



DEPARTMENT OF INFORMATION TECHNOLOGY
WALCHAND COLLEGE OF ENGINEERING, SANGLI
(An Autonomous Institute)

2024-2025

Database Engineering Lab
Code: 6IT351

Journal
T4 - Batch

CERTIFICATE



This is to certify that the report entitled “Database Engineering Lab, Code: 6IT351” submitted by T4 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. Mohini Mane
(Course Teacher)

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

INDEX

Sr. No.	Title	PRN	Experiment Performed on	Experiment Submitted on	Signature
1.	To construct ER Model for given database.	22610012	01/08/2024	08/08/2024	
2.	Implement relation algebra such as select project intersect minus and cartesian operations on a given database.	22610013	08/08/2024	22/08/2024	
3.	Study and implementation of DDL and DML commands of SQL with suitable examples.	22610017	22/08/2024	29/08/2024	
4.	Create a database as customer_information and apply DQL and TCL SQL commands.	22610026	29/08/2024	05/08/2024	
5.	Perform various types of functions SQL.	22610029	05/09/2024	12/09/2024	
6.	Study and Implementation of various types of constraints in SQL.	22610039	12/09/2024	19/09/2024	
7.	Implementation of different types of Joins.	22610040	12/09/2024	19/09/2024	
8.	Study and Implementation of various types of clauses and Indexing.	22610041	19/09/2024	03/10/2024	
9.	Study and Implementation of Triggers and View.	22610042	03/10/2024	10/10/2024	
10.	Implementation of procedure in SQL.	22610043	10/10/2024	17/10/2024	
11.	Study and implement the CRUD operations on MongoDB Database.	22610044	10/10/2024	17/10/2024	
12.	Filtering for data efficiently on MongoDB Database.	22610045	17/10/2024	24/10/2024	
13.	Working with command prompts and create database table in MariaDB.	22610083	17/10/2024	24/10/2024	
14.	To perform CRUD operation on MariaDB.	22610089	24/10/2024	25/10/2024	
15.	Implementation of JDBC and ODBC connectivity.	22610090	24/10/2024	25/10/2024	

Rakesh Thorat (22610012)
TY IT (T4 Batch)
(Experiment No. 01)

1) Title:

Study of Mapping Entities and Relationships to Relation Table (Schema Diagram)

2) Aim:

To map an Entity-Relationship (ER) diagram into relational tables (schemas) by defining entities, attributes, and relationships in a structured format, thereby creating a schema diagram that adheres to relational database principles.

3) Objectives:

- To understand the process of translating an ER diagram into relational tables.
- To apply relational database concepts to organize data effectively in tabular formats.
- To develop a schema diagram that maintains relationships and ensures data integrity through primary and foreign keys.

4) Theory:

1. Entities and Attributes:

Entities represent real-world objects or concepts (e.g., Employee, Department) with attributes describing their properties (e.g., ssn, name, salary for Employee). In the relational schema, each entity is mapped to a table, and each attribute becomes a column.

2. Primary Keys and Foreign Keys:

A primary key uniquely identifies each record within a table. Relationships between tables are maintained through foreign keys—attributes that reference primary keys in other tables. For example, employee_ssn in the Dependent table links each dependent to an employee in the Employee table.

3. Mapping Relationships:

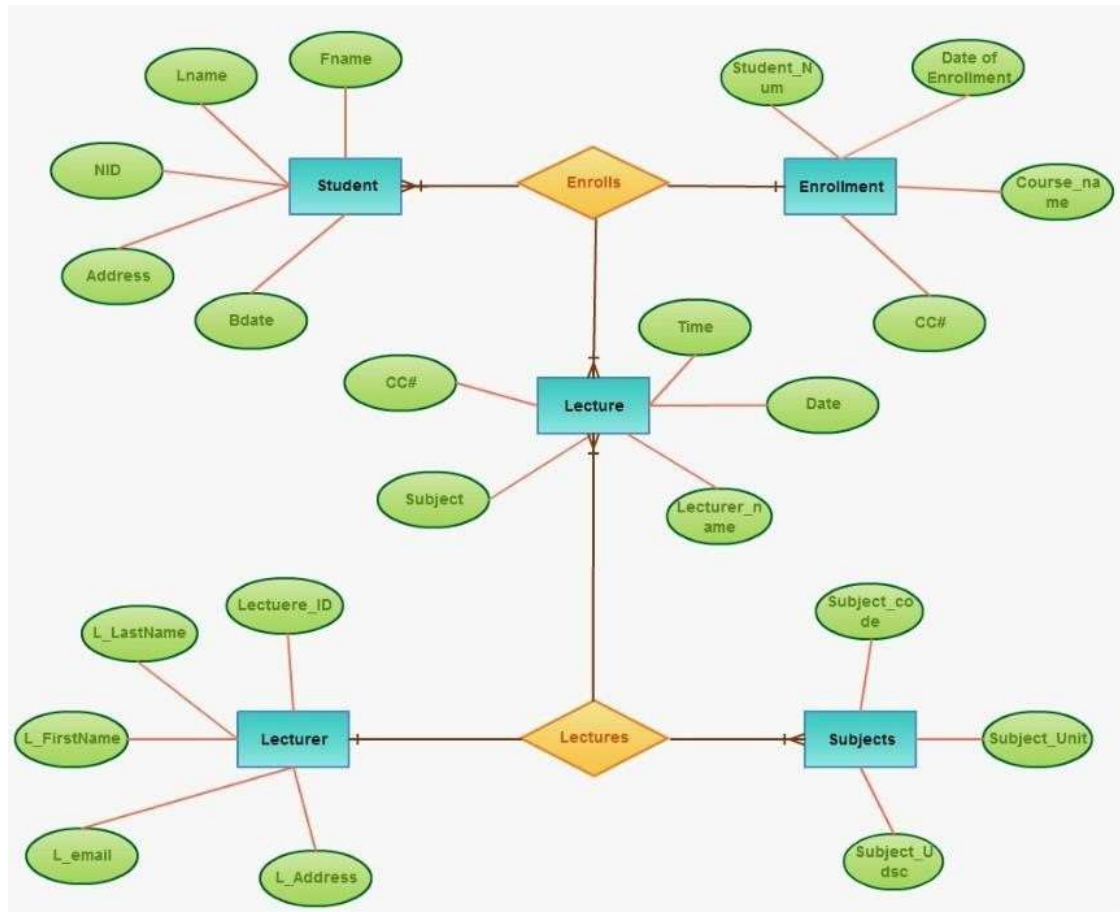
Relationships in ER diagrams (e.g., many-to-many, one-to-many) are translated into table structures. Here's how each type is mapped:

- **One-to-One:** Often merged into one table or represented by foreign keys.
- **One-to-Many:** Implemented using a foreign key in the "many" table referencing the primary key of the "one" table.

- **Many-to-Many:** Converted into a separate table (known as a junction table) containing foreign keys that link the related entities.

5) Implementation:

A-1) ER-Diagram for student database:



A-2) Schema diagram for student database:

Table: Student

Student_Num	Lname	Fname	NID	Address	Bdate
-------------	-------	-------	-----	---------	-------

Schema: Student (Student_Num, Lname, Fname, NID, Address, Bdate)

Table: Lecturer

Lecturer_ID	L_FirstName	L_LastName	L_Address	L_email
-------------	-------------	------------	-----------	---------

Schema: Lecturer (Lecturer_ID, L_FirstName, L_LastName, L_Address, L_email)

Table: Subjects

Subject_code	Subject_Unit	Subject_Unit_desc
--------------	--------------	-------------------

Schema: Subjects (Subject_code, Subject_Unit, Subject_Unit_desc)

Table: Lecture

CC#	Time	Date	Lecturer_name	Subject_code
-----	------	------	---------------	--------------

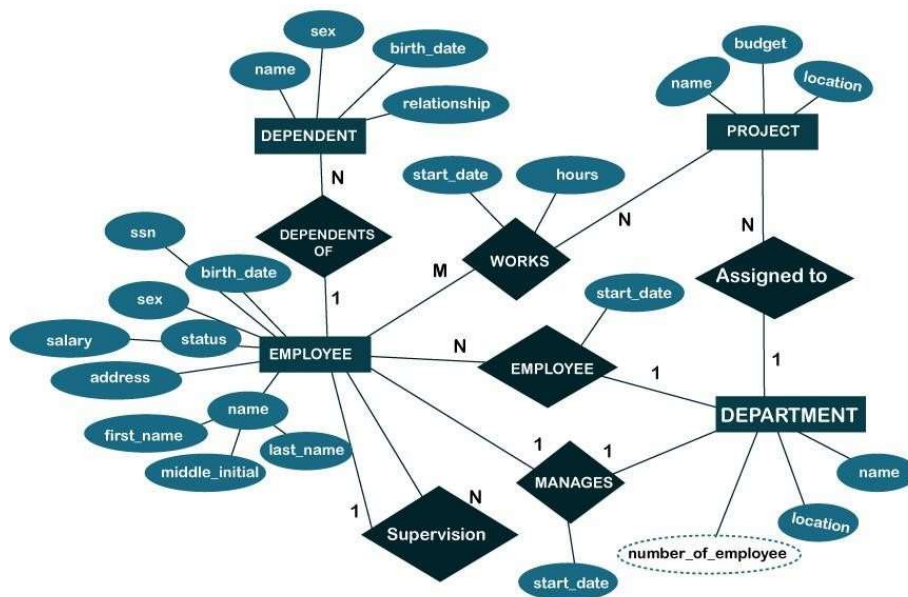
Schema: Lecture (CC#, Time, Date, Lecturer_name, Subject_code)

Table: Enrollment

Enrollment_ID	Student_Num	Date_of_Enrollment	Course_name
---------------	-------------	--------------------	-------------

Schema: Enrollment (Enrollment_ID, Student_Num, Date_of_Enrollment, Course_name)

B-1) ER-Diagram for company database:



B-2) Schema diagram for company database:

Schema Tables

1. Employee Table

ssn	first_name	middle_initial	last_name	birth_date	sex	address	salary	status
-----	------------	----------------	-----------	------------	-----	---------	--------	--------

Schema: Employee(ssn, first_name, middle_initial, last_name, birth_date, sex, address, salary, status)

2. Department Table

department_id	name	location	number_of_employee
---------------	------	----------	--------------------

Schema: Department(department_id, name, location, number_of_employee)

3. Project Table

project_id	name	budget	location
------------	------	--------	----------

Schema: Project(project_id, name, budget, location)

4. Dependent Table

dependent_id	name	sex	birth_date	relationship	employee_ssn
--------------	------	-----	------------	--------------	--------------

Schema: Dependent(dependent_id, name, sex, birth_date, relationship, employee_ssn)

5. Works Table

employee_ssn	project_id	start_date	hours
--------------	------------	------------	-------

Schema: Works(employee_ssn, project_id, start_date, hours)

6. Supervision Table

supervisor_ssn	subordinate_ssn
----------------	-----------------

Schema: Supervision(supervisor_ssn, subordinate_ssn)

7. Manages Table

employee_ssn	department_id	start_date
--------------	---------------	------------

Schema: Manages(employee_ssn, department_id, start_date)

6) Conclusion:

By mapping an ER diagram into relational tables, we successfully structured data into a relational schema that can be implemented in a database management system (DBMS). This schema diagram ensures efficient data storage and retrieval while maintaining data integrity through relationships defined by primary and foreign keys. This process highlights the importance of ER-to-relational mapping as a foundational step in database design, allowing us to represent complex relationships and dependencies in a structured, tabular format.

Om Ravindra Kulsange (22610013)
TY IT (T4 Batch)
(Experiment No. 2)

1) Title:

Implement relation algebra such as select project intersect minus and Cartesian operations on a given database.

2) Aim:

To apply relational algebra operations—select, project, intersect, minus, and Cartesian product—on a database using SQL.

3) Objectives:

- To understand and implement relational algebra operations using SQL.
- To perform operations on an EMPLOYEE table and analyse results based on relational algebra principles.
- To apply these operations for data retrieval and manipulation in a relational database context.

4) Theory:

Relational Algebra is a procedural query language used to query relational databases. It consists of a set of operations that take one or two relations as input and produce a new relation as a result. The primary operations used in this experiment are:

- **Cartesian product (\times):** Combines all rows from two relations.
- **Minus ($-$):** Finds records in one relation that are not present in another.
- **Intersection (\cap):** Retrieves common records from two relations.
- **Select (σ):** Filters rows based on a specified condition.
- **Project (π):** Selects specific columns from a table.

In this experiment, we will use SQL to simulate these operations on a relational database containing employee and department data.

5) Lab Practice – Example Queries:

```
CREATE DATABASE ASSIGNMENT;  
USE ASSIGNMENT;
```

```
CREATE TABLE EMPLOYEE (  
    SSN VARCHAR(20),  
    FNAME VARCHAR(20),  
    LNAME VARCHAR(20),  
    ADDRESS VARCHAR(20),  
    SEX CHAR(1),  
    SALARY INTEGER  
);
```

```
INSERT INTO EMPLOYEE (SSN, FNAME, LNAME, ADDRESS, SEX, SALARY)  
VALUES  
( '123-45-6789', 'John', 'Doe', '123 Elm St', 'M', 60000),  
( '987-65-4321', 'Jane', 'Smith', '456 Maple St', 'F', 75000),  
( '555-55-5555', 'Jim', 'Beam', '789 Oak St', 'M', 55000),  
( '222-33-4444', 'Jake', 'Blues', '321 Pine St', 'M', 70000);
```

```
CREATE TABLE EMPLOYEE2 (  
    SSN VARCHAR(20),  
    FNAME VARCHAR(20),  
    LNAME VARCHAR(20),  
    ADDRESS VARCHAR(20),  
    SEX CHAR(1),  
    SALARY INTEGER  
);
```

```
INSERT INTO EMPLOYEE2 (SSN, FNAME, LNAME, ADDRESS, SEX, SALARY)  
VALUES  
( '987-65-4321', 'Jane', 'Smith', '456 Maple St', 'F', 75000),  
( '555-55-5555', 'Jim', 'Beam', '789 Oak St', 'M', 55000);
```

```
CREATE TABLE DEPARTMENT (  
    DNUMBER INT,  
    DNAME VARCHAR(20)  
);
```

```
INSERT INTO DEPARTMENT (DNUMBER, DNAME) VALUES  
(1, 'HR'),  
(2, 'Engineering');
```

```
SELECT * FROM EMPLOYEE WHERE SALARY > 50000;
```

```
SELECT FNAME, LNAME FROM EMPLOYEE;
```

SELECT * FROM EMPLOYEE WHERE SSN IN (SELECT SSN FROM EMPLOYEE2);
 SELECT * FROM EMPLOYEE WHERE SSN NOT IN (SELECT SSN FROM EMPLOYEE2);

SELECT * FROM EMPLOYEE, DEPARTMENT;

6) Solve Lab Practice:

1. Implement Relational Algebra on given database.
2. Use select, project, intersect, minus and Cartesian operations on a given database.

