Module 4 – Introduction to DBMS

1.Introduction to SQL

LAB EXERCISES:

1: Create a new database named school_db and a table called students with the following columns: student_id, student_name, age, class, and address.

Ans:

step 1: Create the 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 AUTO_INCREMENT,
   student_name VARCHAR(100) NOT NULL,
   age INT,
   class VARCHAR(50),
   address VARCHAR(255)
);
```

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

Ans:

Step 1:INSERT INTO students (student_name, age, class, address)

```
VALUES
('John Smith', 15, '10A', '123 Maple Street'),
('Emily Johnson', 14, '9B', '456 Oak Avenue'),
('Michael Brown', 16, '11C', '789 Pine Road'),
('Sarah Davis', 15, '10A', '321 Birch Lane'),
('David Wilson', 13, '8D', '654 Cedar Drive');
```

Step 2: Retrieve All Records

SELECT * FROM students;

2. SQL Syntax.

LAB EXERCISES:

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

Ans:

SELECT student_name, age FROM students;

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

Ans:

SELECT * FROM students WHERE age > 10;

3. SQL Constraints.

LAB EXERCISES:

1: Create a table teachers with the following columns: teacher_id (Primary Key),

Ans:

teacher name (NOT NULL), subject (NOT NULL), and email (UNIQUE).

```
CREATE TABLE teachers ( teacher_id INT PRIMARY KEY AUTO_INCREMENT,
  teacher_name VARCHAR(100) NOT NULL, subject VARCHAR(100) NOT NULL,
  email VARCHAR(150) UNIQUE );
```

Explanation of Constraints:

PRIMARY KEY: teacher_id uniquely identifies each teacher.

NOT NULL: teacher_name and subject cannot have NULL values.

UNIQUE: email must be unique for each teacher.

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

Ans:

If students already exists (recommended ALTER)

```
ALTER TABLE students ADD COLUMN teacher_id INT NULL,

ADD CONSTRAINT fk_students_teacher FOREIGN KEY (teacher_id)

REFERENCES teachers(teacher_id) ON UPDATE CASCADE

ON DELETE SET NULL;
```

What this does:

- Adds teacher_id (nullable so existing rows remain valid).
- Enforces referential integrity: any value in students.teacher_id must match an existing teacher.
- If a teacher's teacher_id changes, the change cascades to students.
- If a teacher row is deleted, affected students' teacher_id is set to NULL (keeps the student record but removes the link).

If you want *every* student to have a teacher (non-nullable)

Use this after you've backfilled valid teacher IDs for all students:

```
ALTER TABLE students MODIFY teacher_id INT NOT NULL,

DROP FOREIGN KEY fk_students_teacher, ADD CONSTRAINT fk_students_teacher

 FOREIGN KEY (teacher_id) REFERENCES teachers(teacher_id)

 ON UPDATE CASCADE  ON DELETE RESTRICT;
```

Here, deleting a teacher is blocked if students reference them.

Example: Assign existing students to teachers

(Assuming teacher rows with IDs 1–3 exist.)

```
UPDATE students SET teacher_id = 1 WHERE class = '10A';

UPDATE students SET teacher_id = 2 WHERE class = '9B';

UPDATE students SET teacher_id = 3 WHERE class = '11C';
```

Would you like me to:

1. Provide INSERT statements for sample teachers first, then run the ALTER?

2. Backfill all current students with teacher assignments automatically?
3. Show a JOIN query to view students alongside their teachers?

4. Main SQL Commands and Sub-commands (DDL).

LAB EXERCISES:

1: Create a table courses with columns: course_id, course_name, and course_credits. Set the course_id as the primary key.

Ans:

```
CREATE TABLE courses (
 course_id INT PRIMARY KEY AUTO_INCREMENT,  course_name VARCHAR(100) NOT NULL,
  course_credits INT NOT NULL );
```

Explanation:

course_id: Primary key (unique for each course).

course_name: Cannot be NULL.

course_credits: Indicates the credits for the course (cannot be NULL).

2: Use the CREATE command to create a database university_db.

Ans:

```
CREATE DATABASE university_db;
```

Optional Step: Use the database

After creating it, switch to this database to create tables (like courses, students, teachers):

