DATABASE MANAGEMENT SYSTEM LAB

ICT-251

University School of Information Communication and Technology

Guru Gobind Singh Indraprastha University,

Dwarka, Sector 16C,New Delhi

PRACTICAL FILE

Submitted By : Aadesh Kumar Rai
Enrollment No: 05916401523
Course : B.Tech (IT)
Semester : THIRD
YEAR : SECOND

INDEX

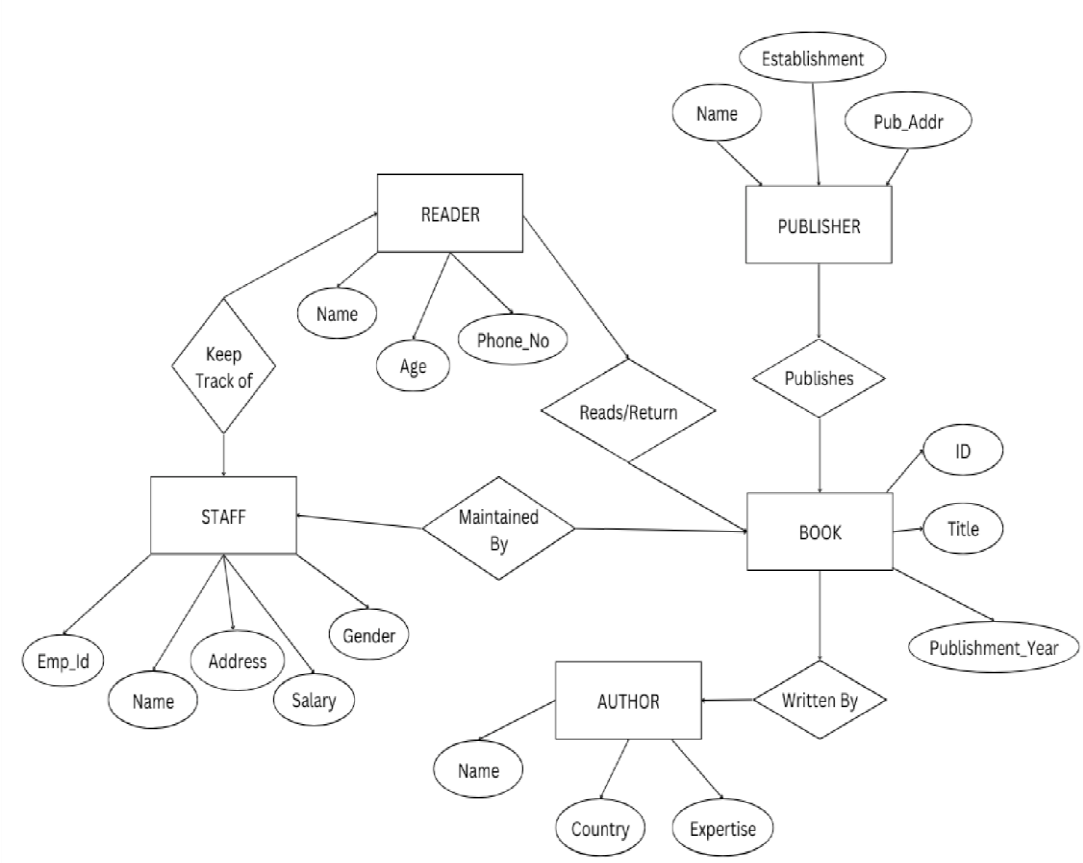
SNO	EXPERIMENTS	DATE	SIGNATURE
1.	Draw E-R Diagram for Library Management System		
2.	Draw E-R Diagram for Banking Management System		
3.	Write SQL queries to implement DDL commands.		
4.	Write SQL queries to implement DML commands.		
5.	Write SQL queries to implement TCL commands		
6.	Write SQL queries to implement Key constraints.		
7.	Write SQL queries to implement Views.		
8.	Write SQL queries to SELECT data using SET, UNION, INTERSECTION and MINUS operations.		
9.	Write SQL queries to implement different types of operators.		
10.	Write SQL queries to implement Joins.		
11.	Write SQL queries using different types of functions.		
12.	Study and implementation of Group By , Having, Order By		
13.	Write PL/SQL code block to implement Triggers.		

EXPERIMENT : 1

AIM : Draw E-R Diagram for Library Management System

The objective of this ER DIAGRAM representation is to provide a detailed understanding of the Entity-Relationship (ER) Diagram for a Library Management System (LMS). This diagram serves as a blueprint for the database design, illustrating the relationships among various entities involved in the system. The goal is to facilitate efficient management of library resources, streamline operations, and enhance user experience. The ER Diagram for the Library Management System encompasses all essential components, including books, library members, staff, transactions, and inventory management. It aims to capture the intricate relationships between these entities, ensuring comprehensive data integrity and smooth functionality of the system.

ER DIAGRAM REPRESENTATION OF LIBRARY MANAGEMENT SYSTEM



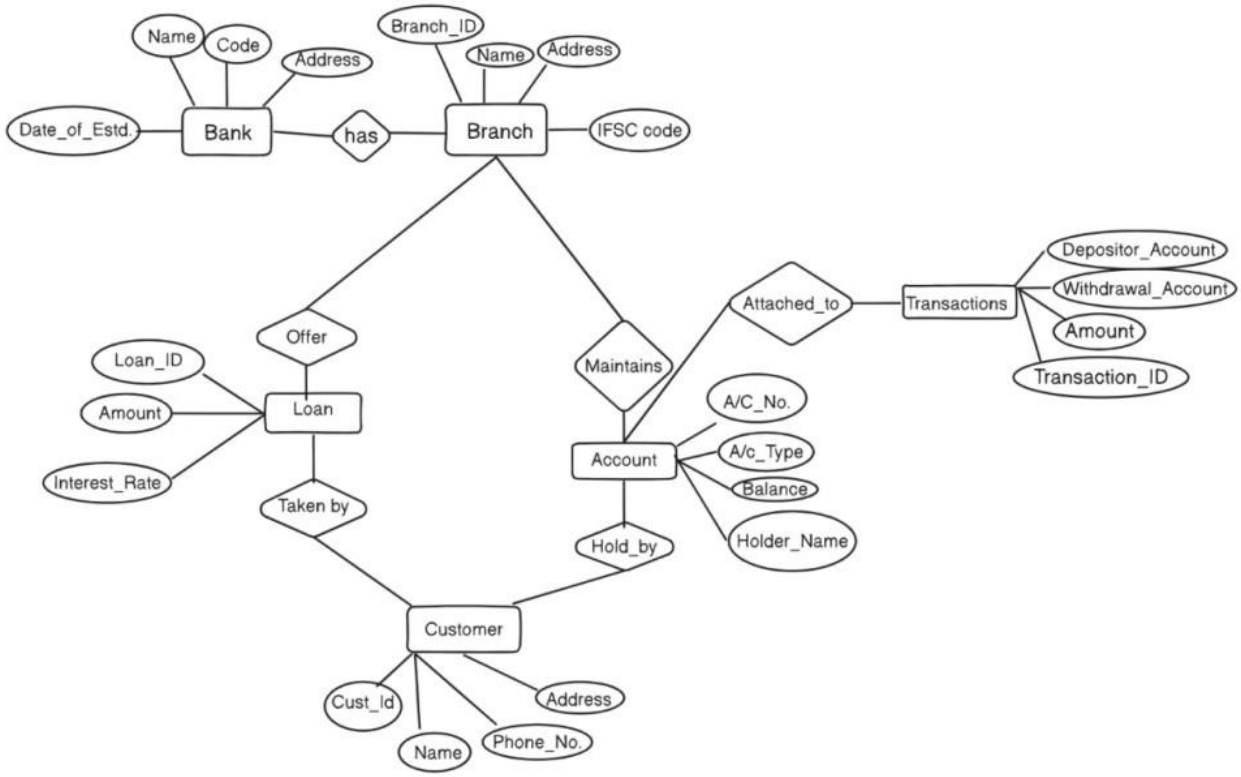
Conclusion: The ER diagram provides a structured overview of a library management system, showing how each entity is related to others.