1) Data:

	SSN	FNAME	LNAME	ADDRESS	SEX	SALARY
▶	123-45-6789	John	Doe	123 Elm St	M	60000
	987-65-4321	Jane	Smith	456 Maple St	F	75000
	555-55-5555	Jim	Beam	789 Oak St	M	55000
	222-33-4444	Jake	Blues	321 Pine St	M	70000

2) Relational Algebra Queries:

- **Select Operation:**
 - **Query**
 - **SELECT * FROM EMPLOYEE WHERE SALARY > 50000**

	SSN	FNAME	LNAME	ADDRESS	SEX	SALARY
▶	123-45-6789	John	Doe	123 Elm St	M	60000
	987-65-4321	Jane	Smith	456 Maple St	F	75000
	555-55-5555	Jim	Beam	789 Oak St	M	55000
	222-33-4444	Jake	Blues	321 Pine St	M	70000

- **Project Operation:**
 - **Query**
 - **SELECT FNAME, LNAME FROM EMPLOYEE**

	FNAME	LNAME
▶	John	Doe
	Jane	Smith
	Jim	Beam
	Jake	Blues

- **Intersection Operation:**
 - **Query**
 - **SELECT * FROM EMPLOYEE WHERE SSN IN (SELECT SSN FROM EMPLOYEE2)**

	SSN	FNAME	LNAME	ADDRESS	SEX	SALARY
▶	987-65-4321	Jane	Smith	456 Maple St	F	75000
	555-55-5555	Jim	Beam	789 Oak St	M	55000

- **Minus Operation:**

- **Query**

- **SELECT * FROM EMPLOYEE
WHERE SSN NOT IN (SELECT SSN FROM EMPLOYEE2)**

	SSN	FNAME	LNAME	ADDRESS	SEX	SALARY
▶	123-45-6789	John	Doe	123 Elm St	M	60000
	222-33-4444	Jake	Blues	321 Pine St	M	70000

- **Cartesian Product Operation:**

- **Query**

- **SELECT * FROM EMPLOYEE, DEPARTMENT**

	SSN	FNAME	LNAME	ADDRESS	SEX	SALARY	DNUMBER	DNAME
▶	123-45-6789	John	Doe	123 Elm St	M	60000	2	Engineering
	123-45-6789	John	Doe	123 Elm St	M	60000	1	HR
	987-65-4321	Jane	Smith	456 Maple St	F	75000	2	Engineering
	987-65-4321	Jane	Smith	456 Maple St	F	75000	1	HR
	555-55-5555	Jim	Beam	789 Oak St	M	55000	2	Engineering
	555-55-5555	Jim	Beam	789 Oak St	M	55000	1	HR
	222-33-4444	Jake	Blues	321 Pine St	M	70000	2	Engineering
	222-33-4444	Jake	Blues	321 Pine St	M	70000	1	HR

7) Conclusion:

The experiment successfully demonstrated the application of relational algebra operations on a relational database using SQL commands. Each operation—select, project, intersect, minus, and Cartesian product—was applied to retrieve specific information, showing how relational algebra can be used to manipulate and retrieve data in various ways in a relational database system.

Vinit Sanjay Mohite (22610017)
TY IT (T4)
(Experiment No.3)

1) Title: Study and Implementation of DDL and DML commands of SQL with suitable examples.

2) Aim:

The aim of this experiment is to study and implement Data Definition Language (DDL) and Data Manipulation Language (DML) commands of SQL. This experiment will demonstrate how to define and manipulate data using SQL commands.

3) Objectives:

- Understand the concepts of DDL and DML commands in SQL.
- Learn how to create and alter tables using DDL commands.
- Implement DML commands such as INSERT, UPDATE, and DELETE.
- Develop the ability to query and manipulate database records efficiently.

4) Theory:

Data Definition Language (DDL):

DDL commands are used to define the structure of the database. These commands are responsible for creating, modifying, and deleting database objects such as tables. The key DDL commands are:

- **CREATE:** Used to create new database objects like tables.

```
CREATE TABLE DEPARTMENT (  
    DEPTNO NUMBER(38) NOT NULL,  
    DNAME VARCHAR2(10),  
    LOC VARCHAR2(4),  
    PINCODE NUMBER(6) NOT NULL  
);
```
- **ALTER:** Used to modify an existing database structure.

```
ALTER TABLE DEPARTMENT  
ADD MANAGER VARCHAR2(20);
```
- **DROP:** Used to delete a database object.

```
DROP TABLE DEPARTMENT;
```

Data Manipulation Language (DML):

DML commands are used to manage data within database tables. These commands allow us to insert, update, delete, and retrieve data. The key DML commands are:

- **INSERT:** Adds new records to a table.

```

INSERT INTO DEPARTMENT (DEPTNO, DNAME, LOC,
PINCODE)
VALUES
(101, 'HR', 'NYC', 10001),
(102, 'Finance', 'LA', 90001),
(103, 'IT', 'SF', 94105),
(104, 'Marketing', 'CHI', 60601),
(105, 'Operations', 'BOS', 02108),
(106, 'Research', 'SEA', 98101);

```

- **UPDATE:** Modifies existing records.

```

UPDATE DEPARTMENT
SET LOC = 'LA'
WHERE DEPTNO = 101;

```
- **DELETE:** Removes records from a table.

```

DELETE FROM DEPARTMENT
WHERE DEPTNO = 101;

```
- **SELECT:** Retrieves data from one or more tables.

```

SELECT * FROM DEPARTMENT;

```

5) Outputs:

Let's define a table named DEPARTMENT using DDL commands and perform some DML operations on it.

Queries for DDL and DML:

Create Table (DDL):

Alter Table(DDL) :

DEPTNO	DNAME	LOC	PINCODE	MANAGER
102	Finance	LA	90001	-
103	IT	SF	94105	-
104	Marketing	CHI	60601	-
105	Operations	BOS	2108	-
106	Research	SEA	98101	-

Insert Data (DML) :

DEPTNO	DNAME	LOC	PINCODE
101	HR	NYC	10001
102	Finance	LA	90001
103	IT	SF	94105
104	Marketing	CHI	60601
105	Operations	BOS	2108
106	Research	SEA	98101

Update Data (DML) :

DEPTNO	DNAME	LOC	PINCODE
101	HR	LA	10001
102	Finance	LA	90001
103	IT	SF	94105
104	Marketing	CHI	60601
105	Operations	BOS	2108
106	Research	SEA	98101

Delete Data (DML) :

DEPTNO	DNAME	LOC	PINCODE
102	Finance	LA	90001
103	IT	SF	94105
104	Marketing	CHI	60601
105	Operations	BOS	2108
106	Research	SEA	98101

Select Data (DML) :

DEPTNO	DNAME	LOC	PINCODE
102	Finance	LA	90001
103	IT	SF	94105
104	Marketing	CHI	60601
105	Operations	BOS	2108
106	Research	SEA	98101

6) Solve Lab Practice:

Here are additional tasks:

1. Create an EMPLOYEE table with columns EmpID, EmpName, DeptNo, Salary, and JoinDate.

```
CREATE TABLE EMPLOYEE (  
    EmpID INT PRIMARY KEY,  
    EmpName VARCHAR(50),  
    DeptNo INT,  
    Salary DECIMAL(10, 2),
```

```
JoinDate DATE  
);
```

2. Insert at least three records into the EMPLOYEE table.

```
INSERT INTO EMPLOYEE (EmpID, EmpName, DeptNo,  
Salary, JoinDate) VALUES  
(1, 'John Doe', 101, 75000, '2021-05-15'),  
(2, 'Jane Smith', 102, 58000, '2022-03-20'),  
(3, 'Alice Johnson', 103, 62000, '2023-01-10');
```

EmpID	EmpName	DeptNo	Salary	JoinDate
1	John Doe	101	75000	2021-05-15
2	Jane Smith	102	58000	2022-03-20
3	Alice Johnson	103	62000	2023-01-10

2. Retrieve employees who have a salary greater than 60,000.

```
SELECT * FROM EMPLOYEE  
WHERE Salary > 60000;
```

EmpID	EmpName	DeptNo	Salary	JoinDate
1	John Doe	101	75000	2021-05-15
3	Alice Johnson	103	62000	2023-01-10

7) Conclusion:

In this experiment, we have learned how to use DDL commands to define the structure of a database table and DML commands to manipulate data in the table. We successfully created the DEPARTMENT table and performed various operations such as inserting, updating, deleting, and selecting data. Understanding DDL and DML commands is fundamental to managing a relational database system effectively.

Elizabeth Suresh Pawar (22610026)
TY IT (T4 Batch)
Experiment No 4

1. Title:

Creating a Bank Database and Applying DQL and TCL SQL Commands

2. Aim:

To create a bank information database, implement DQL (Data Query Language), TCL (Transaction Control Language) commands, and practice using various SQL operators.

3. Objective:

- To create a bank database and a corresponding accounts table.
- To perform basic DQL queries to retrieve data from the database.
- To use TCL commands like SAVEPOINT, ROLLBACK, and COMMIT for handling transactions.
- To demonstrate the use of SQL operators in queries (e.g., comparison, logical, arithmetic, and pattern-matching operators).

4. Theory:

SQL

Overview:

SQL (Structured Query Language) is used to manage and manipulate relational databases. It includes different categories of commands such as:

- **DQL (Data Query Language):** Primarily used to retrieve data from the database using SELECT statements.
- **TCL (Transaction Control Language):** Used to manage transactions in SQL, specifically COMMIT, ROLLBACK, and SAVEPOINT.

Transaction Management:

Transactions allow multiple SQL commands to be treated as a single unit of work.

Transactions are essential in database systems to ensure data integrity. The TCL commands play a role in controlling transaction boundaries:

1. **COMMIT:** Finalizes the transaction and makes changes permanent.
2. **ROLLBACK:** Undoes changes made in the transaction.
3. **SAVEPOINT:** Marks a point in the transaction to which a rollback can occur without affecting the entire transaction.
4. **Operators:**
SQL operators are used within queries to compare data, filter results, and manipulate records:
 - **Comparison Operators:** =, >, <, >=, <=, <>.
 - **Logical Operators:** AND, OR, NOT.
 - **Pattern Matching (LIKE):** Matches strings using % and _.
 - **Range Operators (BETWEEN, IN):** Filters values within a certain range or set.

5. Output:

The screenshot displays the MySQL Workbench interface. The left sidebar contains the 'MANAGEMENT' tab with options like Server Status, Client Connections, Users and Privileges, Status and System Variables, Data Export, and Data Import/Restore. Below this is the 'INSTANCE' tab with Start/Shutdown, Server Logs, and Options File. The 'PERFORMANCE' tab includes Dashboard, Performance Reports, and Performance Schema Setup. The main editor shows a SQL script with the following queries:

```
16 INSERT INTO Accounts (AccountID, AccountHolder, AccountType, Balance) VALUES
17 (1, 'Alice Johnson', 'Savings', 1500.00),
18 (2, 'Bob Smith', 'Checking', 800.00),
19 (3, 'Charlie Brown', 'Savings', 1200.50);
20
21 -- 5. DQ: Queries
22 -- Retrieve all accounts
23 SELECT * FROM Accounts;
24
25 -- Retrieve accounts with a balance greater than 1000
26 SELECT * FROM Accounts WHERE Balance > 1000;
27
28 -- Use pattern matching to find account holders with names starting with 'A'
```

The 'Result Grid' shows the output of the first query, displaying three rows of account data:

AccountID	AccountHolder	AccountType	Balance
1	Alice Johnson	Savings	2000.00
2	Bob Smith	Checking	600.00
3	Charlie Brown	Savings	1200.50

The 'Output' tab shows the 'Action Output' for the executed queries, listing the time, action, message, and duration for each step. The actions include creating the database, creating the table, inserting data, and executing the SELECT queries.

#	Time	Action	Message	Duration / Fetch
1	05:40:00	CREATE DATABASE BankDB	1 row(s) affected	0.000 sec
2	05:40:00	USE BankDB	0 row(s) affected	0.000 sec
3	05:40:04	CREATE TABLE Accounts (AccountID INT PRIMARY KEY, AccountHolder VARCHAR(50) NOT NULL, AccountType VARCHAR(20), Balance	0 row(s) affected	0.016 sec
4	05:40:10	INSERT INTO Accounts (AccountID, AccountHolder, AccountType, Balance) VALUES (1, 'Alice Johnson', 'Savings', 1500.00), (2, 'Bob Smith', 'Checkin...	3 row(s) affected Records: 3 Duplicates: 0 Warnings: 0	0.016 sec
5	05:40:10	SELECT * FROM Accounts LIMIT 0, 1000	3 row(s) returned	0.000 sec / 0.000 sec
6	05:40:18	SELECT * FROM Accounts WHERE Balance > 1000 LIMIT 0, 1000	2 row(s) returned	0.000 sec / 0.000 sec
7	05:40:18	SELECT * FROM Accounts WHERE AccountHolder LIKE 'A%' LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec
8	05:40:25	START TRANSACTION	0 row(s) affected	0.000 sec
9	05:40:25	UPDATE Accounts SET Balance = Balance + 500 WHERE AccountID = 1	1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0	0.000 sec
10	05:40:25	SAVEPOINT BeforeWithdraw	0 row(s) affected	0.000 sec
11	05:40:25	UPDATE Accounts SET Balance = Balance - 200 WHERE AccountID = 2	1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0	0.000 sec
12	05:40:33	COMMIT	0 row(s) affected	0.016 sec
13	05:40:43	SELECT * FROM Accounts LIMIT 0, 1000	3 row(s) returned	0.000 sec / 0.000 sec

The bottom status bar shows the user 'BSE mickap' with a CPU usage of 0.03%, the system language set to 'ENG IN', and the date and time as '05:40 AM 17-10-2024'.

