MODULE 4 – INTRODUCTION TO DBMS(LAB PRACTICAL)

➢ Introduction to SQL

Task 1:

Create a new database named school_db and a table called students with the following columns:

- **student_id**
- **student_name**
- **age**
- **class**
- **address**

solution:-

```
-- Step 1: Create a new database
CREATE DATABASE school_db;

-- Step 2: Use the database
USE school_db;

-- Step 3: Create the students table
CREATE TABLE students (
   student_id INT PRIMARY KEY,
   student_name VARCHAR(50),
   age INT,
   class VARCHAR(20),
   address VARCHAR(100)
);
```

Task 2:

Insert five records into the students table and retrieve all records using the SELECT statement.

Solution:

```
INSERT INTO students (student_id, student_name, age, class, address)
VALUES
(1, 'Rahul Sharma', 12, '6th', 'Delhi'),
(2, 'Priya Verma', 11, '5th', 'Mumbai'),
(3, 'Aman Gupta', 13, '7th', 'Jaipur'),
(4, 'Sneha Patel', 10, '4th', 'Ahmedabad'),
(5, 'Vikram Singh', 12, '6th', 'Chennai');

SELECT * FROM students;
```

Expected Output:-

student_id	student_name	age	class	address
1	Rahul Sharma	12	6th	Delhi
2	Priya Verma	11	5th	Mumbai
3	Aman Gupta	13	7th	Jaipur
4	Sneha Patel	10	4th	Ahmedabad
5	Vikram Singh	12	6th	Chennai

➢ SQL Syntax

Task 1 :

Write SQL queries to retrieve specific columns (student_name and age) from the students table.

Solution:

```
SELECT student_name, age
FROM students;
```

Output :

student_name	age
Rahul Sharma	12
Priya Verma	11
Aman Gupta	13
Sneha Patel	10
Vikram Singh	12

Task 2:

Write SQL queries to retrieve all students whose age is greater than 10.

Solution :

```
SELECT *
FROM students
WHERE age > 10;
```

Output :

student_id	student_name	age	class	address
1	Rahul Sharma	12	6th	Delhi
2	Priya Verma	11	5th	Mumbai

student_id	student_name	age	class	address
3	Aman Gupta	13	7th	Jaipur
5	Vikram Singh	12	6th	Chennai

➢ SQL Constraints

Task1:

Create a table teachers with the following columns:

- **teacher_id (Primary Key)**
- **teacher_name (NOT NULL)**
- **subject (NOT NULL)**
- **email (UNIQUE)**

Solution:

```
CREATE TABLE teachers (
   teacher_id INT PRIMARY KEY,
   teacher_name VARCHAR(50) NOT NULL,
   subject VARCHAR(50) NOT NULL,
   email VARCHAR(100) UNIQUE
);
```

Task 2

Implement a FOREIGN KEY constraint to relate the teacher_id from the teachers table with the students table.

Solution:

First, we need to add a column teacher_id to the students table:

Sql quary:-

```
ALTER TABLE students
ADD teacher_id INT;
```

Then set the foreign key:

```
ALTER TABLE students
ADD CONSTRAINT fk_teacher
FOREIGN KEY (teacher_id) REFERENCES teachers(teacher_id);
```

Output :

- **ALTER TABLE students ADD teacher_id INT; → Query OK, 0 rows affected**
- **ALTER TABLE students ADD CONSTRAINT fk_teacher... → Query OK, 0 rows affected**

➢ Main SQL Commands and Sub-commands (DDL)

Task 1:

Create a table courses with columns:

- **course_id**
- **course_name**
- **course_credits**

Set course_id as the primary key.

Solution:

```
CREATE TABLE courses (
   course_id INT PRIMARY KEY,
   course_name VARCHAR(50),
   course_credits INT
);
```

Expected Output:

Query OK, 0 rows affected

Task 2:

Use the CREATE command to create a database university_db.

Solution:

```
CREATE DATABASE university_db;
```

Expected Output:

Query OK, 1 row affected

➢ ALTER Command

Task 1:

Modify the courses table by adding a column course_duration using the ALTER command.

Solution:

```
ALTER TABLE courses
ADD course_duration VARCHAR(20);
```

Expected Output:

Query OK, 0 rows affected

Task 2:

Drop the course_credits column from the courses table.

Solution:

```
ALTER TABLE courses
DROP COLUMN course_credits;
```

Expected Output:

Query OK, 0 rows affected

➢ . DROP Command

Task 1 :

Drop the teachers table from the school_db database.

Solution:

```
DROP TABLE teachers;
```

Expected Output:

Query OK, 0 rows affected

Task 2

Drop the students table from the school_db database and verify that the table has been removed.