EXPERIMENT : **2**

AIM : Draw E-R Diagram for Banking Management System

The objective of this ER DIAGRAM representation is to provide a detailed understanding of the Entity-Relationship (ER) Diagram for a Banking Management System (BMS). This diagram serves as a blueprint for the database design, illustrating the relationships among various entities involved in the system. The goal is to facilitate efficient management of Banking resources, streamline operations, and enhance user experience. The ER Diagram for the Banking Management System encompasses all essential components entities, attributes, and relationships that represent the various processes and data flow within the system. Key entities typically include Customers, Accounts, Loans, Branches, Transactions, Employees, and Departments.

ER DIAGRAM REPRESENTATION OF BANKING MANAGEMENT SYSTEM



Conclusion: The ER diagram provides a structured overview of a Banking management system, showing how each entity is related to others.

EXPERIMENT : **3**

AIM : Implementation of DDL commands

Introduction

Data Definition Language (DDL) Commands are essential in defining and managing the structure of database objects. This experiment demonstrates the use of key DDL Commands:

CREATE TABLE, ALIER TABLE, and DROP TABLE.

Theory

DDL commands are used to create, alter, and drop database structures Such as tables, indexes, and views.

- CREATE DATABASE is used to create a new database ● CREATE TABLE is used to create a new table in the database.
- ALTER TABLE modifies an existing table by adding. deleting. or modifying columns and constraints.
- DROP TABLE removes a table and all its data from the database permanently.

Procedure

1. Create Database:

Query: CREATE DATABASE USICT_DB;

```
mysql> CREATE DATABASE USICT_DB;
Query OK, 1 row affected (0.03 sec)
```

2.Use Database :

Query: USE USICT_DB;

```
mysql> USE USICT_DB;
Database changed
```

3.Create Table:

Query: CREATE TABLE EMPLOYEE(
-> EMP_ID INT PRIMARY KEY,
-> EMP_NAME VARCHAR(50) NOT NULL,
-> EMP_PNO BIGINT,
-> EMP_SALARY INT);

```
mysql> CREATE TABLE EMPLOYEE(
    -> EMP_ID INT PRIMARY KEY,
    -> EMP_NAME VARCHAR(50) NOT NULL,
    -> EMP_PNO BIGINT,
    -> EMP_SALARY INT);
Query OK, 0 rows affected (0.11 sec)
```

Query: CREATE TABLE DEPARTMENT(
-> DEPT_ID INT PRIMARY KEY,
-> DEPT_NAME VARCHAR(50) NOT NULL,
-> EMP_ID INT,
-> FOREIGN KEY (EMP_ID) REFERENCES EMPLOYEE(EMP_ID));

```
mysql> CREATE TABLE DEPARTMENT(
    -> DEPT_ID INT PRIMARY KEY,
    -> DEPT_NAME VARCHAR(50) NOT NULL,
    -> EMP_ID INT,
    -> FOREIGN KEY (EMP_ID) REFERENCES EMPLOYEE(EMP_ID));
Query OK, 0 rows affected (0.03 sec)
```

4. Alter table:

Query: ALTER TABLE EMPLOYEE ADD EMP_EMAIL VARCHAR(70);

```
mysql> ALTER TABLE EMPLOYEE ADD EMP_EMAIL VARCHAR(70);
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

5. Drop table:

Query: DROP TABLE DEPARTMENT;

SELECT * FROM DEPARTMENT;

```
mysql> DROP TABLE department;
Query OK, 0 rows affected (0.02 sec)
```

EXPERIMENT : 4

AIM: Implementation of DML commands

Introduction

Data Manipulation Language (DML) commands are crucial for managing and manipulating the data stored within a database. This experiment demonstrates the use of key DML commands: INSERT, UPDATE, and DELETE. These commands facilitate the insertion of new records, modification of existing data, and removal of unnecessary entries, allowing for efficient data management and retrieval within a database system.

Theory

- INSERT is used to add new records into a table in the database.
- UPDATE modifies existing records in a table based on specified conditions.
- DELETE removes records from a table, permanently eliminating them from the database.

Procedure:

1. Insert Data:

Queries:

```
INSERT INTO EMPLOYEE (EMP_ID, EMP_NAME, EMP_PNO,
   -> EMP_SALARY, EMP_EMAIL)
   -> VALUES
   -> (101,"EMP-1",110001,50000,"emp.usict@gmail.com"),
   -> (102,"EMP-2",110002,55000,"emp2.usict@gmail.com"),
   -> (103,"EMP-3",110003,25000,"emp3.usict@gmail.com"),
   -> (104,"EMP-4",110004,50000,"emp4.usict@gmail.com"),
   -> (105,"EMP-5",110005,25000,"emp5.usict@gmail.com"),
   -> (106,"EMP-6",110006,50000,"emp6.usict@gmail.com");
```

```
mysql> INSERT INTO EMPLOYEE (EMP_ID, EMP_NAME, EMP_PNO,
    -> EMP_SALARY, EMP_EMAIL)
    -> VALUES
    -> (101,"EMP-1",110001,50000,"emp.usict@gmail.com"),
    -> (102,"EMP-2",110002,55000,"emp2.usict@gmail.com"),
    -> (103,"EMP-3",110003,25000,"emp3.usict@gmail.com"),
    -> (104,"EMP-4",110004,50000,"emp4.usict@gmail.com"),
    -> (105,"EMP-5",110005,25000,"emp5.usict@gmail.com"),
    -> (106,"EMP-6",110006,50000,"emp6.usict@gmail.com");
Query OK, 6 rows affected (0.02 sec)
Records: 6  Duplicates: 0  Warnings: 0
```

```
INSERT INTO DEPARTMENT (DEPT_ID, DEPT_NAME,
   -> EMP_ID)  VALUES
   -> (201,'Human Resources',101),
   -> (202,'Finance',102),
   -> (203,'Engineering',103),
   -> (204,'Sales',104),
   -> (205,'Marketing',105),
   -> (206,'Support',106);
SELECT * FROM EMPLOYEE;
SELECT * FROM DEPARTMENT;
```

```
mysql> INSERT INTO DEPARTMENT (DEPT_ID, DEPT_NAME,
    -> EMP_ID)  VALUES
    -> (201,'Human Resources',101),
    -> (202,'Finance',102),
    -> (203,'Engineering',103),
    -> (204,'Sales',104),
    -> (205,'Marketing',105),
    -> (206,'Support',106);
Query OK, 6 rows affected (0.01 sec)
Records: 6  Duplicates: 0  Warnings: 0
```

```
mysql> SELECT * FROM EMPLOYEE;
+--------+----------+---------+------------+----------------------+
| EMP_ID | EMP_NAME | EMP_PNO | EMP_SALARY | EMP_EMAIL            |
+--------+----------+---------+------------+----------------------+
|    101 | EMP-1    |  110001 |      50000 | emp.usict@gmail.com  |
|    102 | EMP-2    |  110002 |      55000 | emp2.usict@gmail.com |
|    103 | EMP-3    |  110003 |      25000 | emp3.usict@gmail.com |
|    104 | EMP-4    |  110004 |      50000 | emp4.usict@gmail.com |
|    105 | EMP-5    |  110005 |      25000 | emp5.usict@gmail.com |
|    106 | EMP-6    |  110006 |      50000 | emp6.usict@gmail.com |
+--------+----------+---------+------------+----------------------+
6 rows in set (0.00 sec)
```

```
+---------+-----------------+--------+
| DEPT_ID | DEPT_NAME       | EMP_ID |
+---------+-----------------+--------+
|     201 | Human Resources |    101 |
|     202 | Finance         |    102 |
|     203 | Engineering     |    103 |
|     204 | Sales           |    104 |
|     205 | Marketing       |    105 |
|     206 | Support         |    106 |
+---------+-----------------+--------+
6 rows in set (0.00 sec)
```

2. UPDATE DATA

Query:

```
UPDATE EMPLOYEE
SET EMP_SALARY=10000
WHERE EMP_ID=102;
SELECT * FROM EMPLOYEE;
```

```
mysql> UPDATE EMPLOYEE
    -> SET EMP_SALARY=10000
    -> WHERE EMP_ID=102;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM EMPLOYEE;
+--------+----------+---------+------------+----------------------+
| EMP_ID | EMP_NAME | EMP_PNO | EMP_SALARY | EMP_EMAIL            |
+--------+----------+---------+------------+----------------------+
|    101 | EMP-1    |  110001 |      50000 | emp.usict@gmail.com  |
|    102 | EMP-2    |  110002 |      10000 | emp2.usict@gmail.com |
|    103 | EMP-3    |  110003 |      25000 | emp3.usict@gmail.com |
|    104 | EMP-4    |  110004 |      50000 | emp4.usict@gmail.com |
|    105 | EMP-5    |  110005 |      25000 | emp5.usict@gmail.com |
|    106 | EMP-6    |  110006 |      50000 | emp6.usict@gmail.com |
+--------+----------+---------+------------+----------------------+
6 rows in set (0.00 sec)
```

3. DELETE DATA

QUERY:

DELETE FROM DEPARTMENT WHERE DEPT_ID=205;

DELETE FROM EMPLOYEE WHERE EMP_ID=105;

SELECT * FROM EMPLOYEE;