```
USE university_db;
```

5. ALTER Command.

LAB EXERCISES:

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

Ans:

```
ALTER TABLE courses ADD COLUMN course_duration VARCHAR(50);
```

Explanation:

ALTER TABLE: Used to modify an existing table structure.

ADD COLUMN course_duration VARCHAR(50): Adds a new column course_duration of type VARCHAR(50) to store duration information (e.g., "6 months", "1 year").

2: Drop the course_credits column from the courses table.

Ans:

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

What this does:

The course_credits column and all its data will be permanently removed from the table.

6. DROP Command.

LAB EXERCISES:

1: Drop the teachers table from the school_db database.

Ans:

Step1: Select the database (if not already in school_db)

```
USE school_db;
```

Step 2: Drop the teachers table

```
DROP TABLE teachers;
```

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

Ans:

Step 1:Select the Database

USE school_db;

Step 2: Drop the students Table

DROP TABLE students;

Step 3: Verify Table Removal

You can check if the table has been removed by listing all tables in the database:

SHOW TABLES;

If students no longer appears in the list, it has been successfully dropped.

Would you like me to give you a **single SQL script** that will:

1. Drop students and teachers (if they exist).
2. Show remaining tables in school_db.

7. Data Manipulation Language (DML).

LAB EXERCISES:

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

Ans:

```
INSERT INTO courses (course_name, course_duration) VALUES

('Computer Science', '6 months'),

('Mathematics', '4 months'),

 ('Physics', '5 months');
```

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

Ans:

```
UPDATE courses

SET course_duration = '6 months'

WHERE course_name = 'Mathematics';
```

To verify the update:

SELECT * FROM courses;

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

Ans:

DELETE FROM courses

WHERE course_id = 2;

To verify deletion:

SELECT * FROM courses;

8. Data Query Language (DQL).

LAB EXERCISES:

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

Ans:

SELECT * FROM courses;

What this does:

* selects all columns (course_id, course_name, course_duration, etc.).

Displays all records currently in the courses table.

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

Ans:

SELECT * FROM courses ORDER BY course_duration DESC;

Explanation:

SELECT * retrieves all columns from the courses table.

ORDER BY course_duration DESC arranges the rows in descending order (from highest to lowest) based on the course_duration column.

You can use ASC (ascending order) instead of DESC if needed.

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

Ans:

SELECT *

FROM courses

ORDER BY course duration DESC LIMIT 2

9. Data Control Language (DCL).

LAB EXERCISES:

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

Ans:

1. Create two new users

CREATE USER 'user1'@'localhost' IDENTIFIED BY 'password1';

CREATE USER 'user2'@'localhost' IDENTIFIED BY 'password2';

2. Grant SELECT permission to user1 on the courses table

GRANT SELECT ON database_name.courses TO 'user1'@'localhost';

3. Apply the changes

FLUSH PRIVILEGES;

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

Ans:

1. Revoke INSERT permission from user1

REVOKE INSERT ON database_name.courses FROM 'user1'@'localhost';

2. Grant INSERT permission to user2

GRANT INSERT ON database_name.courses TO 'user2'@'localhost';

3. Apply the changes

FLUSH PRIVILEGES;

Explanation:

REVOKE INSERT removes the ability of user1 to insert new records into the courses table.

GRANT INSERT gives user2 the permission to insert data.

FLUSH PRIVILEGES ensures the updated permissions take effect.

10. Transaction Control Language (TCL).

LAB EXERCISES:

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

Ans:

1. Start a transaction

```
START TRANSACTION;
```

2. Insert rows into the courses table

```
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (101, 'SQL Basics', 30),
       (102, 'Advanced SQL', 45),
       (103, 'Database Design', 60);
```

3. Save the changes

```
COMMIT;
```

Explanation:

START TRANSACTION begins a new transaction, ensuring that the following SQL operations are treated as a single unit.

INSERT INTO adds new rows into the courses table with specified values.

COMMIT permanently saves the changes made during the transaction to the database.

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

Ans:

1. Start a transaction