6. Solve Lab Practice:

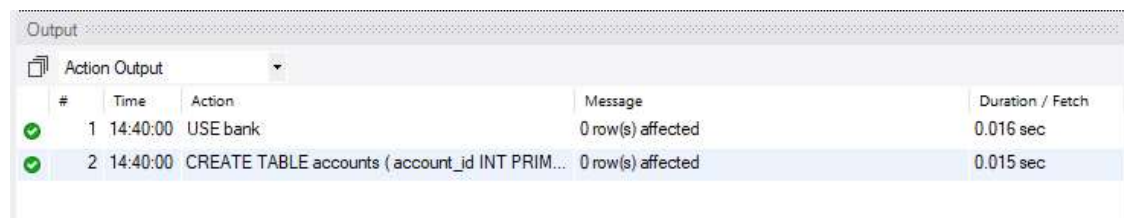
Step 1: Create the Bank Database and Table

sql

```
CREATE DATABASE bank;
```

```
USE bank;
```

```
CREATE TABLE accounts (  
  account_id INT PRIMARY KEY AUTO_INCREMENT,  
  account_holder VARCHAR(100),  
  account_type VARCHAR(20),  
  balance DECIMAL(10, 2),  
  branch VARCHAR(50)  
);
```



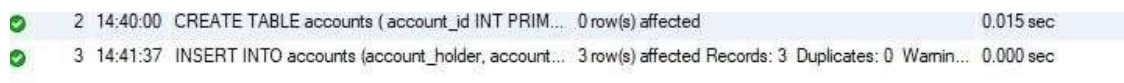
The screenshot shows the 'Output' window in SQL Server Enterprise Manager. It contains a table with the following data:

#	Time	Action	Message	Duration / Fetch
1	14:40:00	USE bank	0 row(s) affected	0.016 sec
2	14:40:00	CREATE TABLE accounts (account_id INT PRIM...	0 row(s) affected	0.015 sec

Step 2: Insert Data into the Table

sql

```
INSERT INTO accounts (account_holder, account_type, balance, branch)  
VALUES  
( 'Alice Johnson', 'Savings', 1500.00, 'New York'),  
( 'Bob Smith', 'Checking', 3000.00, 'Los Angeles'),  
( 'Charlie Williams', 'Savings', 5000.00, 'Chicago');
```



The screenshot shows the continuation of the 'Output' window. It contains a table with the following data:

2	14:40:00	CREATE TABLE accounts (account_id INT PRIM...	0 row(s) affected	0.015 sec
3	14:41:37	INSERT INTO accounts (account_holder, account...	3 row(s) affected Records: 3 Duplicates: 0 Wamin...	0.000 sec

Task 1: Write a query to implement the savepoint.

sql

```
START TRANSACTION;
```

```
-- Insert new account
```

```
INSERT INTO accounts (account_holder, account_type, balance, branch)  
VALUES ( 'Emily Clark', 'Checking', 2500.00, 'San Francisco');
```

```
-- Savepoint created before committing the new record
```

```
SAVEPOINT sp1;
```

✓	4	14:42:41	START TRANSACTION	0 row(s) affected	0.000 sec
✓	5	14:42:41	INSERT INTO accounts (account_holder, account...	1 row(s) affected	0.000 sec
✓	6	14:42:41	SAVEPOINT sp1	0 row(s) affected	0.000 sec

Task 2: Write a query to implement the rollback.

sql

-- Roll back to the savepoint sp1, undoing changes made after the savepoint
ROLLBACK TO sp1;

✓	6	14:42:41	SAVEPOINT sp1	VALUES ('Emily Clark', 'Checking', 2500.00, 'San Francisco')	0.000 sec
✓	7	14:43:20	ROLLBACK TO sp1	0 row(s) affected	0.000 sec

Task 3: Write a query to implement the commit.

sql

-- Commit the transaction to finalize the changes
COMMIT;

✓	7	14:43:20	ROLLBACK TO sp1	0 row(s) affected	0.000 sec
✓	8	14:43:57	COMMIT	0 row(s) affected	0.016 sec

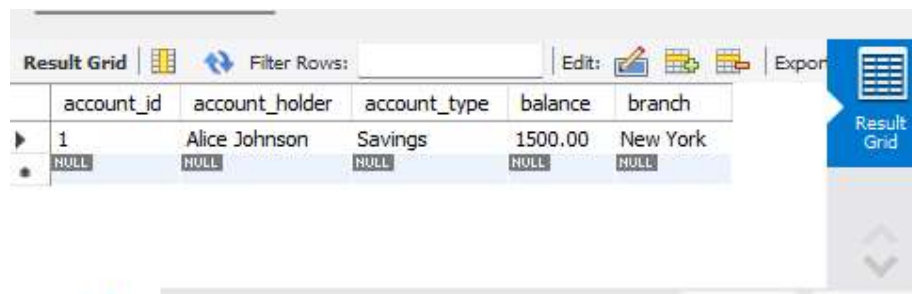
Task 4: Use of SQL Operators

Example queries for various SQL operators:

- **Comparison Operator (=):**

sql

SELECT * FROM accounts WHERE account_holder = 'Alice Johnson';



	account_id	account_holder	account_type	balance	branch
▶	1	Alice Johnson	Savings	1500.00	New York
•	NULL	NULL	NULL	NULL	NULL

- **Logical Operators (AND, OR):**

sql

SELECT * FROM accounts WHERE branch = 'New York' OR branch = 'Los Angeles';

Result Grid					
	account_id	account_holder	account_type	balance	branch
▶	1	Alice Johnson	Savings	1500.00	New York
	2	Bob Smith	Checking	3000.00	Los Angeles
	5	Alice Johnson	Savings	1500.00	New York
	6	Bob Smith	Checking	3000.00	Los Angeles
*	NULL	NULL	NULL	NULL	NULL

- **Pattern Matching (LIKE):**

sql

```
SELECT * FROM accounts WHERE account_holder LIKE '%Smith%';
```

Result Grid					
	account_id	account_holder	account_type	balance	branch
▶	2	Bob Smith	Checking	3000.00	Los Angeles
	6	Bob Smith	Checking	3000.00	Los Angeles
*	NULL	NULL	NULL	NULL	NULL

- **Range Operators (IN, BETWEEN):**

sql

```
SELECT * FROM accounts WHERE balance BETWEEN 1000 AND 4000;
```

Result Grid					
	account_id	account_holder	account_type	balance	branch
▶	1	Alice Johnson	Savings	1500.00	New York
	2	Bob Smith	Checking	3000.00	Los Angeles
	4	Emily Clark	Checking	2500.00	San Francisco
	5	Alice Johnson	Savings	1500.00	New York
	6	Bob Smith	Checking	3000.00	Los Angeles
	8	Emily Clark	Checking	2500.00	San Francisco
*	NULL	NULL	NULL	NULL	NULL

7. Conclusion:

In this lab experiment, students created a bank database and applied DQL and TCL commands, successfully using SELECT to retrieve data and demonstrating transaction control with SAVEPOINT, ROLLBACK, and COMMIT. This hands-on practice enhanced their skills in managing database transactions and effectively querying and filtering data from relational databases.

Rucha Ikare (22610029)
TY IT (T4 Batch)
(Experiment No. 5)

1) Title:

Perform various types of function in SQL

2) Aim:

To demonstrate the creation and manipulation of a SQL database and table, and to utilize various SQL functions including string, aggregate, mathematical, and date functions.

3) Objectives:

- Create a database named company.
- Create a table named employees with relevant columns.
- Insert sample employee data into the table.
- Perform various SQL operations using string, aggregate, mathematical, and date functions.

4) Theory:

SQL (Structured Query Language) is a standard programming language used to manage and manipulate relational databases. In this experiment, we focus on:

- **Creating Databases and Tables:** Establishing the structure for storing data.
- **Inserting Data:** Adding records to the table.
- **String Functions:** Manipulating string data, such as concatenation and case conversion.
- **Aggregate Functions:** Performing calculations on sets of data, such as counting and averaging.
- **Mathematical Functions:** Conducting mathematical operations on numeric data.
- **Date Functions:** Managing and manipulating date values.

5) Outputs:

The outputs from the SQL queries demonstrate the results of the various functions applied:

```

18 -- STRING FUNCTION
19 • SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM employees;
20

```

#	full_name
1	Aditi Sharma
2	Sneha Singh
3	Pooja Patel
4	Neha Iyer
5	Ananya Nair

string functions

Full names: Concatenation of first and last names.

Uppercase last names.

```

18 -- STRING FUNCTION
19 • SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM employees;
20
21 • SELECT UPPER(last_name) AS uppercase_last_name FROM employees;
22

```

#	uppercase_last_name
1	SHARMA
2	SINGH
3	PATEL
4	IYER
5	NAIR

Aggregate functions

Total number of employees.

```

23 -- AGGREGATE FUNCTION
24 • SELECT COUNT(*) AS total_employees FROM employees;
25

```

#	total_employee:
1	5

Average salary of employees.


```

23  -- AGGREGATE FUNCTION
24 • SELECT COUNT(*) AS total_employees FROM employees;
25
26 • SELECT AVG(salary) AS average_salary FROM employees;
27
28  -- MATH FUNCTION
29 • SELECT ROUND(salary) AS rounded_salary FROM employees;

```

#	average_salary
1	67000.000000

Mathematical Functions:

Rounded salaries

```

28  -- MATH FUNCTION
29 • SELECT ROUND(salary) AS rounded_salary FROM employees;
30
31 • SELECT ABS(salary - 50000) AS salary_difference FROM employees;
32

```

#	rounded_salary
1	60000
2	75000
3	50000
4	80000
5	70000

Absolute salary differences from a specified value.

```

28  -- MATH FUNCTION
29 • SELECT ROUND(salary) AS rounded_salary FROM employees;
30
31 • SELECT ABS(salary - 50000) AS salary_difference FROM employees;
32
33  -- DATE FUNCTION
34 • SELECT first_name, DATEDIFF(CURRENT_DATE, hire_date) AS days_since_hire FROM

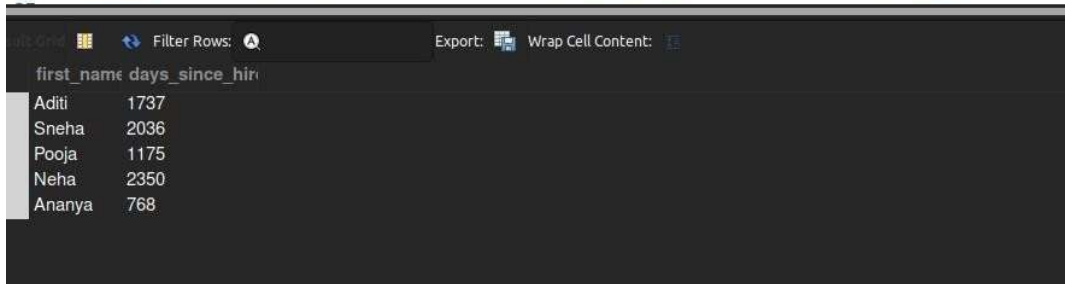
```

#	salary_difference
1	10000.00
2	25000.00
3	0.00
4	30000.00
5	20000.00

Date function

Number of days since each employee was hired.

```
--
33  -- DATE FUNCTION
34 • SELECT first_name, DATEDIFF(CURRENT_DATE, hire_date) AS days_since_hire FROM employees;
35
36 • SELECT first_name, DATE_FORMAT(hire_date, '%Y-%m-%d') AS formatted_hire_date FROM employees;
```

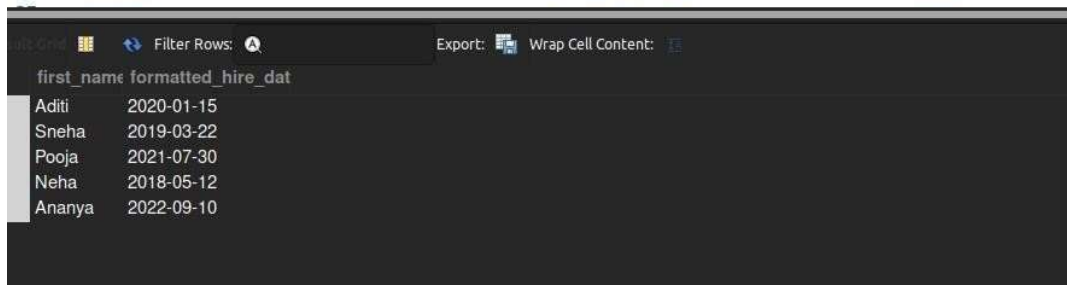


The screenshot shows a SQL query result in a dark-themed interface. The query is: `SELECT first_name, DATEDIFF(CURRENT_DATE, hire_date) AS days_since_hire FROM employees;`. The result table has two columns: `first_name` and `days_since_hire`. The data rows are:

first_name	days_since_hire
Aditi	1737
Sneha	2036
Pooja	1175
Neha	2350
Ananya	768

Formatted hire dates.

```
--
33  -- DATE FUNCTION
34 • SELECT first_name, DATEDIFF(CURRENT_DATE, hire_date) AS days_since_hire FROM employees;
35
36 • SELECT first_name, DATE_FORMAT(hire_date, '%Y-%m-%d') AS formatted_hire_date FROM employees;
```



The screenshot shows a SQL query result in a dark-themed interface. The query is: `SELECT first_name, DATE_FORMAT(hire_date, '%Y-%m-%d') AS formatted_hire_date FROM employees;`. The result table has two columns: `first_name` and `formatted_hire_date`. The data rows are:

first_name	formatted_hire_date
Aditi	2020-01-15
Sneha	2019-03-22
Pooja	2021-07-30
Neha	2018-05-12
Ananya	2022-09-10

6) Lab Practice - Example Queries:

Here is the complete SQL code used in the experiment:

```
CREATE DATABASE company; USE company;
```

```
CREATE TABLE employees ( id INT PRIMARY KEY,
first_name VARCHAR(50), last_name VARCHAR(50), salary DECIMAL(10, 2), hire_date DATE
);
```

```
INSERT INTO employees (id, first_name, last_name, salary, hire_date) VALUES (1, 'Aditi', 'Sharma', 60000,
'2020-01-15'),
```

```
(2, 'Sneha', 'Singh', 75000, '2019-03-22'),
```

```
(3, 'Pooja', 'Patel', 50000, '2021-07-30'),
```

```
(4, 'Neha', 'Iyer', 80000, '2018-05-12'),
```

```
(5, 'Ananya', 'Nair', 70000, '2022-09-10');
```

```
-- STRING FUNCTION
```

```

18 -- STRING FUNCTION
19 • SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM employees;
20