```
mysql> DELETE FROM DEPARTMENT WHERE DEPT_ID=205;
Query OK, 1 row affected (0.00 sec)

mysql> DELETE FROM EMPLOYEE WHERE EMP_ID=105;
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM EMPLOYEE;
+--------+----------+---------+------------+----------------------+
| EMP_ID | EMP_NAME | EMP_PNO | EMP_SALARY | EMP_EMAIL            |
+--------+----------+---------+------------+----------------------+
|    101 | EMP-1    |  110001 |      50000 | emp.usict@gmail.com  |
|    102 | EMP-2    |  110002 |      10000 | emp2.usict@gmail.com |
|    103 | EMP-3    |  110003 |      25000 | emp3.usict@gmail.com |
|    104 | EMP-4    |  110004 |      50000 | emp4.usict@gmail.com |
|    106 | EMP-6    |  110006 |      50000 | emp6.usict@gmail.com |
+--------+----------+---------+------------+----------------------+
5 rows in set (0.00 sec)
```

Conclusion

The DML commands were executed to insert, update, and delete records in the Employee table, illustrating the essential operations required for managing and manipulating data within the database. These operations enable efficient data handling, ensuring the accuracy and relevance of information stored in the database.

Result

- The Employee table was populated with new records.
- The salary of an employee were successfully updated.
- The record for the employee was deleted from the table

EXPERIMENT : 5

AIM: Implementation of TCL commands

Introduction

Transaction Control Language (TCL) commands are essential for managing transactions in a database. These commands ensure the integrity and consistency of data by allowing users to define and control transactions. Key TCL commands include COMMIT, which saves all changes made during the current transaction; ROLLBACK, which undoes changes made in the transaction if errors occur; and SAVEPOINT, which sets a point within a transaction to which you can later roll back. Together, these commands provide robust control over data operations, facilitating reliable and consistent database management.

Theory

TCL commands are used to manage transactions in a database.

- The COMMIT command saves all changes made during the current transaction.
- The ROLLBACK command undoes any changes made since the last commit, restoring the database to a previous state.
- The SAVEPOINT command sets a point within a transaction, allowing partial rollbacks to that point if necessary.

Procedure

Procedure

1.START TRANSACTION

```
mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)
```

QUERY:

START TRANSACTION;

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.01 sec)
```

2. Savepoint

QUERY:

```
INSERT INTO EMPLOYEE (EMP_ID, EMP_NAME, EMP_PNO,
EMP_SALARY,EMP_EMAIL) VALUES (107, 'EMP',110007,
10000,'emp7.usict.gmail.com');
UPDATE EMPLOYEE SET EMP_NAME='Emp-7' WHERE EMP_ID=107;
SAVEPOINT savepoint1;
UPDATE EMPLOYEE SET EMP_EMAIL='emp1.usict@gmail.com' WHERE EMP_ID=101;
SAVEPOINT savepoint2;
DELETE FROM DEPARTMENT WHERE EMP_ID = 206;
```

```
mysql> INSERT INTO EMPLOYEE (EMP_ID, EMP_NAME, EMP_PNO,
    -> EMP_SALARY,EMP_EMAIL) VALUES (107, 'EMP',110007,
    -> 10000,'emp7.usict.gmail.com');
Query OK, 1 row affected (0.01 sec)

mysql> UPDATE EMPLOYEE SET EMP_NAME='Emp-7' WHERE EMP_ID=107;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SAVEPOINT savepoint1;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE EMPLOYEE SET EMP_EMAIL='emp1.usict@gmail.com' WHERE
    -> EMP_ID=101;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SAVEPOINT savepoint2;
Query OK, 0 rows affected (0.00 sec)

mysql> DELETE FROM DEPARTMENT WHERE EMP_ID = 206;
Query OK, 0 rows affected (0.00 sec)
```

3. ROLLBACK

QUERY:
ROLLBACK TO savepoint1;

```
mysql> ROLLBACK TO savepoint1;
Query OK, 0 rows affected (0.00 sec)
```

4.COMMIT

QUERY: COMMIT;
SELECT * FROM EMPLOYEE;

```
mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)
```

Conclusion

The TCL commands were executed to manage transactions in the Employee table. The COMMIT command was used to save changes made during a transaction, ensuring data integrity. The ROLLBACK command successfully undid changes made to an employee record, restoring the previous state. Additionally, the SAVEPOINT command allowed for partial rollbacks, providing flexibility in managing data modifications.

Result

- The new employee record was successfully inserted into the Employee table and committed to the database.
- Changes made to the employee's NAME were rolled back, restoring the previous value.
- The use of savepoints allowed for controlled transaction management, enabling selective retention of changes in the database.

EXPERIMENT : 6

AIM: Implementation of key constraints

Introduction

Key constraints are essential for maintaining data integrity and establishing relationships between tables in a database. They ensure that data adheres to specific rules, preventing duplication and ensuring that relationships are accurately represented. This guide will focus on implementing key constraints using Data Definition Language (DDL) commands in SQL, specifically through the use of primary keys and foreign keys.

Theory

Key constraints are critical elements in database design. The two primary types of key constraints are:

- **Primary Key:** A primary key uniquely identifies each record in a table. It cannot contain NULL values and must be unique across all records. Implementing a primary key helps enforce the integrity of the data.

- **Foreign Key**: A foreign key creates a relationship between two tables by referencing the primary key of another table. It allows for the establishment of relationships and helps maintain referential integrity, ensuring that a foreign key value must match an existing primary key value or be NULL.

Procedure

Step 1: Define the Key Constraints

1. **Primary Key**: Identify the column(s) that will serve as the primary key, ensuring uniqueness and non-nullability.
2. **Foreign Key:** If applicable, determine any foreign keys that will reference primary keys in other tables.

Query: CREATE TABLE EMPLOYEE(
-> EMP_ID INT PRIMARY KEY,
-> EMP_NAME VARCHAR(50) NOT NULL,
-> EMP_PNO BIGINT,
-> EMP_SALARY INT);

Query: CREATE TABLE DEPARTMENT(
-> DEPT_ID INT PRIMARY KEY,
-> DEPT_NAME VARCHAR(50) NOT NULL,
-> EMP_ID INT,
-> FOREIGN KEY (EMP_ID) REFERENCES EMPLOYEE(EMP_ID));

```
mysql> CREATE TABLE EMPLOYEE(
    -> EMP_ID INT PRIMARY KEY,
    -> EMP_NAME VARCHAR(50) NOT NULL,
    -> EMP_PNO BIGINT,
    -> EMP_SALARY INT);
Query OK, 0 rows affected (0.11 sec)
```

```
mysql> CREATE TABLE DEPARTMENT(
    -> DEPT_ID INT PRIMARY KEY,
    -> DEPT_NAME VARCHAR(50) NOT NULL,
    -> EMP_ID INT,
    -> FOREIGN KEY (EMP_ID) REFERENCES EMPLOYEE(EMP_ID));
Query OK, 0 rows affected (0.03 sec)
```

SELECT d.DEPT_NAME, e.EMP_NAME, e.EMP_EMAIL
FROM Department d
JOIN Employee e ON d.EMP_ID = e.EMP_ID;

```
mysql> SELECT d.DEPT_NAME, e.EMP_NAME, e.EMP_EMAIL
    -> FROM Department d
    -> JOIN Employee e ON d.EMP_ID = e.EMP_ID;
+-----------------+----------+---------------------+
| DEPT_NAME       | EMP_NAME | EMP_EMAIL           |
+-----------------+----------+---------------------+
| Human Resources | EMP-1    | emp.usict@gmail.com  |
| Finance         | EMP-2    | emp2.usict@gmail.com |
| Engineering     | EMP-3    | emp3.usict@gmail.com |
| Sales           | EMP-4    | emp4.usict@gmail.com |
| Support         | EMP-6    | emp6.usict@gmail.com |
+-----------------+----------+---------------------+
5 rows in set (0.00 sec)
```

Conclusion:

The DDL commands were executed to create and manage the **Employee** and **Department** tables, establishing key constraints between them. This process demonstrated the fundamental operations required to enforce data integrity and uphold the relational structure within the database schema. By implementing primary and foreign key constraints, we ensured accurate relationships between the entities, thereby enhancing the reliability and organization of the data.

