salary) AS AvgSalary FROM employee;
```

Output:

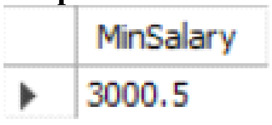


```
AvgSalary
26000.166666666668
```

MIN():

Query:
```
-- Minimum salary
SELECT MIN(salary) AS MinSalary FROM employee;
```

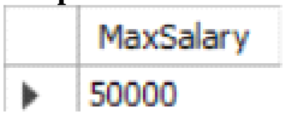
Output:



```
MinSalary
3000.5
```

MAX():

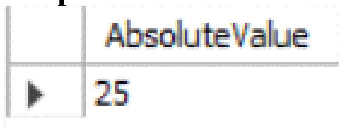
Query:
-- Maximum salary
SELECT MAX(salary) AS MaxSalary FROM employee;

Output:

```
MaxSalary
50000
```

3. Math Functions:

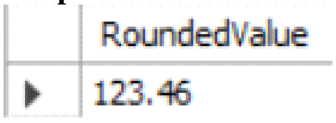
ABS():

Query:
-- Absolute value of -25
SELECT ABS(-25) AS AbsoluteValue;

Output:

```
AbsoluteValue
25
```

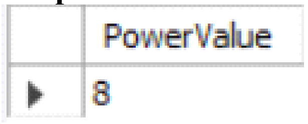
ROUND():

Query:
-- round figure of 123.4567
SELECT ROUND(123.4567, 2) RoundedValue;

Output:

```
RoundedValue
123.46
```

POWER():

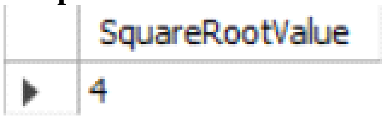
Query:
-- Power 2 raised to 3
SELECT POWER(2, 3) AS PowerValue;

Output:

```
PowerValue
8
```

SQRT():

Query:
-- Squareroot of 16
SELECT SQRT(16) AS SquareRootValue;

Output:

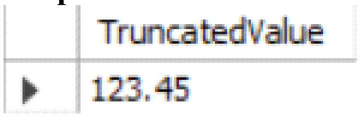


SquareRootValue
4

TRUNCATE():

Query:
-- Truncate the value of 123.4567
SELECT TRUNCATE(123.4567, 2) AS TruncatedValue;

Output:



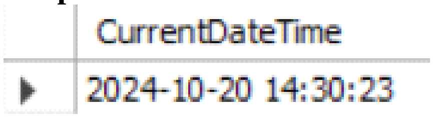
TruncatedValue
123.45

4. Date Functions:

NOW():

Query:
-- Current date and time
SELECT NOW() AS CurrentDateTime;

Output:

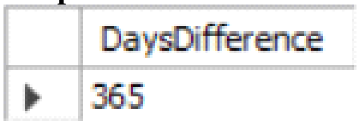


CurrentDateTime
2024-10-20 14:30:23

DATEDIFF():

Query:
-- difference between two date
SELECT DATEDIFF('2024-12-31', '2024-01-01') AS DaysDifference;

Output:



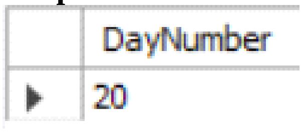
DaysDifference
365

DAY():

Query:

```
-- Select day form date
SELECT DAY(‘2024-10-20’) AS DayNumber;
```

Output:



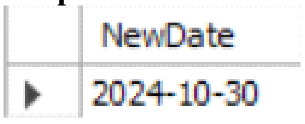
```
DayNumber
20
```

ADD():

Query:

```
-- Add date
SELECT DATE_ADD(‘2024-10-20’, INTERVAL 10 DAY) AS NewDate;
```

Output:



```
NewDate
2024-10-30
```

Conclusion: This experiment explored SQL String, Aggregate, Mathematical, and Date functions, highlighting their importance in text processing, data summarization, complex calculations, and date management. These functions demonstrate SQL's power for efficient database operations.

Mrudula Kamble (22610019)
TY IT (T1 Batch)

Experiment No. 6

Title: Study and Implementation of various types of Constraint in SQL

Aim: To study and implement various types of constraints in SQL by creating a table called EMP with specified constraints and performing SQL queries to ensure data integrity and enforce business rules.

Objectives:

- **Understand and implement various SQL constraints**: Practice defining and applying constraints such as NOT NULL, CHECK, UNIQUE, and PRIMARY KEY to ensure data integrity.
- **Create a structured and valid table**: Design a table with constraints to manage the integrity and accuracy of the data being stored.
- **Perform practical SQL queries**: Execute SQL queries that adhere to these constraints and demonstrate how they enforce business rules.
- **Learn error handling through constraints**: Explore how violating constraints (such as inserting NULL values in NOT NULL columns) triggers errors and how SQL prevents invalid data from being inserted.

Theory: In relational databases, constraints are rules enforced on data columns in a table to ensure accuracy, reliability, and consistency. Constraints define limits on the types of data that can be inserted into a table, thereby preserving the integrity of the database.

In relational databases, there are three main types of relational constraints that ensure data integrity and accuracy across relationships between tables:

1. **Domain Constraints:** Define the permissible values for a given attribute (column) by setting data types and restrictions. Ensure that values stored in each column are of the correct type (e.g., integer, text) and within a valid range.
   Example: Setting data types (e.g., INTEGER, VARCHAR) and value ranges or formats.

2. **Entity Integrity Constraint:** Ensures that each table has a unique identifier (primary key) so each row is distinct and identifiable. Enforces that primary key fields cannot be NULL, ensuring that every record has a unique and valid primary key.

3. **Referential Integrity Constraint:** Maintains consistent relationships between tables by ensuring that foreign key values match primary key values in the referenced (parent) table. Prevents actions that would leave orphaned records in related tables, such as deleting a referenced primary key in the parent table.

Key constraints are essential for defining and enforcing rules on data uniqueness and relationships within and across tables. Types of Key Constraints:

1. **NOT NULL Constraint**: Ensures that a column cannot have a NULL value. It forces the user to provide a valid value when inserting a record.
   Example: If a column has a NOT NULL constraint, attempting to insert a NULL value into that column will generate an error.

2. **CHECK Constraint**: The CHECK constraint ensures that all values in a column meet a specific condition.
   Example: You can use this constraint to ensure that a column value is greater than a certain number, or that it follows a certain pattern.

3. **UNIQUE Constraint**: This constraint ensures that all values in a column are unique across the table.
   Example: If a column has a UNIQUE constraint, no two rows can have the same value in this column. It is useful when certain attributes must not be repeated, such as employee ID or email addresses.

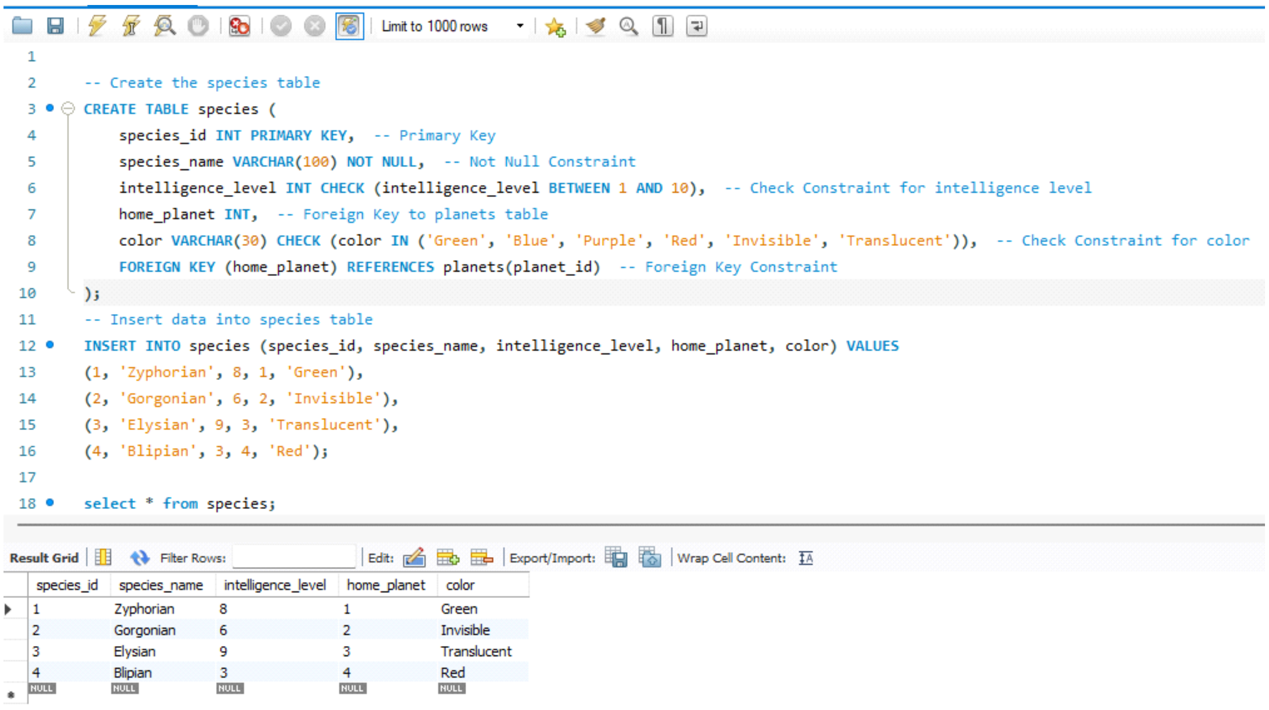
4. **PRIMARY KEY Constraint**: A primary key uniquely identifies each record in a table. It is a combination of the UNIQUE and NOT NULL constraints, meaning that a column designated as a primary key cannot contain NULL values and must have unique entries.
   Example: In a table, the EMPNO column might be designated as the primary key, which ensures that each employee has a unique number that identifies them in the database.

5. **FOREIGN KEY Constraint:** A foreign key enforces a relationship between two tables by linking a column in one table to the primary key of another. It ensures referential integrity, meaning that a value in a foreign key column must match an existing primary key value in the referenced table.
   Example: Consider a database with two tables: Employees and Departments. The Departments table has a primary key column DeptID, which uniquely identifies each department. The Employees table includes a DeptID column, which is a foreign key referencing Departments.DeptID, linking each employee to a department.

Constraints play a critical role in maintaining **data integrity** within a relational database. They help prevent invalid or inconsistent data from being entered, ensuring that the database remains reliable and accurate. By enforcing rules such as uniqueness, non-null values, and valid ranges for numerical data, constraints improve both the quality and reliability of the stored data.

- **Data Integrity**: Ensures that the data is accurate, consistent, and free from anomalies. Constraints prevent duplicate records, missing essential data, or invalid entries.
- **Data Validation**: Constraints help enforce business rules directly in the database, which provides a consistent mechanism for validating data before it is stored.

Implementation:



```sql

-- Create the species table
CREATE TABLE species (
    species_id INT PRIMARY KEY,  -- Primary Key
    species_name VARCHAR(100) NOT NULL,  -- Not Null Constraint
    intelligence_level INT CHECK (intelligence_level BETWEEN 1 AND 10),  -- Check Constraint for intelligence level
    home_planet INT,  -- Foreign Key to planets table
    color VARCHAR(30) CHECK (color IN ('Green', 'Blue', 'Purple', 'Red', 'Invisible', 'Translucent')),  -- Check Constraint for color
    FOREIGN KEY (home_planet) REFERENCES planets(planet_id)  -- Foreign Key Constraint
);
-- Insert data into species table
INSERT INTO species (species_id, species_name, intelligence_level, home_planet, color) VALUES
(1, 'Zyphorian', 8, 1, 'Green'),
(2, 'Gorgonian', 6, 2, 'Invisible'),
(3, 'Elysian', 9, 3, 'Translucent'),
(4, 'Blipian', 3, 4, 'Red');

select * from species;
```

species_id	species_name	intelligence_level	home_planet	color
1	Zyphorian	8	1	Green
2	Gorgonian	6	2	Invisible
3	Elysian	9	3	Translucent
4	Blipian	3	4	Red
NULL	NULL	NULL	NULL	NULL

Query:

Inserting Valid Data:

INSERT INTO planets (planet_id, planet_name, gravity, is_habitable) VALUES
(5, 'Naboo', 9.0, TRUE); -- This will succeed
INSERT INTO species (species_id, species_name, intelligence_level, home_planet, color) VALUES
(6, 'Naboolean', 7, 5, 'Blue'); -- This will succeed

Insert with a Foreign Key Violation

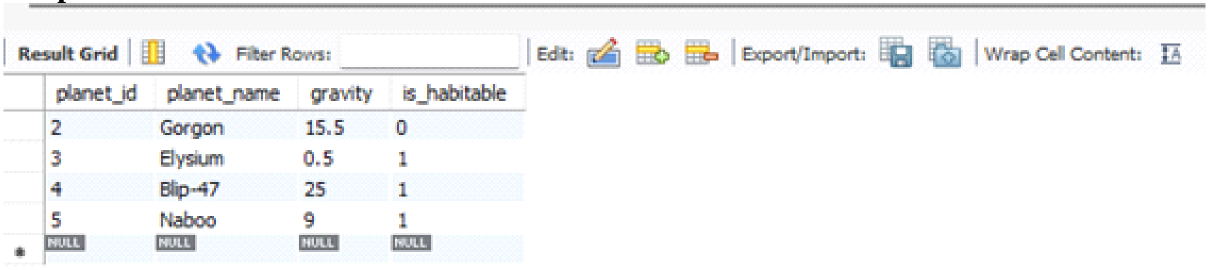
INSERT INTO species (species_id, species_name, intelligence_level, home_planet, color) VALUES
(7, 'Ghostly', 5, 10, 'Invisible'); -- This will fail due to foreign key constraint

Insert Invalid Data

INSERT INTO species (species_id, species_name, intelligence_level, home_planet, color) VALUES
(5, 'Invalid Alien', 11, 1, 'Green'); -- This will fail due to the intelligence_level constraint

SELECT * FROM planets;

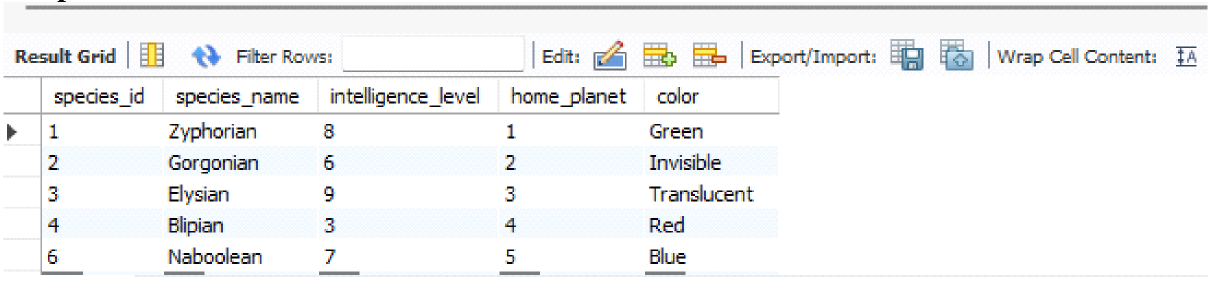
Output:



planet_id	planet_name	gravity	is_habitable
2	Gorgon	15.5	0
3	Elysium	0.5	1
4	Blip-47	25	1
5	Naboo	9	1
NULL	NULL	NULL	NULL

Query:
ALTER TABLE species ADD CONSTRAINT unique_color_per_planet UNIQUE (home_planet, color);

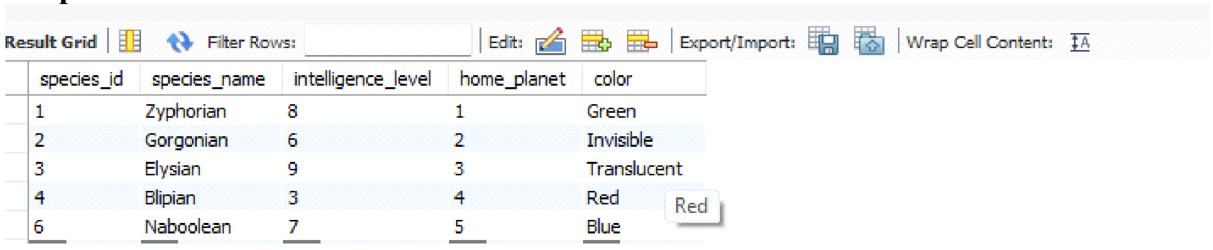
Output:



species_id	species_name	intelligence_level	home_planet	color
1	Zyphorian	8	1	Green
2	Gorgonian	6	2	Invisible
3	Elysian	9	3	Translucent
4	Blipian	3	4	Red
6	Naboolean	7	5	Blue

Query:
ALTER TABLE species ADD CONSTRAINT fk_home_planet FOREIGN KEY (home_planet) REFERENCES planets(planet_id) ON DELETE CASCADE; -- Add foreign key with ON DELETE CASCADE

Output:



species_id	species_name	intelligence_level	home_planet	color
1	Zyphorian	8	1	Green
2	Gorgonian	6	2	Invisible
3	Elysian	9	3	Translucent
4	Blipian	3	4	Red
6	Naboolean	7	5	Blue

Conclusion: This experiment demonstrated the implementation of SQL constraints like NOT NULL, CHECK, UNIQUE, and PRIMARY KEY to ensure data integrity and consistency. By creating the EMP table, we showed how constraints enforce business rules and prevent invalid data, ensuring reliable data storage.

Ajay Sapkal (22610054)
TY IT (T1 Batch)

Experiment No. 7

Title: Implementation of different types of Joins.

Aim: To understand and implement different types of SQL joins and queries using a relational schema consisting of Sailors, Boats, and Reserves tables.

Objectives:

- To learn and apply SQL joins such as Inner Join, Outer Join, and Natural Join.
- To retrieve and analyze data from multiple related tables using SQL queries.
- To practice using aggregation and filtering techniques in SQL.

Theory: In SQL, a JOIN is used to combine rows from two or more tables based on a related column between them. There are several types of joins, each serving different purposes for retrieving data from multiple tables.

Inner Join: The Inner Join retrieves records that have matching values in both tables. It's the most commonly used join in SQL.
Example: to find all sailors who reserved boat number 101:

```
SELECT Sailors.* FROM Sailors
INNER JOIN Reserves ON Sailors.sid = Reserves.sid
WHERE Reserves.bid = 101;
```

Outer Join: An Outer Join returns all records from one table and the matched records from another. If there's no match, the result contains NULL values for non-matching columns.
Example: if you want to retrieve all sailors along with their reservations, including sailors who haven't reserved any boats, you would use a Left Join:

```
SELECT Sailors.*, Reserves.bid
FROM Sailors
LEFT JOIN Reserves ON Sailors.sid = Reserves.sid;
```

1. **Left Outer Join:** The Left Outer Join (or Left Join) returns all records from the Left table and the matched records from the right table. If there's no match, the result contains NULL for columns from the right table.
   Example: Retrieve all sailors along with the boats they reserved, including sailors who haven't reserved any boats.
   ```
   SELECT Sailors.*, Boats.bname
   FROM Sailors
   LEFT JOIN Reserves ON Sailors.sid = Reserves.sid
   LEFT JOIN Boats ON Reserves.bid = Boats.bid;
   ```

2. **Right Outer Join:** The Right Outer Join (or Right Join) returns all records from the right table and the matched records from the left table. If there's no match, the result contains NULL for columns from the left table.
   Example: Retrieve all boats along with the sailors who reserved them, including boats that haven't been reserved by any sailor.