```

#	full_name
1	Aditi Sharma
2	Sneha Singh
3	Pooja Patel
4	Neha Iyer
5	Ananya Nair

SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM employees; SELECT UPPER(last_name) AS uppercase_last_name FROM employees;

```

18 -- STRING FUNCTION
19 • SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM employees;
20
21 • SELECT UPPER(last_name) AS uppercase_last_name FROM employees;
22

```

#	uppercase_last_name
1	SHARMA
2	SINGH
3	PATEL
4	IYER
5	NAIR

-- AGGREGATE FUNCTION

SELECT COUNT(*) AS total_employees FROM employees;

```

23 -- AGGREGATE FUNCTION
24 • SELECT COUNT(*) AS total_employees FROM employees;
25

```

#	total_employee:
1	5

SELECT AVG(salary) AS average_salary FROM employees;

```

28  -- MATH FUNCTION
29 • SELECT ROUND(salary) AS rounded_salary FROM employees;
30
31 • SELECT ABS(salary - 50000) AS salary_difference FROM employees;
32
33  -- DATE FUNCTION
34 • SELECT first_name, DATEDIFF(CURRENT_DATE, hire_date) AS days_since_hire FROM

```

#	salary_difference
1	10000.00
2	25000.00
3	0.00
4	30000.00
5	20000.00

```

23  -- AGGREGATE FUNCTION
24 • SELECT COUNT(*) AS total_employees FROM employees;
25
26 • SELECT AVG(salary) AS average_salary FROM employees;
27
28  -- MATH FUNCTION
29 • SELECT ROUND(salary) AS rounded_salary FROM employees;

```

#	average_salary
1	67000.000000

-- MATHFUNCTION

SELECT ROUND(salary) AS rounded_salary FROM employees;

```

28  -- MATH FUNCTION
29 • SELECT ROUND(salary) AS rounded_salary FROM employees;
30
31 • SELECT ABS(salary - 50000) AS salary_difference FROM employees;
32

```

#	rounded_salary
1	60000
2	75000
3	50000
4	80000
5	70000

SELECT ABS(salary - 50000) AS salary_difference FROM employees;

```

28  -- MATH FUNCTION
29 • SELECT ROUND(salary) AS rounded_salary FROM employees;
30
31 • SELECT ABS(salary - 50000) AS salary_difference FROM employees;
32
33  -- DATE FUNCTION
34 • SELECT first_name, DATEDIFF(CURRENT_DATE, hire_date) AS days_since_hire FROM

```

#	salary_difference
1	10000.00
2	25000.00
3	0.00
4	30000.00
5	20000.00

-- DATE FUNCTION

```
SELECT first_name, DATEDIFF(CURRENT_DATE, hire_date) AS days_since_hire FROM employees;
```

```

33  -- DATE FUNCTION
34 • SELECT first_name, DATEDIFF(CURRENT_DATE, hire_date) AS days_since_hire FROM employees;
35
36 • SELECT first_name, DATE_FORMAT(hire_date, '%Y-%m-%d') AS formatted_hire_date FROM employees;

```

first_name	days_since_hire
Aditi	1737
Sneha	2036
Pooja	1175
Neha	2350
Ananya	768

```
SELECT first_name, DATE_FORMAT(hire_date, '%Y-%m-%d') AS formatted_hire_date FROM employees;
```

```

33  -- DATE FUNCTION
34 • SELECT first_name, DATEDIFF(CURRENT_DATE, hire_date) AS days_since_hire FROM employees;
35
36 • SELECT first_name, DATE_FORMAT(hire_date, '%Y-%m-%d') AS formatted_hire_date FROM employees;

```

first_name	formatted_hire_date
Aditi	2020-01-15
Sneha	2019-03-22
Pooja	2021-07-30
Neha	2018-05-12
Ananya	2022-09-10

8) Conclusion:

This experiment successfully demonstrated the creation of a SQL database and table, the insertion of data, and the execution of various SQL functions. The outputs illustrate how to manipulate and analyze data effectively using SQL, showcasing its utility in managing relational databases.

Kranti Ananda Varekar (22610039)
TY IT (T4 Batch)
(Experiment No. 06)

1) Title:

Study and Implementation of various types of constraint in SQL.

2) Aim:

To explore and implement various types of constraints in SQL to ensure data integrity, enforce business rules, and manage the relationships between data in a relational database effectively.

3) Objectives:

- To create a table with specific column types and constraints.
- To apply various constraints such as NOT NULL, CHECK, UNIQUE, and PRIMARY KEY.
- To perform data manipulation operations (insert, update, delete) while adhering to constraints.
- To analyse the behaviour of constraints through practical examples.

3) Theory:

Constraints in SQL are rules enforced on data columns in a table to maintain the accuracy and integrity of the data. Common types of constraints include:

1. Domain Constraint

- **Definition:** Limits the type, format, or range of values that can be stored in a column.
- **Example:** Ensuring that a column for age only contains positive integers.

2. Entity Integrity Constraint

- **Definition:** Ensures that each row (or record) in a table has a unique and non-null primary key.
- **Primary Key Constraint:** A primary key is a unique identifier for each record, and no two rows can have the same primary key.
- **Example:** A `student_id` column in a `Students` table must contain unique values for each student.

3. Referential Integrity Constraint

- **Definition:** Ensures that a foreign key value in one table must match a primary key value in another table.
- **Foreign Key Constraint:** A foreign key establishes a relationship between two tables and ensures that relationships between records are valid.

- **Example:** In an `Orders` table, the `customer_id` must match an existing `customer_id` in the `Customers` table.

4. Unique Constraint

- **Definition:** Ensures that all values in a column (or a group of columns) are unique across the table.
- **Example:** In an `Employees` table, the email address must be unique, meaning no two employees can have the same email address.

5. NOT NULL Constraint

- **Definition:** Ensures that a column cannot have a null value.
- **Example:** In a `Users` table, the `username` column may have a `NOT NULL` constraint, ensuring that every user must have a username.

6. Check Constraint

- **Definition:** Ensures that all values in a column satisfy a specific condition.
- **Example:** In an `Employees` table, a `Check` constraint can ensure that the salary is greater than zero (`salary > 0`).

7. Default Constraint

- **Definition:** Provides a default value for a column when no value is specified.
- **Example:** In an `Orders` table, if the `status` column has a default constraint of 'Pending', all new records will have 'Pending' as the default status unless specified otherwise.

8. Key Constraints

- **Definition:** These involve special constraints applied to columns that uniquely identify records within a table.
 - **Super Key:** A set of one or more columns that can uniquely identify a record.
 - **Candidate Key:** A minimal super key; a column or combination of columns that can uniquely identify a record.
 - **Primary Key:** The chosen candidate key that is used to uniquely identify each row.
 - **Alternate Key:** The candidate keys that are not chosen as the primary key.

9. Tuple Uniqueness Constraint

- **Definition:** Ensures that no two rows in a table are identical.

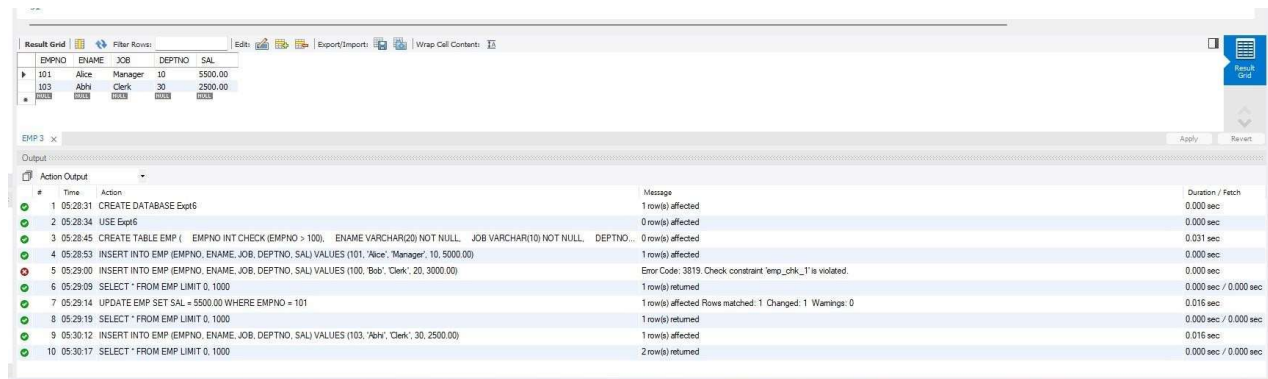
These constraints help maintain the integrity and validity of the database by preventing invalid or inconsistent data from being inserted.

These constraints help in maintaining the reliability of the data stored in the database.

5) Outputs

After executing the SQL commands to create the EMP table and inserting data, the outputs include:

- Successful creation of the EMP table with the specified structure.
- Confirmation messages for successful data insertion.
- Error messages for any attempts to violate constraints (e.g., inserting a NULL value where not allowed).



The screenshot displays the SQL Developer interface. At the top, the 'Result Grid' shows the structure of the EMP table with columns EMPNO, ENAME, JOB, DEPTNO, and SAL. Below this, the 'Output' window shows the 'Action Output' for a series of SQL commands. The commands include creating the database, using the database, creating the EMP table with constraints, and inserting data. The output shows successful execution for most commands, with an error message for an attempt to insert a NULL value into the DEPTNO column.

#	Time	Action	Message	Duration / Fetch
1	05:28:31	CREATE DATABASE Expt6	1 row(s) affected	0.000 sec
2	05:28:34	USE Expt6	0 row(s) affected	0.000 sec
3	05:28:45	CREATE TABLE EMP (EMPNO INT CHECK (EMPNO > 100), ENAME VARCHAR(20) NOT NULL, JOB VARCHAR(10) NOT NULL, DEPTNO...	0 row(s) affected	0.031 sec
4	05:28:53	INSERT INTO EMP (EMPNO, ENAME, JOB, DEPTNO, SAL) VALUES (101, 'Alice', 'Manager', 10, 5000.00)	1 row(s) affected	0.000 sec
5	05:29:00	INSERT INTO EMP (EMPNO, ENAME, JOB, DEPTNO, SAL) VALUES (100, 'Bob', 'Clerk', 20, 3000.00)	Error Code: 3819: Check constraint 'emp_chk_1' is violated.	0.000 sec
6	05:29:09	SELECT * FROM EMP LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec
7	05:29:14	UPDATE EMP SET SAL = 5500.00 WHERE EMPNO = 101	1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0	0.016 sec
8	05:29:19	SELECT * FROM EMP LIMIT 0, 1000	1 row(s) returned	0.000 sec / 0.000 sec
9	05:30:12	INSERT INTO EMP (EMPNO, ENAME, JOB, DEPTNO, SAL) VALUES (103, 'Ah', 'Clerk', 30, 2500.00)	1 row(s) affected	0.016 sec
10	05:30:17	SELECT * FROM EMP LIMIT 0, 1000	2 row(s) returned	0.000 sec / 0.000 sec

6) Solve Lab Practice

SQL Implementation

1. Create the Database and Table:

```
CREATE DATABASE DB;
```

```
USE DB;
```

```
CREATE TABLE EMP (
```

```
EMPNO INT CHECK (EMPNO > 100),
```

```
ENAME VARCHAR(20) NOT NULL,
```

```
JOB VARCHAR(10) NOT NULL,
```

```
DEPTNO INT UNIQUE,
```

```
SAL DECIMAL(7,2),
```

```
PRIMARY KEY (EMPNO)
```

);

2. Inserting Data:

-- Valid insert

```
INSERT INTO EMP (EMPNO, ENAME, JOB, DEPTNO, SAL) VALUES (101, 'Alice',  
'Manager', 10, 5000.00);
```

-- Invalid insert (EMPNO <= 100)

```
INSERT INTO EMP (EMPNO, ENAME, JOB, DEPTNO, SAL) VALUES (100, 'Bob', 'Clerk',  
20, 3000.00);
```

-- Valid insert (NULL ENAME)

```
INSERT INTO EMP (EMPNO, ENAME, JOB, DEPTNO, SAL) VALUES (103, 'Abhi', 'Clerk',  
30, 2500.00);
```

3. Querying Data:

```
SELECT * FROM EMP;
```

4. Updating Data:

```
UPDATE EMP SET SAL = 5500.00 WHERE EMPNO = 101;
```

7) Solve Lab Practice:

- The code demonstrates various SQL constraints: a **Primary Key** on EMPNO to ensure unique and non-null values.
- a **CHECK** constraint ensuring EMPNO is greater than 100
- **NOT NULL** constraints on ENAME and JOB
- a **Unique** constraint on DEPTNO.
- **DECIMAL** data type for precise salary values

These constraints maintain data integrity and validation in the "EMP" table.

7) Conclusion:

Through this experiment, we successfully created a table in SQL with various constraints to ensure data integrity. We observed the behaviour of different constraints by attempting to insert and update records, confirming that the database enforced the rules as expected. Understanding these constraints is crucial for maintaining a robust and reliable database structure, preventing invalid data entries, and ensuring that business rules are upheld. This foundational knowledge is essential for future database management tasks.

Manali Khedekar (22610040)
TY IT (T4)
(Experiment No. 7)

1) Title:

Implementation of different types of Joins

- Inner Join
- Outer Join
- Natural Join etc.

2) Aim:

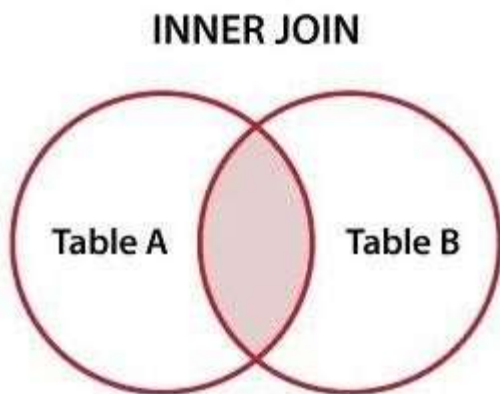
To implement various SQL joins (Inner, Outer, Natural) and perform data retrieval operations on the Sailors, Boats, and Reserves tables.

3) Objectives:

- To learn how to use Inner Join to fetch matching records across multiple tables.
- To apply Outer Join to retrieve both matching and non-matching records from tables.
- To understand the usage of Natural Join for automatically combining tables based on common attributes.
- To practice filtering and sorting data using SQL conditions and functions.
- To practice filtering and sorting data using SQL conditions and functions.

4) Theory:

1. **INNER JOIN:** An INNER JOIN returns only the rows that have matching values in both tables. If there's no match, the row is excluded from the results.