Solution:

```
DROP TABLE students;
SHOW TABLES;
```

Expected Output:

- **DROP TABLE students; → Query OK, 0 rows affected**
- **SHOW TABLES; → The students table will not appear in the list**

➢ Data Manipulation Language (DML)

Task 1

Insert three records into the courses table using the INSERT command.

Solution:

```
INSERT INTO courses (course_id, course_name, course_duration)
VALUES
(101, 'Mathematics', '6 months'),
(102, 'Science', '4 months'),
(103, 'English', '3 months');
```

Expected Output:

Query OK, 3 rows affected

Task 2

Update the course duration of a specific course using the UPDATE command.

Solution:

```
UPDATE courses
SET course_duration = '5 months'
WHERE course_id = 102;
```

Expected Output:

Query OK, 1 row affected

Task 3

Delete a course with a specific course_id from the courses table using the DELETE command.

Solution:

```
DELETE FROM courses
WHERE course_id = 103;
```

Expected Output:

Query OK, 1 row affected

➢ Data Query Language (DQL)

Task 1

Retrieve all courses from the courses table using the SELECT statement.

Solution:

```
SELECT * FROM courses;
```

Expected Output:

course_id	course_name	course_duration
101	Mathematics	6 months
102	Science	5 months

Task 2

Sort the courses based on course_duration in descending order using ORDER BY.

Solution:

```
SELECT * FROM courses
ORDER BY course_duration DESC;
```

Expected Output:

course_id	course_name	course_duration
101	Mathematics	6 months
102	Science	5 months

Task 3

Limit the results of the SELECT query to show only the top two courses using LIMIT.

Solution:

```
SELECT * FROM courses
LIMIT 2;
```

Expected Output:

course_id	course_name	course_duration
101	Mathematics	6 months
102	Science	5 months

- **Data Control Language (DCL)**

Task 1

Create two new users user1 and user2 and grant user1 permission to SELECT from the courses table.

Solution:

```
-- Create users
CREATE USER 'user1'@'localhost' IDENTIFIED BY 'password1';
CREATE USER 'user2'@'localhost' IDENTIFIED BY 'password2';

-- Grant SELECT permission to user1
GRANT SELECT ON school_db.courses TO 'user1'@'localhost';
```

Expected Output:

- **CREATE USER → Query OK, 0 rows affected**
- **GRANT → Query OK, 0 rows affected**

Task 2

Revoke the INSERT permission from user1 and give it to user2.

Solution:

```
REVOKE INSERT ON school_db.courses FROM 'user1'@'localhost';
GRANT INSERT ON school_db.courses TO 'user2'@'localhost';
```

Expected Output:

- **REVOKE → Query OK, 0 rows affected**
- **GRANT → Query OK, 0 rows affected**

➢ . Transaction Control Language (TCL)

Task 1

Insert a few rows into the courses table and use COMMIT to save the changes.

Solution:

```
START TRANSACTION;

INSERT INTO courses (course_id, course_name, course_duration)
VALUES
(104, 'Computer Science', '6 months'),
(105, 'History', '4 months');

COMMIT;
```

Expected Output:

- **START TRANSACTION; → Query OK**
- **INSERT ... → 2 rows affected**
- **COMMIT; → Query OK, changes saved**
-

Task 2

Insert additional rows, then use ROLLBACK to undo the last insert operation.

Solution:

```
START TRANSACTION;

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (106, 'Geography', '3 months');

ROLLBACK;
```

Expected Output:

- **INSERT → 1 row affected (not saved)**
- **ROLLBACK; → Query OK, changes undone**

Task 3

Create a SAVEPOINT before updating the courses table, and use it to roll back specific changes.

Solution:

```
START TRANSACTION;

SAVEPOINT sp1;
UPDATE courses
SET course_duration = '8 months'
WHERE course_id = 101;

ROLLBACK TO sp1;
COMMIT;
```

Expected Output:

- **SAVEPOINT sp1; → Savepoint created**
- **UPDATE → 1 row affected (not saved)**
- **ROLLBACK TO sp1; → Rolled back to savepoint**
- **COMMIT; → Query OK**

➢ SQL Joins

Task 1

Create two tables: departments and employees. Perform an INNER JOIN to display employees along with their respective departments.

Solution:

```
-- Create departments table
CREATE TABLE departments (
   dept_id INT PRIMARY KEY,
   dept_name VARCHAR(50)
);

-- Create employees table
CREATE TABLE employees (
```

```
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);

-- Insert data
INSERT INTO departments VALUES (1, 'HR'), (2, 'IT'), (3, 'Finance');

INSERT INTO employees VALUES
(101, 'Amit Kumar', 1),
(102, 'Neha Sharma', 2),
(103, 'Ravi Patel', 3);

-- Perform INNER JOIN
SELECT employees.emp_name, departments.dept_name
FROM employees
INNER JOIN departments
ON employees.dept_id = departments.dept_id;
```

Expected Output:

emp_name	dept_name
Amit Kumar	HR
Neha Sharma	IT
Ravi Patel	Finance

Task 2

Use a LEFT JOIN to show all departments, even those without employees.