```
SELECT Boats.*, Sailors.sname
FROM Boats
RIGHT JOIN Reserves ON Boats.bid = Reserves.bid
RIGHT JOIN Sailors ON Reserves.sid = Sailors.sid;
```

3. **Full Outer Join:** The Full Outer Join returns all records when there is a match in either left or right table. If there’s no match, NULL is returned for columns from the table without a match.
   Example: Retrieve all sailors and boats, including those who have not made any reservations or those boats that haven't been reserved.
   ```
   SELECT Sailors.*, Boats.*
   FROM Sailors
   FULL OUTER JOIN Reserves ON Sailors.sid = Reserves.sid
   FULL OUTER JOIN Boats ON Reserves.bid = Boats.bid;
   ```

Natural Join: A Natural Join automatically joins tables based on all columns with the same name in both tables.
Example:
```
SELECT sname, bname
FROM Sailors
NATURAL JOIN Reserves
NATURAL JOIN Boats;
```

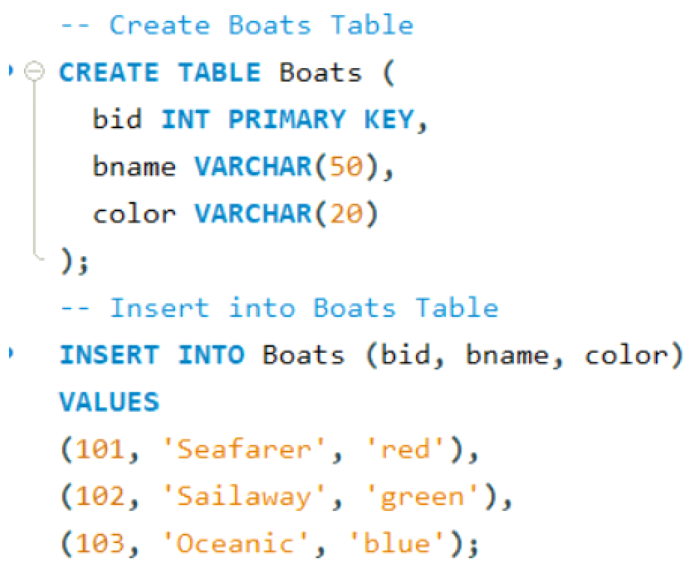
Cross Join: A Cross Join returns the Cartesian product of both tables. This means every row in the first table is combined with every row in the second table, often used when generating all possible combinations.
Example: To get all possible combinations of sailors and boats.
```
SELECT Sailors.sname, Boats.bname
FROM Sailors
CROSS JOIN Boats;
```

Self Join: A Self Join joins a table to itself, useful for comparing rows within the same table.
Example: Find pairs of sailors where one sailor has a higher rating than the other.
```
SELECT S1.sname AS Sailor1, S2.sname AS Sailor2
FROM Sailors S1
INNER JOIN Sailors S2 ON S1.rating > S2.rating;
```

Implementation

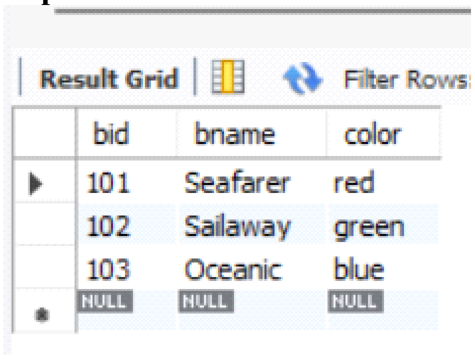
Boats Table:

Query:



```
-- Create Boats Table
CREATE TABLE Boats (
  bid INT PRIMARY KEY,
  bname VARCHAR(50),
  color VARCHAR(20)
);
-- Insert into Boats Table
INSERT INTO Boats (bid, bname, color)
VALUES
(101, 'Seafarer', 'red'),
(102, 'Sailaway', 'green'),
(103, 'Oceanic', 'blue');
```

Output:



Result Grid | Filter Rows:

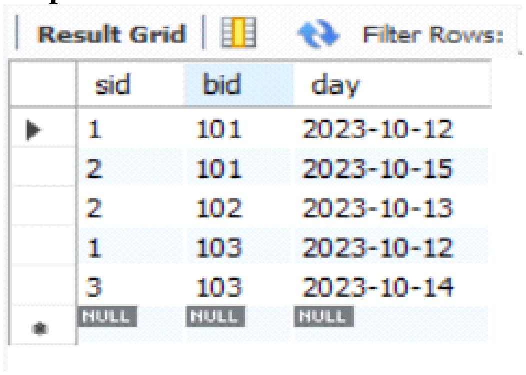
bid	bname	color
101	Seafarer	red
102	Sailaway	green
103	Oceanic	blue
NULL	NULL	NULL

Reserves Table:

Query:

```
-- Create Reserves Table
CREATE TABLE Reserves (
  sid INT,
  bid INT,
  day DATE,
  PRIMARY KEY (sid, bid, day),
  FOREIGN KEY (sid) REFERENCES Sailors(sid),
  FOREIGN KEY (bid) REFERENCES Boats(bid)
);
-- Insert into Reserves Table
INSERT INTO Reserves (sid, bid, day)
VALUES
(1, 101, '2023-10-12'),
(2, 102, '2023-10-13'),
(3, 103, '2023-10-14'),
(1, 103, '2023-10-12'),
(2, 101, '2023-10-15');
```

Output:



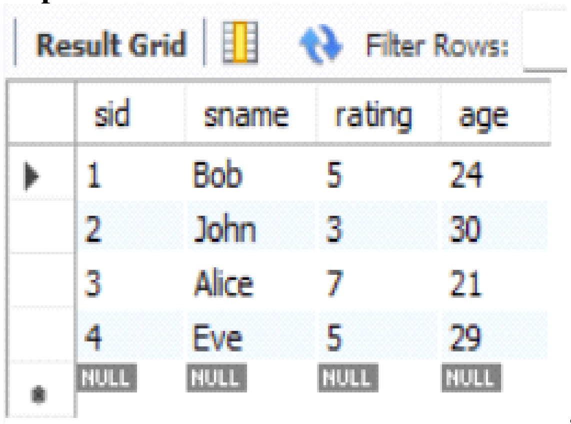
sid	bid	day
1	101	2023-10-12
2	101	2023-10-15
2	102	2023-10-13
1	103	2023-10-12
3	103	2023-10-14
NULL	NULL	NULL

Sailors Table:

Query:

```
-- Create Sailors Table
CREATE TABLE Sailors (
  sid INT PRIMARY KEY,
  sname VARCHAR(50),
  rating INT,
  age INT
);
-- Insert into Sailors Table
INSERT INTO Sailors (sid, sname, rating, age)
VALUES
(1, 'Bob', 5, 24),
(2, 'John', 3, 30),
(3, 'Alice', 7, 21),
(4, 'Eve', 5, 29);
```

Output:



sid	sname	rating	age
1	Bob	5	24
2	John	3	30
3	Alice	7	21
4	Eve	5	29
NULL	NULL	NULL	NULL

:

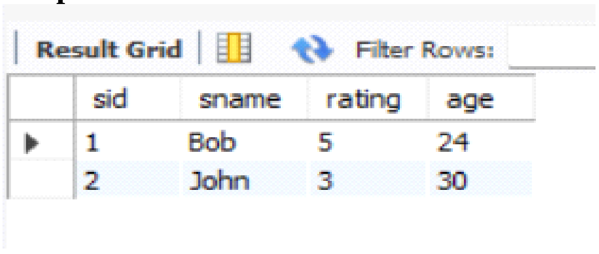
.

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

Query:

```
SELECT Sailors.*
FROM Sailors
INNER JOIN Reserves ON Sailors.sid = Reserves.sid
WHERE Reserves.bid = 101;
```

Output:



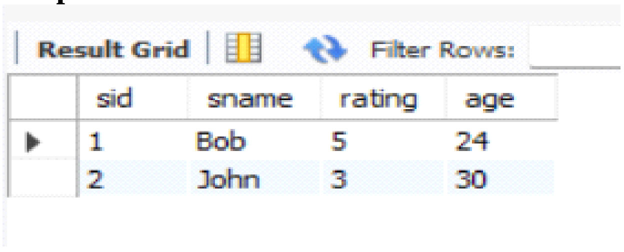
```
   sid  sname  rating  age
▶  1    Bob    5       24
   2    John   3       30
```

Find the name of boat reserved by Bob.

Query:

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

Output:



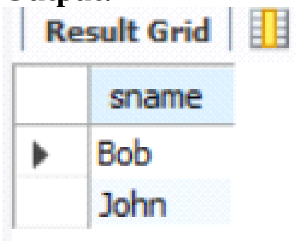
```
   sid  sname  rating  age
▶  1    Bob    5       24
   2    John   3       30
```

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

Query:

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

Output:

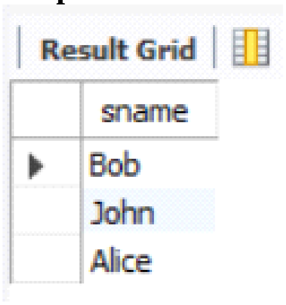


```
   sname
▶  Bob
   John
```

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

Query:
```
SELECT DISTINCT Sailors.sname
FROM Sailors
INNER JOIN Reserves ON Sailors.sid = Reserves.sid;
```

Output:

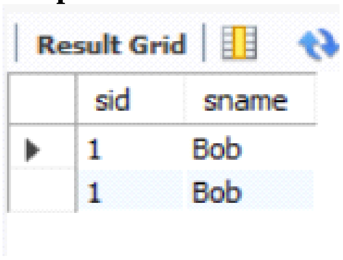


```
sname
Bob
John
Alice
```

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

Query:
```
SELECT Sailors.sid, Sailors.sname
FROM Sailors
INNER JOIN Reserves R1 ON Sailors.sid = R1.sid
INNER JOIN Reserves R2 ON Sailors.sid = R2.sid
WHERE R1.bid != R2.bid AND R1.day = R2.day;
```

Output:

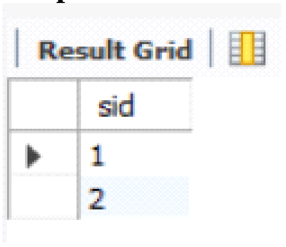


```
sid   sname
1     Bob
1     Bob
```

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

Query:
```
SELECT DISTINCT Sailors.sid
FROM Sailors
INNER JOIN Reserves ON Sailors.sid = Reserves.sid
INNER JOIN Boats ON Reserves.bid = Boats.bid
WHERE Boats.color = 'red' OR Boats.color = 'green';
```

Output:



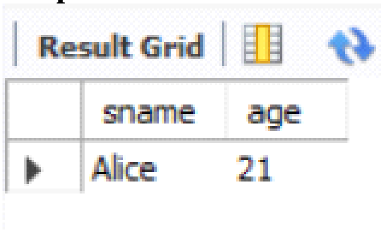
	sid
▶	1
	2

Find the name and the age of the youngest sailor.

Query:

```
SELECT sname, age
FROM Sailors
ORDER BY age ASC
LIMIT 1;
```

Output:



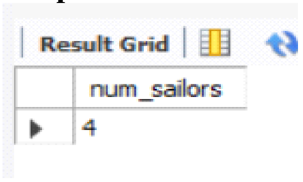
	sname	age
▶	Alice	21

Count the number of different sailor names.

Query:

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

Output:



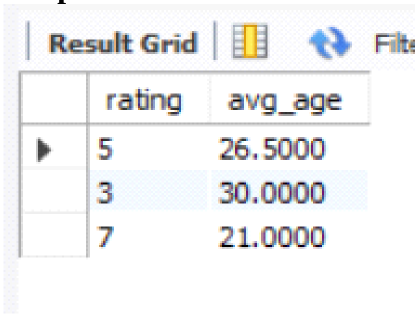
	num_sailors
▶	4

Find the average age of sailors for each rating level.

Query:

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

Output:



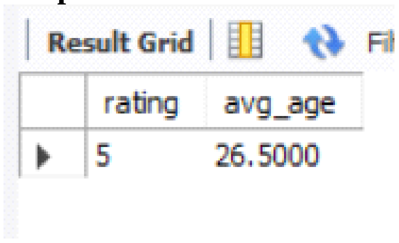
```
rating  avg_age
5       26.5000
3       30.0000
7       21.0000
```

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

Query:

```sql
SELECT rating, AVG(age) AS avg_age
FROM Sailors
GROUP BY rating
HAVING COUNT(*) >= 2;
```

Output:



```
rating  avg_age
5       26.5000
```

Conclusion: This assignment involved implementing SQL joins to retrieve data from multiple tables. We practiced using Inner, Outer, and Natural Joins, along with filtering and aggregation functions, gaining essential skills for managing relational databases efficiently.

Siddhant Supe (22610056)
TY IT (T1 Batch)
Experiment No. 8

Title: Study & Implementation of various types of Clauses and Indexing.

Aim: To understand the concept of SQL Group by & Having Clause, Order by Clause, Where clause and Indexing.

Objectives:

- To create and understand the functionality of various SQL clauses, such as GROUP BY, HAVING, ORDER BY, and WHERE.
- To implement indexing for optimizing database queries.
- To test the effectiveness of indexing and clauses using sample data.

Theory:

Clause: A Clause is a component of a SQL statement that specifies a particular part of the query, typically with specific conditions, actions, or constraints. Clauses are used to define what the query should do and how the data should be retrieved or modified. SQL clauses are used to filter, group, order, and modify the results of SQL queries.

1. **Group by & Having Clause:** The GROUP BY clause is used in SQL to arrange identical data into groups. The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.
   Example: Consider the query that groups data by item name and only selects items where the total quantity exceeds 15.
   ```
   SELECT ItemName, SUM(Quantity) AS TotalQuantity
   FROM Sales
   GROUP BY ItemName
   HAVING SUM(Quantity) > 15;
   ```

2. **Order by Clause:** The ORDER BY clause is used to sort the result-set in ascending or descending order. By default, it sorts the records in ascending order. To sort the records in descending order, the DESC keyword is used.
   Example: Sorting the sales by SaleDate in descending order.
   ```
   SELECT SaleID, ItemName, Quantity, Price, SaleDate
   FROM Sales
   ORDER BY SaleDate DESC;
   ```

3. **Where Clause:** The WHERE clause is used to filter records based on a specific condition.
   Example: The following query selects only the sales where the quantity is greater than 10.
   ```
   SELECT SaleID, ItemName, Quantity, Price
   FROM Sales
   WHERE Quantity > 10;
   ```

Indexing: Indexing is used to enhance the performance of queries by allowing the database engine to quickly locate records.

1. **Ordered Indexing:** In ordered indexing, the entries are stored in a specific order based on the indexed columns. This makes it easy to search for data in a sorted manner, allowing for efficient retrieval of ranges of values and exact matches.

   Example: If you have an Employees table and need to find employees with salaries between a specific range, an ordered index on the salary column will speed up the query:
   SELECT * FROM Employees WHERE salary BETWEEN 50000 AND 100000;

2. **Clustering Indexing:** Clustering indexing refers to the physical arrangement of data on disk based on the index. In a clustered index, the actual data is sorted and stored in the order of the index key. There can be only one clustered index per table because the data can only be physically sorted in one way.

   Example: A clustered index on the employee_id column in the Employees table would store the actual employee records in the order of employee_id:
   CREATE CLUSTERED INDEX idx_employee_id ON Employees(employee_id);

3. **Secondary Indexing:** Secondary indexes are additional indexes created on non-primary key columns. These indexes do not affect the physical storage of the table but provide efficient access to data based on columns that are not the primary key.

   Example: To create a secondary index on the last_name column:
   CREATE NONCLUSTERED INDEX idx_last_name ON Employees(last_name);

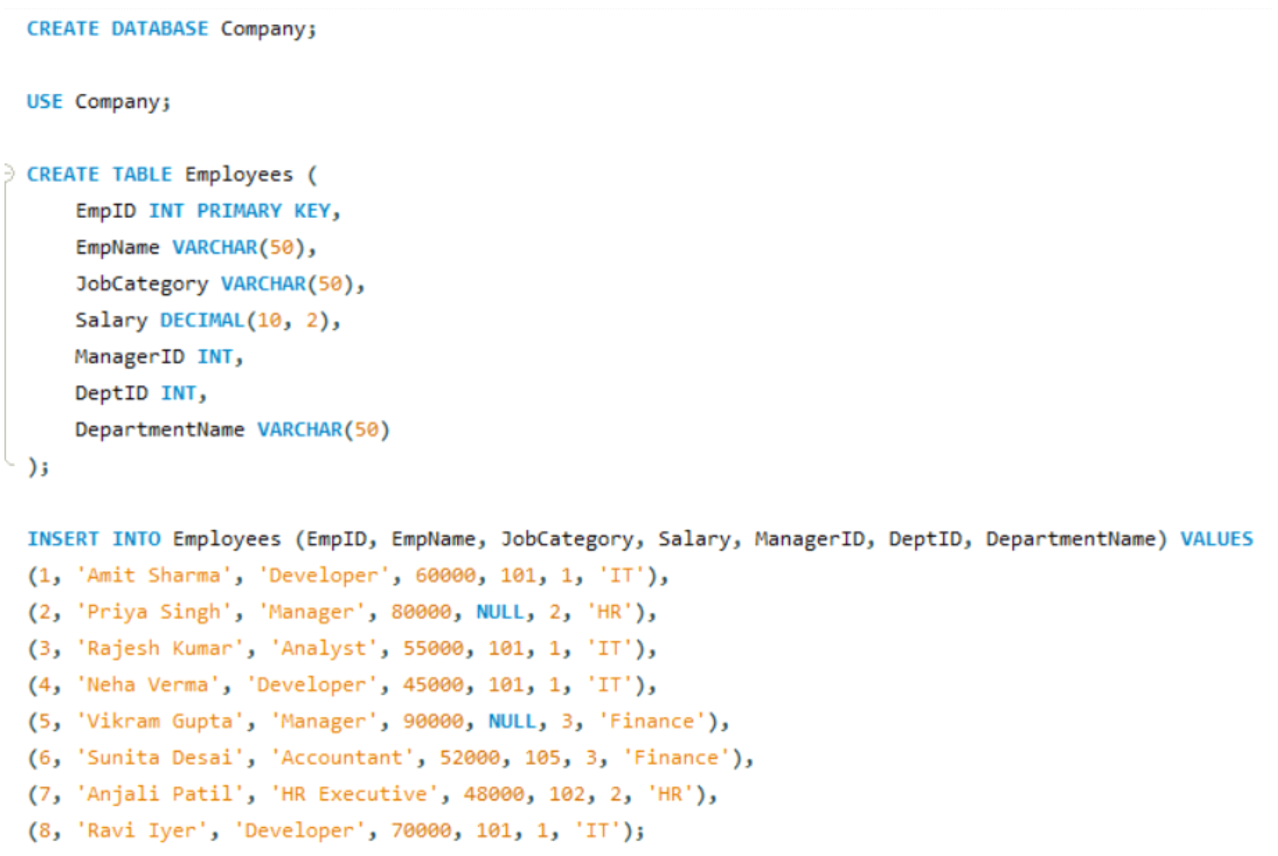
4. **Primary Indexing:** A primary index is created on the primary key of a table. This index is unique, and the data in the table is organized according to the primary key.

   - **Dense Primary Index:** A dense primary index contains an index entry for every record in the table. Each index entry points to the actual row in the table.

   - **Sparse Primary Index:** A sparse primary index only includes index entries for some of the records in the table, typically for records that are key values or require frequent lookup. Sparse indexing is common when tables have large amounts of data, and it is unnecessary to index every row.

Implementation:

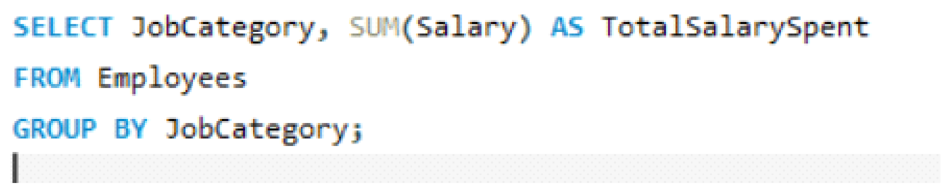
Database:



```sql
CREATE DATABASE Company;

USE Company;

CREATE TABLE Employees (
    EmpID INT PRIMARY KEY,
    EmpName VARCHAR(50),
    JobCategory VARCHAR(50),
    Salary DECIMAL(10, 2),
    ManagerID INT,
    DeptID INT,
    DepartmentName VARCHAR(50)
);

