SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

SRM NAGAR,KATTANKULATHUR-603203

DEPARTMENT OF INFORATION TECHNOLOGY

LAB MANUAL

CS-3463 DATABASE MANAGEMENT SYSTEM LABORATORY

B.E.-IT-4thSEMESTER

REGULATIONS 2023

Prepared by

Ms. G. Santhiya/Assistant Professor(Sr.G)/IT

Ms.D.Nisha/Assistant Professor(Sr.G)/IT

CS3466 DATABASE MANAGEMENT SYSTEMS LABORATORY **L T P C**
0 0 3 1.5

OBJECTIVES:

• To learn and implement important commands in SQL.

• To learn the usage of nested and joint queries.

• To understand functions, procedures and procedural extensions of databases.

• To understand design and implementation of typical database applications.

• To be familiar with the use of a front end tool for GUI based application development.

LIST OF EXPERIMENTS

1. Create a database table, add constraints (primary key, unique, check, Not null), insert rows, update and delete rows using SQL DDL and DML commands.
2. Create a set of tables, add foreign key constraints and incorporate referential integrity.
3. Query the database tables using different 'where' clause conditions and also implement aggregate functions.
4. Query the database tables and explore sub queries and simple join operations.
5. Query the database tables and explore natural, equi and outer joins.
6. Write user defined functions and stored procedures in SQL.
7. Execute complex transactions and realize DCL and TCL commands.
8. Write SQL Triggers for insert, delete, and update operations in a database table.
9. Create View and index for database tables with a large number of records.
10. Case Study using any of the real life database applications from the following list
    a) Inventory Management for a EMart Grocery Shop
    b) Society Financial Management
    c) Cop Friendly App – Eseva
    d) Property Management – eMall
    e) Star Small and Medium Banking and Finance

• Build Entity Model diagram. The diagram should align with the business and functional goals stated in theapplication.

• Apply Normalization rules in designing the tables in scope.

• Prepared applicable views, triggers (for auditing purposes), and functions for enabling enterprise grade features.

11. To Study and develop the application using concept of MongoDB.

TOTAL: 45 PERIODS

SOFTWARE:

Systems with MySql, Visual Studio, Systems with Oracle 11g Client, NoSQL and MongoDB.

OUTCOMES:

At the end of this course, the students should be able to:

• Create databases with different types of key constraints.

• Construct simple and complex SQL queries using DML and DCL commands.

• Use advanced features such as stored procedures

• Create a trigger for the database.

• Create and manipulate database application.

CO – PO – PSO Mapping

CO	PO												PSO			
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4
1	3	3	3	3	-	-	-	-	3	1	3	2	2	3	2	-
2	2	2	3	2	2	-	-	-	1	2	3	3	2	1	2	-
3	3	3	2	1	1	-	-	-	1	1	1	3	2	3	3	-
4	1	3	3	3	1	-	-	-	1	1	3	2	3	1	3	-
5	3	2	1	1	1	-	-	-	2	2	3	1	3	1	2	-
Avg	2.4	2.6	2.4	2.0	1.25	-	-	-	1.6	1.4	2.6	2.2	2.4	1.8	2.4	-

Aim and Procedure/ E-R Diagram	SQLQuery Program/forms /Tables	Execution and Results	Viva-voce	Record	Total
20	30	30	10	10	100

SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution, Affiliated to Anna University, Chennai)

B.TECH. INFORMATION TECHNOLOGY

REGULATIONS – 2023

1. VISION AND MISSION OF INSTITUTION

Vision

"Educate to Excel in Social Transformation"

To accomplish and maintain international eminence and become a model institution for higher learning through dedicated development of minds, advancement of knowledge and professional application of skills to meet the global demands.

Mission

• To contribute to the development of human resources in the form of professional engineers and managers of international excellence and competence with high motivation and dynamism, who besides serving as ideal citizen of our country will contribute substantially to the economic development and advancement in their chosen areas of specialization.

• To build the institution with international repute in education in several areas at several levels with specific emphasis to promote higher education and research through strong instituteindustry interaction and consultancy.

2. VISION AND MISSION OF DEPARTMENT

Vision

To become a model for higher learning through development to prepare self-disciplined, creative culturally competent and dynamic Information Technocrats while remaining sensitive to ethical, societal and environmental issues.

Mission

M1: To mould the students as innovative and high quality IT professionals to meet the global challenges and Entrepreneurs of international excellence as global leaders capable of contributing towards technological innovations learning process, participation citizenship in their neighborhood and economic growth.

M2: To impart value-based IT education to the students and enrich their knowledge and to achieve effective interaction between industry and institution for mutual benefits.

1. PROGRAMME EDUCATIONAL OBJECTIVES (PEOs):

PEO1: To afford the necessary background in the field of Information Technology to deal with engineering problems to excel as engineering professionals in industries.
PEO2: To improve the qualities like creativity, leadership, teamwork and skill thus contributing towards the growth and development of society.
PEO3: To develop ability among students towards innovation and entrepreneurship that caters to the needs of Industry and society.
PEO4: To inculcate and attitude for life-long learning process through the use of information technology sources.
PEO5: To prepare then to be innovative and ethical leaders, both in their chosen profession and in other activities.

2. PROGRAMME OUTCOMES (POs):

Engineering Graduates will be able to:
1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate public health and safety, and the cultural, societal, and environmental considerations.
4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the

engineering practice.

9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

3. PROGRAMME SPECIFIC OUTCOMES (PSOs)

By the completion Bachelor of Technology in Information Technology program the student will have following Program specific outcomes.

PSO1: Design secured database applications involving planning, development and maintenance using state of the art methodologies based on ethical values.

PSO2: Design and develop solutions for modern business environments coherent with the advanced technologies and tools.

PSO3: Design, plan and setting up the network that is helpful for contemporary business environments using latest hardware components.

PSO4: Planning and defining test activities by preparing test cases that can predict and correct errors ensuring a socially transformed product catering all technological needs.

Ex.No:1	DATA DEFINITION LANGUAGE (DDL) & DML COMMANDS
Date:	

AIM:

Creation of a database and writing SQL commands to retrieve information from the database.

Theory:

Data Definition Language (DDL) (Without Constraints)

Data Definition Language (DDL) is a subset of SQL (Structured Query Language) used to define and manage all structures in a database. DDL commands are used to create, modify, and delete database objects such as tables, views, indexes, and schemas.
When we use DDL **without constraints**, we are focusing only on defining the structure of the database objects **without applying any rules or restrictions** (like primary keys, foreign keys, unique constraints, etc.).

Data Definition Language (DDL) (**With Constraints)**

They enforce **rules and restrictions** on the data to maintain **data integrity** and **consistency** in the database.
DML (Data Manipulation Language)

DML is a set of SQL commands used to **manipulate the data** stored in database tables.

Constraints

Constraints are rules applied to columns in a table to limit the type of data that can be stored in them. They ensure accuracy and reliability of the data in the database.

Types of DDL Commands

DDL Command	Description
CREATE	Used to create new database objects (e.g., tables, views).
ALTER	Used to modify existing database structures (add, modify, or drop columns/constraints).
DROP	Deletes an entire database object permanently (e.g., table, view).
TRUNCATE	Deletes all records from a table but keeps the structure intact.
RENAME	Renames a table or other database object.

Types of Constraints

Constraint Type	Description	Example
NOT NULL	Ensures a column cannot have a NULL value.	Name VARCHAR(100) NOT NULL
UNIQUE	Ensures all values in a column are unique.	Email VARCHAR(100) UNIQUE
PRIMARY KEY	Uniquely identifies each row in a table. Combines NOT NULL and UNIQUE.	StudentID INT PRIMARY KEY
FOREIGN KEY	Ensures the value in one table matches a value in another table (maintains relationships).	FOREIGN KEY (DeptID) REFERENCES Department(ID)
CHECK	Validates that values in a column meet a specific condition.	Age INT CHECK (Age >= 18)
DEFAULT	Sets a default value for a column when no value is provided.	Status VARCHAR(10) DEFAULT 'Active'

Common DML Commands:

Command	Use
SELECT	To retrieve data from tables
INSERT	To add new data into tables
UPDATE	To modify existing data
DELETE	To remove data from tables

SYNTAX:

1. Create Table

The CREATE TABLE statement is used to create a relational table

```
CREATE TABLE table_name
(
        column_name1 data_type [constraints],
        column_name1 data_type [constraints],
        column_name1 data_type [constraints],
        ……..
);
```

2. Alter Table

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table

a. To Add a column

```
ALTER TABLE table_name ADD column_name datatype
```

b. To delete a column in a table

```
ALTER TABLE table_name DROP COLUMN column_name
```

c. To change the data type of a column in a table

```
ALTER TABLE table_name ALTER COLUMN column_name datatype
```

3. Drop Table

Used to delete the table permanently from the storage

```
DROP TABLE table_name
```

CREATE THE TABLE

```
CREATE TABLE emp
        (empno INT,
        empname VARCHAR(25),
        dob DATE,
        salary INT,
        designation VARCHAR(20)
        );
```

- Table Created

// Describe the table emp

DESC emp;

Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
EMPNO	INT	11	-	-	-	-	-	-
EMPNAME	VARCHAR	25	-	-	-	-	-	-
DOB	DATE	7	-	-	-	-	-	-
SALARY	INT	11	-	-	-	-	-	-
DESIGNATION	VARCHAR	20	-	-	-	-	-	-

1. ALTER THE TABLE

a. ADD

// To alter the table emp by adding new attribute department

```
ALTER TABLE emp ADD department VARCHAR(50);
```

- Table Altered

DESC emp;

Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
EMPNO	INT	11	-	-	-	-	-	-
EMPNAME	VARCHAR	25	-	-	-	-	-	-

DOB	DATE	7	-	-	-	-	-
-							
SALARY	INT	11	-	-	-	-	-
-	VARCHAR	20	-	-	-	-	-
DESIGNATION							
DEPARTMENT	VARCHAR2	50	-	-	-	-	--

b. MODIFY

// To alter the table emp by modifying the size of the attribute department

ALTER TABLE emp MODIFY department VARCHAR2(100);

- Table Altered

DESC emp;

Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default
Comment							
EMPNO	INT	11	-	-	-	-	-
EMPNAME	VARCHAR	25	-	-	-	-	-
-							
DOB	DATE	7	-	-	-	-	-
-	INT	11	-	-	-	-	
SALARY							
DESIGNATION	VARCHAR	20	-	-	-	-	-
-							
DEPARTMENT	VARCHAR	100	-	-	-	-	-

c. DROP

// To alter the table emp by deleting the attribute department

ALTER TABLE emp DROP(department);

- Table Altered

DESC emp;

Column	Data Type	Length	Precision	Scale	Primar	Key	Nullable	Default	Comment
EMPNO	INT	11	-	-	-	-	-	-	
EMPNAME	VARCHAR	25	-	-	-	-	-	-	
DOB	DATE	7	-	-	-	-	-	-	
SALARY	INT	11	-	-	-	-	-	-	
DESIGNATION	VARCHAR	20	-	-	-	-	-	-	

d. RENAME

// To alter the table name by using rename keyword

ALTER TABLE emp RENAME TO emp1 ;

- Table Altered

DESC emp1;

Column	**Data Type**	**Length**	**Precision**	**Scale**	**Primary Key**	**Nullable**	**Default**	Comment
EMPNO	INT	11	-	-	-	-	-	
EMPNAME	VARCHAR	25	-	-	-	-	-	
DOB	DATE	7	-	-	-	-	-	
SALARY	INT	22	-	-	-	-	-	
DESIGNATION	VARCHAR	20	-	-	-	-	-	
-								
DEPARTMENT	VARCHAR	100	-	-	-	-	-	
-								

2. DROP

//To delete the table from the database

```
DROP TABLE emp1;
```

- Table Dropped

```
DESC emp1;
```

Object to be described could not be found.

