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 named school_db

```
CREATE DATABASE school_db;


-- Use the database
USE school_db;


-- Create the students table with specified columns
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(100),
    age INT,
    class VARCHAR(50),
    address VARCHAR(255)
);
```

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

- Insert five records into the students table

```
INSERT INTO students (student_id, student_name, age, class, address) VALUES
(1, 'Jay Patel', 15, '10th', 'Ahmedabad'),
(2, 'Rina Shah', 14, '9th', 'Surat'),
(3, 'Kiran Mehta', 16, '11th', 'Vadodara'),
(4, 'Pooja Desai', 15, '10th', 'Rajkot'),
(5, 'Manish Joshi', 13, '8th', 'Bhavnagar');


-- Retrieve all records from the students table
SELECT * FROM students;
```

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

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

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

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

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
);
```

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

Create teachers Table

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

Create students Table with FOREIGN KEY

```
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(100),
    age INT,
```

```
    class VARCHAR(50),
    address VARCHAR(255),
    teacher_id INT,
    CONSTRAINT fk_teacher
        FOREIGN KEY (teacher_id) REFERENCES teachers(teacher_id)
);
```

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
);
```

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

```
CREATE DATABASE university_db;
```

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

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

10. Drop the course_credits column from the courses table.

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

11. Drop the teachers table from the school_db database.

```
-- First, select the database
USE school_db;

-- Drop the teachers table
DROP TABLE teachers;
```

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

```
-- Select the database
USE school_db;

-- Drop the students table
DROP TABLE students;

-- Verify if the table has been removed
SHOW TABLES;
```

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

```
INSERT INTO courses (course_id, course_name, course_duration) VALUES
(1, 'Computer Science', '4 years'),
(2, 'Mathematics', '3 years'),
(3, 'Physics', '3 years');
```

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

```
UPDATE courses
SET course_duration = '4 years'
WHERE course_id = 2;
```

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

```
DELETE FROM courses
WHERE course_id = 3;
```

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

```
SELECT * FROM courses;
```

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

```
SELECT * FROM courses

ORDER BY course_duration DESC;
```

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

```
SELECT * FROM courses

ORDER BY course_duration DESC

LIMIT 2;
```

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

```
-- Create user1 with a password

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


-- Create user2 with a password

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

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

Revoke INSERT Permission from user1

```
REVOKE INSERT ON university_db.courses FROM 'user1'@'localhost';
```

Grant INSERT Permission to user2

```
GRANT INSERT ON university_db.courses TO 'user2'@'localhost';
```

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

inserting new rows

```
-- Insert new courses into the courses table

INSERT INTO courses (course_id, course_name, course_duration) VALUES

(4, 'Chemistry', '3 years'),

(5, 'Biology', '3 years'),

(6, 'English Literature', '2 years');
```

Commit the changes

COMMIT;

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

Insert additional rows

-- Insert new courses into the courses table

INSERT INTO courses (course_id, course_name, course_duration) VALUES

(7, 'History', '3 years'),

(8, 'Geography', '3 years');

Undo the last insert operation using ROLLBACK

ROLLBACK;

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

Create a SAVEPOINT before updating

SAVEPOINT before_update;

Perform updates on the courses table

-- Update course_duration for multiple courses
UPDATE courses
SET course_duration = '4 years'
WHERE course_id IN (9, 10);

Roll back to the SAVEPOINT

ROLLBACK TO SAVEPOINT before_update;

Commit the final state

COMMIT;

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) NOT NULL
);
```

Create employees Table

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100) NOT NULL,
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);
```

Insert Sample Data

```
-- Insert departments
INSERT INTO departments (dept_id, dept_name) VALUES
(1, 'Human Resources'),
(2, 'Finance'),
(3, 'IT');

-- Insert employees
INSERT INTO employees (emp_id, emp_name, dept_id) VALUES
(101, 'Alice', 1),
(102, 'Bob', 2),
(103, 'Charlie', 3),
(104, 'David', 3);
```

Perform INNER JOIN to Display 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;
```

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

```
SELECT d.dept_id, d.dept_name, e.emp_id, e.emp_name

FROM departments d

LEFT JOIN employees e

ON d.dept_id = e.dept_id;
```

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

```
SELECT d.dept_name, COUNT(e.emp_id) AS total_employees

FROM departments d

LEFT JOIN employees e ON d.dept_id = e.dept_id

GROUP BY d.dept_name;
```

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

Modify employees **to include a** salary **column (if not already present):**

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

Example data insertion with salaries:

```
UPDATE employees SET salary = 50000 WHERE emp_id = 101;
UPDATE employees SET salary = 60000 WHERE emp_id = 102;
UPDATE employees SET salary = 70000 WHERE emp_id = 103;
UPDATE employees SET salary = 65000 WHERE emp_id = 104;
```

Query using AVG **and** GROUP BY**:**

```
SELECT d.dept_name, AVG(e.salary) AS avg_salary
FROM departments d
LEFT JOIN employees e ON d.dept_id = e.dept_id
GROUP BY d.dept_name;
```

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

Create the Stored Procedure

```
DELIMITER //

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

DELIMITER ;
```

How to Call the Procedure

```
CALL GetEmployeesByDepartment(3);
```

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

Create the Stored Procedure

```
DELIMITER //

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

DELIMITER ;
```