INSERT INTO Employees (EmpID, EmpName, JobCategory, Salary, ManagerID, DeptID, DepartmentName) VALUES
(1, 'Amit Sharma', 'Developer', 60000, 101, 1, 'IT'),
(2, 'Priya Singh', 'Manager', 80000, NULL, 2, 'HR'),
(3, 'Rajesh Kumar', 'Analyst', 55000, 101, 1, 'IT'),
(4, 'Neha Verma', 'Developer', 45000, 101, 1, 'IT'),
(5, 'Vikram Gupta', 'Manager', 90000, NULL, 3, 'Finance'),
(6, 'Sunita Desai', 'Accountant', 52000, 105, 3, 'Finance'),
(7, 'Anjali Patil', 'HR Executive', 48000, 102, 2, 'HR'),
(8, 'Ravi Iyer', 'Developer', 70000, 101, 1, 'IT');
```

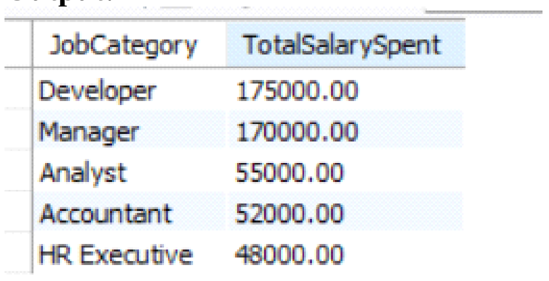
Display total salary spent for each job category.

Query:



```sql
SELECT JobCategory, SUM(Salary) AS TotalSalarySpent
FROM Employees
GROUP BY JobCategory;
```

Output:



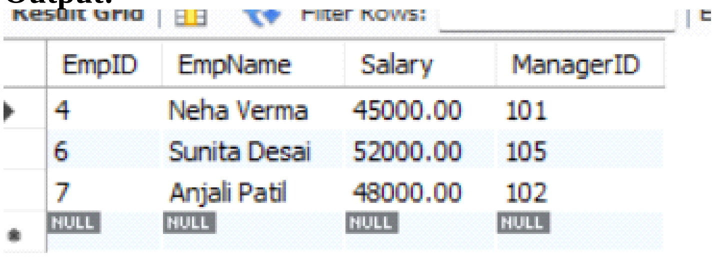
```
JobCategory    TotalSalarySpent
Developer      175000.00
Manager        170000.00
Analyst        55000.00
Accountant     52000.00
HR Executive   48000.00
```

Display lowest paid employee details under each manager.

Query:

```
SELECT EmpID, EmpName, Salary, ManagerID
FROM Employees AS e1
WHERE Salary = (
    SELECT MIN(Salary)
    FROM Employees AS e2
    WHERE e1.ManagerID = e2.ManagerID
) AND ManagerID IS NOT NULL;
```

Output:



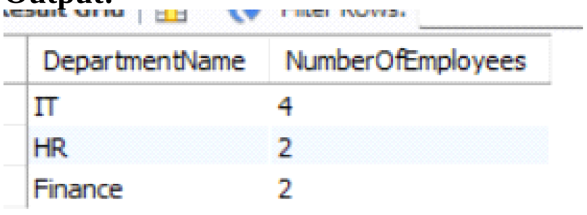
EmpID	EmpName	Salary	ManagerID
4	Neha Verma	45000.00	101
6	Sunita Desai	52000.00	105
7	Anjali Patil	48000.00	102
NULL	NULL	NULL	NULL

Display number of employees working in each department and their department name.

Query:

```
SELECT DepartmentName, COUNT(EmpID) AS NumberOfEmployees
FROM Employees
GROUP BY DepartmentName;
```

Output:



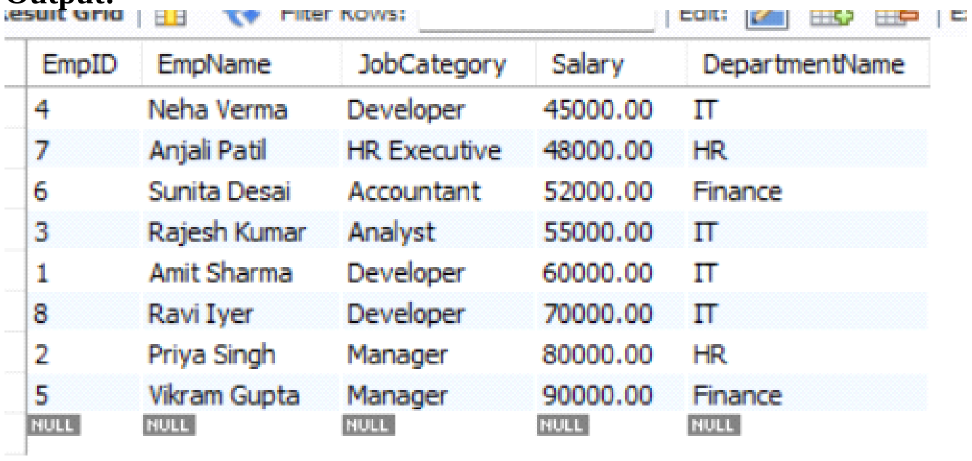
DepartmentName	NumberOfEmployees
IT	4
HR	2
Finance	2

Display the details of employees sorting the salary in increasing order.

Query:

```
SELECT EmpID, EmpName, JobCategory, Salary, DepartmentName
FROM Employees
ORDER BY Salary ASC;
```

Output:



Result Grid | Filter Rows: | Edit:

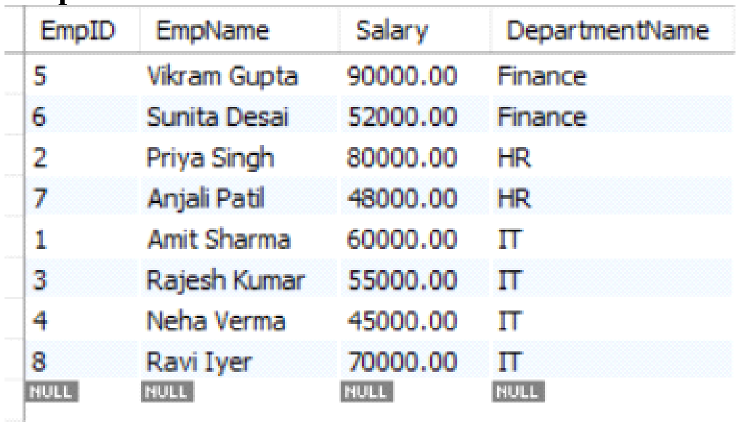
EmpID	EmpName	JobCategory	Salary	DepartmentName
4	Neha Verma	Developer	45000.00	IT
7	Anjali Patil	HR Executive	48000.00	HR
6	Sunita Desai	Accountant	52000.00	Finance
3	Rajesh Kumar	Analyst	55000.00	IT
1	Amit Sharma	Developer	60000.00	IT
8	Ravi Iyer	Developer	70000.00	IT
2	Priya Singh	Manager	80000.00	HR
5	Vikram Gupta	Manager	90000.00	Finance
NULL	NULL	NULL	NULL	NULL

Show the record of employee earning salary greater than 16000 in each department.

Query:

```sql
SELECT EmpID, EmpName, Salary, DepartmentName
FROM Employees
WHERE Salary > 16000
ORDER BY DepartmentName;
```

Output:



EmpID	EmpName	Salary	DepartmentName
5	Vikram Gupta	90000.00	Finance
6	Sunita Desai	52000.00	Finance
2	Priya Singh	80000.00	HR
7	Anjali Patil	48000.00	HR
1	Amit Sharma	60000.00	IT
3	Rajesh Kumar	55000.00	IT
4	Neha Verma	45000.00	IT
8	Ravi Iyer	70000.00	IT
NULL	NULL	NULL	NULL

Create Unique and Non-Clustered Index on given Database.

Unique Index

Query:

```sql
CREATE UNIQUE INDEX idx_unique_empid ON Employees (EmpID);
SHOW INDEX FROM Employees;
```

Output:

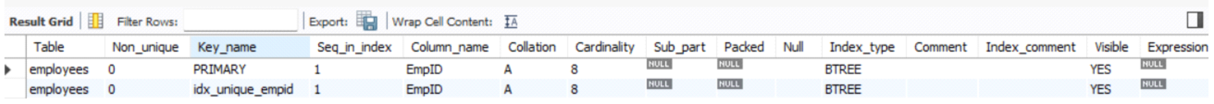
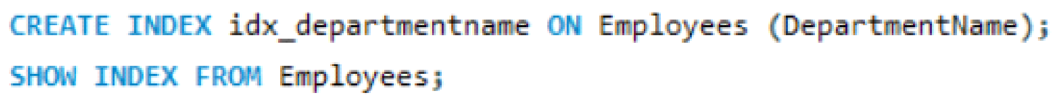


Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
employees	0	PRIMARY	1	EmpID	A	8	NULL	NULL		BTREE			YES	NULL
employees	0	idx_unique_empid	1	EmpID	A	8	NULL	NULL		BTREE			YES	NULL

Non- Clustered Index:

Query:



```sql
CREATE INDEX idx_departmentname ON Employees (DepartmentName);
SHOW INDEX FROM Employees;
```

Output:

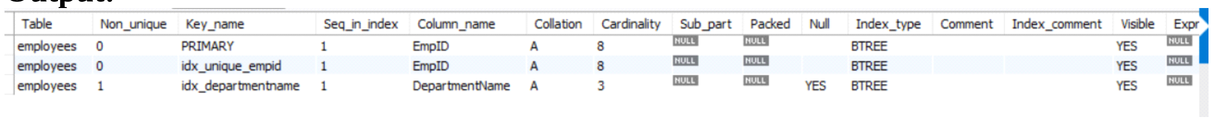


Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expr
employees	0	PRIMARY	1	EmpID	A	8	NULL	NULL		BTREE			YES	NULL
employees	0	idx_unique_empid	1	EmpID	A	8	NULL	NULL		BTREE			YES	NULL
employees	1	idx_departmentname	1	DepartmentName	A	3	NULL	NULL	YES	BTREE			YES	NULL

Conclusion: This experiment demonstrated database creation and management, using SQL queries and clauses like GROUP BY, WHERE, and ORDER BY for efficient data analysis of employee records. Indexing optimized query performance, improving data retrieval and handling of large datasets. These SQL concepts enhance database management and data analysis, forming a strong foundation for future work.

Sagar Meshram (22610069)
TY IT (T1 Batch)
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:

- To create and understand the functionality of triggers in SQL.
- To implement views in SQL for data abstraction.
- To test triggers and views with sample data.

Theory:

1. Trigger: A trigger is an automatic procedure in SQL that executes when a specific event (INSERT, UPDATE, DELETE) occurs on a table. Triggers help enforce data rules, automate tasks, and maintain database integrity.

Types of Triggers:

1. **BEFORE Trigger**: Executes before the event (e.g., data validation).
2. **AFTER Trigger**: Executes after the event (e.g., logging or auditing).
3. **INSTEAD OF Trigger**: Replaces the event (commonly used with views).

Key Terms:

1. **Event**: Action that fires the trigger (INSERT, UPDATE, DELETE).
2. **Timing**: Specifies when the trigger fires (BEFORE, AFTER, INSTEAD OF).
3. **Trigger Body**: SQL code executed by the trigger.
4. **OLD/NEW Values**: References to the row’s values before (OLD) and after (NEW) the event.

Trigger Syntax:

```
CREATE TRIGGER trigger_name
BEFORE|AFTER|INSTEAD OF event
ON table_name
FOR EACH ROW
BEGIN
--Trigger Body
END;
```

Trigger Commands:

1. **Create a Trigger**:
   ```
   CREATE TRIGGER trigger_name
   BEFORE|AFTER event
   ON table_name
   FOR EACH ROW
   BEGIN
   --SQL Statements
   END;
   ```
2. **Delete a Trigger**:
   ```
   DROP TRIGGER trigger_name;
   ```

2. View: A view is a virtual table based on a SELECT query. It simplifies data access and enhances security by restricting access to specific data without storing it physically.

Types of Views:

1. **Simple View**: Based on a single table.
2. **Complex View**: Based on multiple tables or queries with joins.
3. **Materialized View**: Stores query results physically and refreshes periodically.

Key Terms

1. **Base Table**: The table(s) from which the view is derived.
2. **Updatable View**: Allows data modification (INSERT, UPDATE, DELETE) under certain conditions.
3. **Non-Updatable View**: Does not allow data modification due to complex queries or joins.

View Syntax:
```
CREATE VIEW view_name AS
SELECT column1, column2
FROM table_name
WHERE condition;
```

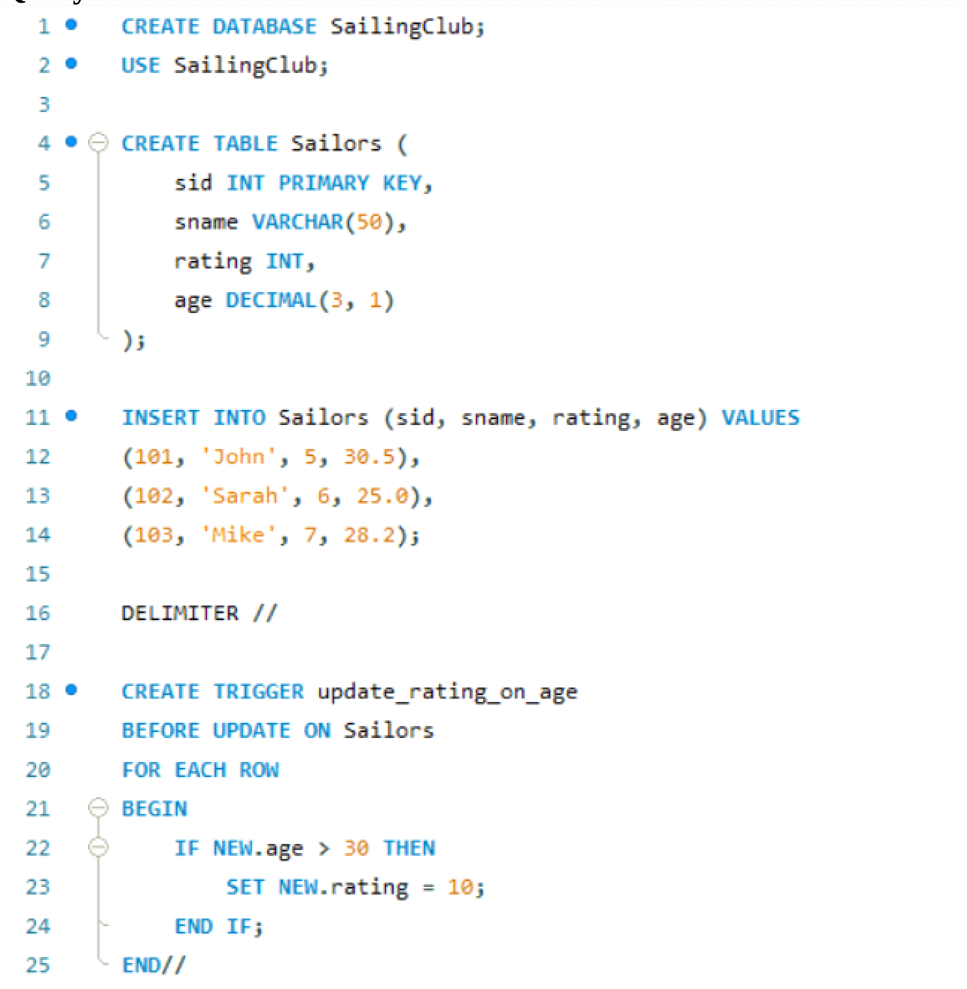
View Commands:

1. **Create a View**:
   ```
   CREATE VIEW view_name AS
   SELECT column1, column2
   FROM table_name
   WHERE condition;
   ```
2. **Delete a View**:
   ```
   DROP VIEW view_name;
   ```

Implementation:

1. Trigger:

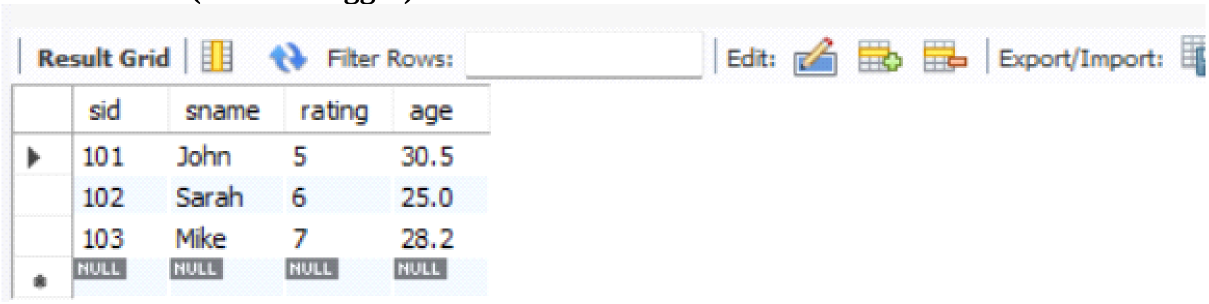
Query:



```
CREATE DATABASE SailingClub;
USE SailingClub;

CREATE TABLE Sailors (
    sid INT PRIMARY KEY,
    sname VARCHAR(50),
    rating INT,
    age DECIMAL(3, 1)
);

INSERT INTO Sailors (sid, sname, rating, age) VALUES
(101, 'John', 5, 30.5),
(102, 'Sarah', 6, 25.0),
(103, 'Mike', 7, 28.2);