1. CREATE THE TABLE with constraints

// To create a table student

```
CREATE TABLE student
(
        studentID INT PRIMARY KEY,
        sname VARCHAR(30) NOT NULL,
        department VARCHAR(5),
        sem INT,
        dob DATE,
        email_id VARCHAR(20) UNIQUE,
        college VARCHAR(20) DEFAULT 'MEC'
);
```

// Describe the table student

```
DESC student;
```

Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
STUDENTID	INT	22	-	-	1	-		
SNAME	VARCHAR	30	-	-	-	-		
DEPARTMENT	VARCHAR	5	-	-	-	-		

SEM	INT	22	-	-	-	-	-
		7	-	-	-	-	-
DOB	DATE						
EMAIL_ID	VARCHAR	20	-	-	-	-	-
COLLEGE	VARCHAR	20	-	-	-	-	
'MEC' -							

//To create a table exam

```
CREATE TABLE exam
(
        examID INT ,
        studentID INT REFERENCES student(studentID),
        department VARCHAR(5) NOT NULL,
        mark1 INT CHECK (mark1<=100 and mark1>=0),
        mark2 INT CHECK (mark2<=100 and mark2>=0),
        mark3 INT CHECK (mark3<=100 and mark3>=0),
        mark4 INT CHECK (mark4<=100 and mark4>=0),
        mark5 INT CHECK (mark5<=100 and mark5>=0),
        total INT,
        average INT,
        grade VARCHAR(1)
);
```

//Describe the table exam

```
DESC exam;
```

Column	**Data Type**	**Length**	**Precision**	**Scale**	**Primary Key**	**Nullable**	**Default**	**Comment**
EXAMID	INT	22	-	-	-	-	-	-
STUDENTID	INT	22	-	-	-	-	-	-
DEPARTMENT	VARCHAR	5	-	-	-	-	-	-
MARK1	INT	22	-	-	-	-	-	-
MARK2	INT	22	-	-	-	-	-	-
MARK3	INT	22	-	-	-	-	-	-
MARK4	INT	22	-	-	-	-	-	-
MARK5	INT	22	-	-	-	-	-	-
TOTAL	INT	22	-	-	-	-	-	-
AVERAGE	INT	22	-	-	-	-	-	-
GRADE	VARCHAR	1	-	-	-	-	-	-

ALTER THE TABLE

ADD

//To alter the table student by adding new attribute address

ALTER TABLE student ADD address VARCHAR2(100);

- Table Altered

DESC student;

Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
STUDENTID	INT	22	-	-	1	-	-	-
SNAME	VARCHAR	30	-	-	-	-	-	-
DEPARTMENT	VARCHAR	5	-	-	-		-	-
SEM	INT	22	-	-	-			
DOB	DATE	7	-	-	-		-	-
EMAIL_ID	VARCHAR	20	-	-	-		-	-
COLLEGE	VARCHAR	20	-	-	-		-	
ADDRESS	VARCHAR	100	-	-	-		-	-

'MEC'

-

MODIFY

//To alter the table student by modifying the size of the attribute address

ALTER TABLE student MODIFY address VARCHAR (150);

- Table Altered

DESC student;

//To alter the table student by deleting the attribute address

ALTER TABLE student DROP(address);

Table Altered;

Desc Student

- ;

Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
STUDENTID	INT	22	-	-	1	-	-	-
SNAME	VARCHAR	30	-	-	-	-	-	-
DEPARTMENT	VARCHAR	5	-	-	-	-	-	-
SEM	INT	22	-	-	-	-	-	-
DOB	DATE	7	-	-	-	-	-	-
EMAIL_ID	VARCHAR	20	-	-	-	-	-	-
COLLEGE	VARCHAR	20	-	-	-	-	'MEC'	-

D. RENAME

// To alter the table name by using rename keyword

ALTER TABLE student RENAME TO student1 ;

Table

Altered

Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
STUDENTID	INT	22	-	-	1	-	-	-
SNAME	VARCHAR	30	-	-	-	-	-	-

- -
DEPARTMENT CHAR 5 - - - -
- -
SEM INT 22 - - - -
- -
DOB DATE 7 - - - -
- -
EMAIL_ID VARCHAR 20 - - - -
- -
COLLEGE VARCHAR 20 - - - -
'MEC' -

DESC

student1;

ALTER TABLE student1 RENAME TO student ;

- Table Altered

1. DROP

// To delete the table from the database

DROP TABLE exam;

Table dropped.

-

Object to be described could not be found.

DML COMMANDS

```
CREATE TABLE Employees (
   EmployeeID INT PRIMARY KEY,              -- Primary Key Constraint
   FirstName VARCHAR(50) NOT NULL,          -- NOT NULL Constraint
   LastName VARCHAR(50) NOT NULL,           -- NOT NULL Constraint
   Email VARCHAR(100) UNIQUE,               -- UNIQUE Constraint
   Salary DECIMAL(10, 2) CHECK (Salary > 0)    -- CHECK Constraint
);
```

OUPUT:
table "Employees" created successfully.

DML: Inserting Records into the Table

-- Inserting valid rows

```
INSERT INTO Employees (EmployeeID, FirstName, LastName, Email,
Salary)
VALUES (1, 'Alice', 'Johnson', 'alice@example.com', 55000.00);
```

1 row inserted.

```
INSERT INTO Employees (EmployeeID, FirstName, LastName, Email,
Salary)
VALUES (2, 'Bob', 'Smith', 'bob@example.com', 62000.00);
```

1 row inserted.

SELECT * FROM Employees;

EmployeeID	FirstName	LastName	Email	Salary
1	Alice	Johnson	'alice@example.com'	55000.00
2	Bob	Smith	bob@example.com	62000.00

-- Attempting invalid inserts (Uncomment one by one to test)

-- Duplicate Primary Key

```
-- INSERT INTO Employees (EmployeeID, FirstName, LastName, Email,
Salary)
-- VALUES (2, 'Sam', 'Taylor', 'sam@example.com', 48000.00);
```

ERROR: Duplicate entry '2' for key 'PRIMARY'

```
-- NULL value in NOT NULL column
-- INSERT INTO Employees (EmployeeID, FirstName, LastName, Email,
Salary)
-- VALUES (3, NULL, 'Green', 'green@example.com', 40000.00);
```

ERROR: Column 'FirstName' cannot be null

```
-- Duplicate Email (Unique Constraint)
-- INSERT INTO Employees (EmployeeID, FirstName, LastName, Email,
Salary)
-- VALUES (4, 'Jane', 'Doe', 'bob@example.com', 30000.00);
```

ERROR: Duplicate entry 'bob@example.com' for key 'Email'

```
-- Invalid Salary (Check Constraint)
-- INSERT INTO Employees (EmployeeID, FirstName, LastName,
Email, Salary)
-- VALUES (5, 'Rick', 'Grimes', 'rick@example.com', -1000.00);
```

ERROR: Check constraint 'Salary > 0' failed

DML: Updating a Record

```
-- Update salary of employee with ID = 2
UPDATE Employees
SET Salary = 70000.00
WHERE EmployeeID = 2;
```

1 row updated.

SELECT * FROM Employees;

EmployeeID	FirstName	LastName	Email	Salary
1	Alice	Johnson	alice@example.com'	55000.00
2	Bob	Smith	bob@example.com	70000.00

DML: Deleting a Record

```
-- Delete employee with ID = 1
DELETE FROM Employees
WHERE EmployeeID = 1;
```

1 row deleted.

Viewing Records in the Table

SELECT * FROM Employees;

EmployeeID	FirstName	LastName	Email	Salary
2	Bob	Smith	bob@example.com	70000.00

RESULT:

Databases are created and retrieved information successfully.

Ex No:2 Creating Tables with Foreign Key Constraints and Enforcing Referential Integrity

Aim

To create tables with primary and foreign key constraints and enforce referential integrity between related tables using SQL.

Theory

Primary Key: A field (or combination of fields) that uniquely identifies each record in a table.

Foreign Key: A field in one table that refers to the primary key in another table.

Referential Integrity: A system of rules that ensures that relationships between tables remain consistent.