Solution:

```
SELECT departments.dept_name, employees.emp_name
FROM departments
LEFT JOIN employees
ON departments.dept_id = employees.dept_id;
```

Expected Output:

dept_name	emp_name
HR	Amit Kumar
IT	Neha Sharma
Finance	Ravi Patel

- **SQL Group By**

  Task 1

  Group employees by department and count the number of employees in each department using GROUP BY.

  Solution:

```
SELECT dept_id, COUNT(emp_id) AS employee_count
FROM employees
GROUP BY dept_id;
```

  Expected Output:

dept_id	employee_count
1	1
2	1
3	1

Task 2

Use the AVG aggregate function to find the average salary of employees in each department.

(We need a salary column, so we add it first.)

Solution:

```
-- Add salary column
ALTER TABLE employees ADD salary INT;

-- Update salaries
UPDATE employees SET salary = 40000 WHERE emp_id = 101;
UPDATE employees SET salary = 50000 WHERE emp_id = 102;
UPDATE employees SET salary = 45000 WHERE emp_id = 103;

-- Find average salary per department
SELECT dept_id, AVG(salary) AS avg_salary
FROM employees
GROUP BY dept_id;
```

Expected Output:

dept_id	avg_salary
1	40000
2	50000
3	45000

➢ SQL Stored Procedure

Task 1

Write a stored procedure to retrieve all employees from the employees **table based on department.**

Solution:

```
DELIMITER $$

CREATE PROCEDURE GetEmployeesByDept(IN dept INT)
BEGIN
   SELECT emp_id, emp_name, salary
   FROM employees
   WHERE dept_id = dept;
END$$

DELIMITER ;

-- Call the procedure
CALL GetEmployeesByDept(2);
```

Expected Output:

emp_id	emp_name	salary
102	Neha Sharma	50000

Task 2

Write a stored procedure that accepts course_id as input and returns the course details.

Solution:

```
DELIMITER $$

CREATE PROCEDURE GetCourseDetails(IN cid INT)
BEGIN
```

```
    SELECT * FROM courses
    WHERE course_id = cid;
END$$

DELIMITER ;

-- Call the procedure
CALL GetCourseDetails(101);
```

Expected Output:

course_id	course_name	course_duration
101	Mathematics	6 months

➢ . SQL View

Task 1

Create a view to show all employees along with their department names.

Solution:

```
CREATE VIEW employee_department AS
SELECT e.emp_name, d.dept_name, e.salary
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id;

-- To see the view
SELECT * FROM employee_department;
```

Expected Output:

emp_name	dept_name	salary
Amit Kumar	HR	40000
Neha Sharma	IT	50000
Ravi Patel	Finance	45000

Task 2

Modify the view to exclude employees whose salaries are below 50,000.

Solution:

```
CREATE OR REPLACE VIEW employee_department AS
SELECT e.emp_name, d.dept_name, e.salary
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
WHERE e.salary >= 50000;

-- Check the updated view
SELECT * FROM employee_department;
```

Expected Output:

emp_name	dept_name	salary
Neha Sharma	IT	50000

➢ SQL Triggers

Task 1

Create a trigger to automatically log changes to the employees table when a new employee is added.

Solution:

```
-- Create log table
CREATE TABLE employee_log (
   log_id INT AUTO_INCREMENT PRIMARY KEY,
   emp_id INT,
   action VARCHAR(20),
   log_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Create trigger
DELIMITER $$

CREATE TRIGGER after_employee_insert
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
   INSERT INTO employee_log(emp_id, action)
   VALUES (NEW.emp_id, 'INSERT');
END$$
DELIMITER ;
```

Expected Output:

Query OK, 0 rows affected

(Whenever you insert into employees, a log will be added to employee_log.)

Task 2

Create a trigger to update the last_modified timestamp whenever an employee record is updated.

Solution:

```
-- Add last_modified column
ALTER TABLE employees ADD last_modified TIMESTAMP;

-- Create trigger
DELIMITER $$

CREATE TRIGGER before_employee_update
BEFORE UPDATE ON employees
FOR EACH ROW
BEGIN
   SET NEW.last_modified = CURRENT_TIMESTAMP;
END$$

DELIMITER ;
```

Expected Output:

- **ALTER TABLE → Query OK**
- **CREATE TRIGGER → Query OK**

➢ . Introduction to PL/SQL

Task1

Write a PL/SQL block to print the total number of employees from the employees table.