DELIMITER //

CREATE TRIGGER update_rating_on_age
BEFORE UPDATE ON Sailors
FOR EACH ROW
BEGIN
    IF NEW.age > 30 THEN
        SET NEW.rating = 10;
    END IF;
END//
```

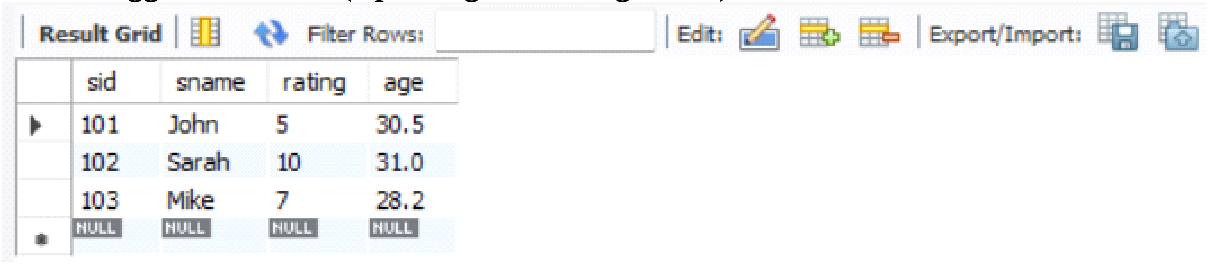
Output:

Sailors Table (Before Trigger):



sid	sname	rating	age
101	John	5	30.5
102	Sarah	6	25.0
103	Mike	7	28.2
NULL	NULL	NULL	NULL

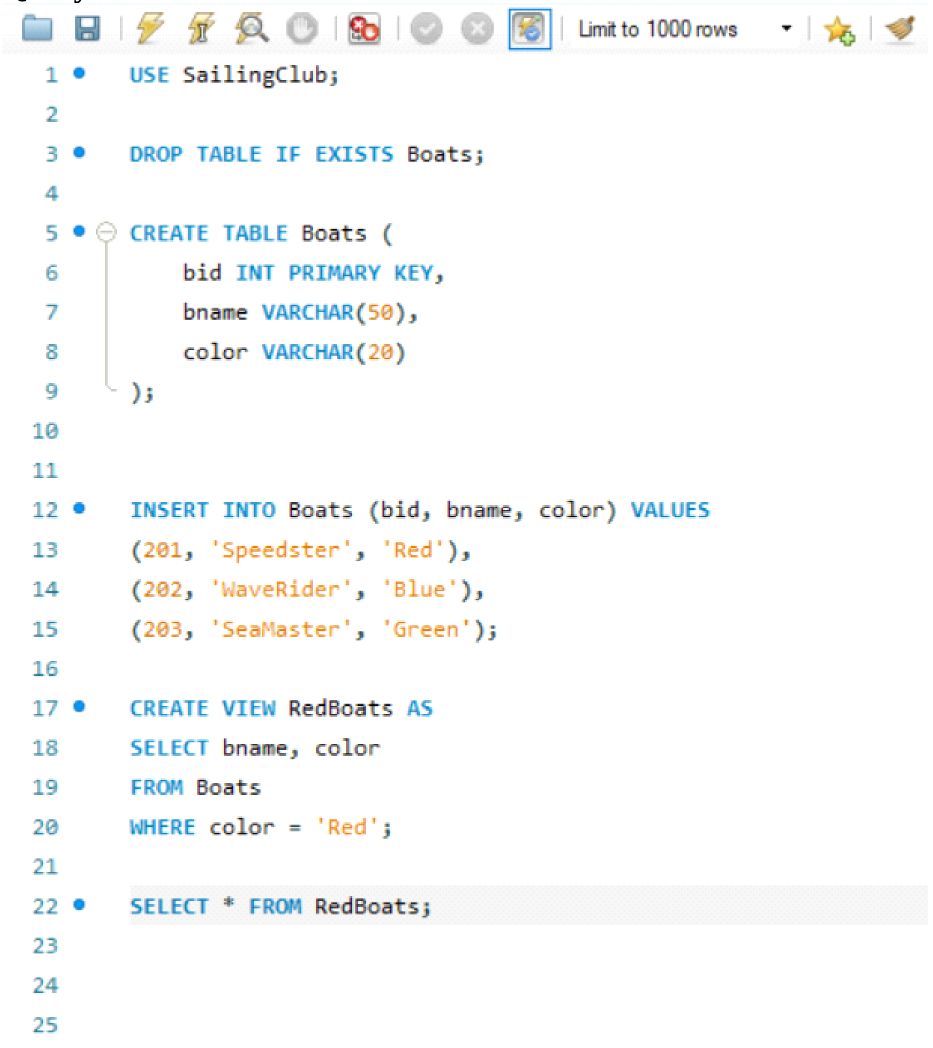
After Trigger Execution (Updating Sarah's age to 31):



sid	sname	rating	age
101	John	5	30.5
102	Sarah	10	31.0
103	Mike	7	28.2
NULL	NULL	NULL	NULL

2. View:

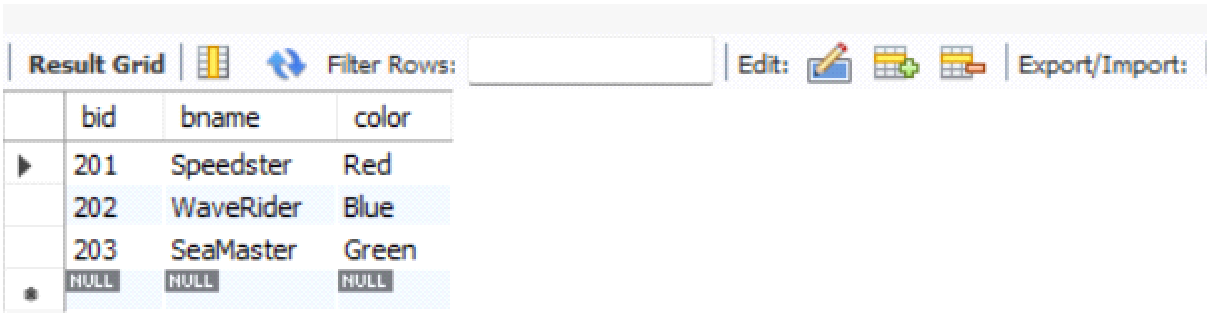
Query:



```sql
USE SailingClub;

DROP TABLE IF EXISTS Boats;

CREATE TABLE Boats (
    bid INT PRIMARY KEY,
    bname VARCHAR(50),
    color VARCHAR(20)
);


INSERT INTO Boats (bid, bname, color) VALUES
(201, 'Speedster', 'Red'),
(202, 'WaveRider', 'Blue'),
(203, 'SeaMaster', 'Green');

CREATE VIEW RedBoats AS
SELECT bname, color
FROM Boats
WHERE color = 'Red';

SELECT * FROM RedBoats;



```

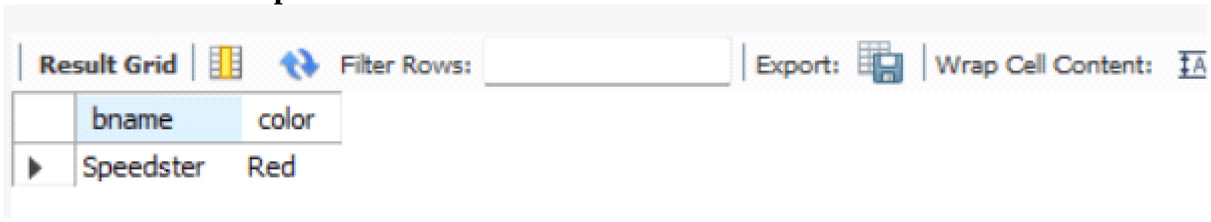
Output:

Boats Table:



	bid	bname	color
▶	201	Speedster	Red
	202	WaveRider	Blue
	203	SeaMaster	Green
*	NULL	NULL	NULL

RedBoats View Output:



	bname	color
▶	Speedster	Red

Conclusion: Triggers and views enhance database management by automating tasks, ensuring data integrity, and simplifying data access. Triggers enforce rules and automate updates, while views organize data and improve security. Together, they streamline operations and improve database efficiency.

Gopal Wagh (22610087)
TY IT (T1 Batch)
Experiment No. 10

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

Aim: The objective of this experiment is to understand and implement basic CRUD operations in MongoDB, a NoSQL database, using its native query language. This will help in managing databases and collections, which store documents in BSON format, and performing common database operations efficiently.

Objectives: The main objectives of the study and implementation of CRUD operations on MongoDB are:

- **Understand NoSQL Concepts:** To familiarize with NoSQL databases, particularly MongoDB, and comprehend the differences between NoSQL and traditional relational databases.
- **Learn Basic CRUD Operations:**
  1. **Create**: To learn how to insert documents into collections within a MongoDB database.
  2. **Read**: To query and retrieve data from MongoDB collections using various query filters and projection techniques.
  3. **Update**: To modify existing data in the MongoDB database, including adding, updating, or removing specific fields.
  4. **Delete**: To delete documents or collections from the database using appropriate delete commands.

Theory: MongoDB is a popular NoSQL database known for its flexibility and scalability. Unlike traditional relational databases, MongoDB stores data in collections (analogous to tables in SQL) as documents in BSON (Binary JSON) format, allowing for a more dynamic schema. Documents in MongoDB are key-value pairs and can have varying structures, making MongoDB ideal for applications where data requirements change over time.

Databases and Collections in MongoDB:

1. **Database**: A container for collections. Each MongoDB instance can have multiple databases.
2. **Collection**: A group of MongoDB documents. Collections are analogous to tables in relational databases.
3. **Document**: A record in MongoDB, typically represented in BSON format. It is analogous to a row in a relational database table.

Steps to Work with MongoDB on Windows:

1. Installation of MongoDB on Windows:

1. Download MongoDB
2. Run the Installer
3. Configure MongoDB as a Service

2. Starting MongoDB Service:

1. **Start MongoDB:** manually start the MongoDB service, open Command Prompt and run: net start MongoDB

2. **Stop MongoDB:** To stop the MongoDB service, use: net stop MongoDB
3. **Verify MongoDB is Running:** To check if the MongoDB service is running, use the command: net start

3. Accessing MongoDB via Command Prompt:

1. **Open Command Prompt:** Open Command Prompt (cmd) as an administrator.
2. **Start the MongoDB Shell:** To interact with MongoDB, run the following command to launch the MongoDB shell: mongosh

4. Create a New User in MongoDB: Once you have logged into the MongoDB shell, you can create a new user with specific privileges.

1. **Switch to the Admin Database:** use admin
2. **Create a New User:** You can create a new user with a specific role using the following command:
   ```
   db.createUser({
   user: "username",
   pwd: "password",
   roles: [{ role: "readWrite", db: "test" }]});
   ```
3. **Verify User Creation:** You can check the created users with: show users

5. Connect to MongoDB Using the Created User: Once the user is created, you can connect to MongoDB as that user:

1. **Authenticate as the New User:** To log in using the created user:
```
mongosh -u username -p password --authenticationDatabase admin
```

CRUD Operations in MongoDB:

1. **Create (Insert Operation):** The insertOne() and insertMany() operations are used to add new documents to a collection. MongoDB allows documents to be inserted without enforcing a strict schema, enabling flexible data storage
   Example: db.users.insertOne({ name: "Alice", age: 25, email: alice@example.com })

2. **Read:** The find() function retrieves documents from a collection based on a query filter. It can be combined with projection to fetch specific fields. MongoDB offers various operators such as $gt, $lt, and $regex for building complex queries.

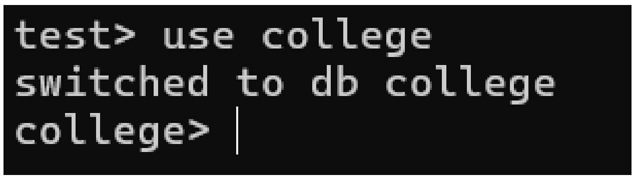
3. **Update Operation:** The updateOne() and updateMany() functions are used to modify existing documents. You can use operators like $set to add or update fields and $unset to remove them**.**

4. **Delete Operation:** The deleteOne() and deleteMany() functions remove documents from a collection. Deleting entire collections is done with the drop() function.

Implementation:

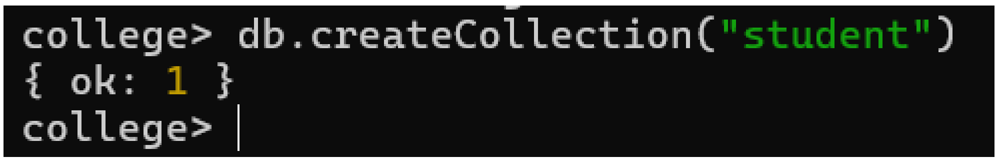
1. Create :

Create database :



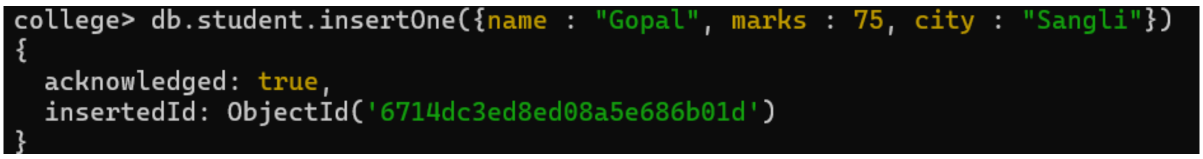
```
test> use college
switched to db college
college>
```

Create Collection :

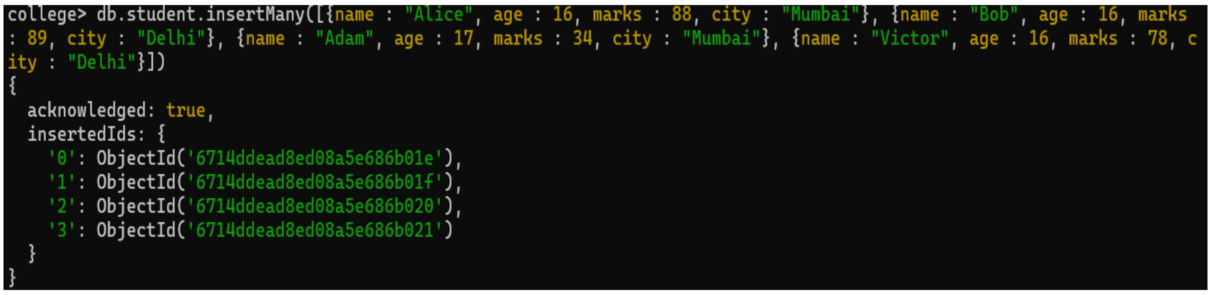


```
college> db.createCollection("student")
{ ok: 1 }
college>
```

2. Insert :

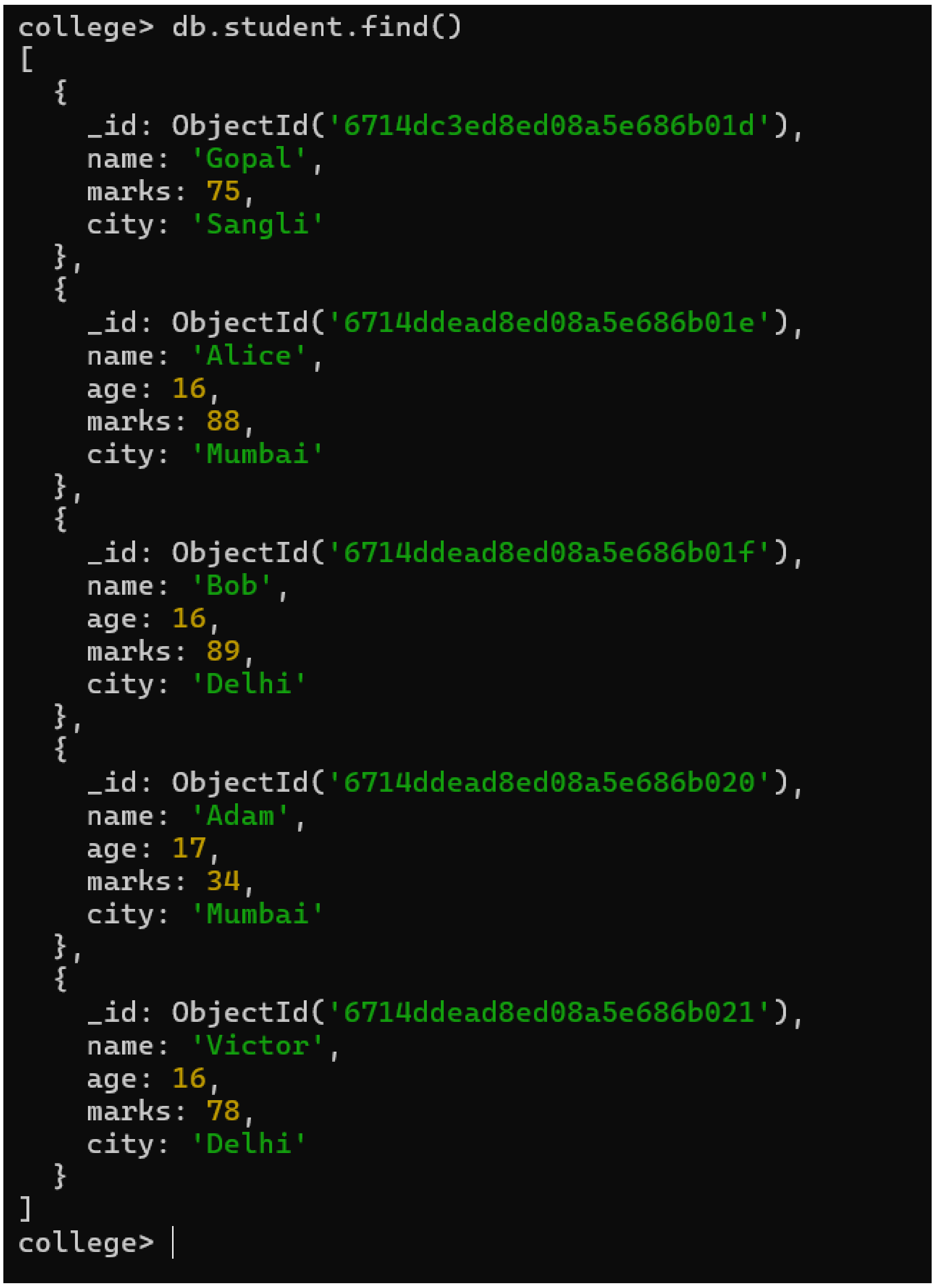


```
college> db.student.insertOne({name : "Gopal", marks : 75, city : "Sangli"})
{
  acknowledged: true,
  insertedId: ObjectId('6714dc3ed8ed08a5e686b01d')
}
```

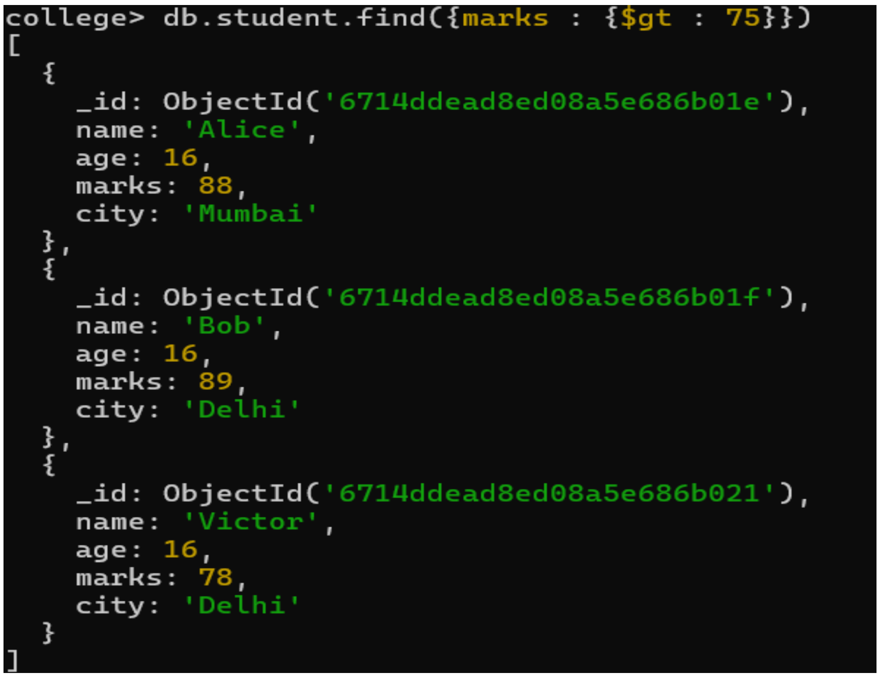


```
college> db.student.insertMany([{name : "Alice", age : 16, marks : 88, city : "Mumbai"}, {name : "Bob", age : 16, marks
: 89, city : "Delhi"}, {name : "Adam", age : 17, marks : 34, city : "Mumbai"}, {name : "Victor", age : 16, marks : 78, c
ity : "Delhi"}])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('6714ddead8ed08a5e686b01e'),
    '1': ObjectId('6714ddead8ed08a5e686b01f'),
    '2': ObjectId('6714ddead8ed08a5e686b020'),
    '3': ObjectId('6714ddead8ed08a5e686b021')
  }
}
```

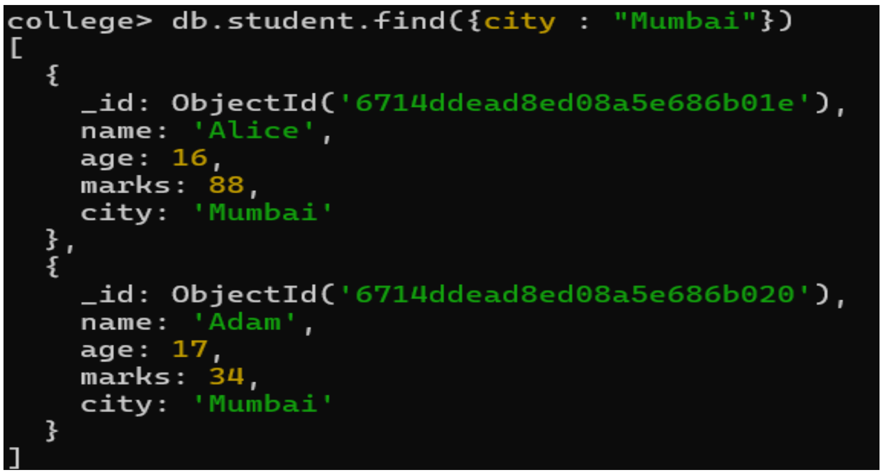
3. Read :