Explanation of Referential Actions

Action	Description
ON DELETE CASCADE	Deletes related rows in the child table when a row in the parent is deleted
ON DELETE SET NULL	Sets the foreign key in child table to NULL when parent row is deleted
ON UPDATE CASCADE	Automatically updates the foreign key when the referenced primary key changes

SQL Syntax

```
-- Create a parent table
CREATE TABLE parent_table (
    id INT PRIMARY KEY,
    name VARCHAR(50)
);

-- Create a child table with a foreign key reference
CREATE TABLE child_table (
    child_id INT PRIMARY KEY,
    parent_id INT,
    description VARCHAR(100),
    FOREIGN KEY (parent_id) REFERENCES parent_table(id)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

Example: University Database

1. Create Department Table (Parent Table)

```
CREATE TABLE Department (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(100) NOT NULL
);
```

Table department created successfully

2. Create Student Table (Child Table)

```
CREATE TABLE Student (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(100),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES Department(dept_id)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);
```

Table Student created successfully

3. Create Course Table

```
CREATE TABLE Course (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(100),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES Department(dept_id)
        ON DELETE CASCADE
);
```

Table Course created successfully

4. Create Enrollment Table (with multiple foreign keys)

```
CREATE TABLE Enrollment (
    enrollment_id INT PRIMARY KEY,
    student_id INT,
    course_id INT,
    enrollment_date DATE,
    FOREIGN KEY (student_id) REFERENCES Student(student_id)
        ON DELETE CASCADE,
    FOREIGN KEY (course_id) REFERENCES Course(course_id)
        ON DELETE CASCADE
);
```

Table Enrollmentt created successfully

Sample Data Insertion

```
-- Insert into Department
INSERT INTO Department VALUES (1, 'Computer Science'), (2, 'Physics');
```

2 rows inserted

Output:

Select * from Department

dept_id	dept_name
1	Computer Science
2	Physics

```
-- Insert into Student
INSERT INTO Student VALUES (101, 'Alice', 1), (102, 'Bob', 2);
```

2 rows inserted

SELECT * FROM Student;

Output:

student_id student_name dept_id

101 Alice 1

102 Bob 2

-- Insert into Course

INSERT INTO Course VALUES (501, 'Data Structures', 1), (502, 'Quantum Mechanics', 2);

2 rows inserted

SELECT * FROM Course;

Output:

course_id course_name dept_id

501 Data Structures 1

502 Quantum Mechanics 2

-- Insert into Enrollment

INSERT INTO Enrollment VALUES (1, 101, 501, '2025-01-10'), (2, 102, 502, '2025-01-12');

2 rows inserted

SELECT * FROM Enrollment;

Output:

enrollment_id student_id course_id enrollment_date

1 101 501 2025-01-10

2 102 502 2025-01-12

Output/Observations

Deleting a department:

- Removes associated courses due to ON DELETE CASCADE

- Sets dept_id to NULL in the Student table due to ON DELETE SET NULL

Updating a department's dept_id:

- Updates related foreign keys in the Student table due to ON UPDATE CASCADE

SQL Queries Demonstrating Foreign Key Effects

1. Deleting a Department (ON DELETE CASCADE and SET NULL)

DELETE FROM Department WHERE dept_id = 2;

Output:

Course Table (ON DELETE CASCADE)

- Any course with dept_id = 2 will be **deleted**.

Before:

course_id	**course_name**	**dept_id**
501	Data Structures	1
502	Quantum Mechanics	2

After:

course_id	**course_name**	**dept_id**
501	Data Structures	1

Course with dept_id = 2 is gone.

Expected Effects:

The course with dept_id = 2 will be deleted from the Course table (ON DELETE CASCADE).

The dept_id of students in the Student table with dept_id = 2 will be set to NULL (ON DELETE SET NULL).

Student Table (ON DELETE SET NULL)

- Any student with dept_id = 2 will have their dept_id **set to NULL**.

Before:

student_id	student_name	dept_id
101	Alice	1
102	Bob	2

After:

student_id	student_name	dept_id
101	Alice	1
102	Bob	NULL

Bob's department is now NULL.

Enrollment Table

- Still **unchanged** for now, because we haven't deleted any student or course involved.

enrollment_id	student_id	course_id	enrollment_date
1	101	501	2025-01-10

Enrollment #2 (for Bob in course 502) is automatically deleted because **course 502 was deleted** due to ON DELETE CASCADE.

Final Table States:

Department

dept_id	dept_name
1	Computer Science

Student

student_id	student_name	dept_id
101	Alice	1
102	Bob	NULL

Course

course_id	course_name	dept_id
501	Data Structures	1

Enrollment

enrollment_id	student_id	course_id	enrollment_date
1	101	501	2025-01-10

To verify the result:

SELECT * FROM Course; -Should not contain any course with dept_id = 2

SELECT * FROM Student; -- Students with dept_id = 2 should now have dept_id = NULL

2. Updating a Department ID (ON UPDATE CASCADE)

UPDATE Department SET dept_id = 5 WHERE dept_id = 1;

Expected Effects:

The dept_id in the Student table is updated to 5 where it was previously 1.

The dept_id in the Course table is also updated to 5 where it was previously 1.

1. Updates the Department table:

dept_id	dept_name
5	Computer Science

Automatically updates foreign keys in:

→ Student table:

student_id	student_name	dept_id
101	Alice	5
102	Bob	NULL

→ Course table:

course_id	course_name	dept_id
501	Data Structures	5

Final Table States After UPDATE:

Department

dept_id	dept_name
5	Computer Science

Student

student_id	student_name	dept_id
101	Alice	5
102	Bob	NULL

Course

course_id	course_name	dept_id
501	Data Structures	5

Enrollment

enrollment_id	student_id	course_id	enrollment_date
1	101	501	2025-01-10

To verify the result:

```
SELECT * FROM Student WHERE dept_id = 5;

 -- Students from previous dept_id 1 now have dept_id 5

SELECT * FROM Course WHERE dept_id = 5;

 -- Courses from previous dept_id 1 now have dept_id 5
```

Result:

Successfully Created Tables with Foreign Key Constraints and Enforcing Referential Integrity.

Ex.No:3	QUERY THE DATABASE TABLES USING DIFFERENT 'WHERE CLAUSE CONDITIONS AND ALSO IMPLEMENT AGGREGATE FUNCTIONS

AIM:

Creation of a database and execute Aggregate Functions to retrieve information from the database.

Theory:

Structured Query Language (SQL) is used to manage and manipulate relational databases. Two common operations in SQL are filtering data using the WHERE clause and performing calculations using aggregate functions such as COUNT, SUM, AVG, MAX, and MIN. These tools allow users to retrieve specific information from large datasets and generate useful summaries.

1. WHERE Clause

The WHERE clause in SQL is used to filter records that meet a certain condition. It helps in extracting only those records that fulfill specified criteria, making the data more meaningful.

- Common conditions used in the WHERE clause include:
- Comparison Operators: =, !=, >, <, >=, <=
- Logical Operators: AND, OR, NOT
- Pattern Matching: LIKE, IN, BETWEEN

2. Aggregate Functions

Aggregate functions in SQL perform a calculation on a set of values and return a single value. They are often used with the GROUP BY clause to group rows sharing a property so that an aggregate value can be calculated for each group.

- Common aggregate functions include:
- COUNT(): Returns the number of rows.
- SUM(): Returns the total sum of a numeric column.
- AVG(): Returns the average value of a numeric column.
- MAX(): Returns the highest value in a column.
- MIN(): Returns the lowest value in a column

1. Basic WHERE Clause Conditions

- Find all employees in the IT department:

```
SELECT * FROM employees
WHERE department = 'IT';
```

- Find employees with salary greater than 50,000:

```
SELECT * FROM employees
WHERE salary > 50000;
```

- Find employees hired after 2020:

```
SELECT * FROM employees
WHERE hire_date > '2020-01-01';
```

- Find employees in HR or Marketing:

```
SELECT * FROM employees
WHERE department IN ('HR', 'Marketing');
```

- Find employees whose names start with 'D':

```
SELECT * FROM employees
WHERE name LIKE 'D%';
```

Sample employees Table:

employee_id	name	department	salary	hire_date
1	Alice	IT	55000	2019-06-01
2	Bob	HR	48000	2021-03-15
3	Charlie	Marketing	51000	2022-07-10
4	David	IT	60000	2023-01-20
5	Diana	Finance	45000	2020-11-05
6	Daniel	HR	53000	2021-12-01

Query Results:

1. Employees in the IT department:

SELECT * FROM employees WHERE department = 'IT';

employee_id	name	department	salary	hire_date
1	Alice	IT	55000	2019-06-01
4	David	IT	60000	2023-01-20

---

2. Employees with salary > 50,000:

SELECT * FROM employees WHERE salary > 50000;

employee_id	name	department	salary	hire_date
1	Alice	IT	55000	2019-06-01
3	Charlie	Marketing	51000	2022-07-10
4	David	IT	60000	2023-01-20
6	Daniel	HR	53000	2021-12-01

---

3. Employees hired after 2020:

SELECT * FROM employees WHERE hire_date > '2020-01-01';

employee_id	name	department	salary	hire_date
2	Bob	HR	48000	2021-03-15
3	Charlie	Marketing	51000	2022-07-10
4	David	IT	60000	2023-01-20
5	Diana	Finance	45000	2020-11-05
6	Daniel	HR	53000	2021-12-01

---

4. Employees in HR or Marketing:

SELECT * FROM employees WHERE department IN ('HR', 'Marketing');

employee_id	name	department	salary	hire_date
2	Bob	HR	48000	2021-03-15
3	Charlie	Marketing	51000	2022-07-10
6	Daniel	HR	53000	2021-12-01

---

5. Employees whose names start with 'D':

SELECT * FROM employees WHERE name LIKE 'D%';

employee_id	name	department	salary	hire_date
4	David	IT	60000	2023-01-20
5	Diana	Finance	45000	2020-11-05
6	Daniel	HR	53000	2021-12-01

2. **Using Aggregate Functions**

- Count total employees:

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

- Total salary of all employees:

```
SELECT SUM(salary) AS total_salary
FROM employees;
```

- Average salary:

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

- Highest and lowest salary:

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

3. **Aggregate Functions with GROUP BY**

- Total salary by department:

```
SELECT department, SUM(salary) AS total_salary
FROM employees
GROUP BY department;
```

- Average salary by department:

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department;
```

- Count of employees per department:

```
SELECT department, COUNT(*) AS num_employees
FROM employees
GROUP BY department;
```

4. **HAVING Clause (Used with Aggregates)**

- Departments with average salary greater than 52,000:

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department
HAVING AVG(salary) > 52000;
```

Output:

Aggregate Functions

Total number of employees:

SELECT COUNT(*) AS total_employees FROM employees;

Output:

total_employees

6

Total salary of all employees:

SELECT SUM(salary) AS total_salary FROM employees;

Output:

total_salary

312000

Average salary:

SELECT AVG(salary) AS average_salary FROM employees;

Output:

average_salary

52000.00

Highest and lowest salary:

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

Output:

highest_salary lowest_salary

60000 45000

Aggregate Functions with GROUP BY

Total salary by department:

SELECT department, SUM(salary) AS total_salary FROM employees GROUP BY department;

Output:

department total_salary

IT 115000

department	total_salary
HR	101000
Marketing	51000
Finance	45000

Average salary by department:

SELECT department, AVG(salary) AS avg_salary FROM employees GROUP BY department;

Output:

department	avg_salary
IT	57500.00
HR	50500.00
Marketing	51000.00
Finance	45000.00

Count of employees per department:

SELECT department, COUNT(*) AS num_employees FROM employees GROUP BY department;

Output:

department	num_employees
IT	2
HR	2
Marketing	1
Finance	1

---

3. Using HAVING Clause

Departments with average salary greater than 52,000:

SELECT department, AVG(salary) AS avg_salary