Result:

1. **Creation of Tables with Key Constraints**: The `Employee` table was successfully created with a primary key (`EMP_ID`), and the `Department` table was also created with a primary key (`DEPT_ID`). A foreign key constraint was established, linking `EMP_ID` in the `Department` table to `EMP_ID` in the `Employee` table, ensuring that each department is associated with a valid employee.
2. **Data Integrity Enforcement**: Any attempts to insert records with non-existent `EMPLOYEE_ID` values in the `Department` table are prevented due to the foreign key constraint. This enforces referential integrity, ensuring only valid relationships between employees and departments are allowed.
3. **Successful Data Retrieval**: After populating both tables with data, a join query retrieves a comprehensive list of departments along with their respective employee details. This demonstrates the established relationships and ensures that only valid, related entries are displayed.

EXPERIMENT : 7

AIM: Write SQL queries to implement Views. Introduction

Introduction

Views in SQL are virtual tables that represent the result of a stored query. They are an essential component of database management as they simplify complex queries, enhance security, and present data in a specific format without altering the underlying tables. By creating views, users can focus on relevant data, facilitate easier access, and enforce security by restricting user access to specific data subsets. This experiment demonstrates the creation, modification, and management of views within a database, highlighting their practical applications in data retrieval and organization.

Theory

Definition:

A **view** is a virtual table that dynamically displays data based on the results of a query involving one or more tables, without physically storing the data.

Simplification:

Views simplify complex queries, allowing users to access relevant information more easily without understanding the details of the underlying tables.

Security:

Views enhance data security by restricting user access to specific data subsets, allowing permissions to be granted on views rather than directly on tables.

Abstraction:

Views provide a layer of abstraction, enabling users to interact with data without needing knowledge of the database schema's complexities.

Updatability:

Some views are updatable, meaning that data within them can be modified, depending on the complexity of the view and the database system's capabilities.

Flexibility:

Views can be created, modified, or dropped easily, offering flexibility in data presentation as requirements evolve.

Performance:

Although views improve data organization and security, they may impact performance; thus, optimizing the view definition is essential.

Procedure

1. **Create a View**: Use the `CREATE VIEW` statement to define a new view based on a query.

Query:

CREATE VIEW EmployeeDetails AS
SELECT e.EMP_ID, e.EMP_NAME, e.EMP_SALARY, d.DEPT_NAME
FROM Employee e
JOIN Department d ON e.EMP_ID = d.EMP_ID;

```
mysql> CREATE VIEW EmployeeDetails AS
    -> SELECT e.EMP_ID, e.EMP_NAME, e.EMP_SALARY, d.DEPT_NAME
    -> FROM Employee e
    -> JOIN Department d ON e.EMP_ID = d.EMP_ID;
Query OK, 0 rows affected (0.03 sec)

mysql> SELECT * FROM EmployeeDetails;
```

2. **Query the View**: Retrieve data from the view as if it were a table.

Query:

SELECT * FROM EmployeeDetails;

```
mysql> SELECT * FROM EmployeeDetails;
+--------+----------+------------+-----------------+
| EMP_ID | EMP_NAME | EMP_SALARY | DEPT_NAME       |
+--------+----------+------------+-----------------+
|    101 | EMP-1    |      50000 | Human Resources |
|    102 | EMP-2    |      10000 | Finance         |
|    103 | EMP-3    |      25000 | Engineering     |
|    104 | EMP-4    |      50000 | Sales           |
|    106 | EMP-6    |      50000 | Support         |
+--------+----------+------------+-----------------+
5 rows in set (0.01 sec)
```

3.**Modify the View**: Use the CREATE OR REPLACE VIEW statement to modify an existing view.

Query:

CREATE OR REPLACE VIEW EmployeeDetails AS
SELECT e.EMP_ID, e.EMP_NAME, e.EMP_SALARY, e.EMP_EMAIL, d.DEPT_NAME
FROM Employee e
JOIN Department d ON e.EMP_ID = d.EMP_ID;

```
mysql> CREATE OR REPLACE VIEW EmployeeDetails AS
    -> SELECT e.EMP_ID, e.EMP_NAME, e.EMP_SALARY, e.EMP_EMAIL, d.DEPT_NAME
    -> FROM Employee e
    -> JOIN Department d ON e.EMP_ID = d.EMP_ID;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM EmployeeDetails;
+--------+----------+------------+----------------------+-----------------+
| EMP_ID | EMP_NAME | EMP_SALARY | EMP_EMAIL            | DEPT_NAME       |
+--------+----------+------------+----------------------+-----------------+
|    101 | EMP-1    |      50000 | emp.usict@gmail.com  | Human Resources |
|    102 | EMP-2    |      10000 | emp2.usict@gmail.com | Finance         |
|    103 | EMP-3    |      25000 | emp3.usict@gmail.com | Engineering     |
|    104 | EMP-4    |      50000 | emp4.usict@gmail.com | Sales           |
|    106 | EMP-6    |      50000 | emp6.usict@gmail.com | Support         |
+--------+----------+------------+----------------------+-----------------+
5 rows in set (0.01 sec)
```

4. Drop the View: Use the `DROP VIEW` statement to remove a view that is no longer needed.
Query:

DROP VIEW EmployeeDetails;

```
mysql> DROP VIEW EmployeeDetails;
Query OK, 0 rows affected (0.02 sec)
```

5. Check Existing Views: Query the information schema to list all views in the database.
Query:

SELECT TABLE_NAME FROM information_schema.VIEWS WHERE TABLE_SCHEMA =”usict_db”;

```
mysql> SELECT TABLE_NAME FROM information_schema.VIEWS WHERE TABLE_SCHEMA = 'usict_db';
Empty set (0.01 sec)
```

Conclusion

This procedure outlines the steps to create, query, modify, and drop views in SQL, providing an efficient means for data management and organization. Views improve data accessibility and security while adding flexibility to data presentation.

Result

1. **View Created**: The EmployeeDetails view was successfully created to display relevant employee and department information.
2. **Data Retrieved:** Querying the view returned data on employees and their associated departments.
3. **View Modified:** Updates made to the view, such as adding columns, were reflected immediately.
4. **View Dropped:** The EmployeeDetails view was successfully removed from the database.
5. **Existing Views Listed**: A query of the information schema confirmed that the view was deleted.

EXPERIMENT : **8**

AIM: Write SQL queries to SELECT data using SET, UNION, INTERSECTION and MINUS operations.

Introduction

SQL set operations are used to combine the results of two or more SELECT queries. These operations include:

1. UNION: Combines the results of two SELECT queries and returns unique rows.

2. INTERSECT: Returns only the rows that are present in both SELECT query results.

3. EXCEPT (or MINUS): Returns rows from the first SELECT query that are not present in the second query.

Procedure

1. Create Tables:

We'll create the employees_2023 and employees_2024 tables with two columns: employee_id (primary key) and employee_name.