```
college> db.student.find()
[
  {
    _id: ObjectId('6714dc3ed8ed08a5e686b01d'),
    name: 'Gopal',
    marks: 75,
    city: 'Sangli'
  },
  {
    _id: ObjectId('6714ddead8ed08a5e686b01e'),
    name: 'Alice',
    age: 16,
    marks: 88,
    city: 'Mumbai'
  },
  {
    _id: ObjectId('6714ddead8ed08a5e686b01f'),
    name: 'Bob',
    age: 16,
    marks: 89,
    city: 'Delhi'
  },
  {
    _id: ObjectId('6714ddead8ed08a5e686b020'),
    name: 'Adam',
    age: 17,
    marks: 34,
    city: 'Mumbai'
  },
  {
    _id: ObjectId('6714ddead8ed08a5e686b021'),
    name: 'Victor',
    age: 16,
    marks: 78,
    city: 'Delhi'
  }
]
college> 
```

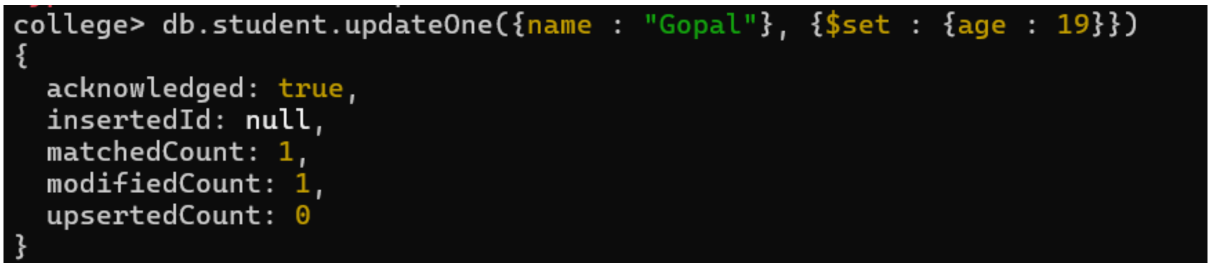


```
college> db.student.find({marks : {$gt : 75}})
[
  {
    _id: ObjectId('6714ddead8ed08a5e686b01e'),
    name: 'Alice',
    age: 16,
    marks: 88,
    city: 'Mumbai'
  },
  {
    _id: ObjectId('6714ddead8ed08a5e686b01f'),
    name: 'Bob',
    age: 16,
    marks: 89,
    city: 'Delhi'
  },
  {
    _id: ObjectId('6714ddead8ed08a5e686b021'),
    name: 'Victor',
    age: 16,
    marks: 78,
    city: 'Delhi'
  }
]
```

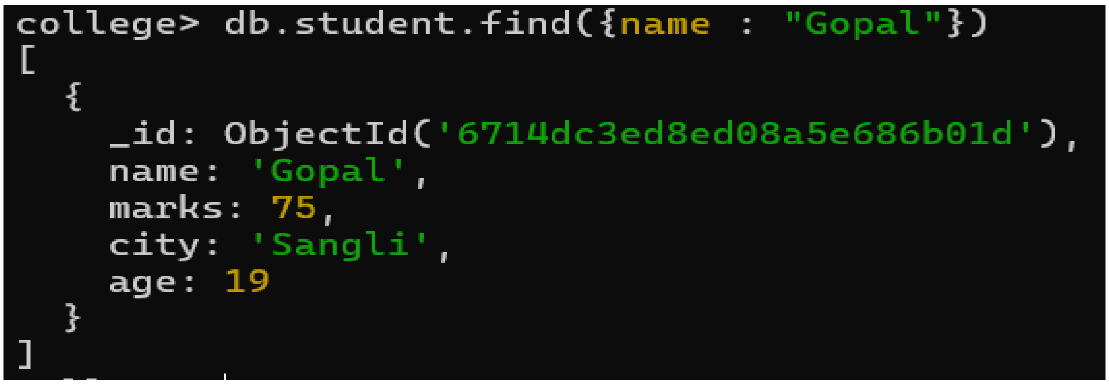


```
college> db.student.find({city : "Mumbai"})
[
  {
    _id: ObjectId('6714ddead8ed08a5e686b01e'),
    name: 'Alice',
    age: 16,
    marks: 88,
    city: 'Mumbai'
  },
  {
    _id: ObjectId('6714ddead8ed08a5e686b020'),
    name: 'Adam',
    age: 17,
    marks: 34,
    city: 'Mumbai'
  }
]
```

4. Update :

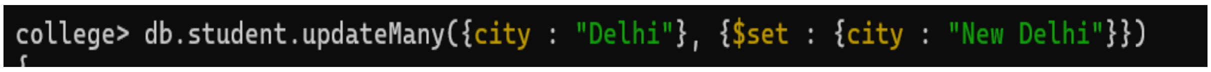


```
college> db.student.updateOne({name : "Gopal"}, {$set : {age : 19}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```



```
college> db.student.find({name : "Gopal"})
[
  {
    _id: ObjectId('6714dc3ed8ed08a5e686b01d'),
    name: 'Gopal',
    marks: 75,
    city: 'Sangli',
    age: 19
  }
]
```

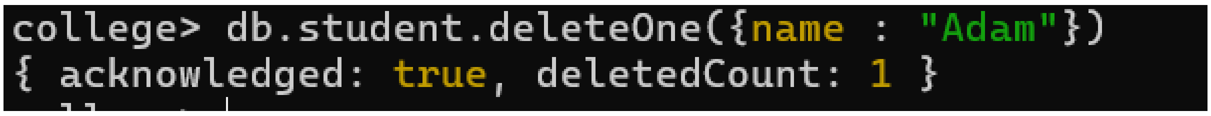
5. Replace:



```
college> db.student.updateMany({city : "Delhi"}, {$set : {city : "New Delhi"}})
```

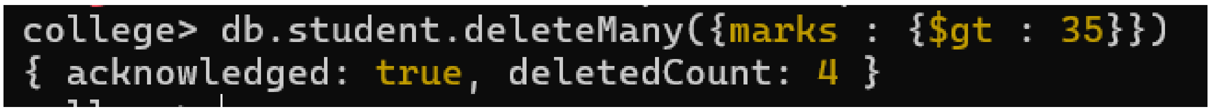
6. Delete :

Delete document where name = Alice



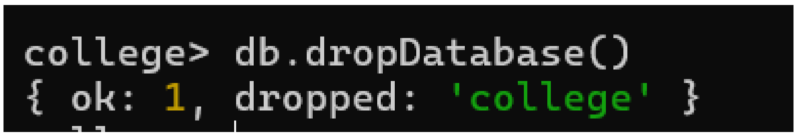
```
college> db.student.deleteOne({name : "Adam"})
{ acknowledged: true, deletedCount: 1 }
```

Delete all documents where marks greater than 35



```
college> db.student.deleteMany({marks : {$gt : 35}})
{ acknowledged: true, deletedCount: 4 }
```

Delete current database



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

Conclusion: This experiment successfully implemented basic CRUD operations in MongoDB, demonstrating its flexibility with dynamic schemas and powerful querying capabilities, making it an ideal choice for NoSQL database management in diverse applications.

Shivam Mathapati (22610088)
TY IT (T1 Batch)
Experiment No. 11

Title: Filtering for Data Efficiently on MongoDB Database.

Aim: The aim of this experiment is to study and implement efficient filtering techniques in MongoDB, focusing on how to query data effectively using MongoDB's powerful querying capabilities. This includes learning how to use operators, indexes, and filters to retrieve specific data sets quickly and efficiently.

Objectives:

- **Understand MongoDB Queries:** Learn how to use the find() method and various operators like $eq, $ne, $gt, $lt, $in, and $regex for filtering data.
- **Implement Efficient Filtering:** Study and implement techniques like indexing, projection, and pagination to optimize query performance.
- **Explore Real-world Scenarios:** Apply filtering techniques to real-world datasets, such as filtering customer or product data based on different attributes.
- **Optimize Query Performance:** Explore how indexes can reduce query execution time and how to avoid performance pitfalls like full collection scans.

Theory: MongoDB is a NoSQL database that allows for flexible and powerful querying. The find() method is at the heart of querying in MongoDB, and it allows for retrieving documents from collections based on specific filter criteria. Efficient filtering not only ensures correct results but also impacts the performance, especially for large datasets.

Common Query Operators:
Equality ($eq): Filters documents that match a specific value.
Example: Retrieves all documents where age is 25.
{ age: { $eq: 25 } }

Inequality ($ne, $gt, $lt): Filters documents that do not match or are greater/less than a value.
Example: Retrieves documents where age is not 25.
{ age: { $ne: 25 } }

Logical Operators ($and, $or): Combine multiple conditions.
Example: Retrieves documents where age is between 20 and 30.
{ $and: [{ age: { $gt: 20 } }, { age: { $lt: 30 } }] }

Inclusion ($in): Filters documents that match any value in an array.
Example: Retrieves documents where age is 20, 25, or 30.
{ age: { $in: [20, 25, 30] } }

Regular Expressions ($regex): Used for pattern matching.
Example: Retrieves documents where name starts with "A".
{ name: { $regex: '^A', $options: 'i' }

Optimizing Queries:
Indexing: MongoDB allows the creation of indexes on fields to improve the speed of query execution.
Example: Indexing on age
db.collection.createIndex({ age: 1 }).

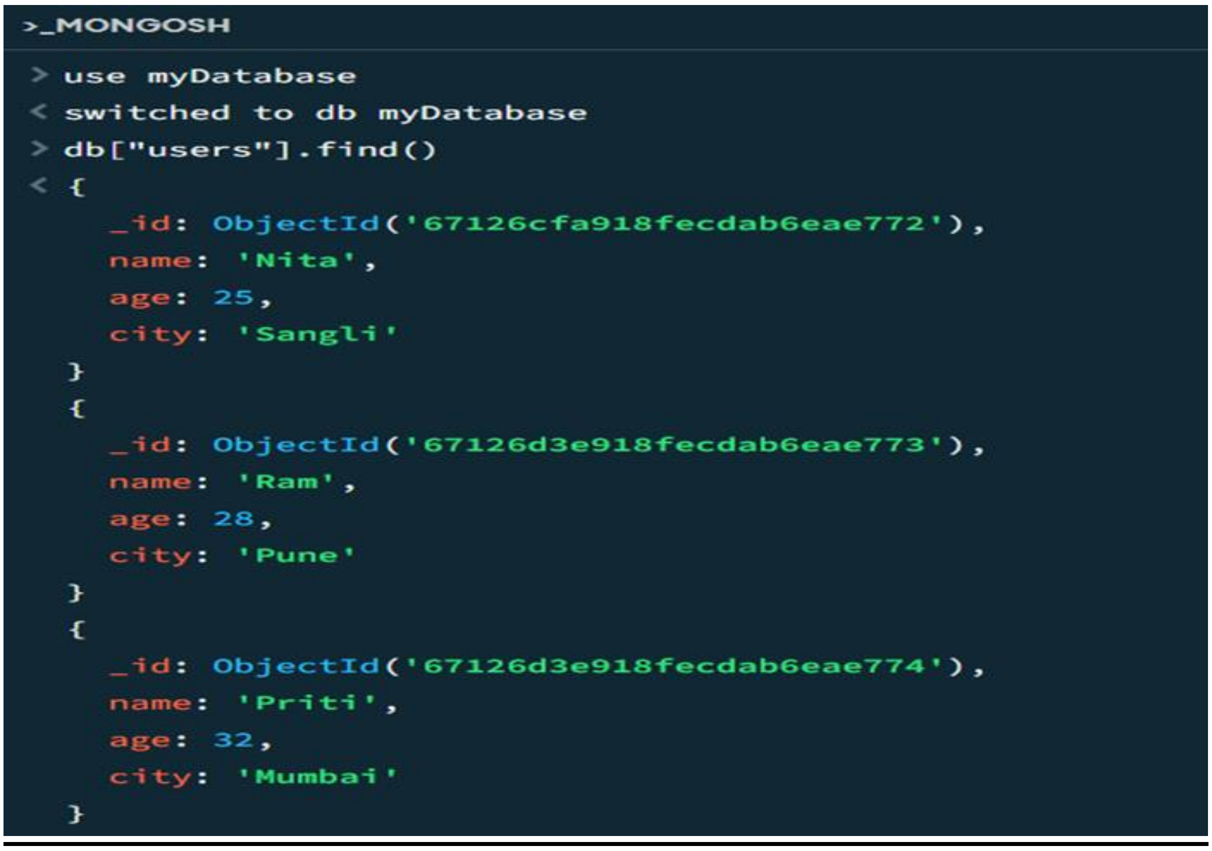
Projection: Limiting the fields returned by a query to reduce data transfer and improve performance.
Example: Retrieves only the name and age fields.
db.collection.find({ age: { $gt: 25 } }, { name: 1, age: 1 })

Pagination: Using skip() and limit() methods to paginate through results.
Example: Skips the first 10 documents and limits the result to 5 documents.
db.collection.find().skip(10).limit(5)

Differences Between SQL and MongoDB Querying:

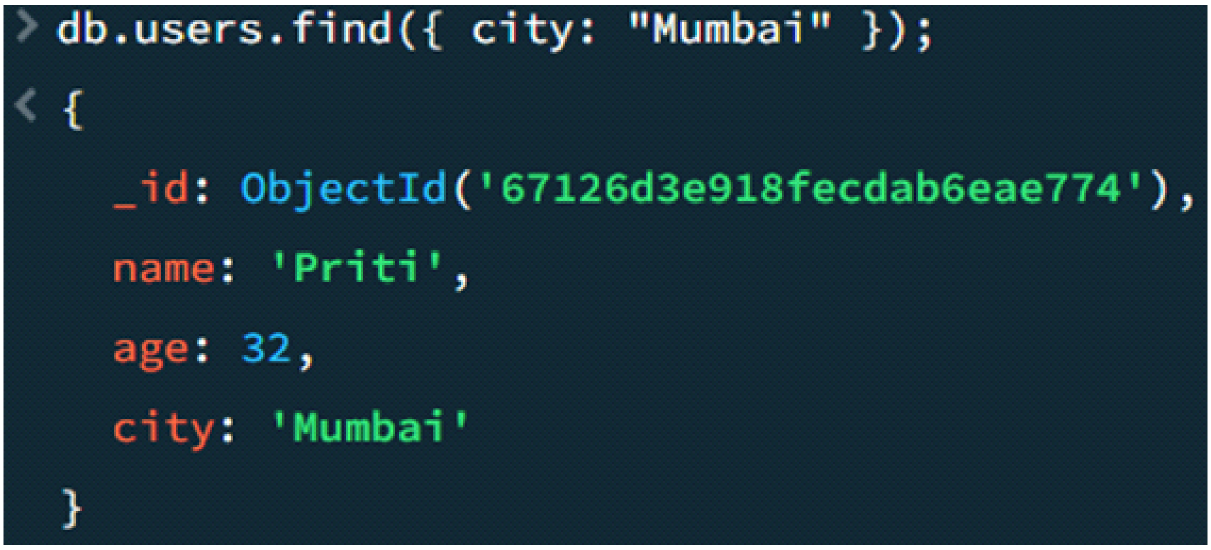
- SQL: Relational database system with structured schema and complex joins. Focuses on tabular data and relationships between tables.
- MongoDB: NoSQL system with flexible schema. Focuses on documents (JSON-like) and horizontal scalability. Uses dot notation for nested objects and arrays.

Implementation:



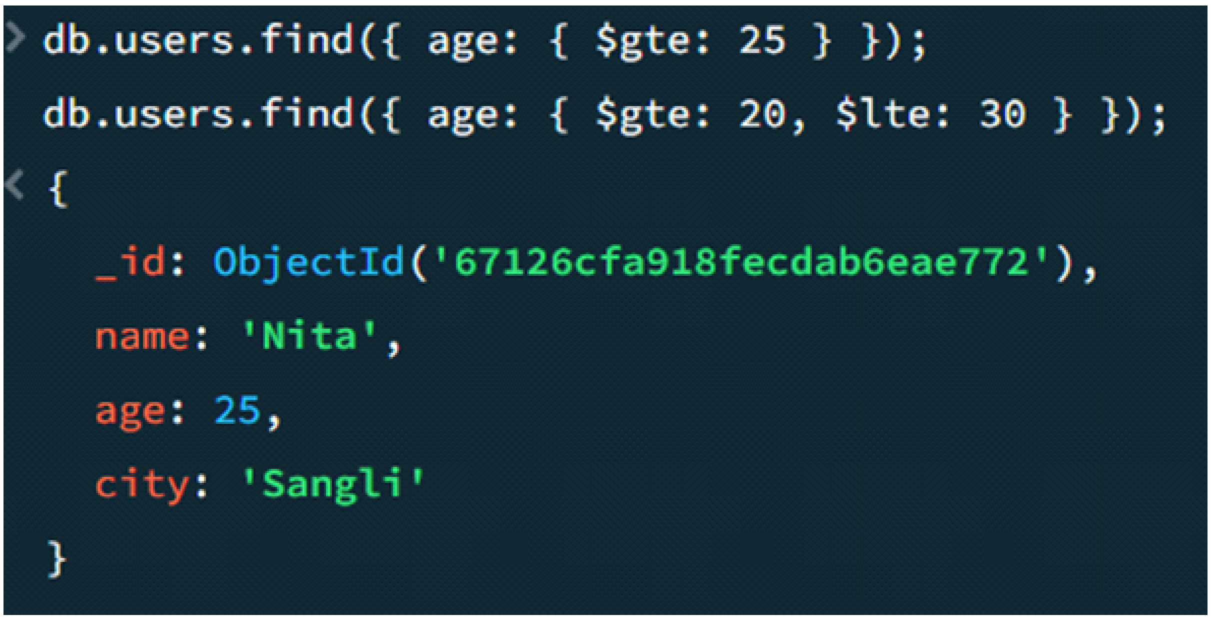
```
>_MONGOSH
> use myDatabase
< switched to db myDatabase
> db["users"].find()
< {
    _id: ObjectId('67126cfa918fecdab6eae772'),
    name: 'Nita',
    age: 25,
    city: 'Sangli'
  }
  {
    _id: ObjectId('67126d3e918fecdab6eae773'),
    name: 'Ram',
    age: 28,
    city: 'Pune'
  }
  {
    _id: ObjectId('67126d3e918fecdab6eae774'),
    name: 'Priti',
    age: 32,
    city: 'Mumbai'
  }
```

1. Basic Filter:



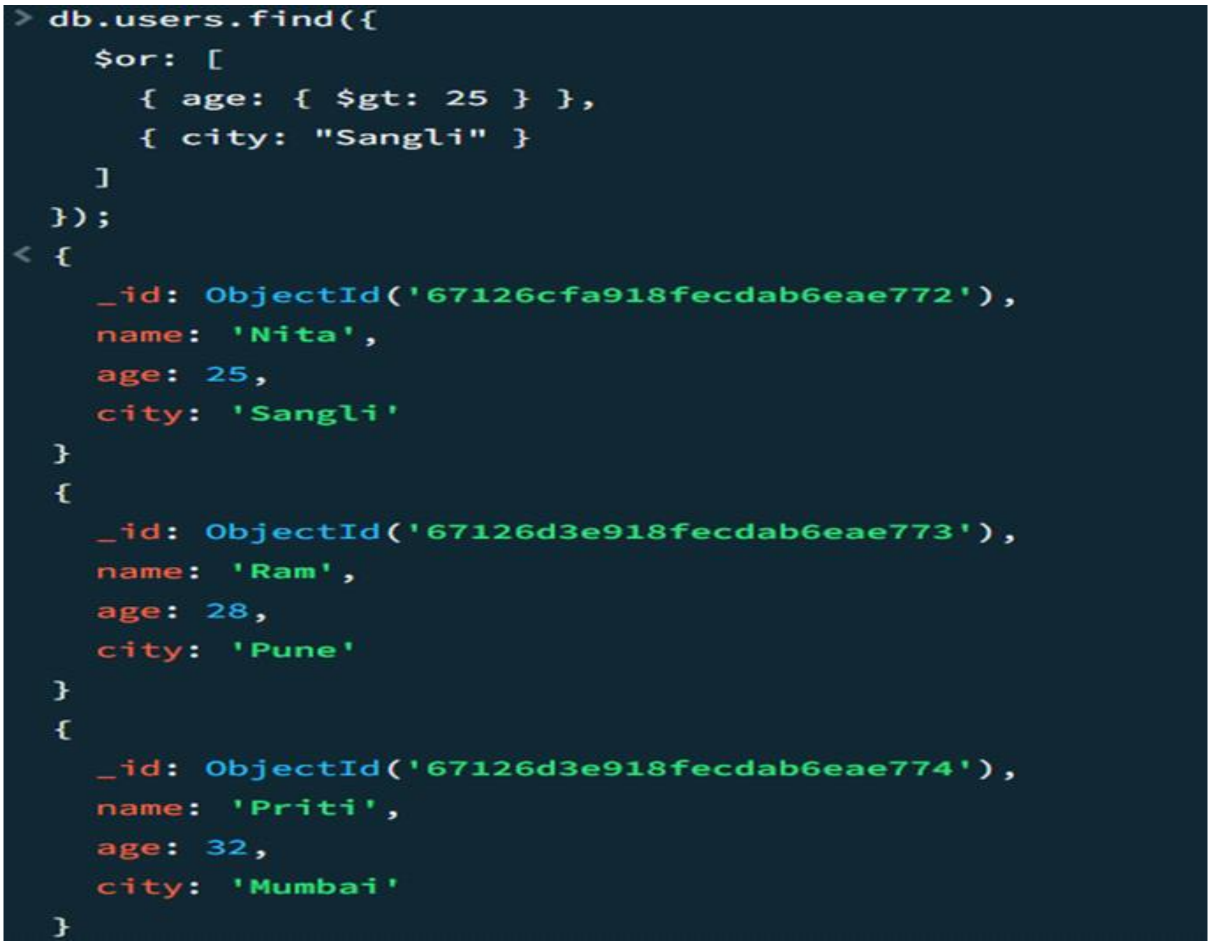
```
> db.users.find({ city: "Mumbai" });
< {
    _id: ObjectId('67126d3e918fecdab6eae774'),
    name: 'Priti',
    age: 32,
    city: 'Mumbai'
  }
```