```
START TRANSACTION;
```

2. Insert additional rows into the courses table

```
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (104, 'Data Warehousing', 50), (105, 'Big Data Analytics', 70);
```

3. Undo the insert operation

ROLLBACK;

Explanation:

START TRANSACTION begins a transaction.

The INSERT statement adds new rows temporarily (not yet saved).

ROLLBACK undoes all operations performed after the transaction began, meaning the newly inserted rows (104 and 105) will not be saved in the courses table.

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

Ans:

1. Start a transaction
START TRANSACTION;
2. Create a SAVEPOINT
SAVEPOINT sp1;
3. Update the courses table
UPDATE courses
SET course_duration = 80
WHERE course_id = 101;
UPDATE courses
SET course_duration = 90
WHERE course_id = 102;
4. Roll back to the SAVEPOINT (undo last update)
ROLLBACK TO sp1;
5. Commit the remaining changes (if any)
COMMIT;
Explanation:
>START TRANSACTION starts a new transaction.
>SAVEPOINT sp1 creates a marker within the transaction to which you can roll back later.
>Two UPDATE statements modify rows in the courses table.
>ROLLBACK TO sp1 undoes all changes made after the SAVEPOINT sp1 but keeps changes before it.
>COMMIT permanently saves the changes that remain.

11. SQL Joins.

LAB EXERCISES:

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

Ans:

```
Create departments table
CREATE TABLE departments (
   department_id INT PRIMARY KEY,
   department_name VARCHAR(50) NOT NULL
);

-- Create employees table
CREATE TABLE employees (
   employee_id INT PRIMARY KEY,
   employee_name VARCHAR(50) NOT NULL,
   department_id INT,
   FOREIGN KEY (department_id) REFERENCES departments(department_id)
);

-- Insert sample data into departments
INSERT INTO departments (department_id, department_name) VALUES
(1, 'Human Resources'),
(2, 'Finance'),
(3, 'IT');

-- Insert sample data into employees
INSERT INTO employees (employee_id, employee_name, department_id) VALUES
(101, 'Alice', 1),
(102, 'Bob', 2),
(103, 'Charlie', 3),
(104, 'David', 2);
-- Perform INNER JOIN
SELECT e.employee_id,
       e.employee_name,
       d.department_name
FROM employees e
INNER JOIN departments d
ON e.department_id = d.department_id;
Result of INNER JOIN:
employee_id  employee_name  department_name
101           Alice           Human Resources
102           Bob              Finance
103           Charlie          IT
104           David            Finance
```

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

Ans:

```
SELECT d.department_id,
       d.department_name,
       e.employee_id,
       e.employee_name
FROM departments d
LEFT JOIN employees e
ON d.department_id = e.department_id;
```

Explanation:

The LEFT JOIN ensures that all rows from the departments table are displayed.

If a department has no employees, the employee columns (employee_id and employee_name) will show NULL.

Example Output:

markdown

Copy

Edit

```
department_id  department_name   employee_id  employee_name

1              Human Resources    101          Alice
2               Finance           102          Bob
2               Finance           104          David
3                  IT             103          Charlie
4              Marketing          NULL         NULL
```

12. SQL Group By.

LAB EXERCISES:

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

Ans:

```
SELECT
   departments.dept_name,
   COUNT(employees.emp_id) AS employee_count FROM
   departments
LEFT JOIN
employees ON
   departments.dept_id = employees.dept_id GROUP BY
   departments.dept_name;
```

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

Ans:

Step 1: Add a salary column (if not already present)

```
ALTER TABLE employees

ADD salary DECIMAL(10, 2);
```

Step 2: Update some salaries for demonstration

```
UPDATE employees SET salary = 50000 WHERE emp_id = 101;

UPDATE employees SET salary = 60000 WHERE emp_id = 102;

UPDATE employees SET salary = 75000 WHERE emp_id = 103;

UPDATE employees SET salary = 80000 WHERE emp_id = 104;
```

Step 3: Use AVG() with GROUP BY to get average salary by department SELECT

```
  departments.dept_name,

  AVG(employees.salary) AS average_salary

FROM

  departments LEFT JOIN

  employees ON departments.dept_id = employees.dept_id GROUP BY

  departments.dept_name;
```

13. SQL Stored Procedure.

LAB EXERCISES:

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

Ans:

```
DELIMITER //

CREATE PROCEDURE get_employees_by_department(IN deptName VARCHAR(100))