QUERY:

```
CREATE TABLE employees_2023 (
employee_id INT PRIMARY KEY,
employee_name VARCHAR(50)
);
CREATE TABLE employees_2024 (
employee_id INT PRIMARY KEY,
employee_name VARCHAR(50)
);
```

```
mysql> CREATE TABLE employees_2023 (
    -> employee_id INT PRIMARY KEY,
    -> employee_name VARCHAR(50)
    -> );
Query OK, 0 rows affected (0.12 sec)
```

```
mysql> CREATE TABLE employees_2024 (
    -> employee_id INT PRIMARY KEY,
    -> employee_name VARCHAR(50)
    -> );
Query OK, 0 rows affected (0.02 sec)
```

2. Insert Dummy Data :

QUERY:

INSERT INTO employees_2023 (employee_id, employee_name) VALUES
(101, 'EMP-1'),
(102, 'EMP-2'),
(103, 'EMP-3'),
(104, ‘EMP-4’),
(105, “EMP-5”);

INSERT INTO employees_2024 (employee_id, employee_name) VALUES
(101, 'EMP-1'),
(103, 'EMP-3'),
(105, “EMP-5”),
(106, “EMP-6”),
(107, “EMP-7”);

```
mysql> INSERT INTO employees_2023 (employee_id, employee_name) VALUES
    -> (101, "EMP-1"),
    -> (102, "EMP-2"),
    -> (103, "EMP-3"),
    -> (104, "EMP-4"),
    -> (105, "EMP-5");
Query OK, 5 rows affected (0.07 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
mysql> INSERT INTO employees_2024 (employee_id, employee_name) VALUES
    -> (101, "EMP-1"),
    -> (103, "EMP-3"),
    -> (105, "EMP-5"),
    -> (106, "EMP-6"),
    -> (107, "EMP-7");
Query OK, 5 rows affected (0.31 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

3. Test Queries

UNION: All Unique Employees Across Both Years

QUERY:

SELECT employee_name FROM employees_2023
UNION
SELECT employee_name FROM employees_2024;

```
mysql> SELECT employee_name FROM employees_2023
    -> UNION
    -> SELECT employee_name FROM employees_2024;
+---------------+
| employee_name |
+---------------+
| EMP-1         |
| EMP-2         |
| EMP-3         |
| EMP-4         |
| EMP-5         |
| EMP-6         |
| EMP-7         |
+---------------+
```

INTERSECT: Employees Who Were in Both 2023 and 2024

QUERY:

SELECT employee_name FROM employees_2023

INTERSECT

SELECT employee_name FROM employees_2024;

```
mysql> SELECT employee_name FROM employees_2023
    -> INTERSECT
    -> SELECT employee_name FROM employees_2024;
+---------------+
| employee_name |
+---------------+
| EMP-1         |
| EMP-3         |
| EMP-5         |
+---------------+
3 rows in set (0.01 sec)
```

MINUS (or EXCEPT): Employees Who Left After 2023

QUERY:

SELECT employee_name FROM employees_2023

EXCEPT

SELECT employee_name FROM employees_2024;

```
mysql> SELECT employee_name FROM employees_2023
    -> EXCEPT
    -> SELECT employee_name FROM employees_2024;
+---------------+
| employee_name |
+---------------+
| EMP-2         |
| EMP-4         |
+---------------+
2 rows in set (0.00 sec)
```

EXPERIMENT : 9

AIM: Write SQL queries to implement different types of operators.

Introduction to SQL Operators
In SQL, operators are symbols or keywords that perform operations on data values. They are the backbone of any query and help filter, combine, compare, and manipulate data. Operators in SQL can be broadly classified into four types:

1. **Arithmetic Operators**: Used for basic arithmetic operations.
2. **Comparison Operators**: Used to compare values.
3. **Logical Operators**: Used to combine multiple conditions.
4. **Set Operators**: Used to combine results from multiple SELECT queries.

1. Arithmetic Operators
Theory: Arithmetic operators perform mathematical operations on numerical data. These operators include addition, subtraction, multiplication, and division.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus(remainder)

Example
Suppose we have a table products with columns product_id, product_name, and price.

QUERIES:

```
SELECT EMP_ID,EMP_NAME,
        EMP_SALARY*0.30 AS BONUS
FROM employee;
```

```
+--------+----------+----------+
| EMP_ID | EMP_NAME | BONUS    |
+--------+----------+----------+
|    101 | EMP-1    | 15000.00 |
|    102 | EMP-2    |  3000.00 |
|    103 | EMP-3    |  7500.00 |
|    104 | EMP-4    | 15000.00 |
|    106 | EMP-6    | 15000.00 |
|    107 | Emp-7    |  3000.00 |
+--------+----------+----------+
6 rows in set (0.07 sec)
```

2. Comparison Operators

Theory: Comparison operators are used to compare values in SQL. These operators return a Boolean value (TRUE or FALSE) depending on the condition's evaluation.

Operator	Description
=	Equal to
!= or <>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
BETWEEN	Between a range of values
IN	Matches any value in a list
LIKE	Matches a pattern (wildcard)

Example

Suppose we have a table employees with columns employee_id, employee_name, and salary.

Queries:

```
SELECT emp_name,emp_salary
FROM employee
WHERE emp_salary > 15000;
```

```
mysql> SELECT emp_name,emp_salary
    -> FROM employee
    -> WHERE emp_salary > 15000;
+----------+------------+
| emp_name | emp_salary |
+----------+------------+
| EMP-1    |      50000 |
| EMP-3    |      25000 |
| EMP-4    |      50000 |
| EMP-6    |      50000 |
+----------+------------+
4 rows in set (0.00 sec)
```

-- Query to find employees whose names start with 'A'

```
SELECT emp_name, emp_salary
FROM employee
WHERE emp_name LIKE '%mp%';
```

```
mysql> SELECT emp_name, emp_salary
    -> FROM employee
    -> WHERE emp_name LIKE '%mp%';
+----------+------------+
| emp_name | emp_salary |
+----------+------------+
| EMP-1    |      50000 |
| EMP-2    |      10000 |
| EMP-3    |      25000 |
| EMP-4    |      50000 |
| EMP-6    |      50000 |
| Emp-7    |      10000 |
+----------+------------+
6 rows in set (0.04 sec)
```

3. Logical Operators

Theory: Logical operators are used to combine multiple conditions in the WHERE clause of an SQL query. They allow complex filtering conditions with the use of AND, OR, and NOT.

Operator	Description
AND	Returns true if both conditions are true
OR	Returns true if at least one condition is true
NOT	Negates a condition

Example

Using the employees table, let's find employees with complex conditions.

Queries:

SELECT emp_name, emp_salary

FROM employee

WHERE emp_salary > 15000 AND emp_id>103;

```
mysql> SELECT emp_name, emp_salary
    -> FROM employee
    -> WHERE emp_salary > 15000 AND emp_id>103;
+----------+------------+
| emp_name | emp_salary |
+----------+------------+
| EMP-4    |      50000 |
| EMP-6    |      50000 |
+----------+------------+
2 rows in set (0.04 sec)
```

SELECT emp_name, emp_salary

FROM employee

WHERE emp_salary < 40000 OR emp_salary > 45000;

```
mysql> SELECT emp_name, emp_salary
    -> FROM employee
    -> WHERE emp_salary < 40000 OR emp_salary > 45000;
+----------+------------+
| emp_name | emp_salary |
+----------+------------+
| EMP-1    |      50000 |
| EMP-2    |      10000 |
| EMP-3    |      25000 |
| EMP-4    |      50000 |
| EMP-6    |      50000 |
| Emp-7    |      10000 |
+----------+------------+
6 rows in set (0.00 sec)
```

EXPERIMENT : 10

AIM: Write SQL queries to implement Joins.

Introduction to SQL Joins

In SQL, JOIN operations allow us to retrieve related data stored across multiple tables. Joins are essential in relational databases, where data is organized into tables with relationships between them. By using joins, we can efficiently combine data based on common fields, enabling complex queries and insights.
SQL supports several types of joins:

1. **INNER JOIN:** Returns records that have matching values in both tables.
2. **LEFT JOIN (or LEFT OUTER JOIN):** Returns all records from the left table and matching records from the right table; if there is no match, NULL values are returned.
3. **RIGHT JOIN (or RIGHT OUTER JOIN):** Returns all records from the right table and matching records from the left table; if there is no match, NULL values are returned.
4. **FULL OUTER JOIN:** Returns all records when there is a match in either left or right table; if there is no match, NULL values are returned.
5. **CROSS JOIN:** Returns the Cartesian product of two tables, pairing each row from the first table with every row from the second table.

For the following examples, we'll use two tables: departments and employees. departments

EMPLOYEE TABLE

```
+--------+----------+---------+------------+----------------------+
| EMP_ID | EMP_NAME | EMP_PNO | EMP_SALARY | EMP_EMAIL            |
+--------+----------+---------+------------+----------------------+
|    101 | EMP-1    |  110001 |      50000 | emp.usict@gmail.com  |
|    102 | EMP-2    |  110002 |      10000 | emp2.usict@gmail.com |
|    103 | EMP-3    |  110003 |      25000 | emp3.usict@gmail.com |
|    104 | EMP-4    |  110004 |      50000 | emp4.usict@gmail.com |
|    106 | EMP-6    |  110006 |      50000 | emp6.usict@gmail.com |
|    107 | Emp-7    |  110007 |      10000 | emp7.usict.gmail.com |
+--------+----------+---------+------------+----------------------+
6 rows in set (0.00 sec)
```

DEPARTMENT TABLE

