Introduction To DBMS

1. **Introduction to SQL**

A. 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 DATABASE school_db;

-> USE school_db;

-> CREATE TABLE students (
   student_id INT PRIMARY KEY,
   student_name VARCHAR(100),
   age INT,
   class VARCHAR(20),
   address VARCHAR(255)
);
```

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

```
-> INSERT INTO students (student_id, student_name, age, class, address)
VALUES
(1, 'Amit Kumar', 14, '8th', 'Delhi'),
(2, 'Sneha Sharma', 13, '7th', 'Mumbai'),
(3, 'Rahul Singh', 15, '9th', 'Kolkata'),
(4, 'Pooja Mehta', 14, '8th', 'Chennai'),
(5, 'Karan Verma', 13, '7th', 'Bangalore');

-> SELECT * FROM students;
```

2. **SQL Syntax**

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

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

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

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

3. SQL Constraints

A. 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(50) NOT NULL,
   email VARCHAR(100) UNIQUE
);
```

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

```
-> CREATE TABLE students (
   student_id INT PRIMARY KEY,
   student_name VARCHAR(100),
   age INT,
   class VARCHAR(20),
   address VARCHAR(255),
   teacher_id INT,
   ADD CONSTRAINT fk_teacher FOREIGN KEY
   (teacher_id)REFERENCES teachers(teacher_id)
);
```

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

A. 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
);
```

B. Use the CREATE command to create a database university_db.

```
-> CREATE DATABASE university_db;
```

5. ALTER Command

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

-> ALTER TABLE courses ADD course_duration VARCHAR(50);

B. Drop the course_credits column from the courses table.

-> ALTER TABLE courses DROP COLUMN course_credits;

6. DROP Command

A. Drop the teachers table from the school_db database.

-> USE school_db;

-> DROP TABLE teachers;

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

-> DROP TABLE students;

-> SHOW TABLES;

7. Data Manipulation Language (DML)

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

-> INSERT INTO courses (course_id, course_name, course_duration)
VALUES
(101, 'Mathematics', '6 months'),
(102, 'Physics', '4 months'),
(103, 'Computer Science', '5 months');

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

-> UPDATE coursesSET course_duration = '6 months'
WHERE course_id = 102;

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

-> DELETE FROM courses WHERE course_id = 103;

8. Data Query Language (DQL)

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

-> SELECT * FROM courses;

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

-> SELECT * FROM courses ORDER BY course_duration DESC;

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

-> SELECT * FROM courses LIMIT 2;

9. Data Control Language (DCL)

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

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

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

-> GRANT SELECT ON school_db.courses TO 'user1'@'localhost';

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

-> REVOKE INSERT ON school_db.courses FROM 'user1'@'localhost';

-> GRANT INSERT ON school_db.courses TO 'user2'@'localhost';

10. Transaction Control Language (TCL)

A. 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, 'Biology', '4 months'),
(202, 'Chemistry', '5 months');

-> COMMIT;
```

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

```
-> INSERT INTO courses (course_id, course_name, course_duration)
VALUES
(203, 'English', '3 months'),
(204, 'History', '4 months');

-> ROLLBACK;
```

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

```
-> INSERT INTO courses (course_id, course_name, course_duration)
VALUES
(205, 'Geography', '6 months');

-> SAVEPOINT before_update;

-> UPDATE courses
SET course_name = 'Advanced Geography', course_duration = '7 months'
WHERE course_id = 205;

-> ROLLBACK TO SAVEPOINT before_update;

-> COMMIT;
```

11. SQL Joins

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

```
-> CREATE TABLE departments (
   department_id INT PRIMARY KEY,
   department_name VARCHAR(100)
);

-> INSERT INTO departments (department_id, department_name)
VALUES (1, 'HR'), (2, 'Finance'), (3, 'IT'), (4, 'Marketing');


-> CREATE TABLE employees (
   employee_id INT PRIMARY KEY,
   employee_name VARCHAR(100),
   department_id INT,
   FOREIGN KEY (department_id) REFERENCES
departments(department_id)
);

-> INSERT INTO employees (employee_id, employee_name, department_id)
VALUES (101, 'Alice', 1), (102, 'Bob', 2), (103, 'Charlie', 3), (104, 'David', 3);


-> SELECT
   employees.employee_id,
   employees.employee_name,
   departments.department_name
FROM
   employees
INNER JOIN
   departments
ON
   employees.department_id = departments.department_id;
```

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

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

12. SQL Group By

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

```
-> SELECT
   department_id,
   COUNT(*) AS employee_count
FROM
   employees
GROUP BY
   department_id;
```

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

```
-> SELECT
   department_id,
   AVG(salary) AS average_salary
FROM
   employees
GROUP BY
   department_id;
```

13. SQL Stored Procedure

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

```
->DELIMITER //

CREATE PROCEDURE GetEmployeesByDepartment(IN dept_id INT)
BEGIN
   SELECT
      employee_id,
      employee_name,
      department_id,
      salary
   FROM
      employees
   WHERE
      department_id = dept_id;
END //

DELIMITER ;
```

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

```
-> DELIMITER //

CREATE PROCEDURE GetCourseDetails(IN cid INT)
BEGIN
   SELECT
      course_id,
      course_name,
      course_duration
   FROM
      courses
   WHERE
      course_id = cid;
END //

DELIMITER ;


-> CALL GetCourseDetails(101);
```