Solution:

```
DECLARE
   total_employees NUMBER;
BEGIN
   SELECT COUNT(*) INTO total_employees FROM employees;
   DBMS_OUTPUT.PUT_LINE('Total Employees: ' || total_employees);
END;
/
```

Expected Output:

Total Employees: 3 (or whatever the actual count is)

Task 2

Create a PL/SQL block that calculates the total sales from an orders table.

Solution:

```
DECLARE
   total_sales NUMBER;
BEGIN
   SELECT SUM(order_amount) INTO total_sales FROM orders;
   DBMS_OUTPUT.PUT_LINE('Total Sales: ' || total_sales);
END;
/
```

Expected Output:

Total Sales: <sum_of_all_orders>

➢ PL/SQL Control Structures

Task 1

Write a PL/SQL block using an IF-THEN condition to check the department of an employee.

Solution:

```
DECLARE
   emp_department VARCHAR2(50);
BEGIN
   SELECT dept_id INTO emp_department
   FROM employees
   WHERE emp_id = 101;

   IF emp_department = '1' THEN
      DBMS_OUTPUT.PUT_LINE('Employee is in HR Department.');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Employee is in another Department.');
   END IF;
END;
/
```

Expected Output:

Employee is in HR Department.

Task 2

Use a FOR LOOP to iterate through employee records and display their names.

Solution:

```
DECLARE
   CURSOR emp_cursor IS
      SELECT emp_name FROM employees;
   emp_name_var employees.emp_name%TYPE;
BEGIN
   FOR emp_record IN emp_cursor LOOP
      DBMS_OUTPUT.PUT_LINE('Employee: ' || emp_record.emp_name);
   END LOOP;
END;
/
```

Expected Output:

Employee: Amit Kumar
Employee: Neha Sharma
Employee: Ravi Patel

➢ SQL Cursors

Task 1

Write a PL/SQL block using an explicit cursor to retrieve and display employee details.

Solution:

```
DECLARE
   CURSOR emp_cursor IS
      SELECT emp_id, emp_name, salary FROM employees;
   v_emp_id employees.emp_id%TYPE;
   v_emp_name employees.emp_name%TYPE;
   v_salary employees.salary%TYPE;
BEGIN
   OPEN emp_cursor;
   LOOP
      FETCH emp_cursor INTO v_emp_id, v_emp_name, v_salary;
      EXIT WHEN emp_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('ID: ' || v_emp_id || ', Name: ' ||
v_emp_name || ', Salary: ' || v_salary);
   END LOOP;
   CLOSE emp_cursor;
END;
/
```

Expected Output:

ID: 101, Name: Amit Kumar, Salary: 40000

ID: 102, Name: Neha Sharma, Salary: 50000

ID: 103, Name: Ravi Patel, Salary: 45000

Task 2

Create a cursor to retrieve all courses and display them one by one.

Solution:

```
DECLARE
   CURSOR course_cursor IS
      SELECT course_id, course_name FROM courses;
   v_id courses.course_id%TYPE;
   v_name courses.course_name%TYPE;
BEGIN
   OPEN course_cursor;
   LOOP
      FETCH course_cursor INTO v_id, v_name;
      EXIT WHEN course_cursor%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Course ID: ' || v_id || ', Name: ' ||
v_name);
   END LOOP;
   CLOSE course_cursor;
END;
/
```

Expected Output:

Course ID: 101, Name: Mathematics

Course ID: 102, Name: Science

Course ID: 104, Name: Computer Science

Course ID: 105, Name: History

➢ Rollback and Commit Savepoint

Task 1

Perform a transaction where you create a savepoint, insert records, then rollback to the savepoint.

Solution:

```
START TRANSACTION;

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (107, 'Biology', '5 months');

SAVEPOINT sp1;

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (108, 'Chemistry', '6 months');

ROLLBACK TO sp1;

COMMIT;
```

Expected Output:

- **The row for course_id 107 (Biology) is saved.**
- **The row for course_id 108 (Chemistry) is rolled back (not saved).**

Task 2

Commit part of a transaction after using a savepoint and then rollback the remaining changes.

Solution:

```
START TRANSACTION;

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (109, 'Physics', '6 months');

SAVEPOINT sp2;

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (110, 'Economics', '4 months');

COMMIT;  -- Saves all changes made before this line

ROLLBACK TO sp2; -- This won't affect committed changes
```

Expected Output:

- **Course 109 (Physics) is saved permanently.**
- **Course 110 (Economics) rollback does nothing because COMMIT already saved it.**