```
mysql> SELECT * FROM DEPARTMENT;
+---------+-----------------+--------+
| DEPT_ID | DEPT_NAME       | EMP_ID |
+---------+-----------------+--------+
|     201 | Human Resources |    101 |
|     202 | Finance         |    102 |
|     203 | Engineering     |    103 |
|     204 | Sales           |    104 |
|     206 | Support         |    106 |
+---------+-----------------+--------+
5 rows in set (0.06 sec)
```

1. INNER JOIN

Theory: An INNER JOIN retrieves only the rows where there is a match in both tables. Rows without a match in either table are excluded from the result.
Procedure:

- Identify the columns that relate the two tables (e.g., department_id in this case).
- Use INNER JOIN with ON to specify the matching column.

Query:

```
SELECT e.emp_id, e.emp_name, d.dept_name
FROM employee e
INNER JOIN department d
ON e.emp_id = d.emp_id;
```

```
mysql> SELECT e.emp_id, e.emp_name, d.dept_name
    -> FROM employee e
    -> INNER JOIN department d
    -> ON e.emp_id = d.emp_id;
+--------+----------+-----------------+
| emp_id | emp_name | dept_name       |
+--------+----------+-----------------+
|    101 | EMP-1    | Human Resources |
|    102 | EMP-2    | Finance         |
|    103 | EMP-3    | Engineering     |
|    104 | EMP-4    | Sales           |
|    106 | EMP-6    | Support         |
+--------+----------+-----------------+
5 rows in set (0.07 sec)
```

2. LEFT JOIN (or LEFT OUTER JOIN)

Theory: A LEFT JOIN returns all rows from the left table (employees), and the matching rows from the right table (departments). If there is no match, NULL values are returned for the columns from the right table.

Procedure:

- Specify the LEFT JOIN with the related column(s) using ON.

Query

```
SELECT e.emp_id, e.emp_name, d.dept_name
FROM employee e
LEFT JOIN department d
ON e.emp_id = d.emp_id;
```

```
mysql> SELECT e.emp_id, e.emp_name, d.dept_name
    -> FROM employee e
    -> LEFT JOIN department d
    -> ON e.emp_id = d.emp_id;
+--------+----------+-----------------+
| emp_id | emp_name | dept_name       |
+--------+----------+-----------------+
|    101 | EMP-1    | Human Resources |
|    102 | EMP-2    | Finance         |
|    103 | EMP-3    | Engineering     |
|    104 | EMP-4    | Sales           |
|    106 | EMP-6    | Support         |
|    107 | Emp-7    | NULL            |
+--------+----------+-----------------+
6 rows in set (0.04 sec)
```

3. RIGHT JOIN (or RIGHT OUTER JOIN)

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

Procedure:

- Specify the RIGHT JOIN with the related column(s) using ON.

Query

```
SELECT e.emp_id, e.emp_name, d.dept_name
FROM employee e
RIGHT JOIN department d
ON e.emp_id = d.emp_id;
```

```
mysql> SELECT e.emp_id, e.emp_name, d.dept_name
    -> FROM employee e
    -> RIGHT JOIN department d
    -> ON e.emp_id = d.emp_id;
+--------+----------+-----------------+
| emp_id | emp_name | dept_name       |
+--------+----------+-----------------+
|    101 | EMP-1    | Human Resources |
|    102 | EMP-2    | Finance         |
|    103 | EMP-3    | Engineering     |
|    104 | EMP-4    | Sales           |
|    106 | EMP-6    | Support         |
+--------+----------+-----------------+
5 rows in set (0.00 sec)
```

4. FULL OUTER JOIN

Theory: A **FULL OUTER JOIN** returns all rows when there is a match in either table. Rows from both tables that do not match are included in the result set, with NULLs for columns where there is no match.

Procedure:

- The first part of the query (LEFT JOIN) retrieves all employees, including those without a matching department, by returning NULL in department_name for unmatched rows.
- The second part of the query (RIGHT JOIN) retrieves all departments, including those without any matching employees, by returning NULL in employee_id and employee_name for unmatched rows.
- The UNION operator combines the results from both parts, effectively simulating a **FULL OUTER JOIN**.

Query

```
SELECT e.emp_id, e.emp_name, d.dept_name FROM employee e LEFT JOIN department
d ON e.emp_id =
d.emp_id UNION SELECT e.emp_id, e.emp_name,
d.dept_name FROM employee e RIGHT JOIN department d ON
e.emp_id = d.emp_id;
```

```
mysql> SELECT e.emp_id, e.emp_name, d.dept_name FROM employee e LEFT JOIN department d ON e.emp_id =
    -> d.emp_id UNION SELECT e.emp_id, e.emp_name,
    -> d.dept_name FROM employee e RIGHT JOIN department d ON
    -> e.emp_id = d.emp_id;
+--------+----------+-----------------+
| emp_id | emp_name | dept_name       |
+--------+----------+-----------------+
|    101 | EMP-1    | Human Resources |
|    102 | EMP-2    | Finance         |
|    103 | EMP-3    | Engineering     |
|    104 | EMP-4    | Sales           |
|    106 | EMP-6    | Support         |
|    107 | Emp-7    | NULL            |
+--------+----------+-----------------+
6 rows in set (0.04 sec)
```

5. CROSS JOIN

Theory: A **CROSS JOIN** returns the Cartesian product of two tables, which means each row from the first table is paired with every row from the second table. It results in a large result set if both tables contain many rows.

Procedure:

- Simply specify CROSS JOIN between the two tables.

Query

```
SELECT e.employee_name, d.department_name
FROM employees e
CROSS JOIN departments d;
```

```
mysql> SELECT e.emp_name, d.dept_name
    -> FROM employee e
    -> CROSS JOIN department d;
+----------+-----------------+
| emp_name | dept_name       |
+----------+-----------------+
| Emp-7    | Human Resources |
| EMP-6    | Human Resources |
| EMP-4    | Human Resources |
| EMP-3    | Human Resources |
| EMP-2    | Human Resources |
| EMP-1    | Human Resources |
| Emp-7    | Finance         |
| EMP-6    | Finance         |
| EMP-4    | Finance         |
| EMP-3    | Finance         |
| EMP-2    | Finance         |
| EMP-1    | Finance         |
| Emp-7    | Engineering     |
| EMP-6    | Engineering     |
| EMP-4    | Engineering     |
| EMP-3    | Engineering     |
| EMP-2    | Engineering     |
| EMP-1    | Engineering     |
| Emp-7    | Sales           |
| EMP-6    | Sales           |
| EMP-4    | Sales           |
| EMP-3    | Sales           |
| EMP-2    | Sales           |
| EMP-1    | Sales           |
| Emp-7    | Support         |
| EMP-6    | Support         |
| EMP-4    | Support         |
| EMP-3    | Support         |
| EMP-2    | Support         |
| EMP-1    | Support         |
+----------+-----------------+
30 rows in set (0.04 sec)
```

EXPERIMENT : 11

AIM: Write SQL queries using different types of functions.

Introduction

SQL functions are integral to data manipulation and analysis, allowing users to perform calculations and transformations on data directly within their queries. Functions can be categorized into various types, including **aggregate functions**, which operate on a set of values and return a single summary value **(e.g., `SUM`, `AVG`, `COUNT`); scalar functions**, which operate on individual values and return a single value **(e.g., `UPPER`, `LOWER`, `ROUND`);** and **string functions,** which are specifically designed for manipulating string data **(e.g., `CONCAT`, `SUBSTRING`).**

Using these functions effectively enables users to derive meaningful insights from their datasets, perform data validation, and enhance reporting capabilities. This experiment will focus on writing SQL queries that utilize different types of functions, demonstrating how they can be applied to enhance data analysis and reporting within a relational database context.

Theory

1. **Aggregate Functions:** These functions, such as `COUNT()`, `SUM()`,`AVG()`, `MAX()`, and `MIN()`, operate on multiple rows of data, returning a single summary value. They are essential for generating insights, such as total counts, averages, and extreme values.
2. **Scalar Functions:** These functions process individual values and return a single result. Examples include `UPPER()` and `LOWER()` for string manipulation, and `ROUND()` for numerical formatting. They are useful for standardizing data and formatting outputs.
3. **String Functions:** Functions like `CONCAT()`, `SUBSTRING()`, and `LENGTH()` allow users to manipulate string data, enabling tasks such as combining names or extracting specific parts of a string.
4. **Date Functions:** Functions such as `DATEDIFF()` and `DATE_FORMAT()` facilitate operations on date and time values, helping to analyze durations and present dates in user-friendly formats.

Procedure

Implement Aggregate Functions:

1. COUNT(): Use the COUNT function to determine the number of employees in a specific department.