2. Filter with Comparison Operators:



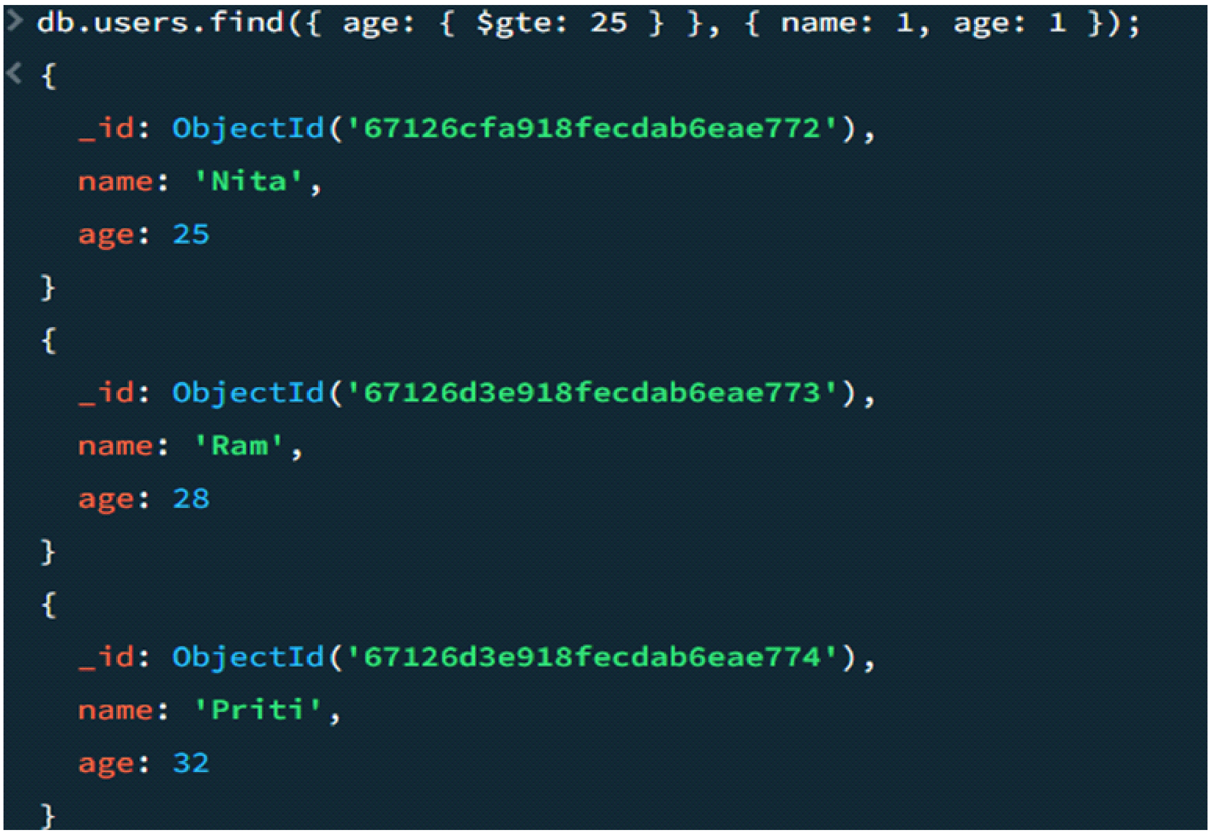
```
> db.users.find({ age: { $gte: 25 } });
  db.users.find({ age: { $gte: 20, $lte: 30 } });
< {
    _id: ObjectId('67126cfa918fecdab6eae772'),
    name: 'Nita',
    age: 25,
    city: 'Sangli'
  }
```

3. Filter with Logical Operators:



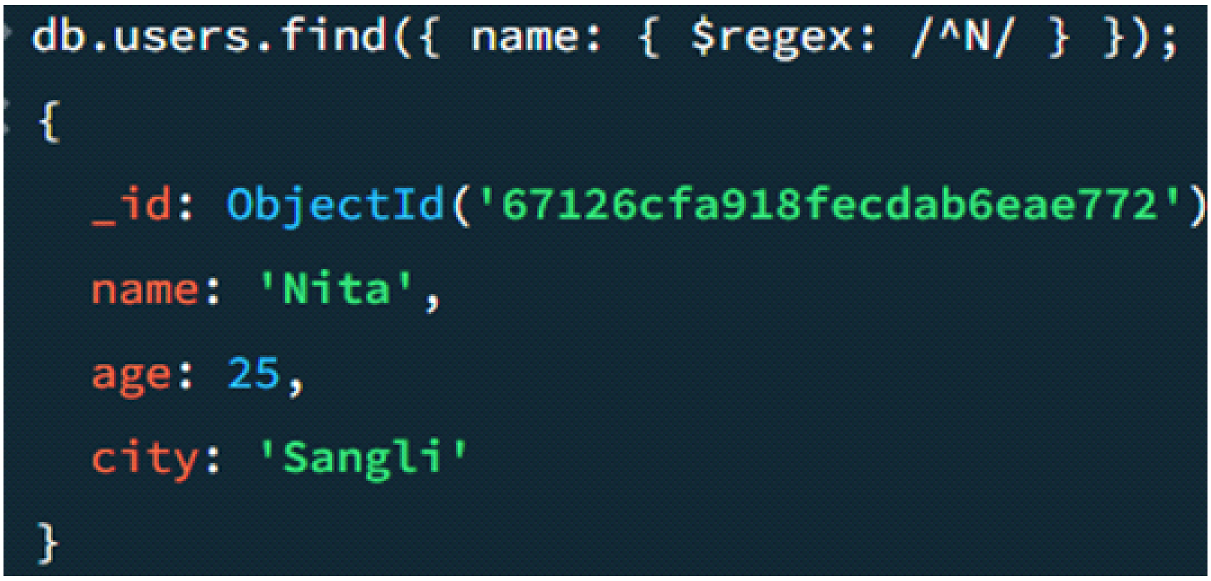
```
> db.users.find({
    $or: [
      { age: { $gt: 25 } },
      { city: "Sangli" }
    ]
  });
< {
    _id: ObjectId('67126cfa918fecdab6eae772'),
    name: 'Nita',
    age: 25,
    city: 'Sangli'
  }
  {
    _id: ObjectId('67126d3e918fecdab6eae773'),
    name: 'Ram',
    age: 28,
    city: 'Pune'
  }
  {
    _id: ObjectId('67126d3e918fecdab6eae774'),
    name: 'Priti',
    age: 32,
    city: 'Mumbai'
  }
```

4. Projection for Filtering Field:



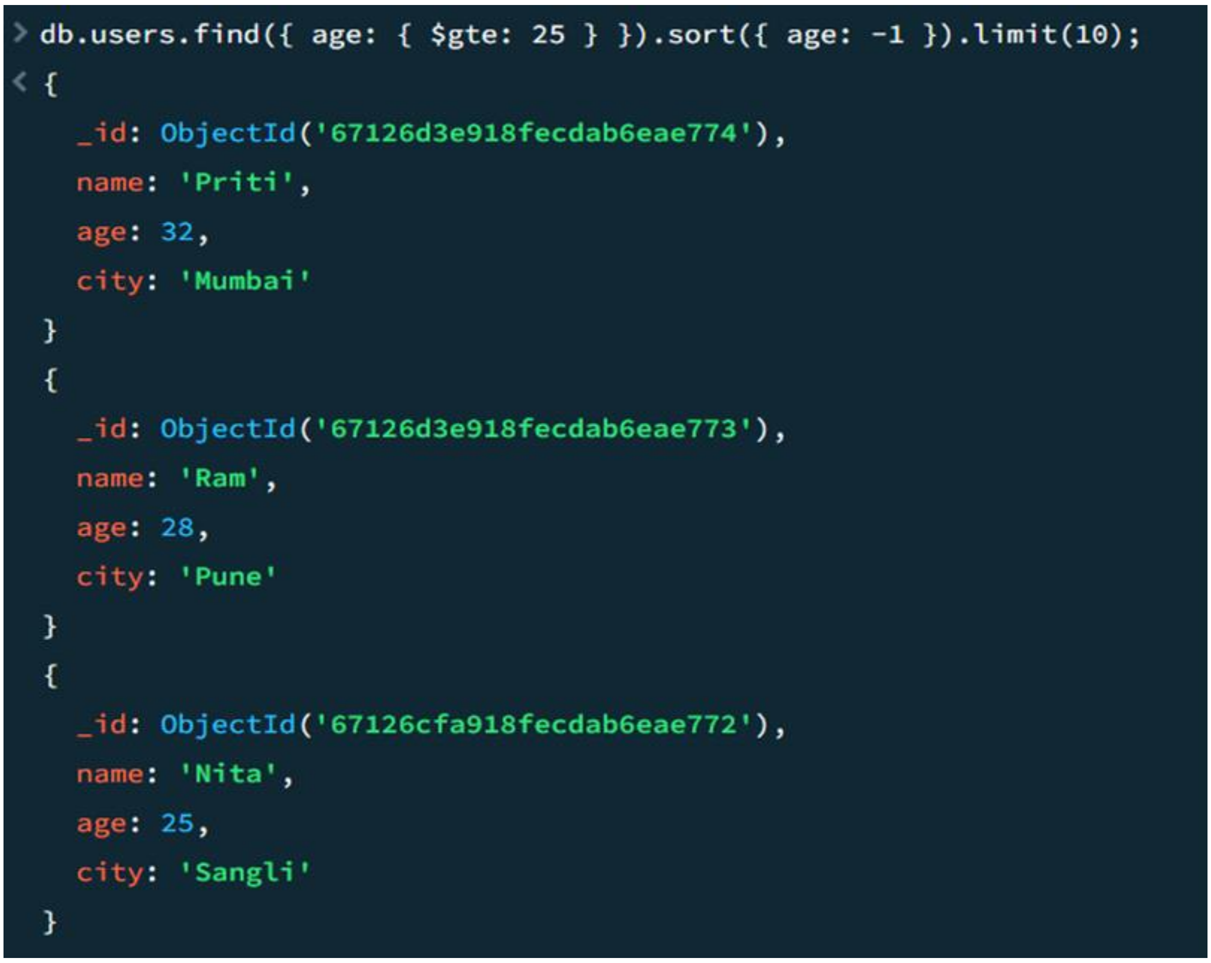
```
> db.users.find({ age: { $gte: 25 } }, { name: 1, age: 1 });
< {
    _id: ObjectId('67126cfa918fecdab6eae772'),
    name: 'Nita',
    age: 25
  }
  {
    _id: ObjectId('67126d3e918fecdab6eae773'),
    name: 'Ram',
    age: 28
  }
  {
    _id: ObjectId('67126d3e918fecdab6eae774'),
    name: 'Priti',
    age: 32
  }
```

5. Regex Filtering for Pattern Matching:



```
db.users.find({ name: { $regex: /^N/ } });
{
   _id: ObjectId('67126cfa918fecdab6eae772')
   name: 'Nita',
   age: 25,
   city: 'Sangli'
}
```

6. Sorting and Pagination:



```
> db.users.find({ age: { $gte: 25 } }).sort({ age: -1 }).limit(10);
< {
    _id: ObjectId('67126d3e918fecdab6eae774'),
    name: 'Priti',
    age: 32,
    city: 'Mumbai'
  }
  {
    _id: ObjectId('67126d3e918fecdab6eae773'),
    name: 'Ram',
    age: 28,
    city: 'Pune'
  }
  {
    _id: ObjectId('67126cfa918fecdab6eae772'),
    name: 'Nita',
    age: 25,
    city: 'Sangli'
  }
```

Conclusion: Efficient data filtering in MongoDB, using the find() method, operators, and indexing, improves performance and accuracy, especially for large datasets. Indexing, projection, and pagination optimize queries for scalability, making MongoDB well-suited for real-world applications.

Atharva Shende (22210036)
TY IT (T1 Batch)
Experiment No. 12

Title: Working with Command Prompts and Creating a Database Table in MariaDB.

Aim: To understand and execute basic commands in MariaDB using the command prompt, and to create a database table with appropriate fields and constraints.

Objectives:

- To explore the MariaDB environment through the command-line interface.
- To learn how to create a database and its associated tables.
- To understand the importance of field data types and constraints in a relational database.
- To perform basic database operations like creating, inserting, and querying data.

Theory: MariaDB is an open-source relational database management system (RDBMS) that serves as a drop-in replacement for MySQL. It supports SQL (Structured Query Language) for querying and managing data. In database management, tables are created with specific fields (columns) and constraints to ensure data integrity. To begin working with MariaDB, follow these steps:

1. **Installation:** Install MariaDB by downloading it from the official site or using the package manager on your operating system. For example, on a Linux-based system, you can install it using:
   sudo apt-get install mariadb-server

2. **Starting MariaDB Service:** After installation, start the MariaDB service using the following command:
   sudo service mariadb start

3. **Accessing MariaDB via Command Prompt:** Once MariaDB is installed and running, you can interact with it using the command prompt:
   - **Log into MariaDB:** Use the following command to log into MariaDB as the root user:
     sudo mariadb -u root -p
   - **Create a New User:** If you want to create a new user, use:
     CREATE USER 'username'@'localhost' IDENTIFIED BY 'password';
   - **Grant Privileges:** Grant the necessary privileges to the new user:
     GRANT ALL PRIVILEGES ON *.* TO 'username'@'localhost' WITH GRANT OPTION;

A table in MariaDB is composed of rows and columns, where each row represents a record and each column represents a field with a specific data type (such as INT, VARCHAR, etc.). Common SQL commands involved in creating and managing tables include:

Basic Database Commands:

1. **Show all databases**: SHOW DATABASES;
2. **Create a new database**: CREATE DATABASE [database_name];
3. **Use a specific database**: USE [database_name];
4. **Drop (delete) a database**:
   DROP DATABASE [database_name];

Table Management Commands:

1. **Show all tables in the current database**: SHOW TABLES;
2. **Create a new table**: CREATE TABLE [table_name] (column1 datatype, column2 datatype, ...);
3. **Describe the table structure (columns**): DESCRIBE [table_name];
4. **Insert data into a table**: INSERT INTO [table_name] (column1, column2, ...) VALUES (value1, value2, ...);
5. **Select (retrieve) data from a table**: SELECT * FROM [table_name];
6. **Update data in a table**: UPDATE [table_name] SET column1 = value1, column2 = value2, ... WHERE condition;
7. **Delete data from a table**: DELETE FROM [table_name] WHERE condition;
8. **Drop (delete) a table**: DROP TABLE [table_name];

User Management Commands:

1. **Create a new user**: CREATE USER '[username]'@'localhost' IDENTIFIED BY '[password]';
2. **Grant privileges to a user**: GRANT ALL PRIVILEGES ON [database_name].* TO '[username]'@'localhost';
3. **Revoke privileges from a user**: REVOKE ALL PRIVILEGES ON [database_name].* FROM '[username]'@'localhost';
4. **Show all users**: SELECT user, host FROM mysql.user;
5. **Drop (delete) a user**: DROP USER '[username]'@'localhost';

Data Querying Commands:

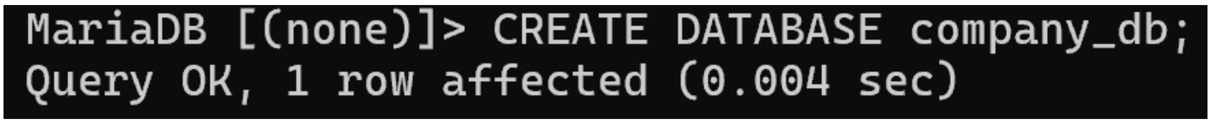
1. **Select specific columns**: SELECT column1, column2 FROM [table_name];
2. **Order results**: SELECT * FROM [table_name] ORDER BY column_name ASC|DESC;
3. **Filtering results (WHERE clause):** SELECT * FROM [table_name] WHERE condition;
4. **Count rows:** SELECT COUNT(*) FROM [table_name];
5. **Join two tables**: SELECT columns FROM [table1] INNER JOIN [table2] ON [table1.column] = [table2.column];

Indexes and Constraints:

1. **Create an index**: CREATE INDEX [index_name] ON [table_name](column_name);
2. **Drop an index**: DROP INDEX [index_name] ON [table_name];
3. **Add a primary key constraint**: ALTER TABLE [table_name] ADD PRIMARY KEY (column_name);
4. **Add a foreign key constraint**: ALTER TABLE [table_name]
   ADD CONSTRAINT [constraint_name]
   FOREIGN KEY (column_name)
   REFERENCES [other_table_name](column_name);

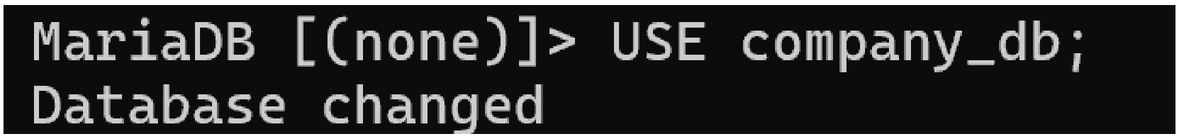
Implementation:

Create database:



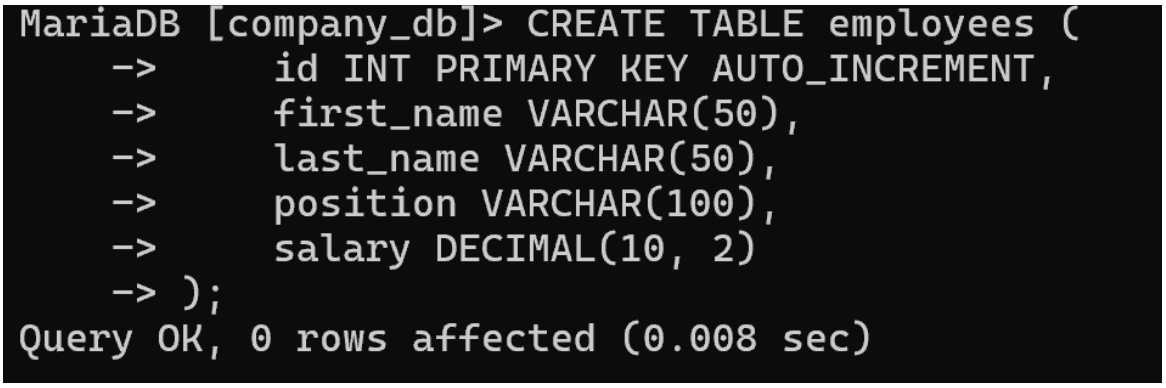
```
MariaDB [(none)]> CREATE DATABASE company_db;
Query OK, 1 row affected (0.004 sec)
```