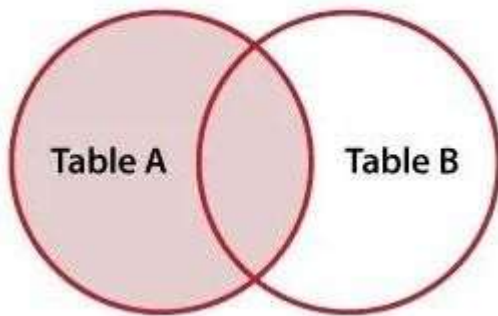
Syntax:

```
SELECT columns FROM table1 INNER JOIN table2 ON table1.column = table2.column;
```

2. **OUTER JOIN:** OUTER JOIN includes three types: LEFT, RIGHT, and FULL.

a. **LEFT JOIN** (or LEFT OUTER JOIN) Theory: A LEFT JOIN returns all rows from the left table and the matched rows from the right table. If there is no match, NULL values are returned for columns from the right table.

LEFT OUTER JOIN

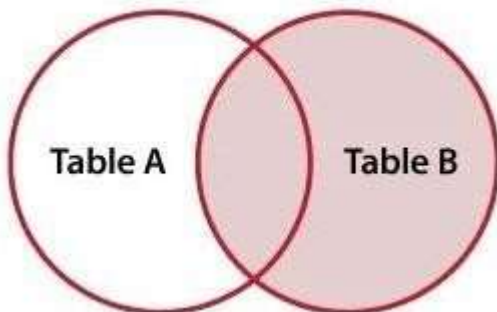


Syntax:

```
SELECT columns FROM table1 LEFT JOIN table2 ON table1.column = table2.column;
```

b. **RIGHT JOIN** (or RIGHT OUTER JOIN) Theory: A RIGHT JOIN returns all rows from the right table and the matched rows from the left table. If there is no match, NULL values are returned for columns from the left table.

RIGHT OUTER JOIN

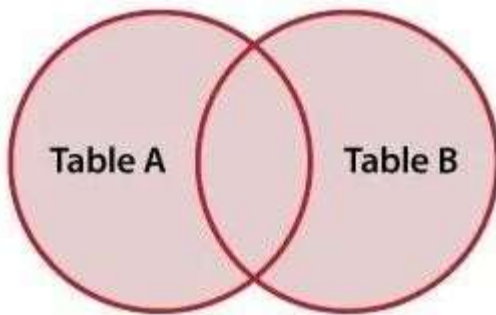


Syntax:

```
SELECT columns FROM table1 RIGHT JOIN table2 ON table1.column = table2.column;
```

c. **FULL JOIN** (or FULL OUTER JOIN) Theory: A FULL JOIN returns all rows when there is a match in either left or right table records. Rows without a match in either table will contain NULL values.

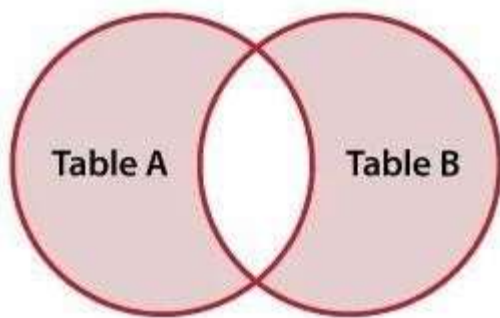
FULL OUTER JOIN



Syntax:

```
SELECT columns FROM table1 FULL OUTER JOIN table2 ON table1.column = table2.column;
```

3. NATURAL JOIN: A NATURAL JOIN automatically joins tables based on columns with the same name and compatible data types. It selects all columns from both tables except duplicate columns.



Syntax:

```
SELECT columns FROM table1 NATURAL JOIN table2;
```

5) Outputs:

Both Tables:

```
select * from Employees;
```

emp_id	emp_name	department_id
1	John	101
2	Alice	102
3	Bob	103
4	David	101
5	Eve	NULL
NULL	NULL	NULL

```
select * from Departments;
```

department_id	department_name
101	Sales
102	HR
103	IT
104	Marketing
NULL	NULL

Inner Join:

```
SELECT Employees.emp_name, Departments.department_name
FROM Employees
INNER JOIN Departments ON Employees.department_id = Departments.department_id;
```

emp_name	department_name
John	Sales
Alice	HR
Bob	IT
David	Sales

Left Join:

```
SELECT Employees.emp_name, Departments.department_name
FROM Employees
LEFT JOIN Departments ON Employees.department_id = Departments.department_id;
```

emp_name	department_name
John	Sales
Alice	HR
Bob	IT
David	Sales
Eve	NULL

Right Join:

```
SELECT Employees.emp_name, Departments.department_name
FROM Employees
RIGHT JOIN Departments ON Employees.department_id = Departments.department_id;
```

emp_name	department_name
David	Sales
John	Sales
Alice	HR
Bob	IT
NULL	Marketing

Full Join:

```
SELECT Employees.emp_name, Departments.department_name
FROM Employees
LEFT JOIN Departments ON Employees.department_id = Departments.department_id
UNION
SELECT Employees.emp_name, Departments.department_name
FROM Employees
RIGHT JOIN Departments ON Employees.department_id = Departments.department_id;
```

emp_name	department_name
John	Sales
Alice	HR
Bob	IT
David	Sales
Eve	NULL
NULL	Marketing

Natural Join:

```
SELECT *
FROM Employees
NATURAL JOIN Departments;
```

department_id	emp_id	emp_name	department_name
101	1	John	Sales
102	2	Alice	HR
103	3	Bob	IT
101	4	David	Sales

6) Solve Lab Practice:

```
CREATE DATABASE sailors;
```

```
USE sailors;
```

```
CREATE TABLE Sailors (
```

```
  sid INT AUTO_INCREMENT PRIMARY KEY,
```

```
  sname VARCHAR(100),
```

```
  rating INT,
```

```
  age INT
```

```
);
```

```
CREATE TABLE Boats (
```

```
  bid INT AUTO_INCREMENT PRIMARY KEY,
```

```
  bname VARCHAR(100),
```

```
  color VARCHAR(50)
```

);

```
CREATE TABLE Reserves (  
    sid INT,  
    bid INT,  
    day DATE,  
    FOREIGN KEY (sid) REFERENCES Sailors(sid),  
    FOREIGN KEY (bid) REFERENCES Boats(bid)  
);
```

INSERT INTO Sailors (sname, rating, age) VALUES

```
('Bob', 5, 22),  
('Alice', 4, 23),  
('Charlie', 5, 20),  
('David', 3, 21),  
('Eve', 4, 19),  
('Frank', 5, 24);
```

sid	sname	rating	age
1	Bob	5	22
2	Alice	4	23
3	Charlie	5	20
4	David	3	21
5	Eve	4	19
6	Frank	5	24
NULL	NULL	NULL	NULL

INSERT INTO Boats (bname, color) VALUES

```
('Boat1', 'red'),  
('Boat2', 'green'),  
('Boat3', 'blue'),  
('Boat4', 'red'),  
('Boat5', 'yellow');
```

bid	bname	color
1	Boat1	red
2	Boat2	green
3	Boat3	blue
4	Boat4	red
5	Boat5	yellow
NULL	NULL	NULL

INSERT INTO Reserves (sid, bid, day) VALUES

(1, 1, '2024-10-01'),

(1, 2, '2024-10-02'),

(2, 3, '2024-10-01'),

(3, 1, '2024-10-03'),

(4, 4, '2024-10-02'),

(5, 5, '2024-10-03'),

(1, 3, '2024-10-04');

sid	bid	day
1	1	2024-10-01
1	2	2024-10-02
2	3	2024-10-01
3	1	2024-10-03
4	4	2024-10-02
5	5	2024-10-03
1	3	2024-10-04

/*1 */

SELECT s.*

FROM Sailors s

INNER JOIN Reserves r ON s.sid = r.sid

WHERE r.bid = 1;

	sid	sname	rating	age
▶	1	Bob	5	22
	3	Charlie	5	20

/*2 */

SELECT b.bname

FROM Boats b

INNER JOIN Reserves r ON b.bid = r.bid

INNER JOIN Sailors s ON r.sid = s.sid

WHERE s.sname = 'Bob';

	bname
▶	Boat1
	Boat2
	Boat3

/*3 */

SELECT DISTINCT s.sname

FROM Sailors s

INNER JOIN Reserves r ON s.sid = r.sid

sname	age
Charlie	20
David	21
Bob	22

```
INNER JOIN Boats b ON r.bid = b.bid
```

```
WHERE b.color = 'red'
```

```
ORDER BY s.age;
```

```
/*4 */
```

```
SELECT DISTINCT s.sname
```

```
FROM Sailors s
```

```
INNER JOIN Reserves r ON s.sid = r.sid;
```

	sname
▶	Bob
	Alice
	Charlie
	David
	Eve

```
/*5 */
```

```
SELECT r1.sid, s.sname
```

```
FROM Reserves r1
```

```
INNER JOIN Reserves r2 ON r1.sid = r2.sid AND r1.bid <> r2.bid AND  
r1.day = r2.day
```

```
INNER JOIN Sailors s ON r1.sid = s.sid
```

```
GROUP BY r1.sid, s.sname;
```

	sid	sname

```
/*6 */
```

```
SELECT DISTINCT r.sid
```

```
FROM Reserves r
```

```
INNER JOIN Boats b ON r.bid = b.bid
```

```
WHERE b.color IN ('red', 'green');
```

```
/*7 */
```

```
SELECT s.sname, s.age
```

```
FROM Sailors s
```

	sid
▶	1
	3
	4

	sname	age
▶	Eve	19

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

/*8 */

SELECT COUNT(DISTINCT sname) AS unique_sailor_count
FROM Sailors;

unique_sailor_count
6

/*9 */

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

rating	average_age
5	22.0000
4	21.0000
3	21.0000

/*10 */

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

rating	average_age
5	22.0000
4	21.0000

7) Conclusion:

This assignment enhances understanding of SQL joins and complex queries, enabling efficient data extraction and manipulation from relational databases.

Anshul Patil (22610041)
TY IT (T4 Batch)
(Experiment No. 08)

1) Title:

Study & Implementation of Various Types of Clauses and Indexing.

2) Aim:

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

3) Objectives:

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

4) Theory:

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

SQL Clauses

1. GROUP BY:

- The GROUP BY clause is used to arrange identical data into groups. This is particularly useful when combined with aggregate functions such as SUM(), COUNT(), AVG(), etc. For instance, grouping employee data by department allows one to calculate the total salary per department.

2. HAVING:

- The HAVING clause is applied to filter records that work with aggregate functions. Unlike the WHERE clause, which filters records before any groupings are made, HAVING filters groups after the aggregation has occurred. For example, one can use HAVING to find departments with an average salary above a certain threshold.

3. ORDER BY:

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

4. WHERE:

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

Indexing:

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

Types of Indexing:

1. Unique Index:

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

2. Clustered Index:

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

3. Non-Clustered Index:

- A non-clustered index creates a separate structure from the data rows, maintaining a pointer to the actual data. This allows multiple non-clustered indexes on a table. For example, a non-clustered index can be created on employee names to quickly find specific employees without having to scan the entire employee table.

6) Solve Lab Practice:

a) Create a relation and implement the following queries:

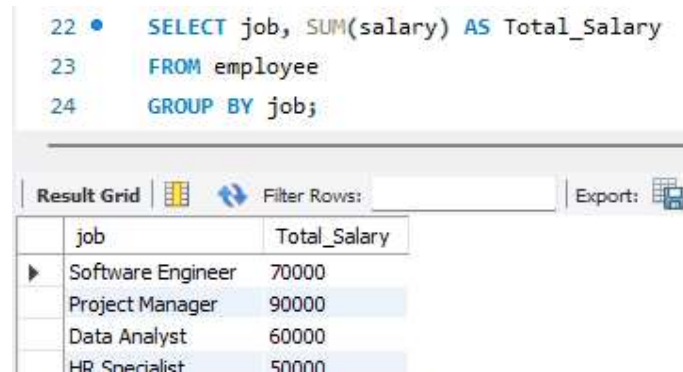
```
CREATE TABLE employee (  
  emp_id INT PRIMARY KEY,  
  name VARCHAR(50),  
  job VARCHAR(50),  
  manager_id INT,  
  department VARCHAR(50),  
  salary INT );
```

```
• show databases;  
• create database employ;  
• use employ;  
• CREATE TABLE employee (  
    emp_id INT PRIMARY KEY,  
    name VARCHAR(50),  
    job VARCHAR(50),  
    manager_id INT,  
    department VARCHAR(50),  
    salary INT  
  );
```

b) Queries Implementation:

1. Total salary spent for each job category:

```
SELECT job, SUM(salary) AS Total_Salary
FROM employee
GROUP BY job;
```



The screenshot shows a SQL query editor with the following code:

```
22 • SELECT job, SUM(salary) AS Total_Salary
23 FROM employee
24 GROUP BY job;
```

Below the editor is a toolbar with 'Result Grid', 'Filter Rows', and 'Export' buttons. The results are displayed in a table:

job	Total_Salary
Software Engineer	70000
Project Manager	90000
Data Analyst	60000
HR Specialist	50000

2. Lowest paid employee details under each manager:

```
SELECT manager_id, name, MIN(salary) AS Min_Salary
FROM employee
GROUP BY manager_id
ORDER BY manager_id;
```



The screenshot shows a SQL query editor with the following code:

```
20 • SELECT manager_id, name, MIN(salary) AS Min_Salary
21 FROM employee
22 GROUP BY manager_id, name
23 ORDER BY manager_id;
24
25
```

Below the editor is a toolbar with 'Result Grid', 'Filter Rows', 'Export', and 'Wrap Cell C' buttons. The results are displayed in a table:

manager_id	name	Min_Salary
1	Alice Smith	90000
2	Bob Johnson	50000
2	John Doe	60000
5	Eve Davis	45000

3. Number of employees working in each department and their department name:

```
SELECT department, COUNT(emp_id) AS Total_Employees
FROM employee
GROUP BY department;
```

```
24 • SELECT department, COUNT(emp_id) AS Total_Employees
25 FROM employee
26 GROUP BY department;
27
```

Result Grid		Filter Rows:	Export:	Wrap Cell Contents
	department	Total_Employees		
▶	IT	2		
	Data	1		
	HR	1		

4. Details of employees sorting the salary in increasing order:

```
SELECT *
FROM employee
ORDER BY salary ASC;
```

```
27 • SELECT *
28 FROM employee
29 ORDER BY salary ASC;
```

Result Grid

Filter Rows:

Edit:

Export/Imp

	emp_id	name	job	manager_id	department	salary
▶	4	Eve Davis	HR Specialist	5	HR	45000
	3	Bob Johnson	Data Analyst	2	Data	50000
	1	John Doe	Software Engineer	2	IT	60000
	2	Alice Smith	Manager	NULL	IT	90000

5. Record of employees earning a salary greater than 16000 in each department:

```
SELECT *
FROM employee
WHERE salary > 16000
ORDER BY department;
```

```
30 • SELECT *
31 FROM employee
32 WHERE salary > 16000
33 ORDER BY department;
```

Result Grid						
Filter Rows: <input type="text"/>						
Edit: Export/Import						
	emp_id	name	job	manager_id	department	salary
▶	3	Bob Johnson	Data Analyst	2	Data	50000
	4	Eve Davis	HR Specialist	5	HR	45000
	1	John Doe	Software Engineer	2	IT	60000
	2	Alice Smith	Manager	NULL	IT	90000

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

-- Create a unique index on emp_id

```
CREATE UNIQUE INDEX idx_emp_id ON employee(emp_id);
```

-- Create a clustered index on salary

```
CREATE CLUSTERED INDEX idx_salary ON employee(salary);
```

```
34 • CREATE UNIQUE INDEX idx_emp_id ON employee(emp_id);
35 • CREATE INDEX idx_salary ON employee(salary);
36 • SHOW INDEX FROM employee;
```

Result Grid												
Filter Rows: <input type="text"/>												
Export: Wrap Cell Content:												
	Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment
▶	employee	0	PRIMARY	1	emp_id	A	4	NULL	NULL		BTREE	YES
	employee	0	idx_emp_id	1	emp_id	A	4	NULL	NULL		BTREE	YES
	employee	1	idx_salary	1	salary	A	4	NULL	NULL	YES	BTREE	YES

7) Conclusion:

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

Aditya Ganesh Khandare (22610042)
TY IT (T4 Batch)
(Experiment No. 09)

1) Title:

Study and Implementation of Triggers and Views.