```
mysql> SELECT DepartmentID, COUNT(*) AS EmployeeCount
    -> FROM employee.hospitalemployee
    -> GROUP BY DepartmentID;
+--------------+---------------+
| DepartmentID | EmployeeCount |
+--------------+---------------+
|            1 |             6 |
|            2 |             6 |
|            3 |             5 |
+--------------+---------------+
3 rows in set (0.01 sec)
```

2. **SUM()**: Calculate the total salary of all employees in the organization.

```
mysql> SELECT SUM(Salary) AS TotalSalary
    -> FROM employee.hospitalemployee;
+--------------+
| TotalSalary  |
+--------------+
|   1785000.00 |
+--------------+
1 row in set (0.00 sec)
```

3. AVG(): Find the average salary of employees in the Nursing department.

```
mysql> SELECT AVG(Salary) AS AverageSalary
    -> FROM employee.hospitalemployee
    -> WHERE DepartmentID = 3;  -- Assuming 3 is the ID for Nursing
+---------------+
| AverageSalary |
+---------------+
| 111000.000000 |
+---------------+
1 row in set (0.00 sec)
```

4. MAX() and MIN(): Retrieve the highest and lowest salaries among all employees.

```
mysql> SELECT MAX(Salary) AS HighestSalary, MIN(Salary) AS LowestSalary
    -> FROM employee.hospitalemployee;
+---------------+--------------+
| HighestSalary | LowestSalary |
+---------------+--------------+
|     150000.00 |     50000.00 |
+---------------+--------------+
1 row in set (0.00 sec)
```

Implement Scalar Functions:

1. UPPER() and LOWER(): Standardize employee names to uppercase.

```
mysql> SELECT UPPER(FirstName) AS UpperFirstName, LOWER(LastName) AS LowerLastName
    -> FROM employee.hospitalemployee;
+----------------+---------------+
| UpperFirstName | LowerLastName |
+----------------+---------------+
| RAHUL          | sharma        |
| ANANYA         | patel         |
| ADITI          | verma         |
| VIKRAM         | singh         |
| PRIYA          | kumar         |
| ROHAN          | reddy         |
| SNEHA          | mehta         |
| RAJESH         | iyer          |
| NEHA           | choudhury     |
| KARAN          | nair          |
| SANYA          | malhotra      |
| AARAV          | bhatia        |
| DIYA           | joshi         |
| VANI           | ghosh         |
| KABIR          | kapoor        |
| NEHA           | verma         |
| RAHUL          | kumar         |
+----------------+---------------+
17 rows in set (0.00 sec)
```

2. ROUND(): Round the salary to the nearest thousand for easier reporting.

```
mysql> SELECT FirstName, LastName, ROUND(Salary, -3) AS RoundedSalary
    -> FROM employee.hospitalemployee;
+-----------+-----------+---------------+
| FirstName | LastName  | RoundedSalary |
+-----------+-----------+---------------+
| Rahul     | Sharma    |        120000 |
| Ananya    | Patel     |        110000 |
| Aditi     | Verma     |         75000 |
| Vikram    | Singh     |        130000 |
| Priya     | Kumar     |         85000 |
| Rohan     | Reddy     |         95000 |
| Sneha     | Mehta     |         65000 |
| Rajesh    | Iyer      |        140000 |
| Neha      | Choudhury |        135000 |
| Karan     | Nair      |        125000 |
```

3. NOW(): Insert the current timestamp into a log table.

```
mysql> SELECT CONCAT(FirstName, ' ', LastName) AS FullName
    -> FROM employee.hospitalemployee;
+----------------+
| FullName       |
+----------------+
| Rahul Sharma   |
| Ananya Patel   |
```

Implement String Functions:

1. CONCAT(): Combine first and last names into a full name.

```
mysql> SELECT CONCAT(FirstName, ' ', LastName) AS FullName
    -> FROM employee.hospitalemployee;
+----------------+
| FullName       |
+----------------+
| Rahul Sharma   |
| Ananya Patel   |
```

2. SUBSTRING(): Extract the first three letters of employee last names.

```
mysql> SELECT SUBSTRING(LastName, 1, 3) AS ShortLastName
    -> FROM employee.hospitalemployee;
+---------------+
| ShortLastName |
+---------------+
| Sha           |
| Pat           |
| Ver           |
| Sin           |
| Kum           |
| Red           |
```

3. LENGTH(): Find the length of each employee's last name.

```
mysql> SELECT LastName, LENGTH(LastName) AS NameLength
    -> FROM employee.hospitalemployee;
+------------+-------------+
| LastName   | NameLength  |
+------------+-------------+
| Sharma     |           6 |
| Patel      |           5 |
| Verma      |           5 |
| Singh      |           5 |
| Kumar      |           5 |
| Reddy      |           5 |
| Mehta      |           5 |
| Iyer       |           4 |
| Choudhury  |           9 |
| Nair       |           4 |
```

Implement Date Functions:

1. DATEDIFF(): Calculate the number of days since an employee was hired.

```
mysql> SELECT FirstName, LastName, DATEDIFF(NOW(), HireDate) AS DaysSinceHired
    -> FROM employee.hospitalemployee;
+-----------+-----------+----------------+
| FirstName | LastName  | DaysSinceHired |
+-----------+-----------+----------------+
| Rahul     | Sharma    |           1738 |
| Ananya    | Patel     |           1318 |
| Aditi     | Verma     |           1947 |
| Vikram    | Singh     |            966 |
| Priya     | Kumar     |           1176 |
| Rohan     | Reddy     |           2381 |
| Sneha     | Mehta     |           1443 |
| Rajesh    | Iyer      |           1129 |
| Neha      | Choudhury |           1980 |
| Karan     | Nair      |           1052 |
| Sanya     | Malhotra  |           1459 |
| Aarav     | Bhatia    |            802 |
| Diya      | Joshi     |            647 |
| Vani      | Ghosh     |           2409 |
| Kabir     | Kapoor    |            854 |
| Neha      | Verma     |           1610 |
| Rahul     | Kumar     |           1876 |
+-----------+-----------+----------------+
17 rows in set (0.00 sec)
```

2. DATE_FORMAT(): Format the hire date in a user-friendly manner.