Use database:



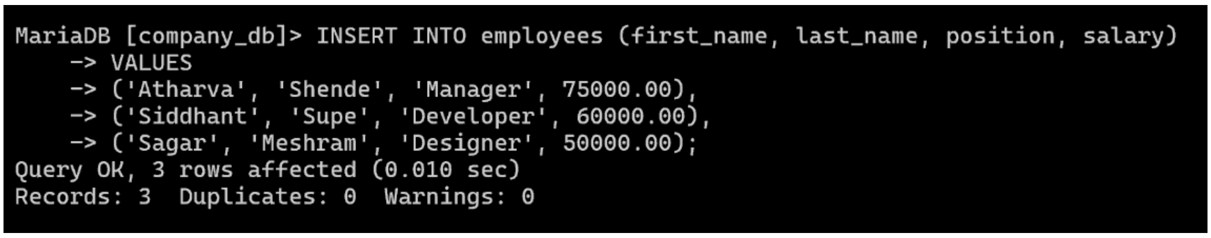
```
MariaDB [(none)]> USE company_db;
Database changed
```

Create Table:



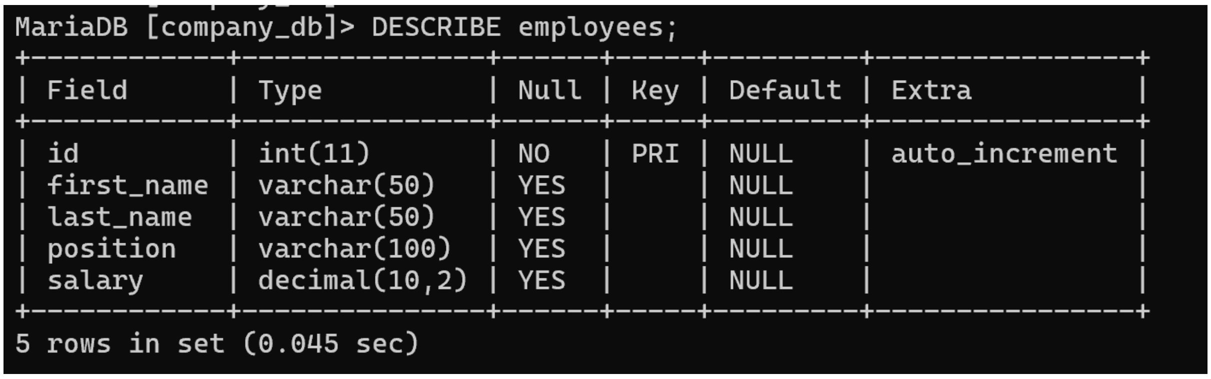
```
MariaDB [company_db]> CREATE TABLE employees (
    ->     id INT PRIMARY KEY AUTO_INCREMENT,
    ->     first_name VARCHAR(50),
    ->     last_name VARCHAR(50),
    ->     position VARCHAR(100),
    ->     salary DECIMAL(10, 2)
    -> );
Query OK, 0 rows affected (0.008 sec)
```

Insert Data:



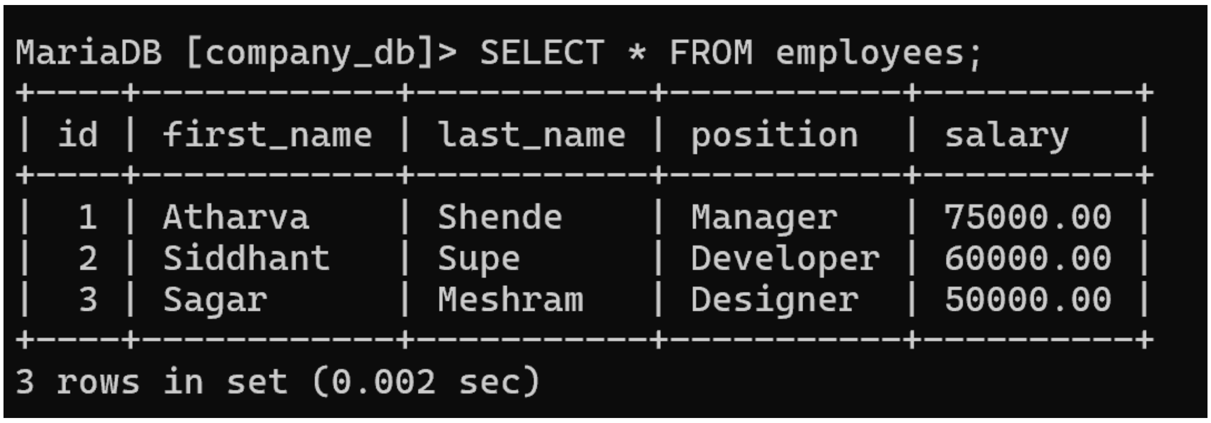
```
MariaDB [company_db]> INSERT INTO employees (first_name, last_name, position, salary)
    -> VALUES
    -> ('Atharva', 'Shende', 'Manager', 75000.00),
    -> ('Siddhant', 'Supe', 'Developer', 60000.00),
    -> ('Sagar', 'Meshram', 'Designer', 50000.00);
Query OK, 3 rows affected (0.010 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

Describe table:



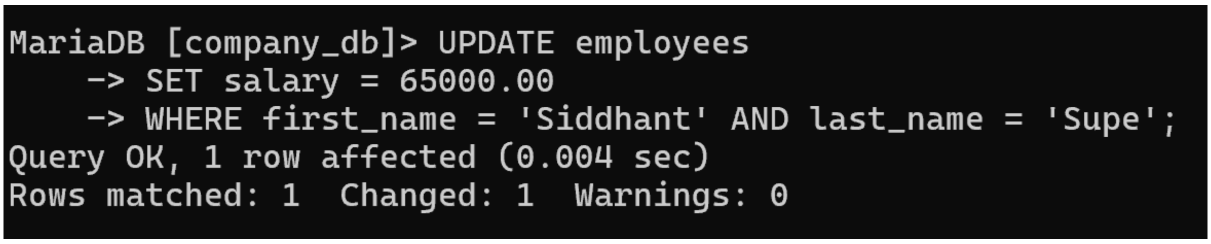
```
MariaDB [company_db]> DESCRIBE employees;
+------------+---------------+------+-----+---------+----------------+
| Field      | Type          | Null | Key | Default | Extra          |
+------------+---------------+------+-----+---------+----------------+
| id         | int(11)       | NO   | PRI | NULL    | auto_increment |
| first_name | varchar(50)   | YES  |     | NULL    |                |
| last_name  | varchar(50)   | YES  |     | NULL    |                |
| position   | varchar(100)  | YES  |     | NULL    |                |
| salary     | decimal(10,2) | YES  |     | NULL    |                |
+------------+---------------+------+-----+---------+----------------+
5 rows in set (0.045 sec)
```

Retrieve data:



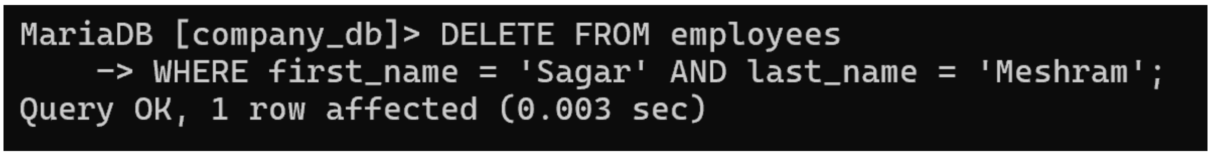
```
MariaDB [company_db]> SELECT * FROM employees;
+----+------------+-----------+-----------+----------+
| id | first_name | last_name | position  | salary   |
+----+------------+-----------+-----------+----------+
|  1 | Atharva    | Shende    | Manager   | 75000.00 |
|  2 | Siddhant   | Supe      | Developer | 60000.00 |
|  3 | Sagar      | Meshram   | Designer  | 50000.00 |
+----+------------+-----------+-----------+----------+
3 rows in set (0.002 sec)
```

Update data:



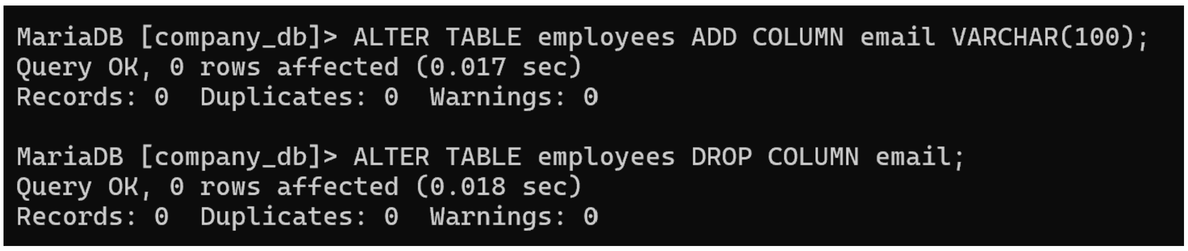
```
MariaDB [company_db]> UPDATE employees
    -> SET salary = 65000.00
    -> WHERE first_name = 'Siddhant' AND last_name = 'Supe';
Query OK, 1 row affected (0.004 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Delete data:



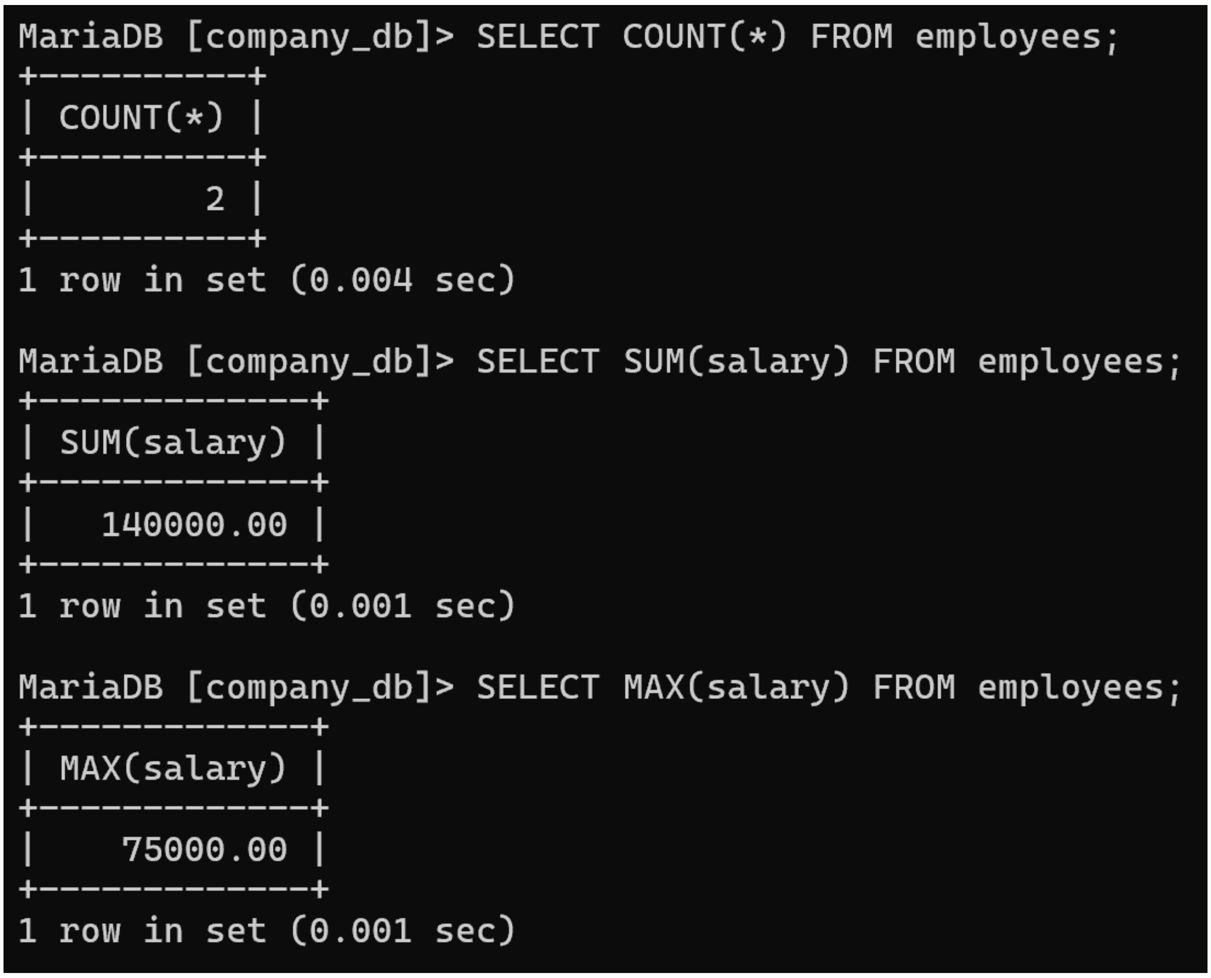
```
MariaDB [company_db]> DELETE FROM employees
    -> WHERE first_name = 'Sagar' AND last_name = 'Meshram';
Query OK, 1 row affected (0.003 sec)
```

Alter table:



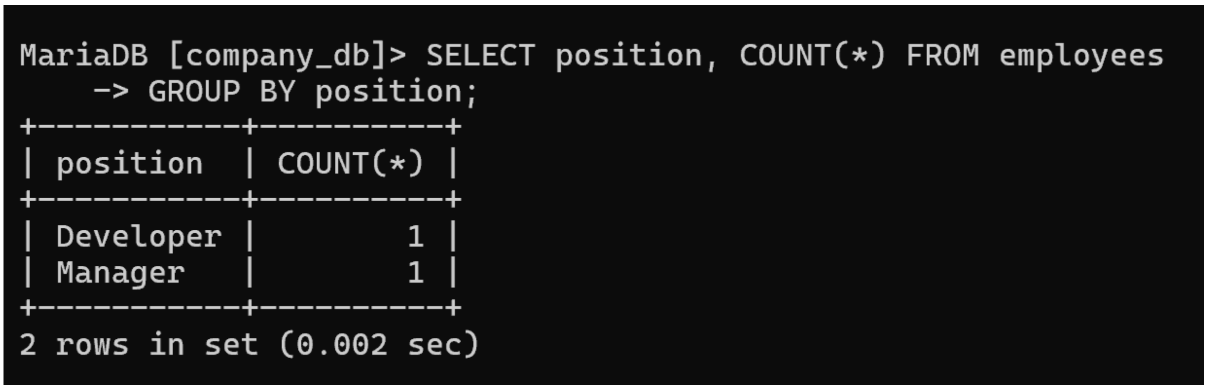
```
MariaDB [company_db]> ALTER TABLE employees ADD COLUMN email VARCHAR(100);
Query OK, 0 rows affected (0.017 sec)
Records: 0  Duplicates: 0  Warnings: 0

MariaDB [company_db]> ALTER TABLE employees DROP COLUMN email;
Query OK, 0 rows affected (0.018 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Aggregate Functions:



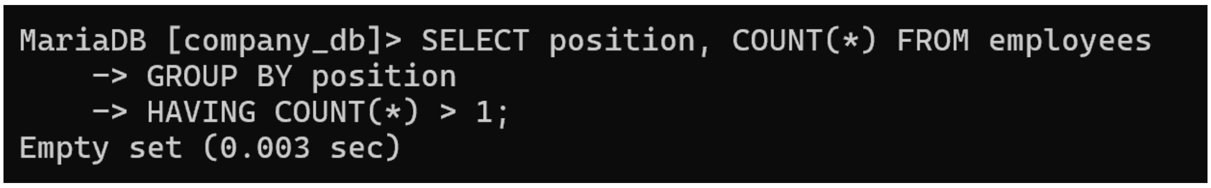
```
MariaDB [company_db]> SELECT COUNT(*) FROM employees;
+----------+
| COUNT(*) |
+----------+
|        2 |
+----------+
1 row in set (0.004 sec)

MariaDB [company_db]> SELECT SUM(salary) FROM employees;
+-------------+
| SUM(salary) |
+-------------+
|   140000.00 |
+-------------+
1 row in set (0.001 sec)

MariaDB [company_db]> SELECT MAX(salary) FROM employees;
+-------------+
| MAX(salary) |
+-------------+
|    75000.00 |
+-------------+
1 row in set (0.001 sec)
```

Grouping:



```
MariaDB [company_db]> SELECT position, COUNT(*) FROM employees
    -> GROUP BY position;
+-----------+----------+
| position  | COUNT(*) |
+-----------+----------+
| Developer |        1 |
| Manager   |        1 |
+-----------+----------+
2 rows in set (0.002 sec)
```

Filtering:



```
MariaDB [company_db]> SELECT position, COUNT(*) FROM employees
    -> GROUP BY position
    -> HAVING COUNT(*) > 1;
Empty set (0.003 sec)
```

Conclusion: Through this experiment, we successfully worked with the command prompt to interact with MariaDB. We created a database and table, inserted data, and performed queries. This hands-on approach deepened our understanding of basic database operations and SQL commands in a MariaDB environment.

Atharva Shende (22210036)
TY IT (T1 Batch)
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 the 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:

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

SQL commands associated with these operations in MariaDB:

1. **Create (Insert Data)-**
   ```
   INSERT INTO table_name (column1, column2, ...) VALUES (value1, value2, ...);
   ```

2. **Read (Select Data)-**
   ```
   SELECT column1, column2, ... FROM table_name WHERE condition;
   ```

3. **Update (Modify Data)-**
   ```
   UPDATE table_name SET column1 = value1, column2 = value2 WHERE condition;
   ```

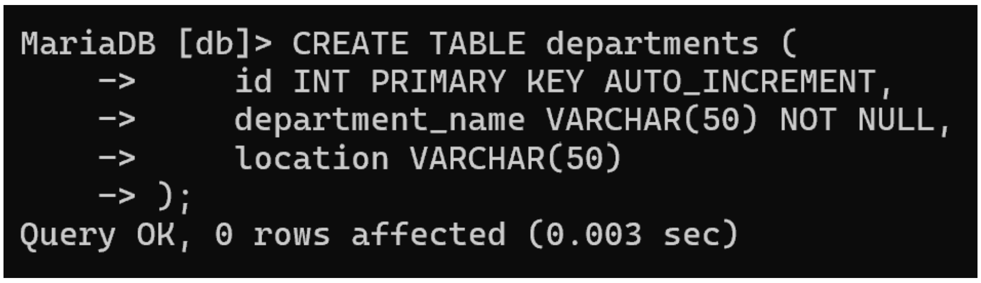
4. **Delete (Remove Data)-**
   ```
   DELETE FROM table_name WHERE condition;
   ```

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.

Implementation:

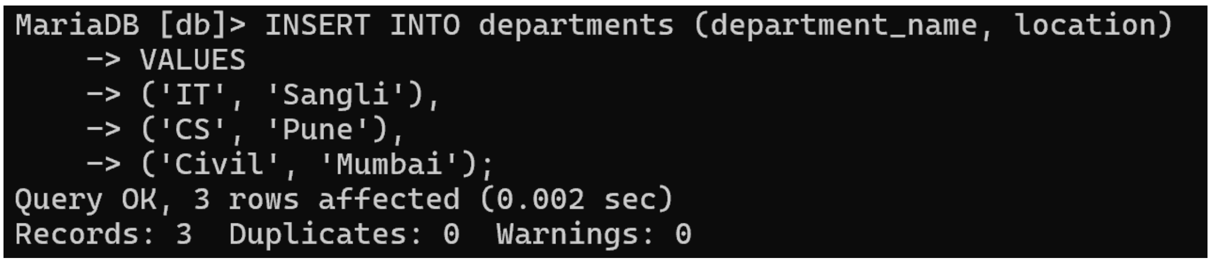
1. Create Operation (Insert Data):

Create the Table:



```
MariaDB [db]> CREATE TABLE departments (
    ->     id INT PRIMARY KEY AUTO_INCREMENT,
    ->     department_name VARCHAR(50) NOT NULL,
    ->     location VARCHAR(50)
    -> );
Query OK, 0 rows affected (0.003 sec)
```

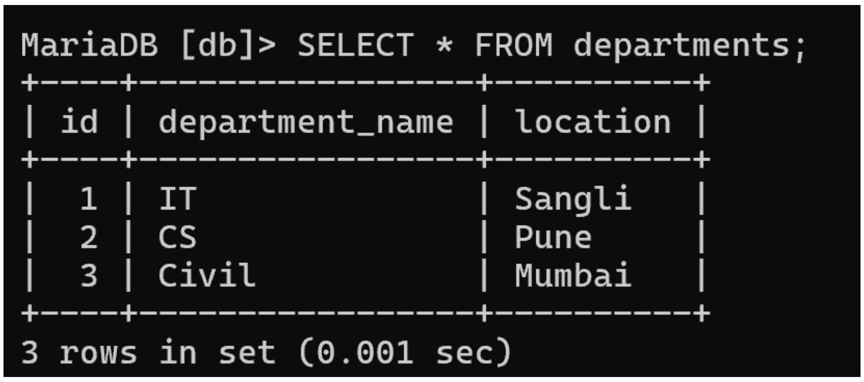
Insert Records:



```
MariaDB [db]> INSERT INTO departments (department_name, location)
    -> VALUES
    -> ('IT', 'Sangli'),
    -> ('CS', 'Pune'),
    -> ('Civil', 'Mumbai');
Query OK, 3 rows affected (0.002 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

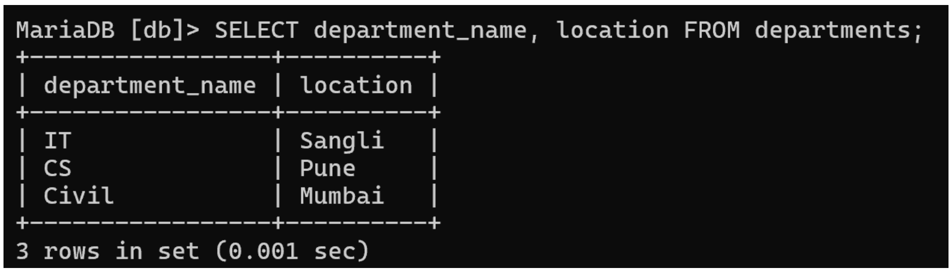
2. Read Operation (Retrieve Data):

Select all records:



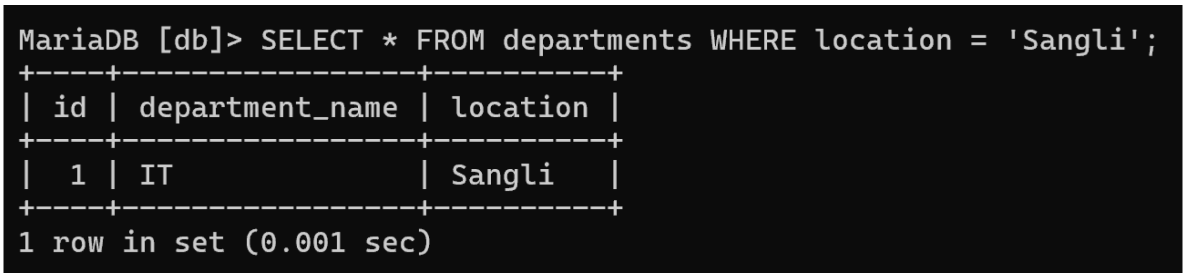
```
MariaDB [db]> SELECT * FROM departments;
+----+-----------------+----------+
| id | department_name | location |
+----+-----------------+----------+
|  1 | IT              | Sangli   |
|  2 | CS              | Pune     |
|  3 | Civil           | Mumbai   |
+----+-----------------+----------+
3 rows in set (0.001 sec)
```

Select specific columns:



```
MariaDB [db]> SELECT department_name, location FROM departments;
+-----------------+----------+
| department_name | location |
+-----------------+----------+
| IT              | Sangli   |
| CS              | Pune     |
| Civil           | Mumbai   |
+-----------------+----------+
3 rows in set (0.001 sec)
```

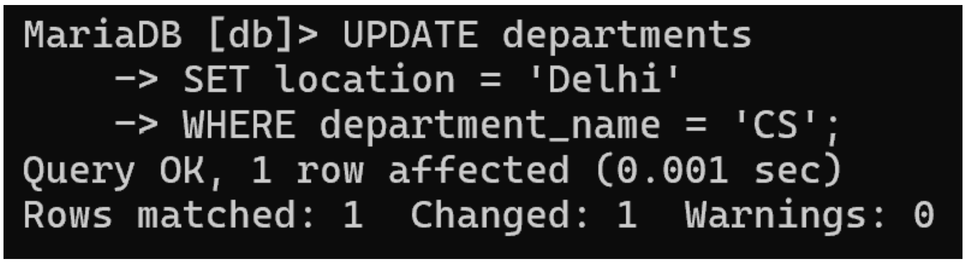
Filter:



```
MariaDB [db]> SELECT * FROM departments WHERE location = 'Sangli';
+----+-----------------+----------+
| id | department_name | location |
+----+-----------------+----------+
|  1 | IT              | Sangli   |
+----+-----------------+----------+
1 row in set (0.001 sec)
```

3. Update Operation (Modify Data):

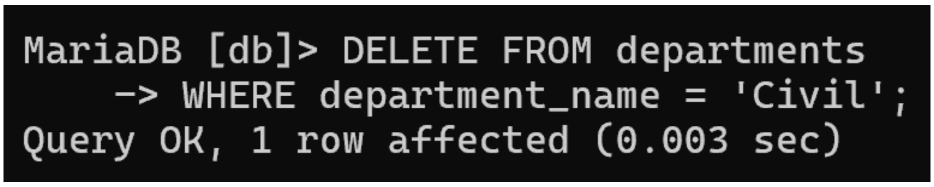
Update Records:



```
MariaDB [db]> UPDATE departments
    -> SET location = 'Delhi'
    -> WHERE department_name = 'CS';
Query OK, 1 row affected (0.001 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

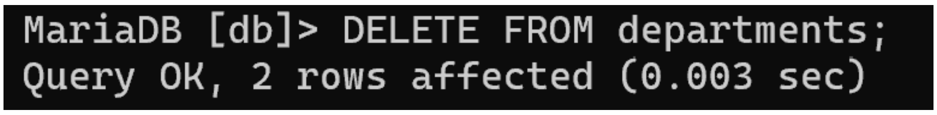
4. Delete Operation (Remove Data):

Delete a specific record:



```
MariaDB [db]> DELETE FROM departments
    -> WHERE department_name = 'Civil';
Query OK, 1 row affected (0.003 sec)
```

Delete all records:



```
MariaDB [db]> DELETE FROM departments;
Query OK, 2 rows affected (0.003 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.

Divya Tarkase (22610014)
TY IT (T1 Batch)
Experiment No. 14

Title: Implementation of JDBC for database connectivity

Aim: To understand and implement Java Database Connectivity (JDBC) to interact with a MySQL database, including inserting, retrieving, and displaying records from a table.

Objectives:

- To set up a MySQL database and create a table for user data.
- To establish a JDBC connection to the MySQL database from a Java application.
- To insert records into the users table.
- To retrieve and display records from the users table using JDBC.

Theory: JDBC stands for Java Database Connectivity, a Java API that serves as an industry standard to facilitate connections between Java applications and databases. It provides classes and methods for managing connection details, executing SQL statements, and handling result sets and metadata. Initially designed as a client-side API, JDBC connects Java applications with data sources like SQL databases.

Understanding JDBC in Java: Before diving into establishing a JDBC connection with databases like SQL, it is crucial to grasp the various concepts, classes, tools, and interfaces involved. Key components include JDBC drivers, relational databases (like SQL), and their integration with JDBC.

JDBC components: JDBC includes several interfaces and classes that form its core, such as:

1. **Driver:** Controls communication with the database server and retrieves information about driver objects.
2. **DriverManager:** Manages a list of database drivers.
3. **Connection:** Represents a session between a Java application and a database, offering methods to establish connections.
4. **Statement:** Used to execute static SQL queries.
5. **ResultSet:** Provides methods for accessing data retrieved from the database, row by row.

Steps to Establish a JDBC Connection: To establish a connection using JDBC, follow these steps:

1. **Import the Required Packages:** First, include java.sql.* in your program. This allows access to the necessary JDBC interfaces and classes.
2. **Load the Database Driver:** Before using the driver, it must be loaded or registered. There are two ways to do this:
    1. **Using Class.forName():** This method loads the driver class into memory during runtime, without the need to create new objects.
       Example: Class.forName("oracle.jdbc.driver.OracleDriver");

    2. **Using DriverManager.registerDriver():** This method utilizes the DriverManager class, which is built into Java. Here, the driver class is registered during compile time. Example: DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

3. **Register the Driver with DriverManager:** After loading the driver, it must be registered with the DriverManager to be used.
4. **Create a Connection Object**: Establish a connection by using the getConnection() method from the DriverManager class:
   Connection con = DriverManager.getConnection(url, user, password);
5. **Create a Statement:** Once connected, create a Statement object to interact with the database. You can use interfaces like CallableStatement, JDBCStatement, or PreparedStatement to execute SQL commands and fetch data:
   Statement st = con.createStatement();
6. **Execute SQL Queries:** You can run various SQL queries, such as those for retrieving or modifying data. Use the executeQuery() method to fetch data, which returns a ResultSet object for reading records. Use executeUpdate() for updates or inserts: ResultSet rs = st.executeQuery("SELECT * FROM employees");
   int updateCount = st.executeUpdate("INSERT INTO employees VALUES ('John', 101)"
7. **Close the Connection:** Once your operations are complete, ensure you close the connection. This will automatically close any Statement and ResultSet objects: con.close();

Implementation:

SQL Commands:

```
CREATE DATABASE mydb;
USE mydb;
CREATE TABLE users (
        id INT AUTO_INCREMENT PRIMARY KEY,
        name VARCHAR(100) NOT NULL,
        email VARCHAR(100) NOT NULL UNIQUE,
        password VARCHAR(255) NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

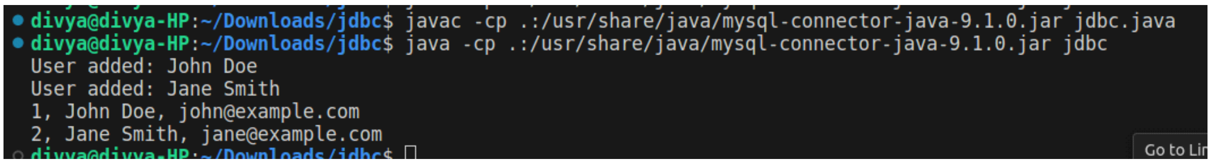
Java Code:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;

public class jdbc {
  public static void main(String[] args) {
     // MySQL database URL
     String jdbcURL = "jdbc:mysql://localhost:3306/mydb?serverTimezone=UTC";
```

```
        String username = "root";
        String password = "newpassword";


        try {
            // Step 1: Load the MySQL JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Step 2: Establish a connection
            Connection connection = DriverManager.getConnection(jdbcURL, username,
password);

            // Insert data into the users table
            insertUser(connection, "John Doe", "john@example.com", "password123");
            insertUser(connection, "Jane Smith", "jane@example.com", "securePassword");

            // Step 3: Create a Statement object
            Statement statement = connection.createStatement();

            // Step 4: Execute a SQL query to retrieve data
            String sql = "SELECT * FROM users";
            ResultSet resultSet = statement.executeQuery(sql);

            // Step 5: Process the result set
            while (resultSet.next()) {
                int id = resultSet.getInt("id");
                String name = resultSet.getString("name");
                String email = resultSet.getString("email");

                System.out.println(id + ", " + name + ", " + email);
            }

            // Step 6: Close resources
            resultSet.close();
            statement.close();
            connection.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Method to insert a user into the users table
    private static void insertUser(Connection connection, String name, String email, String
password) {
        String sql = "INSERT INTO users (name, email, password) VALUES (?, ?, ?)";
        try (PreparedStatement preparedStatement = connection.prepareStatement(sql)) {
            preparedStatement.setString(1, name);
            preparedStatement.setString(2, email);
            preparedStatement.setString(3, password); // Consider hashing the password before
inserting
```

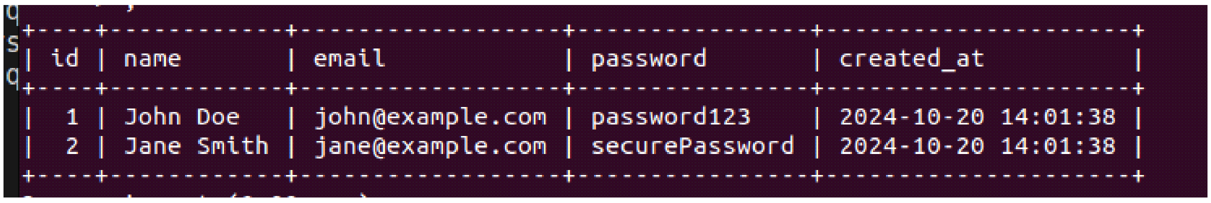
```
            preparedStatement.executeUpdate();
            System.out.println("User added: " + name);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

1. Console:



```
● divya@divya-HP:~/Downloads/jdbc$ javac -cp .:/usr/share/java/mysql-connector-java-9.1.0.jar jdbc.java
● divya@divya-HP:~/Downloads/jdbc$ java -cp .:/usr/share/java/mysql-connector-java-9.1.0.jar jdbc
  User added: John Doe
  User added: Jane Smith
  1, John Doe, john@example.com
  2, Jane Smith, jane@example.com
○ divya@divya-HP:~/Downloads/jdbc$ 
```

2. Database:



```
+----+------------+------------------+----------------+---------------------+
| id | name       | email            | password       | created_at          |
+----+------------+------------------+----------------+---------------------+
|  1 | John Doe   | john@example.com | password123    | 2024-10-20 14:01:38 |
|  2 | Jane Smith | jane@example.com | securePassword | 2024-10-20 14:01:38 |
+----+------------+------------------+----------------+---------------------+
```

Conclusion: This experiment demonstrated using JDBC to interact with a MySQL database, successfully connecting, inserting records, and retrieving data from the users table. This foundational JDBC knowledge is essential for building Java applications with efficient database interaction.

(22610004, 22610008, 23620010)
TY IT (T1 Batch)
Case Study - Foodie-Fi

Introduction:

Subscription based businesses are super popular and Danny realised that there was a large gap in the market - he wanted to create a new streaming service that only had food related content - something like Netflix but with only cooking shows!
Danny finds a few smart friends to launch his new startup Foodie-Fi in 2020 and started selling monthly and annual subscriptions, giving their customers unlimited on-demand access to exclusive food videos from around the world!
Danny created Foodie-Fi with a data driven mindset and wanted to ensure all future investment decisions and new features were decided using data. This case study focuses on using subscription style digital data to answer important business questions.

Table 1: plans:

Customers can choose which plans to join Foodie-Fi when they first sign up.
Basic plan customers have limited access and can only stream their videos and is only available monthly at $9.90
Pro plan customers have no watch time limits and are able to download videos for offline viewing. Pro plans start at $19.90 a month or $199 for an annual subscription.
Customers can sign up to an initial 7 day free trial will automatically continue with the pro monthly subscription plan unless they cancel, downgrade to basic or upgrade to an annual pro plan at any point during the trial.
When customers cancel their Foodie-Fi service - they will have a churn plan record with a null price but their plan will continue until the end of the billing period.

plan_id	plan_name	price
0	trial	0
1	basic monthly	9.90
2	pro monthly	19.90
3	pro annual	199
4	churn	null

Table 2: subscriptions:

Customer subscriptions show the exact date where their specific plan_id starts.
If customers downgrade from a pro plan or cancel their subscription - the higher plan will remain in place until the period is over - the start_date in the subscriptions table will reflect the date that the actual plan changes.
When customers upgrade their account from a basic plan to a pro or annual pro plan - the higher plan will take effect straightaway.
When customers churn - they will keep their access until the end of their current billing period but the start_date will be technically the day they decided to cancel their service.

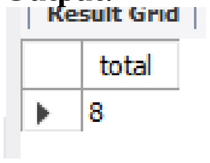
customer_id	plan_id	start_date
1	0	2020-08-01
1	1	2020-08-08
2	0	2020-09-20

2	3	2020-09-27
11	0	2020-11-19
11	4	2020-11-26
13	0	2020-12-15
13	1	2020-12-22
13	2	2021-03-29
15	0	2020-03-17
15	2	2020-03-24
15	4	2020-04-29
16	0	2020-05-31
16	1	2020-06-07
16	3	2020-10-21
18	0	2020-07-06
18	2	2020-07-13
19	0	2020-06-22
19	2	2020-06-29
19	3	2020-08-29

Data Analysis Questions:

- **How many customers has Foodie-Fi ever had?**

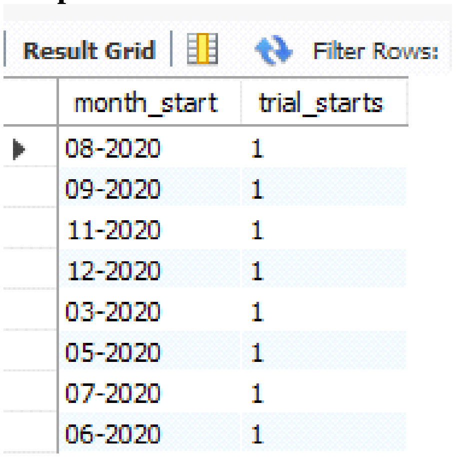
Query:
SELECT COUNT(DISTINCT customer_id) AS total FROM subscriptions;

Output:

```
Result Grid
total
8
```

- **What is the monthly distribution of trial plan start_date values for our dataset - use the start of the month as the group by value.**

Query:
SELECT DATE_FORMAT(start_date, '%m-%Y') AS month_start, COUNT(*) AS trial_starts
FROM subscriptions
WHERE plan_id = 0
GROUP BY month_start;

Output:



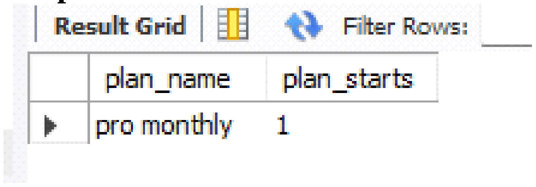
```
month_start  trial_starts
08-2020      1
09-2020      1
11-2020      1
12-2020      1
03-2020      1
05-2020      1
07-2020      1
06-2020      1
```

- **What plan start_date values occur after the year 2020 for our dataset? Show the breakdown by count of events for each plan_name.**

Query-

```
SELECT p.plan_name, COUNT(*) AS plan_starts
FROM subscriptions s
JOIN plans p ON s.plan_id = p.plan_id
WHERE YEAR(start_date) > 2020
GROUP BY p.plan_name;
```

Output-



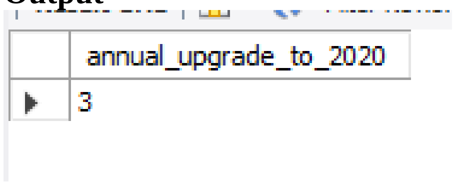
```
plan_name    plan_starts
pro monthly  1
```

- **How many customers have upgraded to an annual plan in 2020?**

Query-

```
SELECT COUNT(DISTINCT customer_id) AS annual_upgrade_to_2020
FROM subscriptions
WHERE plan_id = 3
AND YEAR(start_date) = 2020;
```

Output-



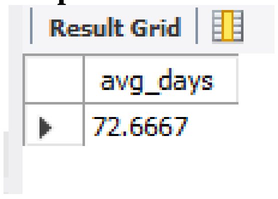
```
annual_upgrade_to_2020
3
```

- **How many days on average does it take for a customer to an annual plan from the day they join Foodie-Fi?**

Query-

```
SELECT AVG(DATEDIFF(s2.start_date, s1.start_date)) AS avg_days
FROM subscriptions s1
JOIN subscriptions s2 ON s1.customer_id = s2.customer_id
WHERE s1.plan_id = 0
AND s2.plan_id = 3;
```

Output-



```
avg_days
72.6667
```