2) Aim:

1. To study and implement **triggers** and **views** in a relational database management system (RDBMS) and understand their usage in automating tasks and managing data efficiently.

3) Objectives:

- To understand the concept and working of **triggers** in an RDBMS.
- To implement **triggers** to automate specific actions when a particular event occurs in the database (e.g., INSERT, UPDATE, or DELETE).
- To explore the concept of **views** and their role in simplifying complex queries and data abstraction.
- To create and manage **views** for better data representation and security.

4) Theory:

Trigger: A trigger is procedural code in SQL that automatically executes in response to specific events (INSERT, UPDATE, DELETE) on a table or view.

Types of Triggers:

1. **BEFORE Trigger:** Executes before the triggering event (e.g., before an INSERT, UPDATE, or DELETE).
2. **AFTER Trigger:** Executes after the triggering event.
3. **INSTEAD OF Trigger:** Replaces the triggering event (mainly used with views).

Trigger Syntax:

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF} {INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
    -- Trigger actions
END;
```

View: A view is a virtual table that shows the result of a stored SQL query. It does not store data but displays data from underlying tables.

View Syntax:

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

Commands:

- **Create View:** CREATE VIEW view_name AS SELECT ...;
- **Modify View:** CREATE OR REPLACE VIEW view_name AS SELECT ...;
- **Delete View:** DROP VIEW view_name;

5) Lab Practice:

Database Schema:

Sailors table: (sid, sname, rating, age)

Boats table: (bid, bname, color)

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

```
CREATE TABLE Boats (  
    bid INT PRIMARY KEY,  
    bname VARCHAR(50),  
    color VARCHAR(30)  
);
```

Trigger Implementation:

Step 1: Create the Sailors table

```
4      -- Step 1: Create the Sailors table to store sailor data  
5      CREATE TABLE Sailors (  
6          sid INT PRIMARY KEY,  
7          sname VARCHAR(50),  
8          rating INT,  
9          age INT  
10     );
```

Step 2: Create the Sailor_Log table for logging insert operations

```
12     -- Step 2: Create the Sailor_Log table for logging insert operations  
13     CREATE TABLE Sailor_Log (  
14         log_id INT AUTO_INCREMENT PRIMARY KEY,  
15         sid INT,  
16         sname VARCHAR(50),  
17         rating INT,  
18         age INT,  
19         inserted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
20     );
```

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

```

22 -- Step 3: Create the trigger to log data after an insert into the Sailors table
23 DELIMITER //
24 * CREATE TRIGGER after_sailor_insert
25 AFTER INSERT ON Sailors
26 FOR EACH ROW
27 BEGIN
28     -- Insert the newly inserted sailor data into the Sailor_Log table
29     INSERT INTO Sailor_Log (sid, sname, rating, age)
30     VALUES (NEW.sid, NEW.sname, NEW.rating, NEW.age);
31 END; //

```

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

```

33 -- Step 4: Insert sample data into the Sailors table
34 * INSERT INTO Sailors (sid, sname, rating, age)
35 VALUES (1, 'Aditya', 7, 25);
36
37 INSERT INTO Sailors (sid, sname, rating, age)
38 VALUES (2, 'Vidyadhar', 6, 30),
39         (3, 'Vinit', 8, 28)

```

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

```

41 -- Step 5: Check the Sailor_Log table to verify
42 SELECT * FROM Sailor_Log;
43

```

Result Grid						
Filter Rows:						
Edit:						
	log_id	sid	sname	rating	age	inserted_at
	1	1	Aditya	7	25	2024-10-17 11:16:34
	2	2	Vidyadhar	6	30	2024-10-17 11:16:42
	3	3	Vinit	8	28	2024-10-17 11:16:42
▶*	NULL	NULL	NULL	NULL	NULL	

View Implementation:

Step 1: Create the Boats table

```

4 -- Step 1: Create the Boats table
5 * CREATE TABLE Boats (
6     bid INT PRIMARY KEY,
7     bname VARCHAR(50),
8     color VARCHAR(30)
9 );

```

Step 2: Insert sample data into the Boats table

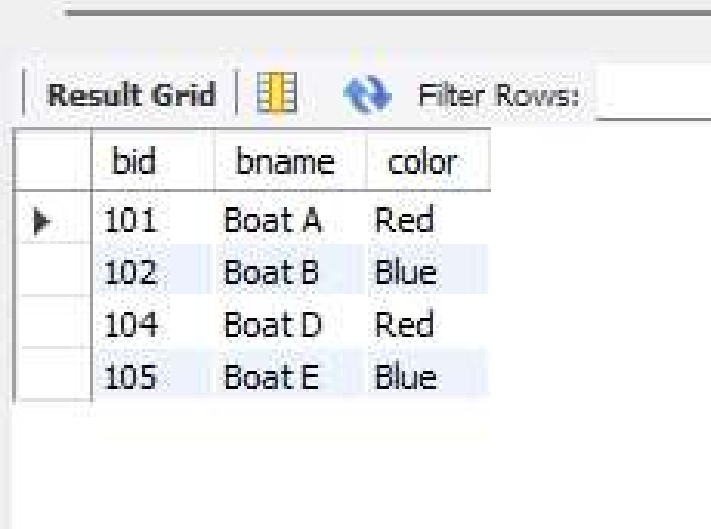
```
11  -- Step 2: Insert sample data into the Boats table
12 • INSERT INTO Boats (bid, bname, color)
13  VALUES (101, 'Boat A', 'Red'),
14          (102, 'Boat B', 'Blue'),
15          (103, 'Boat C', 'Green'),
16          (104, 'Boat D', 'Red'),
17          (105, 'Boat E', 'Blue');
```

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

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

Step 4: Query the RedBlueBoats view to see the results

```
25  -- Step 4: Query the view to display the results
26 • SELECT * FROM RedBlueBoats;
27  |
```



	bid	bname	color
▶	101	Boat A	Red
	102	Boat B	Blue
	104	Boat D	Red
	105	Boat E	Blue

7) Conclusion:

Triggers can automate tasks like logging data or performing calculations based on table events. Views simplify complex queries and enhance database security by limiting access to specific data. Both features are useful for efficient database management and ensuring data integrity.

Vidyadhar Sanjay Dinde (22610043)
TY IT (T4 Batch)
(Experiment No. 10)

1) Title:

Implementation of procedure in SQL.

2) Aim:

To create a table with student details and perform procedures on the table displaying various information.

3) Objectives:

- To create and manage a database for storing student information
- To insert student records into a relational table.
- To retrieve student data based on specific criteria using stored procedures.
- To ensure data integrity through primary keys and auto-incrementing columns. To practice database procedural programming with SQL.

4) Theory:

A **stored procedure** in SQL is a precompiled set of SQL statements stored in the database that can be executed as a single unit, allowing for improved modularity, reusability, and performance. They encapsulate business logic and simplify complex tasks by bundling multiple operations, which can be called repeatedly without rewriting the code. Stored procedures enhance security by restricting direct access to database tables and offer performance benefits since they are precompiled. They can accept input (IN), return output (OUT), or both (INOUT) parameters, allowing flexible interaction with the database. Procedures centralize database operations, making code easier to maintain and ensuring consistent execution across applications. By reducing network traffic and application code complexity, they also improve overall efficiency.

In SQL, stored procedures allow you to perform various operations on data. Here are some common operations that can be performed inside a procedure:

1. Data Retrieval (SELECT):

Stored procedures can be used to retrieve data from tables using the SELECT statement.

Example:

```
CREATE PROCEDURE GetAllEmployees() BEGIN SELECT * FROM employees; END;
```

2. Data Insertion (INSERT):

You can insert new rows into a table within a stored procedure.

Example:

```
CREATE PROCEDURE AddNewEmployee(IN emp_name VARCHAR(50), IN emp_salary  
DECIMAL(10,2))  
BEGIN  
    INSERT INTO employees (name, salary) VALUES (emp_name, emp_salary);  
END;
```

3. Data Update (UPDATE):

Procedures can update existing data in tables.

Example:

```
CREATE PROCEDURE UpdateEmployeeSalary(IN emp_id INT, IN new_salary  
DECIMAL(10,2))  
BEGIN  
    UPDATE employees SET salary = new_salary WHERE employee_id = emp_id;  
END;
```

4. Data Deletion (DELETE):

You can remove rows from tables within a stored procedure.

Example:

```
CREATE PROCEDURE DeleteEmployee(IN emp_id INT)  
BEGIN  
    DELETE FROM employees WHERE employee_id = emp_id;  
END;
```

5. Conditional Logic (IF/ELSE, CASE):

You can use conditional logic to execute different SQL statements based on certain conditions.

Example:

```

CREATE PROCEDURE UpdateSalaryBasedOnPerformance(IN emp_id INT, IN performance
CHAR(1))
BEGIN
    IF performance = 'A' THEN
        UPDATE employees SET salary = salary * 1.1 WHERE employee_id = emp_id;
    ELSE
        UPDATE employees SET salary = salary * 1.05 WHERE employee_id = emp_id;
    END IF;
END;

```

6. Looping (WHILE, REPEAT, FOR):

Stored procedures can include loops to perform repeated tasks.

7. Transaction Control (BEGIN TRANSACTION, COMMIT, ROLLBACK):

You can manage transactions within a procedure to ensure atomicity (all or nothing execution of a group of statements).

Benefits of Stored Procedures:

- **Performance:** Stored procedures are precompiled and optimized, reducing the processing overhead when they are executed multiple times.
- **Reusability:** They allow code reuse across different parts of an application.
- **Maintainability:** Changes made to a procedure are automatically applied wherever it is used.
- **Security:** Access to the database can be restricted through procedures, hiding the underlying logic and data structure.

Basic Syntax:

```

CREATE PROCEDURE procedure_name [ (parameter datatype [IN | OUT | INOUT], ...) ]
BEGIN -- SQL statements to be executed END;

```

5) Lab Practice

```

CREATE DATABASE T4;

```

```
USE T4;
```

```
CREATE TABLE t4_batch (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(100), marks INT, branch VARCHAR(50), city VARCHAR(50),  
    age INT  
);
```

```
INSERT INTO t4_batch (name, marks, branch, city, age) VALUES
```

```
('Vidyadhar', 85, 'Computer Science', 'Mumbai', 21),
```

```
('Manali', 90, 'Mechanical', 'Pune', 22),
```

```
('Aditya', 78, 'Electronics', 'Delhi', 20),
```

```
('Om', 88, 'Civil', 'Bangalore', 23),
```

```
('Vinit', 92, 'Computer Science', 'Chennai', 21),
```

```
('Rakesh', 75, 'Information Technology', 'Hyderabad', 24),
```

```
('Ameya', 95, 'Mechanical', 'Ahmedabad', 22),
```

```
('Aryan', 80, 'Electronics', 'Kolkata', 19),
```

```
('Rucha', 87, 'Civil', 'Jaipur', 21),
```

```
('Kranti', 70, 'Computer Science', 'Lucknow', 20),
```

```
('Eliza', 93, 'Information Technology', 'Surat', 22),
```

```
('Shreyash', 82, 'Mechanical', 'Nagpur', 23),
```

```
('Harsh', 89, 'Civil', 'Indore', 21),
```

```
('Shashank', 91, 'Electronics', 'Vadodara', 20),
```

```
('Anshul', 76, 'Computer Science', 'Visakhapatnam', 22);
```

```
DELIMITER //
```

```
CREATE PROCEDURE GetStudentsByBranch(IN student_branch VARCHAR(50)) BEGIN
```