```
FROM employees
GROUP BY department
HAVING AVG(salary) > 52000;
```

Output:

department	avg_salary
IT	57500.00

Result:

Thus the SQL Queries using where Clause and Aggregate Functions to retrieve information from the table has been executed successfully.

Ex.No:4	QUERY THE DATABASE TABLES AND EXPLORE SUB QUERIES AND SIMPLE JOIN OPERATIONS
Date:	

AIM:

Creation of a database and execute SQL Joins to retrieve information from the database.

Theory:

SQL Subqueries and Join Operations

Subquery

A subquery is a query that is nested inside another SQL query. It is used to perform operations that require intermediate results before applying them in the main query.

Types of Subqueries:

1. Single-row subquery – returns only one row.
2. Multi-row subquery – returns more than one row.
3. Correlated subquery – depends on the outer query for its values.
4. Nested subquery – contains another subquery inside it.

JOIN Operation

A JOIN clause is used to combine rows from two or more tables based on a related column between them.

Types of JOINs:

INNER JOIN – Returns only matching rows from both tables.

LEFT JOIN (or LEFT OUTER JOIN) – Returns all rows from the left table and matched rows from the right table.

RIGHT JOIN – Returns all rows from the right table and matched rows from the left table.

FULL OUTER JOIN – Returns rows when there is a match in either left or right table.

// Table Creation - cseitstudent

```
CREATE TABLE
cseitstudent (
        studentID INT PRIMARY
        KEY, sname VARCHAR(30),
        department
        VARCHAR(5),
        sem INT
);
```

-Table Created

// Table Creation - placement

```
CREATE TABLE
placement (
        PlacementID             INT
        PRIMARY KEY, StudentID
        INT,
        department
        VARCHAR(5),
        Company
        VARCHAR(30),
        salary INT
);
```

-Table Created

// Inserting values into cseitstudent table

```
INSERT INTO cseitstudent (studentID, sname, department, sem) VALUES(101,'reema', 'IT',5);

INSERT INTO cseitstudent (studentID, sname, department, sem) VALUES(102,'reenu', 'IT',3);

INSERT INTO cseitstudent (studentID, sname, department, sem)
VALUES(103,'sheela', 'CSE',3);

INSERT INTO cseitstudent (studentID, sname, department, sem) VALUES(104,'nirmal', 'IT',3);

INSERT INTO cseitstudent (studentID, sname, department, sem)
VALUES(105,'eshwar', 'CSE',5);
```

- 5 row(s) inserted

// Inserting values into placement table

```
INSERT INTO placement VALUES(1, 104, 'IT', 'infosys', 25000);

INSERT INTO placement VALUES(2, 105, 'CSE', 'Wipro', 22000);

INSERT INTO placement VALUES(3, 204, 'MECH', 'HYUNDAI',

30000); INSERT INTO placement VALUES(4, 102, 'IT', 'infosys',

25000); INSERT INTO placement VALUES(5, 103, 'CSE',

'infosys', 25000);
```

- 5 row(s) inserted

// Display the values in the table as rows

```
SELECT * FROM cseitstudent;
```

STUDENTID	SNAME	DEPARTMENT	SEM
101	reema	IT	5
102	reenu	IT	3
103	sheela	CSE	3
104	nirmal	IT	3
105	eshwar	CSE	5

```
SELECT * FROM placement;
```

PLACEMENTID	STUDENTID	DEPARTMENT	COMPANY	SALARY
1	104	IT	infosys	25000
2	105	CSE	Wipro	22000
3	204	MECH	HYUNDAI	30000
4	102	IT	Infosys	25000
5	103	CSE	Infosys	25000

// INNER JOIN

SELECT *FROM cseitstudent INNER JOIN Placement
ON cseitstudent.studentID=placement.StudentID;

STUDENTID	SNAME	DEPARTMENT	SEM	PLACEMENTID	STUDENTID	DEPARTMENT	COMPANY	SALARY
104	nirmal	IT	3	1	104	IT	infosys	25000
105	eshwar	CSE	5	2	105	CSE	Wipro	22000
102	reenu	IT	3	4	102	IT	infosys	25000
103	sheela	CSE	3	5	103	CSE	infosys	25000

SELECT cseitstudent.studentID,cseitstudent.sname,placement.company,
placement.salaryFROM cseitstudent INNER JOIN placement
ON cseitstudent.studentID=placement.studentID;

STUDENTID	SNAME	COMPANY	SALARY
104	nirmal	infosys	25000
105	eshwar	Wipro	22000
102	reenu	infosys	25000
103	sheela	infosys	25000

//SELF JOIN

Returns rows by comparing the values of the same table.

//TABLE CREATION

```
CREATE TABLE
employee (
        empid INT,
        empname
         VARCHAR(25),
         reportingid INT
);
```

- Table Created

//INSERTING VALUES IN THE TABLE

```
INSERT INTO employee VALUES(1, 'Principal', null);
INSERT INTO employee VALUES(2, 'HOD', 1);
INSERT INTO employee VALUES(3, 'PO', 1);
INSERT INTO employee VALUES(4, 'Staff', 2);
INSERT INTO employee VALUES(5, 'Non Teaching Staff', 2);
```

- 5 row(s) inserted

//DISPLAYS VALUES IN THE TABLE

SELECT * FROM employee;

EMPID	EMPNAME	REPORTINGID
1	Principal	-
2	HOD	1
3	PO	1
4	Staff	2
5	Non Teaching Staff	2

SELECT e1.empid, e1.empname, e2.empname AS HeadName FROM employee e1, employee e2 WHERE e1.reportingid=e2.empid;

EMPID	EMPNAME	HEADNAME
2	HOD	Principal
3	PO	Principal
4	Staff	HOD
5	Non Teaching Staff	HOD

// EXISTS

```
SELECT * FROM placement WHERE EXISTS (
SELECT * FROM placement WHERE salary>20000);
```

PLACEMENTID	STUDENTID	DEPAR	COMPANY	SALARY
1	104	IT	infosys	25000
2	105	CSE	Wipro	22000
3	204	MECH	HYUNDAI	30000
4	102	IT	infosys	25000
5	103	CSE	infosys	25000

```
SELECT * FROM employee1 WHERE EXISTS
(SELECT * FROM employee1 WHERE salary>20000 );
```

EMPNO	**EMPNAME**	**SALARY**	**DESIGNATION**
101	RAM	45000	Manager
102	LINGAM	35000	Asst. Manager
103	RUPESH	20000	Clerk
104	BALA	10000	Typist
105	PRAVEEN	15000	Cashier

// NOT EXISTS

```
SELECT * FROM employee1 WHERE
NOT EXISTS (SELECT * FROM employee1 WHERE
    salary>20000 ); no data found
```

RESULT:

Databases are created and SQL queries are retrieved information successfully.

Ex.No:5	QUERY THE DATABASE TABLES AND EXPLORE NATURAL , EQUI AND OUTER JOIN OPERATIONS
Date:	

AIM:

Creation of a database and execute SQL Joins to retrieve information from the database.

Theory:

JOINs in SQL are used to combine rows from two or more tables based on a related column. There are various types of JOINs, each serving a different purpose depending on how you want to retrieve data.

OUTER JOIN

An OUTER JOIN returns records that have a match in one of the tables, plus unmatched rows from one or both tables.

LEFT OUTER JOIN (LEFT JOIN)

- Returns all rows from the left table, and the matched rows from the right table.
- If there is no match, NULLs are returned from the right table.

RIGHT OUTER JOIN (RIGHT JOIN)

- Returns all rows from the right table, and the matched rows from the left table.
- If there is no match, NULLs are returned from the left table

FULL OUTER JOIN

- Returns all rows when there is a match in either left or right table.
- Non-matching rows will contain NULLs for the missing side.

CROSS JOIN

- Returns the Cartesian product of two tables.
- Every row of the first table is paired with every row of the second.

EQUI JOIN

An EQUI JOIN is a JOIN where the condition is based on an equality (=) comparison between columns.

NON EQUI JOIN

A NON EQUI JOIN is a JOIN that uses comparison operators other than (=) like <, >, <=, >=, !=, `BETWEEN`, etc.

// Table Creation - cseitstudent

```
CREATE TABLE
cseitstudent (
        studentID INT PRIMARY
        KEY, sname VARCHAR(30),
        department
        VARCHAR(5),
        sem INT
);
```

-Table Created

// Table Creation - placement

```
CREATE TABLE
placement (
        PlacementID INT PRIMARY
        KEY, StudentID INT,
        department
        VARCHAR(5),
        Company
        VARCHAR(30),
        salary INT
);
```

-Table Created

// Inserting values into cseitstudent table

INSERT INTO cseitstudent (studentID, sname, department, sem) VALUES(101,'reema',

'IT',5); INSERT INTO cseitstudent (studentID, sname, department, sem)

VALUES(102,'reenu', 'IT',3);

INSERT INTO cseitstudent (studentID, sname, department, sem)
VALUES(103,'sheela', 'CSE',3);

INSERT INTO cseitstudent (studentID, sname, department, sem) VALUES(104,'nirmal', 'IT',3);

INSERT INTO cseitstudent (studentID, sname, department, sem)
VALUES(105,'eshwar', 'CSE',5);

- 5 row(s) inserted

// Inserting values into placement table

INSERT INTO placement VALUES(1, 104, 'IT', 'infosys', 25000);

INSERT INTO placement VALUES(2, 105, 'CSE', 'Wipro', 22000);

INSERT INTO placement VALUES(3, 204, 'MECH', 'HYUNDAI',

30000);

INSERT INTO placement VALUES(4, 102, 'IT', 'infosys', 25000);

INSERT INTO placement VALUES(5, 103, 'CSE', 'infosys',

25000);

- 5 row(s) inserted

// Display the values in the table as rows

SELECT * FROM cseitstudent;

STUDENTID	SNAME		DEPARTMENT	SEM
101	reema	IT	5	
102	reenu	IT	3	
103	sheela	CSE	3	
104	nirmal	IT	3	
105	eshwar	CSE	5	

SELECT * FROM placement;

PLACEMENTID	STUDENTID	DEPARTMENT	COMPANY	SALARY
1	104	IT	infosys	25000
2	105	CSE	Wipro	22000
3	204	MECH	HYUNDAI	30000
4	102	IT	Infosys	25000
5	103	CSE	Infosys	25000

//LEFT OUTER JOIN

*SELECT *FROM cseitstudent*
LEFT OUTER JOIN placement
ON cseitstudent.studentID=placement.studentID;

STUDENTID	SNAME	DEPARTMENT	SEM	PLACEMENTID	STUDENTID	DEPARTMENT	COMPANY	SALARY
104	nirmal	IT	3	1	104	IT	infosys	25000
105	eshwar	CSE	5	2	105	CSE	Wipro	22000
102	reenu	IT	3	4	102	IT	infosys	25000
103	sheela	CSE	3	5	103	CSE	infosys	25000
101	reema	IT	5	-	-	-	-	-

SELECT cseitstudent.sname,placement.placementID,
placement.company FROM cseitstudent
LEFT OUTER JOIN placement
ON cseitstudent.studentID=placement.studentID;

SNAME	**PLACEMENTID**	**COMPANY**
nirmal	1	infosys
eshwar	2	Wipro
reenu	4	Infosys

sheela	5	infosys
reema	-	-

//RIGHT OUTER JOIN