```
mysql> SELECT FirstName, LastName, DATE_FORMAT(HireDate, '%M %d, %Y') AS FormattedHireDate
    -> FROM employee.hospitalemployee;
+-----------+-----------+-------------------+
| FirstName | LastName  | FormattedHireDate |
+-----------+-----------+-------------------+
| Rahul     | Sharma    | January 15, 2020  |
| Ananya    | Patel     | March 10, 2021    |
| Aditi     | Verma     | June 20, 2019     |
| Vikram    | Singh     | February 25, 2022 |
| Priya     | Kumar     | July 30, 2021     |
| Rohan     | Reddy     | April 12, 2018    |
| Sneha     | Mehta     | November 05, 2020 |
| Rajesh    | Iyer      | September 15, 2021|
| Neha      | Choudhury | May 18, 2019      |
| Karan     | Nair      | December 01, 2021 |
| Sanya     | Malhotra  | October 20, 2020  |
| Aarav     | Bhatia    | August 08, 2022   |
| Diya      | Joshi     | January 10, 2023  |
| Vani      | Ghosh     | March 15, 2018    |
| Kabir     | Kapoor    | June 17, 2022     |
| Neha      | Verma     | May 22, 2020      |
| Rahul     | Kumar     | August 30, 2019   |
+-----------+-----------+-------------------+
17 rows in set (0.00 sec)
```

Conclusion

SQL functions enhance data manipulation and analysis by providing powerful tools for calculations and transformations. By utilizing aggregate

functions for summarizing data, scalar functions for individual value processing, and string and date functions for formatting, users can derive meaningful insights and streamline data handling. These capabilities improve the efficiency and effectiveness of SQL queries, facilitating better decision-making and reporting.

Result

The SQL queries using different types of functions were executed successfully.

1. **Aggregate Functions:** Summarized data, such as total salaries and employee counts, provided insights into workforce metrics.
2. **Scalar Functions:** Individual value manipulations, such as formatting names and rounding salary figures, enhanced data presentation.
3. **String Functions:** Concatenated names and calculated string lengths, improving the clarity of employee records.
4. **Date Functions:** Calculated the difference between hire dates and current dates, allowing for analysis of employee tenure.

EXPERIMENT : 12

AIM: Study and implementation of Group By , Having, Order By

Introduction

The SQL clauses **`GROUP BY`, `HAVING`,** and **`ORDER BY`** play a crucial role in data organization and analysis within relational databases. These clauses enable users to summarize and filter data effectively. The **`GROUP BY`** clause allows for the aggregation of data into groups, facilitating the use of functions like SUM and COUNT. The **`HAVING`** clause provides a mechanism to filter these grouped results based on specified conditions, ensuring only relevant groups are included in the final output. Lastly, the **`ORDER BY`** clause arranges the resulting dataset in a specified order, enhancing readability and interpretability.

Theory

The SQL clauses `GROUP BY`, `HAVING`, and `ORDER BY` are integral for data manipulation and analysis.

- **GROUP BY:** This clause groups rows that share common values in specified columns, allowing aggregate functions to summarize data for each group. For example, using `GROUP BY` with a `COUNT` function enables the counting of records per category, providing insights into data distributions.
- **HAVING:** Unlike the `WHERE` clause, which filters rows before grouping, `HAVING` filters the results of aggregated data after the `GROUP BY` operation. This allows for conditions to be applied to the summary results. For instance, one can use `HAVING` to display only groups that meet certain criteria, such as counts exceeding a specified number.
- **ORDER BY:** This clause is used to sort the final result set based on one or more columns, either in ascending or descending order. Sorting enhances the clarity of the data presented, making it easier to analyze trends and patterns.

Procedure

1. **GROUP BY**:Use the **GROUP BY** clause to group rows by a specified column.

```
mysql> SELECT Department, COUNT(*) AS EmployeeCount
    -> FROM employee.hospitalemployee
    -> GROUP BY Department;
+------------------+---------------+
| Department       | EmployeeCount |
+------------------+---------------+
| Emergency        |             2 |
| Pediatrics       |             1 |
| Nursing          |             2 |
| Radiology        |             1 |
| Pharmacy         |             1 |
| Administration   |             1 |
| Surgery          |             1 |
| Oncology         |             1 |
| Cardiology       |             1 |
| Nutrition        |             1 |
| Neurology        |             1 |
| Anesthesiology   |             1 |
| Gastroenterology |             1 |
| NULL             |             2 |
+------------------+---------------+
14 rows in set (0.00 sec)
```

2. **HAVING** : Add the **HAVING** clause to filter groups based on aggregate conditions.

```
mysql> SELECT Department, AVG(Salary) AS AverageSalary
    -> FROM employee.hospitalemployee
    -> GROUP BY Department
    -> HAVING AVG(Salary) > 80000;
+------------------+---------------+
| Department       | AverageSalary |
+------------------+---------------+
| Emergency        | 122500.000000 |
| Pediatrics       | 110000.000000 |
| Nursing          |  82500.000000 |
```

1. **ORDER BY :** Use the **`ORDER BY`** clause to sort the results.

```
mysql> SELECT Department, COUNT(*) AS EmployeeCount
    -> FROM employee.hospitalemployee
    -> GROUP BY Department
    -> ORDER BY EmployeeCount DESC;
+------------------+---------------+
| Department       | EmployeeCount |
+------------------+---------------+
| Emergency        |             2 |
| Nursing          |             2 |
| NULL             |             2 |
| Pediatrics       |             1 |
| Radiology        |             1 |
| Pharmacy         |             1 |
| Administration   |             1 |
| Surgery          |             1 |
| Oncology         |             1 |
| Cardiology       |             1 |
| Nutrition        |             1 |
| Neurology        |             1 |
| Anesthesiology   |             1 |
| Gastroenterology |             1 |
+------------------+---------------+
14 rows in set (0.00 sec)
```

Conclusion

The use of the `GROUP BY`, `HAVING`, and `ORDER BY` clauses in SQL provides powerful capabilities for data analysis. By grouping data, users can apply aggregate functions to summarize information effectively. The `HAVING` clause allows for precise filtering of these summarized results, ensuring that only relevant data is presented. Finally, the `ORDER BY` clause enhances data readability by sorting results in a meaningful way. Together, these features facilitate in-depth insights and help in making informed decisions based on the data analysis.

Result

1. **Grouped Data:** Employee counts per department were accurately calculated.
2. **Filtered Results:** Departments with average salaries exceeding a specified threshold were effectively displayed.

EXPERIMENT : 13

AIM: Write a PL/SQL code block to implement Triggers

Introduction

PL/SQL triggers are powerful tools in relational databases that automatically execute in response to specific events on a table or view. These triggers enable the enforcement of business rules and the maintenance of data integrity without manual intervention.
There are two primary types of triggers: Row-Level Triggers, which activate for each row affected by an event (such as INSERT, UPDATE, or DELETE), and Statement-Level Triggers, which execute once for the entire SQL statement. This flexibility allows developers to perform tasks like logging changes, validating data, and enforcing referential integrity .
Overall, PL/SQL triggers enhance automation and reliability in data management processes.

Theory

PL/SQL triggers are automatic procedures that execute in response to specific events on a table or view, enhancing data integrity and automating tasks.

There are two main types of triggers:

1. **Row-Level Triggers:** Fire for each affected row, allowing detailed operations like data validation before insertion or updates.
2. **Statement-Level Triggers:** Execute once per SQL statement, regardless of the number of rows, ideal for tasks like logging changes.

Triggers can be set to run before or after the event, facilitating actions such as modifying data or sending notifications. Common use cases include data validation, auditing changes, and maintaining referential integrity. Overall, triggers play a crucial role in automating processes and ensuring consistent data management.

Procedure

Row-Level Trigger:

- **Definition**: A row-level trigger executes once for each row affected by the triggering event (INSERT, UPDATE, DELETE).

```
mysql> DELIMITER //
mysql>
mysql> CREATE TRIGGER trg_after_insert
    -> AFTER INSERT ON hospitalemployee
    -> FOR EACH ROW
    -> BEGIN
    ->     INSERT INTO logs (LogTime, Action)
    ->     VALUES (NOW(), CONCAT('Inserted employee: ', NEW.FirstName, ' ', NEW.LastName));
    -> END //
Query OK, 0 rows affected (0.01 sec)

mysql>
mysql> DELIMITER ;
mysql>
```

Statement-Level Trigger:

- **Definition**: A statement-level trigger executes once for the entire statement, regardless of how many rows are affected.

```
mysql> DELIMITER //
mysql>
mysql> CREATE TRIGGER trg_after_insert_statement
    -> AFTER INSERT ON hospitalemployee
    -> FOR EACH ROW
    -> BEGIN
    ->     INSERT INTO logs (LogTime, Action)
    ->     VALUES (NOW(), 'Multiple employees inserted');
    -> END //
Query OK, 0 rows affected (0.01 sec)

mysql>
mysql> DELIMITER ;
```

Conclusion

The implementation of triggers in the database enhances data integrity and automated logging.

The row-level trigger effectively captures individual insertions, recording detailed information about each employee added to the `**employee**` table. In contrast, the statement-level trigger provides a summary action for bulk inserts, indicating when multiple employees are added at once. By utilizing these triggers, the database maintains a comprehensive log of changes, facilitating better tracking and auditing of data modifications. This structured approach to data management ensures that all actions are documented, contributing to improved database reliability and transparency.

Result

The triggers were successfully created and are functioning as intended:

1. **Row-Level Trigger:** Each time a new employee is inserted into the `**employee**` table, an entry is logged in the `logs` table, capturing the exact time of insertion and the name of the employee.
2. **Statement-Level Trigger:** Whenever multiple employees are inserted in a single operation, a log entry is created indicating that multiple insertions have occurred.

These triggers provide a robust mechanism for tracking changes within the database, ensuring all insert actions are recorded effectively. The logging enhances audit capabilities, allowing for easier monitoring and analysis of employee data changes over time.