14. SQL View

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

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



-> SELECT * FROM EmployeeDepartmentView;
```

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

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



-> SELECT * FROM EmployeeDepartmentView;
```

15. SQL Triggers

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

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

-> DELIMITER //

CREATE TRIGGER after_employee_insert
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
   INSERT INTO employee_log (employee_id, action)
   VALUES (NEW.employee_id, 'INSERT');
END//

DELIMITER ;
```

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

```
-> ALTER TABLE employees
ADD last_modified TIMESTAMP NULL;


-> DELIMITER //

CREATE TRIGGER before_employee_update
BEFORE UPDATE ON employees
FOR EACH ROW
BEGIN
   SET NEW.last_modified = NOW();
END//

DELIMITER ;
```

16. Introduction to PL/SQL

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

```
-> DECLARE
   v_total_employees NUMBER;
BEGIN
   -- Get the total count
   SELECT COUNT(*)
   INTO v_total_employees
   FROM employees;

   -- Print the result
   DBMS_OUTPUT.PUT_LINE('Total number of employees: ' ||
v_total_employees);
END;
/
```

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

```
-> CREATE TABLE orders (
   order_id NUMBER PRIMARY KEY,
   order_amount NUMBER
);
```

```
-> DECLARE
   v_total_sales NUMBER;
BEGIN
   -- Sum all order amounts
   SELECT SUM(order_amount)
   INTO v_total_sales
   FROM orders;

   -- Handle case when no rows (SUM would be NULL)
   IF v_total_sales IS NULL THEN
      v_total_sales := 0;
   END IF;

   -- Print the result
   DBMS_OUTPUT.PUT_LINE('Total sales: ' || v_total_sales);
END;
/
```

17. PL/SQL Control Structures

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

```
-> DECLARE
   v_employee_id    NUMBER := 101; -- You can change this value
   v_department_id  NUMBER;
BEGIN
   -- Get department_id of the employee
   SELECT department_id
   INTO v_department_id
   FROM employees
   WHERE employee_id = v_employee_id;

   -- IF-THEN logic
   IF v_department_id = 1 THEN
      DBMS_OUTPUT.PUT_LINE('Employee works in HR department.');
   ELSIF v_department_id = 2 THEN
      DBMS_OUTPUT.PUT_LINE('Employee works in Finance department.');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Employee works in another department.');
   END IF;
END;
/
```

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

```
-> DECLARE
   -- Cursor to select all employees
   CURSOR emp_cursor IS
      SELECT employee_name FROM employees;
BEGIN
   -- Loop through cursor
   FOR emp_rec IN emp_cursor LOOP
      DBMS_OUTPUT.PUT_LINE('Employee Name: ' ||
emp_rec.employee_name);
   END LOOP;
END;
/
```

18. SQL Cursors

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

```
-> DECLARE
   -- Define the cursor to get employee details
   CURSOR emp_cursor IS
      SELECT employee_id, employee_name, department_id, salary
      FROM employees;

   -- Variables to hold the data
   v_employee_id   employees.employee_id%TYPE;
   v_employee_name employees.employee_name%TYPE;
   v_department_id employees.department_id%TYPE;
   v_salary        employees.salary%TYPE;
BEGIN
   -- Open the cursor
   OPEN emp_cursor;

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

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

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

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

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

```
-> DECLARE
   -- Define the cursor to get all courses
   CURSOR course_cursor IS
      SELECT course_id, course_name, course_duration
      FROM courses;

   -- Variables to hold course data
   v_course_id       courses.course_id%TYPE;
   v_course_name     courses.course_name%TYPE;
   v_course_duration courses.course_duration%TYPE;
BEGIN
   -- Open the cursor
   OPEN course_cursor;

   LOOP
      -- Fetch each row
      FETCH course_cursor INTO v_course_id, v_course_name,
v_course_duration;

      -- Exit 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 ||
                           ', Duration: ' || v_course_duration);
   END LOOP;

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

19. Rollback and Commit Savepoint

A. 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, 'Art History', '3 months');

-- Insert second record
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (302, 'Philosophy', '4 months');

-- Create a savepoint
SAVEPOINT before_extra_inserts;

-- Insert more records after the savepoint
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (303, 'Psychology', '6 months');

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (304, 'Anthropology', '5 months');

-- Roll back to the savepoint (undo last two inserts)
ROLLBACK TO SAVEPOINT before_extra_inserts;

-- Commit remaining changes (first two inserts)
COMMIT;
```

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

```
-- Start transaction
-> START TRANSACTION;

-- Insert first records
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (401, 'Physics II', '4 months');

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (402, 'Chemistry II', '5 months');

-- Create a savepoint
SAVEPOINT first_batch_done;

-- Insert additional records
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (403, 'Biology II', '6 months');

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (404, 'Mathematics II', '7 months');

-- Commit everything up to this point (all inserts)
COMMIT;

-- Start a new transaction for the next inserts
START TRANSACTION;

-- Insert more records
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (405, 'English Literature', '4 months');

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (406, 'Economics', '5 months');

-- Decide to rollback these last inserts
ROLLBACK;
```