BEGIN

  SELECT

    employees.emp_id,

employees.emp_name,
employees.salary,
departments.dept_name     FROM
employees     INNER JOIN
        departments ON employees.dept_id = departments.dept_id
    WHERE
        departments.dept_name = deptName;
END //
 DELIMITER ;
```

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

Ans:

```
courses (
    course_id INT PRIMARY KEY,     course_name VARCHAR(100),
course_credits INT
)
-----------stored procedure:-
DELIMITER $$
CREATE PROCEDURE GetCourseDetails(IN input_course_id INT)
BEGIN
    SELECT *
    FROM courses
    WHERE course_id = input_course_id;
END $$
DELIMITER ;
```

14. SQL View.

LAB EXERCISES:

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

Ans:

```
creating employee table:- employees (
    employee_id INT PRIMARY KEY,     employee_name VARCHAR(100),
department_id INT
```

)

creating department table:- departments (

department_id INT PRIMARY KEY,   department_name VARCHAR(100)

)

creating view:-

```
CREATE VIEW employee_department_view AS
SELECT
    e.employee_id,
    e.employee_name,
    d.department_name FROM
    employees e INNER JOIN
    departments d ON e.department_id = d.department_id;
```

How to use view:-

```
SELECT * FROM employee_department_view;
```

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

Ans:

```
CREATE OR REPLACE VIEW employee_department_view AS
SELECT
    e.employee_id,
    e.employee_name,
    e.salary,
    d.department_name FROM
    employees e JOIN
    departments d ON e.department_id = d.department_id
WHERE
    e.salary >= 50000;
```

15. SQL Triggers.

LAB EXERCISES:

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

Ans:

Step 1: Create a log table (if it doesn't already exist) CREATE TABLE employee_log (

```
  log_id INT PRIMARY KEY AUTO_INCREMENT,     employee_id INT,

    employee_name VARCHAR(100), action VARCHAR(50),

  log_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP

);
```

Step 2: Create the trigger

For MySQL / PostgreSQL:

```
CREATE TRIGGER log_new_employee

AFTER INSERT ON employees

FOR EACH ROW

BEGIN

  INSERT INTO employee_log (employee_id, employee_name, action)

  VALUES (NEW.employee_id, NEW.employee_name, 'INSERT');

END;
```

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

Ans:

Step 1: Ensure the employees table has a last_modified column

```
ALTER TABLE employees

ADD COLUMN last_modified TIMESTAMP DEFAULT
CURRENT_TIMESTAMP;
```

Step 2: Create the trigger   For MySQL:

```
CREATE TRIGGER update_last_modified

BEFORE UPDATE ON employees

FOR EACH ROW
```

```
BEGIN
   SET NEW.last_modified = CURRENT_TIMESTAMP;
END;
```

16. Introduction to PL/SQL.

LAB EXERCISES:

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

Ans:

```
DECLARE
   total_employees NUMBER;
BEGIN
   SELECT COUNT(*) INTO total_employees
   FROM employees;

   DBMS_OUTPUT.PUT_LINE('Total number of employees: ' || total_employees);
END;
/
```

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

Ans:

```
DECLARE
   total_sales NUMBER(10,2);
BEGIN
     SELECT SUM(order_amount) INTO total_sales FROM orders;

   DBMS_OUTPUT.PUT_LINE('Total Sales: $' || total_sales);
END;
/
```

17. PL/SQL Control Structures.

LAB EXERCISES:

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

Ans:

```
DECLARE
   emp_id       NUMBER := 101;  -- change this to the employee ID you want to check
   emp_dept     VARCHAR2(50);
BEGIN
   SELECT department_name INTO emp_dept
   FROM employees e
   JOIN departments d ON e.department_id = d.department_id
   WHERE e.employee_id = emp_id;

   IF emp_dept = 'Sales' THEN
      DBMS_OUTPUT.PUT_LINE('The employee works in the Sales department.');
   ELSIF emp_dept = 'HR' THEN
      DBMS_OUTPUT.PUT_LINE('The employee works in the HR department.');
    ELSE
      DBMS_OUTPUT.PUT_LINE('The employee works in another department: ' || emp_dept);
   END IF;
END;
/
```

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

Ans:

```
DECLARE
   CURSOR emp_cursor IS
      SELECT employee_name FROM employees;
BEGIN
   FOR emp_rec IN emp_cursor LOOP
      DBMS_OUTPUT.PUT_LINE('Employee Name: ' || emp_rec.employee_name);
   END LOOP;
END;
/
```

18. SQL Cursors.

LAB EXERCISES:

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

Ans:

```
DECLARE
   -- Declare variables to hold employee details     v_employee_id
employees.employee_id%TYPE; v_employee_name employees.employee_name%TYPE;
v_salary       employees.salary%TYPE;

   -- Declare the explicit cursor
   CURSOR emp_cursor IS
      SELECT employee_id, employee_name, salary
      FROM employees;
BEGIN
   -- Open the cursor
   OPEN emp_cursor;

   LOOP
      -- Fetch each record into variables
      FETCH emp_cursor INTO v_employee_id, v_employee_name, v_salary;

      -- Exit when no more rows
      EXIT WHEN emp_cursor%NOTFOUND;

      -- Display employee details
      DBMS_OUTPUT.PUT_LINE('ID: ' || v_employee_id || ', Name: ' || v_employee_name || ',
Salary: ' || v_salary);
   END LOOP;
```

```
     -- Close the cursor
   CLOSE emp_cursor;
END;
/
```

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

Ans:

```
DECLARE
   -- Variables to hold course details     v_course_id
courses.course_id%TYPE;     v_course_name
courses.course_name%TYPE;     v_course_credit
courses.course_credits%TYPE;

   -- Declare the cursor
   CURSOR course_cursor IS
      SELECT course_id, course_name, course_credits
      FROM courses;
BEGIN
   -- Open the cursor
   OPEN course_cursor;

   LOOP
      -- Fetch each course into variables
      FETCH course_cursor INTO v_course_id, v_course_name, v_course_credit;

      -- Exit loop when no more rows
      EXIT WHEN course_cursor%NOTFOUND;

      -- Display course details
      DBMS_OUTPUT.PUT_LINE('Course ID: ' || v_course_id ||
                        ', Name: ' || v_course_name ||
                        ', Credits: ' || v_course_credit);
   END LOOP;

   -- Close the cursor
   CLOSE course_cursor;
END;
/
```

19. Rollback and Commit Savepoint.

LAB EXERCISES:

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

Ans:

```
BEGIN
   -- Start of transaction
   -- First insert
   INSERT INTO employees (employee_id, employee_name, salary, department_id)
     VALUES (201, 'Alice Johnson', 60000, 10);
   -- Create a savepoint after first insert
   SAVEPOINT after_first_insert;

   -- Second insert
   INSERT INTO employees (employee_id, employee_name, salary, department_id)
   VALUES (202, 'Bob Smith', 55000, 20);

   -- Rollback to savepoint (undo Bob's insert, keep Alice's)
   ROLLBACK TO after_first_insert;
   -- Commit the transaction to finalize Alice's insert
   COMMIT;
   DBMS_OUTPUT.PUT_LINE('Transaction complete: Inserted Alice, rolled back Bob.');
```

```
END;
/
```

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

Ans:

```
BEGIN
  -- Insert 1st employee (part to commit)
   INSERT INTO employees (employee_id, employee_name, salary, department_id)
  VALUES (301, 'Ravi Kumar', 60000, 1);

  -- Insert 2nd employee (part to commit)
  INSERT INTO employees (employee_id, employee_name, salary, department_id)
  VALUES (302, 'Anjali Shah', 58000, 2);

  -- Create a savepoint after first two inserts
  SAVEPOINT after_first_two;

  -- Commit the changes up to the savepoint
  COMMIT;

  -- Insert 3rd employee (this will be rolled back)
  INSERT INTO employees (employee_id, employee_name, salary, department_id)
  VALUES (303, 'Deepak Mehta', 62000, 3);

  -- Insert 4th employee (this will also be rolled back)
  INSERT INTO employees (employee_id, employee_name, salary, department_id)
  VALUES (304, 'Pooja Verma', 61000, 4);

   -- Now rollback the remaining uncommitted changes
  ROLLBACK;
```

```
    DBMS_OUTPUT.PUT_LINE('First two inserts committed, remaining rolled back.');
END;
/
```