Module 4 – Introduction to DBMS(practical)

Lab 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.

```
-- Create a new database
CREATE DATABASE school_db;

-- Select the database (needed in MySQL, not in Oracle)
USE school_db;

-- Create the students table
CREATE TABLE students (
   student_id INT PRIMARY KEY,
   student_name VARCHAR(50),
   age INT,
   class VARCHAR(10),
   address VARCHAR(100)
);
```

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

```
-- Insert 5 records into students table
INSERT INTO students (student_id, student_name, age, class, address)
VALUES (1, 'Amit Sharma', 15, '10A', 'Delhi');

INSERT INTO students (student_id, student_name, age, class, address)
VALUES (2, 'Priya Verma', 14, '9B', 'Mumbai');

INSERT INTO students (student_id, student_name, age, class, address)
VALUES (3, 'Rahul Singh', 16, '11C', 'Kolkata');

INSERT INTO students (student_id, student_name, age, class, address)
VALUES (4, 'Neha Patel', 15, '10B', 'Ahmedabad');

INSERT INTO students (student_id, student_name, age, class, address)
VALUES (5, 'Karan Mehta', 17, '12A', 'Pune');

-- Retrieve all records
SELECT * FROM students;
```

Lab 3: Write SQL queries to retrieve specific columns (student_name and age) from the students table.

```
-- Retrieve only student_name and age from students table
SELECT student_name, age
FROM students;
```

Lab 4: Write SQL queries to retrieve all students whose age is greater than 10.

```
-- Retrieve all columns for students whose age is greater than 10
SELECT *
FROM students
WHERE age > 10;
```

Lab 5: Create a table teachers with the following columns: teacher_id (Primary Key), teacher_name (NOT NULL), subject (NOT NULL), and email (UNIQUE).

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

Lab 6: Implement a FOREIGN KEY constraint to relate the teacher_id from the teachers table with the students table.

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

Lab 7: Create a table courses with columns: course_id, course_name, and course_credits. Set the course_id as the primary key.

```
CREATE TABLE courses (
   course_id INT PRIMARY KEY,
   course_name VARCHAR(100),
   course_credits INT
);
```

Lab 8: Use the CREATE command to create a database university_db.

```
CREATE DATABASE university_db;
```

Lab 9: Modify the courses table by adding a column course_duration using the ALTER command.

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

Lab 10: Drop the course_credits column from the courses table.

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

Lab 11: Drop the teachers table from the school_db database. \

```
USE school_db;

DROP TABLE teachers;
```

Lab 12: Drop the students table from the school_db database and verify that the table has been removed.

```
USE school_db;

DROP TABLE students;

-- Verification: check if table exists
SHOW TABLES;
```

Lab 13: Insert three records into the courses table using the INSERT command.

```
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (101, 'Computer Science', '3 Years');

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (102, 'Mathematics', '2 Years');

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (103, 'English Literature', '1 Year');
```

Lab 14: Update the course duration of a specific course using the UPDATE command.

```
UPDATE courses
```

```
SET course_duration = '4 Years'
WHERE course_id = 101;
```

Lab 15: Delete a course with a specific course_id from the courses table using the DELETE command.

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

Lab 16: Retrieve all courses from the courses table using the SELECT statement.

```
SELECT * FROM courses;
```

Lab 17: Sort the courses based on course_duration in descending order using ORDER BY.

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

Lab 18: Limit the results of the SELECT query to show only the top two courses using LIMIT.

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

Lab 19: Create two new users user1 and user2 and grant user1 permission to SELECT from the courses table.

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

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

Lab 20: Revoke the INSERT permission from user1 and give it to user2.

```
-- Revoke INSERT permission from user1
REVOKE INSERT ON school_db.courses FROM 'user1'@'localhost';

-- Grant INSERT permission to user2
GRANT INSERT ON school_db.courses TO 'user2'@'localhost';
```

Lab 21: Insert a few rows into the courses table and use COMMIT to save the changes.

```
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (201, 'Physics', '3 Years');

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (202, 'Chemistry', '2 Years');

-- Save the changes permanently
COMMIT;
```

Lab 22: Insert additional rows, then use ROLLBACK to undo the last insert operation.

```
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (203, 'Biology', '2 Years');

-- Undo the last insert operation
ROLLBACK;
```

Lab 23: Create a SAVEPOINT before updating the courses table, and use it to roll back specific changes.

```
-- Create a savepoint
SAVEPOINT sp_before_update;

-- Update a course
UPDATE courses
SET course_duration = '5 Years'
WHERE course_id = 201;

-- Rollback to savepoint (undo the update, but keep earlier inserts)
ROLLBACK TO sp_before_update;

-- Finally, release the savepoint (optional)
RELEASE SAVEPOINT sp_before_update;
```

Lab 24: Create two tables: departments and employees. Perform an INNER JOIN to display employees along with their respective departments.

```
-- Create departments table
CREATE TABLE departments (
   dept_id INT PRIMARY KEY,
```

```
    dept_name VARCHAR(100)
);

-- Create employees table
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);

-- Insert sample data
INSERT INTO departments VALUES (1, 'HR'), (2, 'IT'), (3, 'Finance');
INSERT INTO employees VALUES (101, 'Alice', 1), (102, 'Bob', 2), (103, 'Charlie', 2);

-- INNER JOIN: show employees with departments
SELECT e.emp_id, e.emp_name, d.dept_name
FROM employees e
INNER JOIN departments d ON e.dept_id = d.dept_id;
```

Lab 25: Use a LEFT JOIN to show all departments, even those without employees.