```
    SELECT * FROM t4_batch WHERE branch = student_branch;    END  
//
```

```
DELIMITER ;
```


CALL GetStudentsByBranch('Computer Science');

7) Table:

	id	name	marks	branch	city	age
▶	1	Vidyadhar	85	Computer Science	Mumbai	21
	2	Manali	90	Mechanical	Pune	22
	3	Aditya	78	Electronics	Delhi	20
	4	Om	88	Civil	Bangalore	23
	5	Vinit	92	Computer Science	Chennai	21
	6	Rakesh	75	Information Technology	Hyderabad	24
	7	Ameya	95	Mechanical	Ahmedabad	22
	8	Aryan	80	Electronics	Kolkata	19
	9	Rucha	87	Civil	Jaipur	21
	10	Kranti	70	Computer Science	Lucknow	20
	11	Eliza	93	Information Technology	Surat	22
	12	Shreyash	82	Mechanical	Nagpur	23
	13	Harsh	89	Civil	Indore	21
	14	Shashank	91	Electronics	Vadodara	20
	15	Anshul	76	Computer Science	Visakhapat...	22
⋈	NULL	NULL	NULL	NULL	NULL	NULL

8) Output:

id	name	marks	branch	city	age
1	Vidyadhar	85	Computer Science	Mumbai	21
5	Vinit	92	Computer Science	Chennai	21
10	Kranti	70	Computer Science	Lucknow	20
15	Anshul	76	Computer Science	Visakhapatnam	22

9) Conclusion:

The SQL code efficiently creates a student database (T4) with an AUTO_INCREMENT primary key for unique identification. It includes a stored procedure, GetStudentsByBranch, to retrieve student records by branch, ensuring modularity and ease of data management.

Harsh Narule (22610044)

TY IT (T4 Batch)

(Experiment No. 11)

1) Title:

Study and Implement CRUD operations on MongoDB Database using MongoDB Compass.

2) Aim:

To Study and Implement CRUD (Create, Read, Update, Delete) operations on a MongoDB database using MongoDB Compass.

3) Objectives:

- To understand the structure and functionality of MongoDB.
- To perform CRUD operations using MongoDB Compass.
- To gain hands-on experience with MongoDB Compass for managing MongoDB databases.

4) Theory:

CRUD operations in MongoDB allow you to manage data by creating, reading, updating, and deleting documents in a database. **Create** operations insert new documents into collections using `insertOne()` for a single document or `insertMany()` for multiple. **Read** operations retrieve data, with `find()` being the most common method to query all or specific documents based on conditions. **Update** operations modify existing documents using methods like `updateOne()`, `updateMany()` where specific fields can be updated or the entire document replaced. Finally, **Delete** operations remove documents from collections using `deleteOne()` or `deleteMany()`, depending on the criteria. These operations enable flexible data management, making MongoDB ideal for handling dynamic schema and large datasets in applications.

CRUD Operations Overview:

- **Create:** Insert new documents.
- **Read:** Retrieve documents based on queries.
- **Update:** Modify existing documents.
- **Delete:** Remove documents from a collection.

Syntax:

1) Create:

```
db.createCollection("collection_name")
db.collection.insertOne({ field1: "value1", field2: "value2" })

db.collection.insertMany([ { field1: "value1", field2: "value2" }, {
field1: "value3", field2: "value4" } ])
```

2) Read:

```
db.collection.find()
```

3) Update:

```
db.collection.updateOne({ field1: "value1" }, { $set: { field2: "new_value" } })
```

```
db.collection.updateMany({ field1: "value1" }, { $set: { field2: "new_value" } })
```

4) Delete:

```
db.collection.deleteOne({field1:"value1"})
```

```
db.collection.deleteMany({ field1: "value1" })
```

5) Lab Practice - Example Queries:

1) Create:

```
db.createCollection("student")
```

```
db.student.insertOne(
```

```
{ "prn":44,"name":"harry","city":"ich","inst":"sied" })
```

```
> use local
< switched to db local
> db["wce"].find()
<
> use wce
< switched to db wce
> db.createCollection("student")
< { ok: 1 }
> db.student.insertOne({
  "prn":44,"name":"harry","city":"ich","inst":"sied"})
< {
  acknowledged: true,
  insertedId: ObjectId('67148b54996ef4aeaf423e57')
}
```

```
db.student.insertMany([  
  {"prn":45,"name":"shaz","city":"jsp","inst":"kps"},  
  {"prn":83,"name":"ssp","city":"mrj","inst":"sacd"},  
  {"prn":80,"name":"rsp","city":"tsg","inst":"sacd"},  
  {"prn":86,"name":"vigh","city":"tsg","inst":"kcp"},  
  {"prn":71,"name":"sat","city":"kol","inst":"new"},  
  {"prn":108,"name":"khade","city":"tsg","inst":"sacd"},  
  {"prn":7,"name":"todkar","city":"krd","inst":"sacd"}  
])
```

```
db.student.insertMany([  
  {"prn":45,"name":"shaz","city":"jsp","inst":"kps"},  
  {"prn":83,"name":"ssp","city":"mrj","inst":"sacd"},  
  {"prn":80,"name":"rsp","city":"tsg","inst":"sacd"},  
  {"prn":86,"name":"vigh","city":"tsg","inst":"kcp"},  
  {"prn":71,"name":"sat","city":"kol","inst":"new"},  
  {"prn":108,"name":"khade","city":"tsg","inst":"sacd"},  
  {"prn":7,"name":"todkar","city":"krd","inst":"sacd"}  
])
```

2) Read: db.student.find();

```
db.student.find()
{
  _id: ObjectId('67148b54996ef4aeaf423e57'),
  prn: 44,
  name: 'harry',
  city: 'ich',
  inst: 'sidd'
}
{
  _id: ObjectId('67148ca0996ef4aeaf423e58'),
  prn: 45,
  name: 'shaz',
  city: 'jsp',
  inst: 'kps'
}
{
  _id: ObjectId('67148ca0996ef4aeaf423e59'),
  prn: 83,
  name: 'ssp',
  city: 'mrj',
  inst: 'sidd'
}
```

```
{
  _id: ObjectId('67148ca0996ef4aeaf423e5a'),
  prn: 80,
  name: 'rsp',
  city: 'tsg',
  inst: 'sidd'
}
{
  _id: ObjectId('67148ca0996ef4aeaf423e5b'),
  prn: 86,
  name: 'vigh',
  city: 'tsg',
  inst: 'kcp'
}
{
  _id: ObjectId('67148ca0996ef4aeaf423e5c'),
  prn: 71,
  name: 'sat',
  city: 'kol',
  inst: 'new'
}
```

```
{
  _id: ObjectId('67148ca0996ef4aeaf423e5d'),
  prn: 108,
  name: 'khade',
  city: 'tsg',
  inst: 'sidd'
}
{
  _id: ObjectId('67148ca0996ef4aeaf423e5e'),
  prn: 7,
  name: 'todkar',
  city: 'krd',
  inst: 'sidd'
}
```

3) Update:

```
db.student.updateOne({"name":"sat"},{$set:{'inst':"nck"}})
```

```
db.student.updateOne({"name":"sat"},{$set:{'inst':"nck"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

```
db.student.find({"name":"sat"})
{
  _id: ObjectId('67148ca0996ef4aeaf423e5c'),
  prn: 71,
  name: 'sat',
  city: 'kol',
  inst: 'nck'
}
```

4) Delete:

```
db.student.deleteOne({"prn":7})
```

```
db.student.deleteOne({"prn":7})
{
  acknowledged: true,
  deletedCount: 1
}
```

6) Conclusion:

This experiment successfully demonstrated the implementation of CRUD operations on a MongoDB database using MongoDB Compass. The user-friendly interface of Compass allowed for efficient management of the database without manual coding. Each operation was performed successfully, providing a practical understanding of document management in MongoDB.

Shashank Shridhar Pujari (22610045)
TY IT (T4 Batch)
(Experiment No. 12)

1) Title:

Filtering Data Efficiently on MongoDB Database.

2) Aim:

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

3) Objectives:

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

4) Theory:

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

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

MongoDB Query Operators

MongoDB's powerful query language (MQL) enables users to perform various filtering operations to retrieve specific data. These queries often utilize filtering operators to refine and match documents based on conditions.

1. Comparison Operators

Comparison operators are used to compare field values to specific conditions. Here are the most commonly used ones:

- **\$eq** (Equal): Matches values that are equal to a specified value.

Query: { "age": { "\$eq": 25 } }

Matches documents where the age field is exactly 25.

- **\$ne** (Not Equal): Matches values that are not equal to a specified value.

Query: { "status": { "\$ne": "inactive" } }

Matches documents where the status is anything but "inactive."

- **\$gt** (Greater Than): Matches values that are greater than a specified value.

Query: { "age": { "\$gt": 30 } }

Matches documents where age is greater than 30.

- **\$gte** (Greater Than or Equal To): Matches values that are greater than or equal to a specified value.

Query: { "score": { "\$gte": 90 } }

Matches documents where the score is 90 or above.

- **\$lt** (Less Than): Matches values that are less than a specified value.

Query: { "age": { "\$lt": 18 } }

Matches documents where age is less than 18.

- **\$lte** (Less Than or Equal To): Matches values that are less than or equal to a specified value.

Query: { "price": { "\$lte": 50 } }

Matches documents where the price is 50 or below.

2. Logical Operators

Logical operators combine multiple query conditions using logical operations.

- **\$and**: Matches documents that satisfy all the specified conditions.

Query: { "\$and": [{ "age": { "\$gte": 18 } }, { "status": "active" }] }

Matches documents where age is 18 or above, and status is "active."

- **\$or**: Matches documents that satisfy at least one of the specified conditions.

Query: { "\$or": [{ "status": "active" }, { "age": { "\$lt": 18 } }] }

Matches documents where status is "active" or age is less than 18.

- **\$not**: Inverts the effect of a query expression.

Query: { "age": { "\$not": { "\$gt": 30 } } }

Matches documents where age is not greater than 30.

3. Element Operators

These operators deal with the presence or absence of fields in a document.

- **\$exists**: Matches documents that either contain or don't contain a field.

Query: { "nickname": { "\$exists": true } }

Matches documents where the "nickname" field exists.

- **\$type**: Matches documents where the value of a field is of a specified BSON type.

Query: { "age": { "\$type": "int" } }

Matches documents where the age field is an integer.

4. Array Operators

These operators are useful when working with arrays within documents.

- **\$in**: Matches documents where a field's value is found in an array of specified values.

Query: { "status": { "\$in": ["active", "pending"] } }

Matches documents where the status is either "active" or "pending."

- **\$all**: Matches arrays that contain all the specified elements.

Query: { "tags": { "\$all": ["mongodb", "nosql"] } }

Matches documents where the "tags" array contains both "mongodb" and "nosql."

- **\$size**: Matches arrays with a specific number of elements.

Query: { "items": { "\$size": 5 } }

Matches documents where the "items" array has exactly 5 elements.

5. Regular Expression Operators

MongoDB allows filtering documents using regular expressions for pattern matching.

- **\$regex**: Matches documents where the field's value matches a specified regular expression.

Query: { "name": { "\$regex": "^A" } }

Matches documents where the name starts with the letter "A."

6. Projection and \$match

In addition to the operators mentioned above, MongoDB allows filtering documents in aggregation pipelines using the **\$match** operator, which behaves similarly to queries in the find() method but is often used in data aggregation pipelines to narrow down results.

Query: {
 "\$match": { "status": "active" }
 }

5) Outputs:

1| Data:

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

2) Data filtering queries:

```
> var ageAbove28 = db.users.find({ age: { $gt: 28 } }).toArray();
print("Users aged above 28:", JSON.stringify(ageAbove28));
< Users aged above 28:
< [{"_id":"6714b6fc45bc1badd9019724","name":"Vivaan","age":30,"city":"Delhi"},{"_id":"6714b6fc45bc1badd9019727","na

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

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

6) Lab Practice - Example Queries:

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

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

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

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

```

var usersInDelhiAbove25 = db.users.find({
  $and: [
    { age: { $gt: 25 } },
    { city: "Delhi" }
  ]
}).toArray();
print("Users in Delhi aged above 25:", JSON.stringify(usersInDelhiAbove25));

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

```

7) Solve Lab Practice:

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

```

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

```

```

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

```

```

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

```

```

var usersInDelhiAbove25 = db.users.find({
  $and: [
    { age: { $gt: 25 } },
    { city: "Delhi" }
  ]
}).toArray();
print("Users in Delhi aged above 25:", JSON.stringify(usersInDelhiAbove25));
var uniqueCities = db.users.distinct("city");print("Unique cities:",
JSON.stringify(uniqueCities));

```

```

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

```

8) Conclusion:

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

Shreyash Suresh Pawar (22610083)
TY IT (T4 Batch)
Experiment No: 13

1) Title:

Working with Command Prompts to Create a Database Table in MariaDB

2) Aim:

The aim of this assignment is to understand the installation, configuration, and operation of MariaDB through command-line interactions. The objective includes creating a database, defining a table, and performing basic operations like insertion and querying data.

3) Objectives:

- To understand the basics of MariaDB.
- To install and configure MariaDB on Windows.
- To create a database and tables using command-line prompts.
- To insert data into tables and retrieve data using SQL queries.

4) Theory:

What is MariaDB?

MariaDB is an open-source relational database management system (RDBMS) that originated as a fork of MySQL. It supports SQL queries and offers several storage engines, security features, and performance enhancements. MariaDB is widely used for web applications and provides compatibility with MySQL.

Features of MariaDB

- Multi-threaded performance with excellent scalability.
- Open-source and community-driven.
- Supports stored procedures, views, and triggers.
- Advanced security with data encryption and authentication plugins.

Installation and Configuration of MariaDB on Windows

1. Download MariaDB:

- Visit the official MariaDB website: <https://mariadb.org/download/>
- Download the Windows installer package.

2. Installation Steps:

- Run the installer and follow the setup wizard.
- Choose the components (e.g., MariaDB Server, client tools).
- Configure the root user password during setup.
- Select the appropriate port (default is **3306**).
- Complete the installation and restart the system if needed.

3. Verify Installation:

- Open the Command Prompt (CMD) and type:
`mysql --version`
- If installed correctly, it will display the installed MariaDB version.

4. Starting MariaDB Service:

- Start the MariaDB server using:
net start mariadb

5) Solve Lab Practice:

Step 1: Open Command Prompt

- Search for CMD in the Start menu and open it.

Step 2: Log into MariaDB

```
mysql -u root -p
```

- Enter the password you set during installation to access the MariaDB console.

Step 3: Create a Database

```
CREATE DATABASE school;
```

- Output: Query OK, 1 row affected.

Step 4: Use the Database

```
USE school;
```

Step 5: Create a Table

```
CREATE TABLE students (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50),
    age INT,
    grade VARCHAR(5)
);
```

- Output: Query OK, 0 rows affected.

Step 6: Insert Data into the Table

```
INSERT INTO students (name, age, grade)
VALUES ('Shreyash', 18, 'A'),
       ('Shashank', 19, 'B'),
       ('Harsh', 20, 'A');
```

- Output: Query OK, 3 rows affected.

Step 7: Query the Data

```
SELECT * FROM students;
```

- Output:

```
+---+-----+-----+-----+
| id | name      | age | grade |
+---+-----+-----+-----+
| 1  | Shreyash  | 18  | A      |
| 2  | Shashank  | 19  | B      |
| 3  | Harsh     | 20  | A      |
+---+-----+-----+-----+
```

6) Outputs:

This section contains the screenshots and logs of the operations performed below. Include screenshots of successful operations (database creation, table creation, and data insertion).