```
SELECT *
FROM cseitstudent
RIGHT OUTER JOIN placement
ON cseitstudent.studentID=placement.studentID;
```

STUDENTID	SNAME	DEPARTMENT	SEM	PLACEMENTID	STUDENTID	DEPARTMENT	COMPANY	SALARY
102	reenu	IT	3	4	102	IT	infosys	25000
103	sheela	CSE	3	5	103	CSE	infosys	25000
104	nirmal	IT	3	1	104	IT	Infosys	25000
105	eshwar	CSE	5	2	105	CSE	Wipro	22000
-	-	-	-	3	204	*MECH*	*HYUNDAI*	*30000*

```
SELECT cseitstudent.sname,placement.placementID,
placement.company FROM cseitstudent
RIGHT OUTER JOIN placement
ON cseitstudent.studentID = placement.studentID;
```

SNAME	PLACEMENTID	COMPANY
reenu	4	infosys
sheela	5	infosys
nirmal	1	infosys
eshwar	2	Wipro
	3	HYUNDAI

//FULL OUTER JOIN

```
SELECT *
FROM cseitstudent
FULL OUTER JOIN placement
ON cseitstudent.studentID=placement.studentID;
```

STUDENTID	SNAME	DEPARTMENT	SEM	PLACEMENTID	STUDENTID	DEPARTMENT	COMPANY	SALARY
104	nirmal	IT	3	1	104	IT	infosys	25000
105	eshwar	CSE	5	2	105	CSE	Wipro	22000
-	-	-		3	204	MECH	HYUNDAI	30000
102	reenu	IT	3	4	102	IT	infosys	25000
103	sheela	CSE	3	5	103	CSE	infosys	25000
101	reema	IT	5	-	-	-	-	-

```
SELECT cseitstudent.sname,placement.placementID,
placement.company FROM cseitstudent
FULL OUTER JOIN placement
ON cseitstudent.studentID=placement.studentID;
```

SNAME	PLACEMENTID	COMPANY
nirmal	1	infosys
eshwar	2	Wipro
-	3	HYUNDAI
reenu	4	infosys
sheela	5	infosys
reema	-	-

//EQUI JOIN

SELECT * FROM cseitstudent, placement WHERE cseitstudent.studentID=placement.studentID;

STUDENTID	SNAME	DEPARTMENT	SEM	SALARY	STUDENTID	DEPARTMENT	COMPANY	PLACEMENTID
104	nirmal	IT	3	1	104	IT	infosys	25000
105	eshwar	CSE	5	2	105	CSE	Wipro	22000
102	reenu	IT	3	4	102	IT	infosys	25000
103	sheela	CSE	3	5	103	CSE	infosys	25000

// NON EQUI JOIN

SELECT cseit.studentID, cseit.sname FROM cseitstudent cseit, placement placed WHERE cseit.studentID>placed.studentID;

STUDENTID	SNAME
105	eshwar
105	eshwar
105	eshwar
104	nirmal
104	nirmal
103	sheela

// EXISTS

```
SELECT * FROM placement WHERE EXISTS (
SELECT * FROM placement WHERE
salary>20000);
```

PLACEMENTID	STUDENTID	DEPAR	COMPANY	SALARY
1	104	IT	infosys	25000
2	105	CSE	Wipro	22000
3	204	MECH	HYUNDAI	30000
4	102	IT	infosys	25000
5	103	CSE	infosys	25000

```
SELECT * FROM employee1 WHERE EXISTS
(SELECT * FROM employee1 WHERE salary>20000 );
```

EMPNO	EMPNAME	SALARY	DESIGNATION
101	RAM	45000	Manager
102	LINGAM	35000	Asst. Manager
103	RUPESH	20000	Clerk
104	BALA	10000	Typist
105	PRAVEEN	15000	Cashier

// NOT EXISTS

```
SELECT * FROM employee1 WHERE
NOT EXISTS (SELECT * FROM employee1 WHERE
    salary>20000 ); no data found
```

// NATURAL JOIN

syntax.

```
SELECT *

FROM TABLE1

NATURAL JOIN

TABLE2;
```

Table-1:
department –
Create Table
department

```
(
  DEPT_NAME Varchar(20),
  MANAGER_NAME
  Varchar(255)
);
```

Table-2:
employee –
Create Table
employee

```
(
  EMP_ID int,
  EMP_NAME
  Varchar(20),
  DEPT_NAME Varchar(255)
);
```

Inserting values :

Add value into the tables as follows.

```
INSERT INTO DEPARTMENT(DEPT_NAME,MANAGER_NAME)VALUES(“IT", "ROHAN");
INSERT INTO DEPARTMENT(DEPT_NAME,MANAGER_NAME) VALUES ( "SALES", "RAHUL");
INSERTINTODEPARTMENT(DEPT_NAME,MANAGER_NAME)VALUES("HR","TANMAY"
);
INSERTINTODEPARTMENT(DEPT_NAME,MANAGER_NAME)VALUES("FINANCE","ASHISH")
;
INSERTINTODEPARTMENT(DEPT_NAME,MANAGER_NAME)VALUES("MARKETING",
"SAMAY");
INSERT INTO EMPLOYEE(EMP_ID, EMP_NAME, DEPT_NAME) VALUES (1, "SUMIT",
"HR");
INSERT INTO EMPLOYEE(EMP_ID, EMP_NAME, DEPT_NAME) VALUES (2, "JOEL",
"IT");
INSERT INTO EMPLOYEE(EMP_ID, EMP_NAME, DEPT_NAME) VALUES (3, "BISWA",
"MARKETING");
INSERTINTO EMPLOYEE(EMP_ID, EMP_NAME, DEPT_NAME)VALUEs(4, "VAIBHAV", "IT")
INSERT INTO EMPLOYEE(EMP_ID, EMP_NAME, DEPT_NAME) VALUES (5, "SAGAR",
"SALES");
```

Verifying inserted data :

This is our data inside the table as follows.
*SELECT * FROM EMPLOYEE;*

Output :

EMP_ID	EMP_NAME	DEPT_NAME
1	SUMIT	HR
2	JOEL	IT
3	BISWA	MARKETING
4	VAIBHAV	IT
5	SAGAR	SALES

SELECT * FROM DEPARTMENT;

Output :

DEPT_NAME	MANAGER_NAME
IT	ROHAN
SALES	RAHUL
HR	TANMAY
FINANCE	ASHISH
MARKETING	SAMAY

Query to implement SQL Natural Join :

```
SELECT *
FROM EMPLOYEE
NATURAL JOIN DEPARTMENT;
```

EMP_ID	EMP_NAME	DEPT_NAME	MANAGER_NAME
1	SUMIT	HR	TANMAY
2	JOEL	IT	ROHAN
3	BISWA	MARKETING	SAMAY
4	VAIBHAV	IT	ROHAN
5	SAGAR	SALES	RAHUL

RESULT:

Tables are created and join operation has been executed successfully.

Ex.No:6	USER DEFINED FUNCTIONS AND STORED PROCEDURES IN SQL
Date:	

AIM:

Creation of user defined functions and stored procedures by using Structures Query

Theory

In SQL, **User-Defined Functions (UDFs)** and **Stored Procedures** are used to encapsulate SQL logic for reuse. They help in modularizing code, improving performance, and enhancing readability and maintainability.

User Defined Functions (UDFs)

Scalar Function

A **scalar function** returns a single value.

Table-Valued Function

A **table-valued function** returns a table.

Stored Procedures

Stored procedures allow you to perform actions such as updates or inserts.

Input Table: employees

Id	name	department	salary
1	Alice	HR	50000
2	Bob	Finance	60000
3	Charlie	IT	55000
4	David	Marketing	48000
5	Ethan	HR	47000

Input Table: departments

department_name	location
HR	New York
Finance	Chicago
IT	San Francisco
Marketing	Boston

Scalar Function

A **scalar function** returns a single value.

Below is an example that calculates yearly salary based on a monthly salary input.

```
-- Function to calculate yearly salary
CREATE FUNCTION fn_CalculateYearlySalary (@monthlySalary
INT)
RETURNS INT
AS
BEGIN
   RETURN @monthlySalary * 12;
END;
```

Usage Example:

```
SELECT name, dbo.fn_CalculateYearlySalary(salary) AS
yearly_salary
FROM employees;
```

Sample Output:

name	yearly_salary
Alice	600000
Bob	720000
Charlie	660000

Table-Valued Function

A **table-valued function** returns a table. The example below returns all employees from a specified department.

```
-- Function to get employees by department
CREATE FUNCTION fn_GetEmployeesByDepartment (@dept
VARCHAR(50))
RETURNS TABLE
AS
RETURN (
   SELECT * FROM employees WHERE department = @dept
);
```

Example:

```
SELECT * FROM dbo.fn_GetEmployeesByDepartment('HR');
```

Sample Output:

id	name	department
1	Alice	HR
5	Ethan	HR

Stored Procedures

Example: Update Employee Salary

Stored procedures allow you to perform actions such as updates or inserts. Here's a stored procedure to update an employee's salary:

```
-- Procedure to update employee salary
CREATE PROCEDURE sp_UpdateSalary
   @empId INT,
   @newSalary INT
AS
BEGIN
   UPDATE employees
   SET salary = @newSalary
   WHERE id = @empId;
END;
```

Execution:

```
EXEC sp_UpdateSalary @empId = 3, @newSalary = 58000;
```

Updated Employee Table (Sample Output):

id	name	salary
3	Charlie	58000

RESULT:

Databases are created user defined function and procedure in SQL queries are retrieved information successfully.

Ex.No:7	Execute Complex transactions and realize DCL and TCL Commands
Date:	

AIM:

To write SQL queries using DCL commands to manage the database.

DESCRIPTION:

The DCL language is used for controlling the access to the table and hence securing the Database. DCL is used to provide certain privileges to a particular user. Privileges are rights to be allocated. The privilege commands are namely,

- Grant
- Revoke
- Commit
- Savepoint
- Rollback

GRANT COMMAND: It is used to create users and grant access to the database. It requires database administrator (DBA) privilege, except that a user can change their password. A user can grant access to their database objects to other users.

REVOKE COMMAND: Using this command , the DBA can revoke the granted database privileges from the user.

COMMIT : It is used to permanently save any transaction into database.

SAVEPOINT: It is used to temporarily save a transaction so that you can rollback to that point whenever necessary

ROLLBACK: It restores the database to last commited state. It is also use with savepoint command to jump to a savepoint in a transaction.

Queries:

Consider the following tables namely “DEPARTMENTS” and “EMPLOYEES” Their schemas are as follows ,

Departments (dept _no , dept_ name , dept_location); Employees (emp_id , emp_name , emp_salary);

SQL> Grant all on employees to abcde; Grant succeeded.

SQL> Grant select , update , insert on departments to abcde with grant option; Grant succeeded.

SQL> Revoke all on employees from abcde;
Revoke succeeded.

SQL> Revoke select , update , insert on departments from abcde;

Revoke succeeded.

COMMIT, ROLLBACK and SAVEPOINT:

SQL> select * from class;

NAME	ID
anu	1
brindha	2
chinthiya	3
divya	4
ezhil	5
fairoz	7

SQL> insert into class values('gayathri',9);

1 row created.

SQL> commit;
Commit complete.

SQL> update class set name='hema' where id='9';
1 row updated.

SQL> savepoint A;
Savepoint created.

SQL> insert into class values('indu',11); 1
row created.

SQL> savepoint B; Savepoint created.

SQL> insert into class values('janani',13);
1 row created.

SQL> select * from class;

NAME	ID
Anu	1
brindha	2
chinthiya	3
divya	4
ezhil	5
fairoz	7
hema	9
indu	11
janani	13

9 rows
selected.

SQL> rollback to B;

Rollback complete.

SQL> select * from

class;

NAME	ID
anu	1
brindha	2
chinthiya	3
divya	4
ezhil	5
fairoz	7
hema	9
indu	11

8 rows selected.

SQL> rollback to A;

Rollback complete.

SQL>

select * from class;

NAME	ID
anu	1
brindha	2
chinthiya	3
divya	4
ezhil	5

fairoz	7
hema	9
indu	11

RESULT:

Thus the creation of database with various constraints and the SQL queries to retrieve information from the database using DCL statements has been implemented and the output was verified successfully.

Ex.No:8	SQL TRIGGERS FOR INSERT, DELETE, AND UPDATE OPERATIONS IN DATABASE TABLE
Date:	

AIM:

To Create a SQL triggers for insert, delete and update operations in database table

Theory:

A **Trigger** in SQL is a special type of stored procedure that **automatically executes** (or fires) when a specific event occurs in the database. It is typically used to enforce business rules, maintain audit logs, or automatically update related data in other tables.

Triggers are activated in response to the following **Data Manipulation Language (DML)** events:

- INSERT
- UPDATE
- DELETE

There are also **DDL (Data Definition Language) triggers** and **Logon triggers**, but the most commonly used are DML triggers.

Types of Triggers

1. BEFORE Trigger (Not supported in all RDBMS):

- Executes **before** the actual operation (insert/update/delete).
- Mainly used in Oracle or PostgreSQL.

2. AFTER Trigger:

- Executes **after** the operation has been completed.
- Used for tasks like audit logs or history tracking.

3. INSTEAD OF Trigger:

- Replaces the standard action with custom logic.
- Commonly used on **views** where direct modification is restricted.

TRIGGER

SYNTAX:

```
 CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE| DELETE }
ON table_name FOR EACH
ROW trigger_body;
```

EXAMPLE-1:

```
 mysql>select * from emp;

        empno  empname       offcode
        100    kalai         111
        101    rama          222
        102    began         333

mysql>create table emp_audit(id int auto_increment primary key,
        empno int,empname varchar(20),cdate datetime,action varchar(20));
```

TRIGGER CREATION :

```
mysql>create trigger
        before_emp_update before
        update on emp
        for each row
        insert into
        emp_audit set
        action = 'update',
        empno = old.empno,
        empname =
        old.empname,
        cdate = now();
```

UPDATE EMP TABLE:

```
mysql> update emp set empname = 'gokul' where empno = 100;
```

SELECT-EMP_AUDIT

mysql> select * from emp_audit;

id	empno	empname	cdate	action
1	100	gokul	2019-12-12 13-02-17	update

SELECT –EMP:

mysql> select * from emp1;

empno	empname	offcode
100	gokul	111
101	rama	222
102	began	333

RESULT:

Thus, the Trigger was executed successfully.

Ex.No:9	CREATE VIEW AND INDEX FOR DATABASE TABLES WITH LARGE NUMBER OF RECORDS
Date:	

Aim:

To Create the View and index by using Structures Query Language.

Theory

VIEWS in SQL

A View in SQL is a virtual table that is based on the resultset of a SQL query. Unlike real tables, views do not store data physically. Instead, they display data stored in other tables.

To simplify complex queries.

To hide specific columns or rows for security.

To present data in a particular format or structure.

To reduce code duplication.

INDEXES in SQL

An **Index** is a database object used to **speed up the retrieval of rows** from a table. It works like an index in a book – allowing the database to locate data without scanning every row.

Indexes are especially useful in **large tables** and **frequent search or join operations**.

Syntax :CREATE VIEW view_name AS SELECT

column_name(s) FROM table_name WHERE condition

DEFINITION:

A **view**, is a logical table based on one or more tables or views. A view contains no data itself. The tables upon which a view is based are called **base tables**.

SYNTAX:

CREATE VIEW view_name

AS

<Query Expression>

QUERY:

create table student(sid int, sname varchar(50), dept varchar(5));

OUTPUT:

table STUDENT created.

QUERY:

insert into student values(1,'bala','IT');

insert into student values(2,'rupesh','IT');

insert into student values(3,'arthi','cse');

OUTPUT:

1 rows inserted.

1 rows inserted.

1 rows inserted.

QUERY:

create view studview as select * from student where dept='IT'

OUTPUT:

view STUDVIEW created.

QUERY:

select * from studview;

OUTPUT:

SID	SNAME	DEPT
1	bala	IT
2	rupesh	IT

QUERY:insert into studview values(7,'anand','IT');

OUTPUT:1 rows inserted.

QUERY:select * from studview;

OUTPUT:

SID	SNAME	DEPT
1	bala	IT
2	rupesh	IT

7 anand IT

QUERY:update studview set sid=5 where sid=2;

OUTPUT:

1 rows updated.

QUERY:select * from studview;

OUTPUT:

SID	SNAME	DEPT
1	bala	IT
5	rupesh	IT
7	anand	IT

QUERY:

delete from studview where sid = 5;

OUTPUT:1 rows deleted

QUERY:select * from studview;

OUTPUT:

SID	SNAME	DEPT
1	bala	IT
7	art	IT

QUERY:drop view studview;

OUTPUT:view STUDVIEW dropped.

DEFINITION:An **index** is a performance-tuning method of allowing faster retrieval of records. An index creates an entry for each value that appears in the indexed columns.

SYNTAX:Create index:

CREATE INDEX idx_name ON table_name(attribute);

Drop Index:

DROP INDEX index_name;

QUERY:

create index idx_stud on student(sid);

OUTPUT:

index IDX_STUD created.

QUERY:

drop index idx_stud;

OUTPUT:

index IDX_STUD dropped.

Result:

Thus, the SQL queries using views were successfully executed and verified.

Ex.No:10 Date:	**Case Study (Inventory Management for a EMart Grocery Shop)**

Aim:

Write the coding for Inventory Management System.

Description:

Inventory control system helps in monitoring the sales and various stocks available

DATABASE:

STOCK TABLE

CREATE TABLE stock

(

prodid INT PRIMARY KEY, prodname VARCHAR2(50), quantity INT, unitprice INT, reorderlevel INT

);

SALES TABLE

CREATE TABLE sales

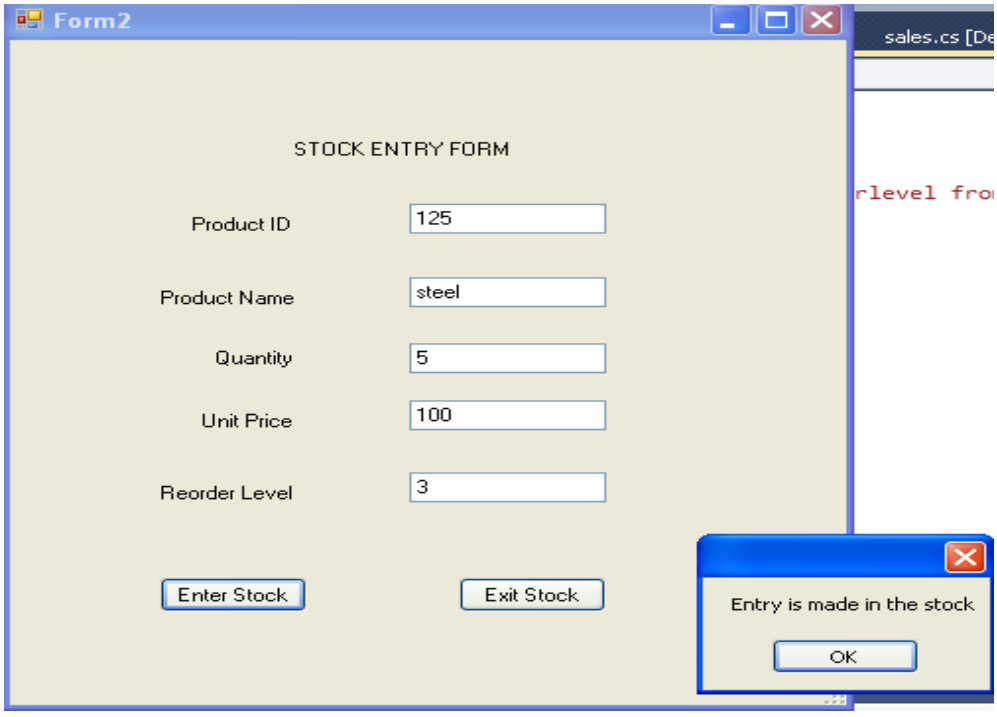
(

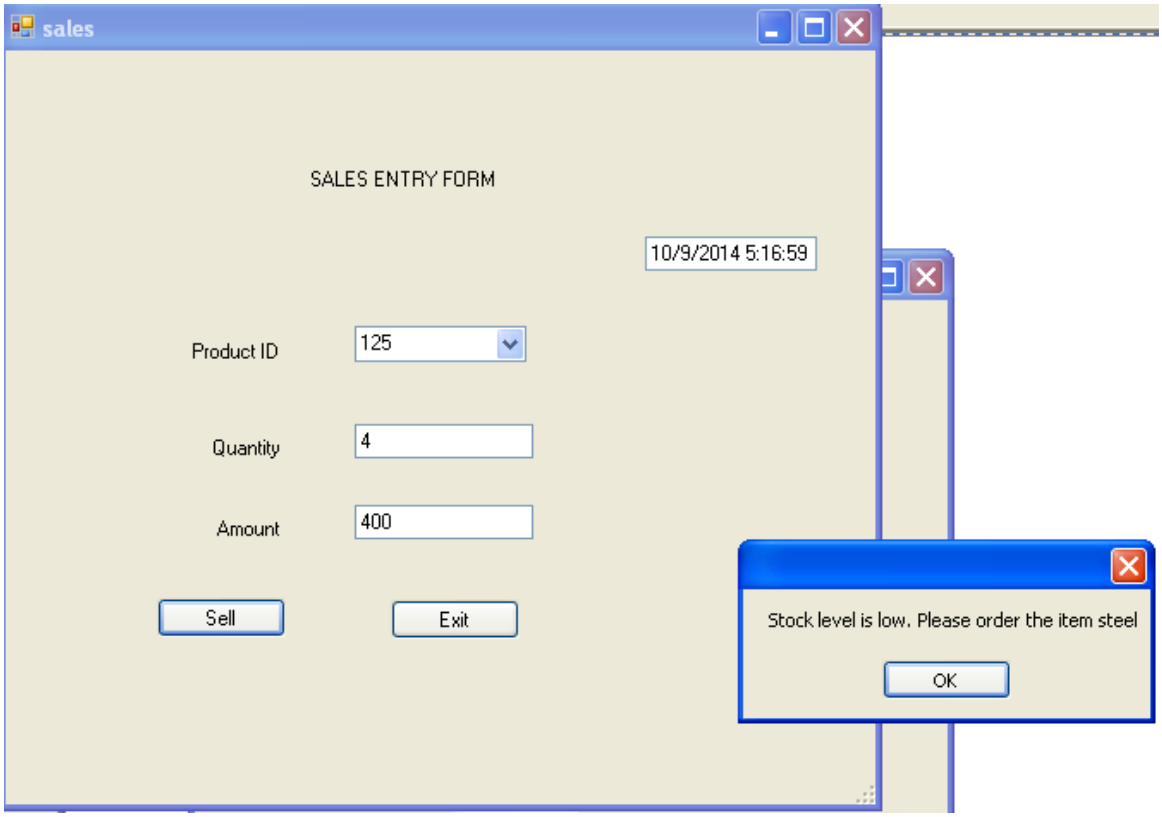
prodid INT REFERENCES stock(prodid), salesqty INT, amount INT, salesdate DATE

);

FORMS:

Inventory control system form:





CODING FOR SALES FORM:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using Oracle.DataAccess.Client;
namespace inventory1
{
   public partial class sales : Form
   {
      public sales()
      {
```

```
            InitializeComponent();
        }
        OracleConnection conn;
        private void sales_Load(object sender, EventArgs e)
        {
            string oradb = "Data Source=172.18.1.6;User Id=cselab;Password=cselab;";
            conn = new OracleConnection(oradb);
            conn.Open();

            string sel1 = "select prodid from cselab.stock";
            OracleCommand cmd = new OracleCommand(sel1, conn);
            OracleDataReader dr = cmd.ExecuteReader();
            while (dr.Read())
            {
                cmboPid.Items.Add(dr["prodid"].ToString());
            }
            DateTime dt = DateTime.Now;
            txtSalesdate.Text = dt.ToString();
        }
        private void btnSell_Click(object sender, EventArgs e)
        {
            Boolean result=validate();
            if (result==true)
            {
                MessageBox.Show("Insufficient Stock");
                return;
            }
            DateTime dt = DateTime.Now;
            String ins="Insert into cselab.sales(prodid,salesqty,amount,salesdate) values
(:prodid,:salesqty,:amount,:salesdate)";
            OracleCommand cmd = new OracleCommand(ins, conn);
            cmd.Parameters.Add("prodid", cmboPid.SelectedItem);
            cmd.Parameters.Add("salesqty", txtSalesqty.Text);
            cmd.Parameters.Add("amount", txtAmt.Text);
            cmd.Parameters.Add("salesdate", dt);
            cmd.ExecuteNonQuery();
            MessageBox.Show("Product sold successfully");
            updatestock();
            clear();
```

```
        }
         public Boolean validate()
        {
            String sel1 = "select quantity,reorderlevel,prodname from cselab.stock where
prodid=:prodid";
            OracleCommand cmd = new OracleCommand(sel1, conn);
            cmd.Parameters.Add("prodid", cmboPid.SelectedItem);
            OracleDataReader dr = cmd.ExecuteReader();
            int relevel = 0;
            int qty = 0;
            if (dr.Read())
            {
                qty = int.Parse(dr["quantity"].ToString());
                relevel = int.Parse(dr["reorderlevel"].ToString());
            }
            int sqty = int.Parse(txtSalesqty.Text);
            if (sqty > qty) // Insuficient stock
            {
                return true;
            }
            int afterSale = qty - sqty;
            if (afterSale < relevel)
            {
                MessageBox.Show("Stock level is low. Please order the item " +
dr["prodname"].ToString());
            }
            return false;
        }
         private void txtSalesqty_Validated(object sender, EventArgs e)
         {
             int amt = getAmount();
             txtAmt.Text = amt.ToString();
         }
        private int getAmount()
        {
            string sel = "SELECT unitprice FROM cselab.stock WHERE prodid=:prodid";
            OracleCommand cmd = new OracleCommand(sel, conn);
            cmd.Parameters.Add("prodid", cmboPid.SelectedItem);
            OracleDataReader dr = cmd.ExecuteReader();
```

```
            int uprice = 0;
            if (dr.Read())
            {
                uprice = int.Parse(dr["unitprice"].ToString());
            }
            int sqty = int.Parse((txtSalesqty.Text).ToString());
            int amt = uprice * sqty;
            return amt;
        }
        public void updatestock()
        {
            string upd = "update cselab.stock set quantity=quantity-:salesqty where prodid=:prodid";
            OracleCommand cmd = new OracleCommand(upd, conn);
            cmd.Parameters.Add("salesqty", txtSalesqty.Text);
            cmd.Parameters.Add("prodid", cmboPid.SelectedItem);
            cmd.ExecuteNonQuery();
        }
        void clear()
        {
            cmboPid.Text = "";
            txtSalesqty.Text = "";
            txtAmt.Text = "";
            txtSalesdate.Text = "";
        }

        private void btnXit1_Click(object sender, EventArgs e)
        {
            Close();
        }
    }
}
```