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

Lab 26: Group employees by department and count the number of employees in each department using GROUP BY.

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

Lab 27: Use the AVG aggregate function to find the average salary of employees in each department.

First, we add a salary column to the employees table:

```
ALTER TABLE employees ADD salary DECIMAL(10,2);
```

Now query:

```
SELECT dept_id, AVG(salary) AS average_salary
FROM employees
```

```
GROUP BY dept_id;
```

Lab 28: Write a stored procedure to retrieve all employees from the employees table based on department.

```
DELIMITER //

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

DELIMITER ;

------- call like this ------
CALL GetEmployeesByDepartment(2);
```

Lab 29: Write a stored procedure that accepts course_id as input and returns the course details.

```
DELIMITER //

CREATE PROCEDURE GetCourseDetails(IN cid INT)
BEGIN
   SELECT * FROM courses
   WHERE course_id = cid;
END //
DELIMITER ;
```

Call it like this:

```
CALL GetCourseDetails(201);
```

Lab 30: Create a view to show all employees along with their department names.

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

To use the view:

```
SELECT * FROM employee_department_view;
```

Lab 31: Modify the view to exclude employees whose salaries are below $50,000.

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

Lab 32: Create a trigger to automatically log changes to the employees table when a new employee is added.

First, create a log table:

```
CREATE TABLE employee_log (
   log_id INT AUTO_INCREMENT PRIMARY KEY,
   emp_id INT,
   action VARCHAR(50),
   action_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Then the 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, 'Employee Added');
END //

DELIMITER ;
```

Lab 33: Create a trigger to update the last_modified timestamp whenever an employee record is updated.

Add column in employees table:

```
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 ;
```

Lab 34: Write a PL/SQL block to print the total number of employees from the employees table.

```
SET SERVEROUTPUT ON;

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

Lab 35: Create a PL/SQL block that calculates the total sales from an orders table.

```
SET SERVEROUTPUT ON;

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

Lab 36: Write a PL/SQL block using an IF-THEN condition to check the department of an employee.

```
SET SERVEROUTPUT ON;

DECLARE
   v_emp_id employees.emp_id%TYPE := 101;
   v_dept_id employees.dept_id%TYPE;
BEGIN
   SELECT dept_id INTO v_dept_id
   FROM employees
   WHERE emp_id = v_emp_id;
```

```
  IF v_dept_id = 1 THEN
    DBMS_OUTPUT.PUT_LINE('Employee works in HR');
  ELSIF v_dept_id = 2 THEN
    DBMS_OUTPUT.PUT_LINE('Employee works in IT');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Employee works in another department');
  END IF;
END;
/
```

Lab 37: Use a FOR LOOP to iterate through employee records and display their names.

```
SET SERVEROUTPUT ON;

DECLARE
  v_name employees.emp_name%TYPE;
BEGIN
  FOR rec IN (SELECT emp_name FROM employees) LOOP
    DBMS_OUTPUT.PUT_LINE('Employee: ' || rec.emp_name);
  END LOOP;
END;
/
```

Lab 38: Write a PL/SQL block using an explicit cursor to retrieve and display employee details.

```
SET SERVEROUTPUT ON;

DECLARE
  CURSOR emp_cursor IS
    SELECT emp_id, emp_name, dept_id FROM employees;
  v_emp employees.emp_id%TYPE;
  v_name employees.emp_name%TYPE;
  v_dept employees.dept_id%TYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO v_emp, v_name, v_dept;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('ID: ' || v_emp || ' | Name: ' || v_name || ' | Dept: ' || v_dept);
  END LOOP;
  CLOSE emp_cursor;
END;
/
```

Lab 39: Create a cursor to retrieve all courses and display them one by one.

```
SET SERVEROUTPUT ON;

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

Lab 40: Perform a transaction where you create a savepoint, insert records, then rollback to the savepoint.

```
-- Start transaction
START TRANSACTION;

-- Insert first record
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (301, 'Economics', '3 Years');

-- Create savepoint
SAVEPOINT sp1;

-- Insert another record
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (302, 'Political Science', '2 Years');

-- Rollback to savepoint (undo second insert only)
ROLLBACK TO sp1;

-- Commit the first insert
COMMIT;
```

Lab 41: Commit part of a transaction after using a savepoint and then rollback the remaining changes.

```
-- Start transaction
START TRANSACTION;

-- Insert first record
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (401, 'History', '3 Years');

-- Savepoint
SAVEPOINT sp2;

-- Insert second record
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (402, 'Philosophy', '2 Years');

-- Commit up to savepoint (keeps History)
RELEASE SAVEPOINT sp2;
COMMIT;

-- Rollback remaining changes (Philosophy won't be saved)
ROLLBACK;
```