```
Command Prompt (MariaDB) x + v
C:\Windows\System32>mysql -u root -p
Enter password: ***
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 3
Server version: 11.5.2-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> CREATE DATABASE school;
Query OK, 1 row affected (0.002 sec)

MariaDB [(none)]> USE school;
Database changed
MariaDB [school]> CREATE TABLE students (
  -> id INT PRIMARY KEY AUTO_INCREMENT,
  -> name VARCHAR(50),
  -> age INT,
  -> grade VARCHAR(5)
  -> );
Query OK, 0 rows affected (0.025 sec)

MariaDB [school]> INSERT INTO students (name, age, grade)
  -> VALUES ('Shreyash', 18, 'A'),
  -> ('Shashank', 19, 'B'),
  -> ('Harsh', 20, 'A');
Query OK, 3 rows affected (0.077 sec)
Records: 3 Duplicates: 0 Warnings: 0

MariaDB [school]> SELECT * FROM students;
+----+-----+-----+-----+
| id | name  | age  | grade |
+----+-----+-----+-----+
| 1  | Shreyash | 18  | A     |
| 2  | Shashank | 19  | B     |
| 3  | Harsh   | 20  | A     |
+----+-----+-----+-----+
3 rows in set (0.005 sec)

MariaDB [school]>
```

7) Conclusion:

In this assignment, we installed and configured MariaDB on a Windows system, creating a database and table, inserting records, and retrieving data via the command prompt. This hands-on experience showcased the power of MariaDB's SQL interface for effective database management.

Ameya Unchagaonkar (22610089)
TY IT (T4 Batch)
(Experiment No. 14)

1) Title:

To perform CRUD operation on MariaDB

2) Aim:

To understand and implement CRUD (Create, Read, Update, Delete) operations on a MariaDB database through practical application using a **students** table.

3) Objectives:

- To create a structured table to store student information.
- To insert bulk data into the table efficiently.
- To retrieve and display specific student records.
- To update student records based on specific criteria.
- To delete unnecessary student records to maintain data integrity.

4) Theory:

CRUD operations are the backbone of any relational database management system (RDBMS), like MariaDB. These four operations allow users to perform essential database interactions, ensuring efficient data manipulation and integrity.

1. Create:

The CREATE operation is responsible for inserting new records into a table. In SQL, the INSERT INTO statement is used to add new rows to the database. Before inserting data, a table must be defined using the CREATE TABLE statement. For example, creating a table for storing student data involves defining the structure (columns such as id, name, age, etc.) and ensuring data types are specified.

2. Read

(Retrieve):

The READ operation, often referred to as Retrieve or Select, involves querying the database to fetch data based on certain conditions or criteria. The SQL SELECT statement is used for this operation, allowing users to retrieve specific columns, filtered rows, or all the records from a table. The ability to filter, sort, and limit the result set makes this operation highly flexible. For example, fetching student names and grades for analysis can be done with this operation.

3. Update:

The UPDATE operation allows modifications to existing records. The UPDATE SQL statement enables users to change data in specific columns based on predefined conditions. This operation is essential when corrections need to be made, or when records need to reflect the latest information. For instance, if a student's grade needs to be updated, the UPDATE command can modify that specific student's record without affecting others.

4. Delete:

The DELETE operation is used to remove records from a table when they are no longer necessary or need to be replaced. The DELETE statement removes rows based on a condition; alternatively, all rows can be deleted if no condition is specified. Proper use

of this command is crucial for maintaining database cleanliness and performance, as unnecessary or outdated records can slow down queries and take up storage space.

5) Outputs:

Table Creation: Successfully created the table with the following query:

```
CREATE TABLE students (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(100) NOT NULL,  
    age INT,  
    grade VARCHAR(10),  
    email VARCHAR(100)  
);
```

```
MariaDB [students]> CREATE TABLE students (  
->     id INT AUTO_INCREMENT PRIMARY KEY,  
->     name VARCHAR(100) NOT NULL,  
->     age INT,  
->     grade VARCHAR(10),  
->     email VARCHAR(100)  
-> );  
Query OK, 0 rows affected (0.004 sec)
```

Table Data Insertion:

```
INSERT INTO students (name, age, grade, email) VALUES  
(  
'Alice Johnson', 20, 'A', 'alice.johnson@example.com'),  
(  
'Bob Smith', 22, 'B', 'bob.smith@example.com'),  
(  
'Charlie Brown', 21, 'C', 'charlie.brown@example.com'),  
(  
'Diana Prince', 19, 'A', 'diana.prince@example.com'),  
(  
'Ethan Hunt', 23, 'B', 'ethan.hunt@example.com');
```

```
MariaDB [students]> INSERT INTO students (name, age, grade, email) VALUES  
-> ('Alice Johnson', 20, 'A', 'alice.johnson@example.com'),  
-> ('Bob Smith', 22, 'B', 'bob.smith@example.com'),  
-> ('Charlie Brown', 21, 'C', 'charlie.brown@example.com'),  
-> ('Diana Prince', 19, 'A', 'diana.prince@example.com'),  
-> ('Ethan Hunt', 23, 'B', 'ethan.hunt@example.com');  
Query OK, 5 rows affected (0.010 sec)  
Records: 5 Duplicates: 0 Warnings: 0
```

Table Updation:

You can update specific records using the UPDATE statement. Here's how to update the age of a student:

1. UPDATE students

SET age = 21

WHERE name = 'Alice Johnson';

```
MariaDB [students]> UPDATE students
-> SET age = 21
-> WHERE name = 'Alice Johnson';
Query OK, 1 row affected (0.008 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

2. UPDATE students

SET grade = 'A+', email = 'alice.new@example.com'

WHERE name = 'Alice Johnson';

```
MariaDB [students]> UPDATE students
-> SET grade = 'A+', email = 'alice.new@example.com'
-> WHERE name = 'Alice Johnson';
Query OK, 1 row affected (0.009 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

Table Data Deletion:

To delete records, use the DELETE statement. For example, to delete a student by name:

1. DELETE FROM students

WHERE name = 'Bob Smith';

```
MariaDB [students]> DELETE FROM students
-> WHERE name = 'Bob Smith';
Query OK, 1 row affected (0.008 sec)
```

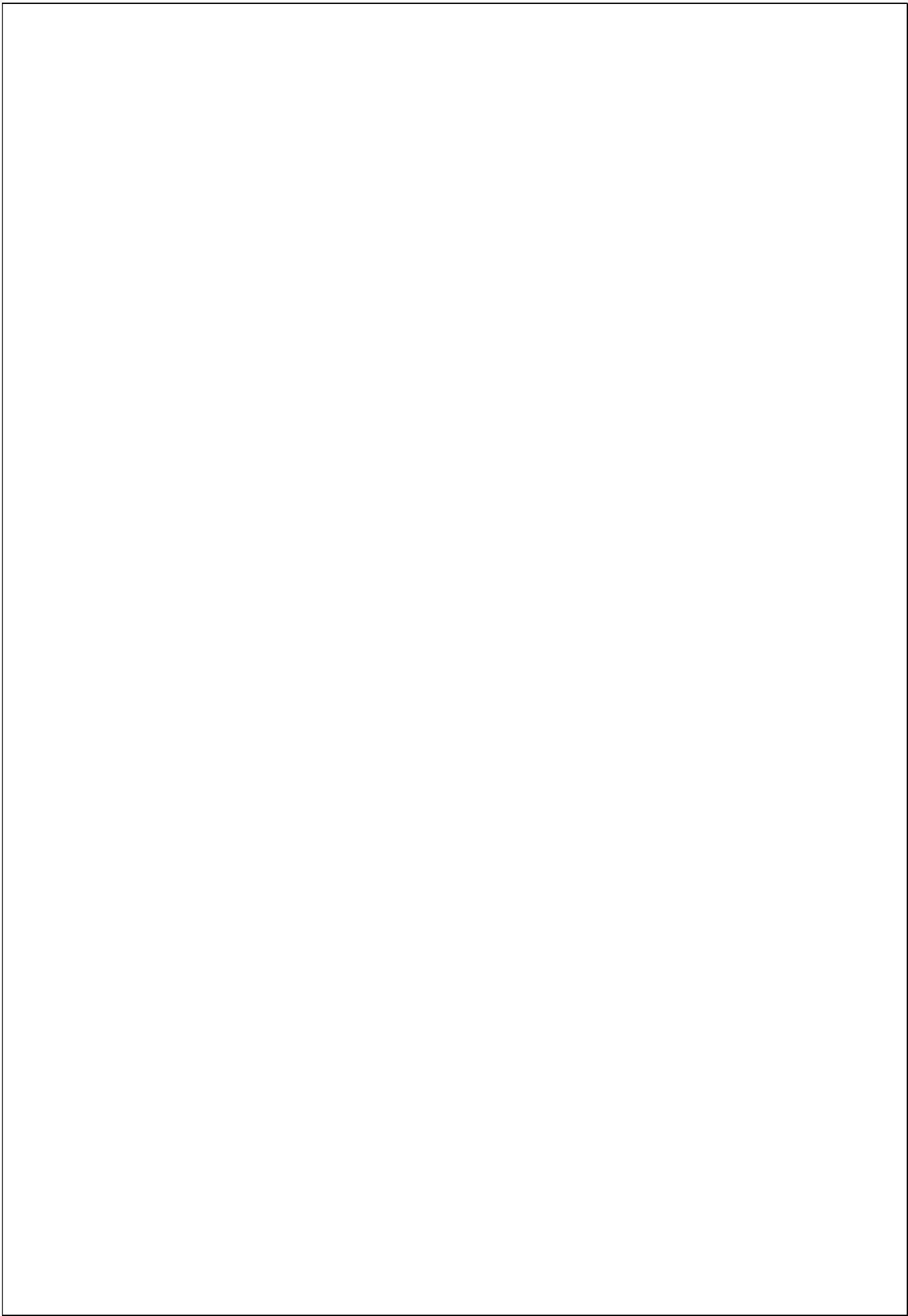
2. DELETE FROM students

WHERE age < 20;

```
MariaDB [students]> DELETE FROM students
-> WHERE age < 20;
Query OK, 1 row affected (0.009 sec)
```

6) Conclusion:

The experiment successfully showcased SQL clauses for data manipulation and retrieval in a relational database by creating an employee records database. It enhanced operational efficiency through various queries and indexing techniques, reinforcing theoretical concepts from class.



Aryan Atul Babar (22610090)
TY IT (T4 Batch)
Experiment No: 15

1. Title:

Implementation of JDBC/ODBC for Database Connectivity

2. Aim:

To implement and demonstrate the functionality of JDBC for establishing a connection between a Java application and a database, performing basic CRUD operations.

3. Objectives:

- To understand the Java Database Connectivity (JDBC) API.
- To establish a connection with databases like Oracle and MySQL using JDBC.
- To perform SQL queries (Insert, Update, Select, Delete) using JDBC.
- To gain hands-on experience with database connectivity in Java programs.

4. Theory:

What is JDBC?

JDBC (Java Database Connectivity) is an API that allows Java programs to interact with databases. It provides methods for querying and updating data in relational databases. The key components of JDBC include the following:

- **Driver:** Controls the communication with the database server.
- **DriverManager:** Manages a list of database drivers and establishes connections.
- **Connection:** A session between the Java application and the database.
- **Statement:** Used for executing static SQL queries.
- **ResultSet:** A table of data representing a database result set, obtained by executing a query.

Prerequisites:

Before establishing a JDBC connection, the following prerequisites must be met:

1. **JDK (Java Development Kit):** Ensure that JDK is installed on the system to compile and run Java programs.
2. **Database Software (e.g., MySQL, Oracle):** A running instance of the database server (like MySQL or Oracle) should be installed and accessible.
3. **JDBC Driver for the Database:** For each database, there is a corresponding JDBC driver. For MySQL, the mysql-connector-java.jar must be included in the project. Similarly, for Oracle, use the ojdbc.jar.
4. **IDE/Editor (e.g., IntelliJ, Eclipse):** To write and run Java programs.
5. **SQL Database Credentials:** Have valid credentials (username, password) and the database's URL, which includes the hostname, port, and database name.

Establishing JDBC Connection:

The process of connecting to a database using JDBC involves:

1. **Importing the Database Library:** Use `import java.sql.*` to import JDBC classes.
2. **Loading the Driver:** Use `Class.forName()` or `DriverManager.registerDriver()` to register the driver.
3. **Establishing a Connection:** Use `DriverManager.getConnection(url, user, password)` to form a connection to the database.
4. **Executing Queries:** Use a `Statement` object to execute SQL queries, and use `ResultSet` to retrieve query results.
5. **Closing the Connection:** Close the `Connection` and `Statement` to free up resources.

JAVA Code to Create Database:

```
// Importing necessary classes for JDBC
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class CreateDB {

    // JDBC URL, username, and password of MySQL server
    static final String DB_URL = "jdbc:mysql://localhost/";
    static final String USER = "root";
    static final String PASS = "5796";

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

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

            // Step 2: Open a connection
            System.out.println("Connecting to database...");
            conn = DriverManager.getConnection(DB_URL, USER, PASS);

            // Step 3: Create a statement object
            System.out.println("Creating database...");
            stmt = conn.createStatement();

            // Step 4: Execute a query to create a database
            String sql = "CREATE DATABASE ADA";
            stmt.executeUpdate(sql);
            System.out.println("Database created successfully...");

        } catch (SQLException e) {
            // Handle SQL exceptions
            e.printStackTrace();
        }
    }
}
```

```

    } catch (ClassNotFoundException e) {
        // Handle ClassNotFoundException
        e.printStackTrace();
    } finally {
        // Step 5: Close resources
        try {
            if (stmt != null) stmt.close();
            if (conn != null) conn.close();
        } catch (SQLException se) {
            se.printStackTrace();
        }
    }
}
}

```

5. Terminal OUTPUT:

```

PS C:\Users\babar\Downloads\SQL\JDBC> c:; cd
vuyszdyhmhcdtmc6soq75.argfile' 'CreateDB'
Connecting to database...
Creating database...
Database created successfully...
PS C:\Users\babar\Downloads\SQL\JDBC> 

```

6. Conclusion:

JDBC allows Java applications to interact with relational databases, enabling CRUD operations through a standard API. The process involves loading drivers, establishing connections, executing SQL queries, and processing results. JDBC is essential for developing Java-based database-driven applications.