CODING FOR STOCK FORM:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
```

```
using System.Text;
using System.Windows.Forms;
using Oracle.DataAccess.Client;
namespace inventory1
{
   public partial class stock : Form
   {
      public stock()
      {
         InitializeComponent();
      }
      OracleConnection conn;
      Boolean IsExist;

      private void stock_Load(object sender, EventArgs e)
      {
         string oradb = "Data Source=172.18.1.6;User Id=cselab;Password=cselab;";
         conn = new OracleConnection(oradb);
         conn.Open();
      }
      void clear()
      {
         txtPid.Text = "";
         txtPname.Text = "";
         txtPrice.Text = "";
         txtQty.Text = "";
         txtReorder.Text = "";
      }
      private void textBox3_TextChanged(object sender, EventArgs e)
      {

      }
      private void btnStock1_Click(object sender, EventArgs e)
      {
          if (IsExist)
          {
            String upd = "update cselab.stock set quantity=quantity + :qty where prodid=:prodid";
            OracleCommand cmd2 = new OracleCommand(upd, conn);
            cmd2.Parameters.Add("qty", txtQty.Text);
```

```
            cmd2.Parameters.Add("prodid", txtPid.Text);
            cmd2.ExecuteNonQuery();
            MessageBox.Show("Quantity is updated");
            clear();
          }
          else
          {
            String ins = "INSERT INTO cselab.stock(prodid, prodname, quantity, unitprice,
reorderlevel) values(:prodid, :prodname, :quantity, :unitprice, :reorderlevel)";
            OracleCommand cmd = new OracleCommand(ins, conn);
            cmd.Parameters.Add("prodid", txtPid.Text);
            cmd.Parameters.Add("prodname", txtPname.Text);
            cmd.Parameters.Add("quantity", txtQty.Text);
            cmd.Parameters.Add("unitprice", txtPrice.Text);
            cmd.Parameters.Add("reorderlevel", txtReorder.Text);
            cmd.ExecuteNonQuery();
            MessageBox.Show("Entry is made in the stock");
            clear(); ;
                  }
      }
      private void btnXit_Click(object sender, EventArgs e)
      {
        Close();
      }
      private void txtPid_Validated(object sender, EventArgs e)
      {
        String sel = "Select  prodid,prodname,unitprice,reorderlevel from cselab.stock where
prodid=:prodid";
        OracleCommand cmd1 = new OracleCommand(sel, conn);
        cmd1.Parameters.Add("prodid", txtPid.Text);
        String pid = txtPid.Text;
        OracleDataReader dr = cmd1.ExecuteReader();
        IsExist = false;
        if (dr.Read())
        {
          txtPname.Text = dr["prodname"].ToString();
          txtPrice.Text = dr["unitprice"].ToString();
          txtReorder.Text = dr["reorderlevel"].ToString();
          txtPname.Enabled = false;
```

```
            txtPrice.Enabled = false;
            txtReorder.Enabled = false;
            IsExist = true;
        }
    }

}
}
```

ESULT:

Thus, the Coding was executed successfully

Ex.No:11 Date:	**To Study and develop the application using concept of MongoDB.**

Aim

To study and implement NoSQL database operations using **MongoDB** by performing CRUD operations, queries, and indexing on a collection.

Software Required

- MongoDB Server
- MongoDB Shell (mongosh)
- Operating System: Windows / Linux

Theory (Brief)

MongoDB is a **NoSQL document-oriented database** that stores data in **JSON-like documents (BSON)**. It is schema-less, scalable, and suitable for big data and real-time applications.

Experiment Description

Create a database for a **Student Management System** and perform:

- Database creation
- Collection creation
- Insert, Read, Update, Delete (CRUD) operations
- Querying with conditions
- Index creation

Step 1: Create Database

use studentDB

Output

switched to db studentDB

Step 2: Create Collection & Insert Documents

```
db.students.insertMany([
  { rollno: 101, name: "Arun", dept: "CSE", marks: 85 },
  { rollno: 102, name: "Meena", dept: "IT", marks: 78 },
  { rollno: 103, name: "Ravi", dept: "CSE", marks: 92 },
  { rollno: 104, name: "Priya", dept: "ECE", marks: 67 }
])
```

Output

```
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("..."),
    '1': ObjectId("..."),
    '2': ObjectId("..."),
    '3': ObjectId("...")
  }
}
```

Step 3: Display All Documents

```
db.students.find()
```

Output

```
{ rollno: 101, name: 'Arun', dept: 'CSE', marks: 85 }
{ rollno: 102, name: 'Meena', dept: 'IT', marks: 78 }
{ rollno: 103, name: 'Ravi', dept: 'CSE', marks: 92 }
{ rollno: 104, name: 'Priya', dept: 'ECE', marks: 67 }
```

Step 4: Query with Condition (marks > 80)

```
db.students.find({ marks: { $gt: 80 } })
```

Output

{ rollno: 101, name: 'Arun', dept: 'CSE', marks: 85 }

{ rollno: 103, name: 'Ravi', dept: 'CSE', marks: 92 }

Step 5: Update Document

```
db.students.updateOne(
  { rollno: 102 },
  { $set: { marks: 82 } }
)
```

Output

{ acknowledged: true, matchedCount: 1, modifiedCount: 1 }

Step 6: Delete Document

```
db.students.deleteOne({ rollno: 104 })
```

Output

{ acknowledged: true, deletedCount: 1 }

Step 7: Display After Deletion

```
db.students.find()
```

Output

{ rollno: 101, name: 'Arun', dept: 'CSE', marks: 85 }

{ rollno: 102, name: 'Meena', dept: 'IT', marks: 82 }

{ rollno: 103, name: 'Ravi', dept: 'CSE', marks: 92 }

Step 8: Sorting Records (Descending Marks)

```
db.students.find().sort({ marks: -1 })
```

Output

{ rollno: 103, name: 'Ravi', dept: 'CSE', marks: 92 }

{ rollno: 101, name: 'Arun', dept: 'CSE', marks: 85 }

{ rollno: 102, name: 'Meena', dept: 'IT', marks: 82 }

Step 9: Create Index

db.students.createIndex({ rollno: 1 })

Output

rollno_1

Result

Thus, the MongoDB NoSQL database was successfully created and CRUD operations, conditional queries, sorting, and indexing were executed with appropriate outputs.

Topic Beyond Syllabus:

Data Masking and Privacy Protection in DBMS

Aim

To implement **data masking techniques** in a relational database to protect sensitive user information such as email IDs and phone numbers while allowing controlled data access.

Software Required

- Operating System: Windows / Linux
- Database: **MySQL 8.0** (or PostgreSQL / Oracle)
- Tool: MySQL Workbench / Command Line

Theory

Data masking is a security technique used to hide sensitive data by replacing original values with masked values.
It is commonly used in **banking, healthcare, and enterprise applications** to protect user privacy while allowing data analysis.

Algorithm

1. Create a database and select it.
2. Create a table with sensitive data.
3. Insert sample records.
4. Display original data.
5. Create a view that masks sensitive fields.
6. Query masked data through the view.
7. Verify that original data remains unchanged.

Steps to Execute

Step 1: Create Database

CREATE DATABASE PrivacyDB;

```
USE PrivacyDB;
```

Step 2: Create Table with Sensitive Data

```
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    email VARCHAR(50) UNIQUE,
    phone VARCHAR(15),
    city VARCHAR(30)
);
```

Step 3: Insert Records

```
INSERT INTO Customers VALUES
(1, 'Rahul', 'rahul@gmail.com', '9876543210', 'Bangalore'),
(2, 'Anita', 'anita@yahoo.com', '9123456780', 'Chennai'),
(3, 'Vikram', 'vikram@outlook.com', '9988776655', 'Mumbai');
```

Step 4: Display Original Data

```
SELECT * FROM Customers;
```

Output

```
+-------------+--------+---------------------+------------+-----------+
| customer_id | name   | email               | phone      | city      |
+-------------+--------+---------------------+------------+-----------+
| 1           | Rahul  | rahul@gmail.com     | 9876543210 | Bangalore |
| 2           | Anita  | anita@yahoo.com     | 9123456780 | Chennai   |
| 3           | Vikram | vikram@outlook.com  | 9988776655 | Mumbai    |
+-------------+--------+---------------------+------------+-----------+
```

Step 5: Create Masked View

```
CREATE VIEW Masked_Customers AS
SELECT
   customer_id,
   name,
   CONCAT(SUBSTRING(email,1,3),'****@****') AS masked_email,
   CONCAT('XXXXXX', SUBSTRING(phone,7,4)) AS masked_phone,
   city
FROM Customers;
```

Step 6: Display Masked Data

```
SELECT * FROM Masked_Customers;
```

Output

```
+-------------+--------+-----------------+--------------+-----------+
| customer_id | name   | masked_email    | masked_phone | city      |
+-------------+--------+-----------------+--------------+-----------+
| 1           | Rahul  | rah****@****    | XXXXXX3210   | Bangalore |
| 2           | Anita  | ani****@****    | XXXXXX6780   | Chennai   |
| 3           | Vikram | vik****@****    | XXXXXX6655   | Mumbai    |
+-------------+--------+-----------------+--------------+-----------+
```

Result

Thus, data masking was successfully implemented using SQL views. Sensitive information such as **email IDs and phone numbers** was protected while maintaining access to non-sensitive data.

Experiment 1: Create a database table, add constraints, and perform DDL & DML operations

1. **What is the difference between DDL and DML commands?**
2. **What is a primary key and why is it important?**
3. **How does a UNIQUE constraint differ from a PRIMARY KEY?**
4. **What is the purpose of the CHECK constraint?**
5. **Can a table have multiple NOT NULL constraints?**
6. **What happens if you try to insert a NULL value into a NOT NULL column?**
7. **What is the syntax for updating a record in a table?**
8. **How does the DELETE command differ from the TRUNCATE command?**
9. **Can constraints be added after table creation? If yes, how?**
10. **What is the difference between DROP and DELETE?**

Experiment 2: Create tables with foreign key constraints and referential integrity

1. **What is a foreign key?**
2. **How does a foreign key maintain referential integrity?**
3. **What happens when a referenced parent record is deleted?**
4. **What is the difference between ON DELETE CASCADE and ON DELETE SET NULL?**
5. **Can a foreign key reference a non-primary key column?**
6. **What is a parent table and a child table?**
7. **How can referential integrity be violated?**
8. **Can a table have multiple foreign keys?**
9. **What are the advantages of enforcing referential integrity?**
10. **How do foreign keys improve database consistency?**

Experiment 3: WHERE clause conditions and aggregate functions

1. **What is the purpose of the WHERE clause?**
2. **What is the difference between WHERE and HAVING clauses?**
3. **Explain the use of BETWEEN and IN operators.**

4. **What are aggregate functions in SQL?**
5. **Explain COUNT(), SUM(), AVG(), MIN(), and MAX().**
6. **Can aggregate functions be used without GROUP BY?**
7. **What is the use of GROUP BY clause?**
8. **How does the LIKE operator work?**
9. **What is the difference between AND and OR conditions?**
10. **Can NULL values affect aggregate function results?**

Experiment 4: Subqueries and simple joins

1. **What is a subquery?**
2. **What is the difference between a subquery and a join?**
3. **What are correlated subqueries?**
4. **Can a subquery return multiple rows?**
5. **Where can subqueries be used in SQL statements?**
6. **What is an INNER JOIN?**
7. **How does a join improve query performance?**
8. **What happens if join conditions are missing?**
9. **Can joins be used with more than two tables?**
10. **Give a real-world example of a subquery.**

Experiment 5: Natural, Equi, and Outer Joins

1. **What is a join in SQL?**
2. **What is a natural join?**
3. **What is an equi join?**
4. **What is the difference between INNER JOIN and OUTER JOIN?**
5. **Explain LEFT OUTER JOIN.**
6. **Explain RIGHT OUTER JOIN.**
7. **What is a FULL OUTER JOIN?**
8. **What happens when matching records are not found?**

9. **Which join returns all rows from both tables?**
10. **When should outer joins be used?**

Experiment 6: User Defined Functions and Stored Procedures

1. **What is a stored procedure?**
2. **What is a user-defined function?**
3. **What is the difference between a function and a procedure?**
4. **Can functions return multiple values?**
5. **What are IN, OUT, and INOUT parameters?**
6. **What are the advantages of stored procedures?**
7. **Can stored procedures include transaction control statements?**
8. **How are functions invoked in SQL?**
9. **Can functions be used in SELECT statements?**
10. **What are the performance benefits of stored procedures?**

Experiment 7: DCL and TCL commands (Transactions)

1. **What is a transaction?**
2. **What are ACID properties?**
3. **What is the purpose of COMMIT?**
4. **What happens when ROLLBACK is executed?**
5. **What is SAVEPOINT?**
6. **What are DCL commands?**
7. **Explain GRANT and REVOKE commands.**
8. **How does TCL differ from DML?**
9. **Can transactions be nested?**
10. **What happens to uncommitted data during a system failure?**

Experiment 8: SQL Triggers

1. **What is a trigger in SQL?**

2. **When does a trigger get executed?**
3. **What is the difference between BEFORE and AFTER triggers?**
4. **Can triggers be used for auditing?**
5. **What events can activate a trigger?**
6. **Can a trigger call a stored procedure?**
7. **How many triggers can be created on a table?**
8. **What are row-level triggers?**
9. **Can triggers affect performance?**
10. **What are the limitations of triggers?**

Experiment 9: Views and Indexes

1. **What is a view in SQL?**
2. **What are the advantages of using views?**
3. **Can data be modified through a view?**
4. **What is an index?**
5. **How does an index improve query performance?**
6. **What are the different types of indexes?**
7. **Can a table have multiple indexes?**
8. **What is the difference between a view and a table?**
9. **When should indexes not be used?**
10. **Can indexes be created on large tables?**

Experiment 10: Case Study (Real-Life Database Application)

1. **What is the purpose of this database application?**
2. **What entities are involved in the system?**
3. **What is an Entity Relationship (ER) diagram?**
4. **How does normalization improve database design?**
5. **What normal forms were applied and why?**
6. **What are views used for in this case study?**

7. **Why are triggers required for auditing?**
8. **What real-life problems does this system solve?**
9. **How does the design ensure data integrity?**
10. **How can this system be scaled in the future?**

Experiment 11: MongoDB Application Development

1. **What is MongoDB?**
2. **How does MongoDB differ from relational databases?**
3. **What is a collection in MongoDB?**
4. **What is a document?**
5. **What is BSON?**
6. **How does MongoDB handle schema?**
7. **What are advantages of MongoDB?**
8. **What is indexing in MongoDB?**
9. **What are CRUD operations in MongoDB?**
10. **Where is MongoDB best suited in real-world applications?**