How to Call the Procedure

For example, to get details for course_id = 2:

```
CALL GetCourseDetails(2);
```

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

Create the View

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

How to Query the View

```
SELECT * FROM EmployeeDepartmentView;
```

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

Drop the Existing View

```
DROP VIEW IF EXISTS EmployeeDepartmentView;
```

Create the Modified View

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

Query the Modified View

```
SELECT * FROM EmployeeDepartmentView;
```

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

Create the employee_log Table

This table will store the log entries whenever a new employee is added.

```
CREATE TABLE employee_log (
   log_id INT AUTO_INCREMENT PRIMARY KEY,
   emp_id INT,
   emp_name VARCHAR(100),
   action_time DATETIME,
   action VARCHAR(50)
);
```

Create the Trigger

```
DELIMITER //

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

DELIMITER ;
```

Verify the Trigger Works

Insert a new employee:

```
INSERT INTO employees (emp_id, emp_name, dept_id, salary)
```

VALUES (105, 'Emma', 2, 52000);

Check the log:

SELECT * FROM employee_log;

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

Add last_modified Column to the employees Table

ALTER TABLE employees
ADD last_modified DATETIME;

Create the Trigger

DELIMITER //

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

DELIMITER ;

✅ Step 3 – Test the Trigger

UPDATE employees
SET salary = 58000
WHERE emp_id = 105;

Then, check the updated record:

SELECT emp_id, emp_name, salary, last_modified FROM employees WHERE emp_id = 105;

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

DECLARE
  total_employees NUMBER;
BEGIN
  -- Retrieve the total number of employees
  SELECT COUNT(*) INTO total_employees FROM employees;

```
   -- Print the total number of employees
   DBMS_OUTPUT.PUT_LINE('Total number of employees: ' || total_employees);
END;
```

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

```
DECLARE
   total_sales NUMBER;
BEGIN
   -- Calculate the total sales by summing the order amounts
   SELECT SUM(order_amount) INTO total_sales FROM orders;

   -- Print the total sales amount
   DBMS_OUTPUT.PUT_LINE('Total sales amount: ' || NVL(total_sales, 0));
END;
```

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

```
DECLARE
   v_emp_id   employees.emp_id%TYPE := 101;  -- Example employee ID
   v_dept_id  employees.dept_id%TYPE;
BEGIN
   -- Get the department ID for the employee
   SELECT dept_id INTO v_dept_id FROM employees WHERE emp_id = v_emp_id;

   -- Check if the employee belongs to department 3 (for example, IT)
   IF v_dept_id = 3 THEN
      DBMS_OUTPUT.PUT_LINE('Employee ' || v_emp_id || ' belongs to the IT department.');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Employee ' || v_emp_id || ' does not belong to the IT
department.');
   END IF;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('No employee found with ID ' || v_emp_id);
END;
```

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

```
BEGIN
   FOR emp_rec IN (SELECT emp_name FROM employees)
   LOOP
      DBMS_OUTPUT.PUT_LINE('Employee Name: ' || emp_rec.emp_name);
   END LOOP;
END;
```

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

```
DECLARE
   -- Define the explicit cursor to select employee details
   CURSOR emp_cursor IS
      SELECT emp_id, emp_name, salary FROM employees;

   -- Define a record variable to hold each row fetched by the cursor
   emp_rec emp_cursor%ROWTYPE;
BEGIN
   -- Open the cursor
   OPEN emp_cursor;

   -- Loop through each record
   LOOP
      -- Fetch the next row into emp_rec
      FETCH emp_cursor INTO emp_rec;

      -- Exit the loop if no more rows are found
      EXIT WHEN emp_cursor%NOTFOUND;

      -- Display the employee details
      DBMS_OUTPUT.PUT_LINE('ID: ' || emp_rec.emp_id || ', Name: ' || emp_rec.emp_name ||
', Salary: ' || emp_rec.salary);
   END LOOP;

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

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

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

   -- Define a record to hold the fetched row
   course_rec course_cursor%ROWTYPE;
BEGIN
   -- Open the cursor
   OPEN course_cursor;

   -- Loop through the rows fetched by the cursor
   LOOP
      -- Fetch a row into the record variable
      FETCH course_cursor INTO course_rec;
```

```
        -- Exit the loop if no more rows are found
        EXIT WHEN course_cursor%NOTFOUND;

        -- Display the course details
        DBMS_OUTPUT.PUT_LINE('Course ID: ' || course_rec.course_id ||
                             ', Name: ' || course_rec.course_name ||
                             ', Duration: ' || course_rec.course_duration);
    END LOOP;

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

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

```
BEGIN
    -- Start the transaction implicitly

    -- Create a savepoint before inserting
    SAVEPOINT before_insert;

    -- Insert a new course record
    INSERT INTO courses (course_id, course_name, course_duration)
    VALUES (6, 'Astronomy', '2 years');

    INSERT INTO courses (course_id, course_name, course_duration)
    VALUES (7, 'Philosophy', '3 years');

    DBMS_OUTPUT.PUT_LINE('Records inserted.');

    -- Rollback to the savepoint, undoing the inserts
    ROLLBACK TO SAVEPOINT before_insert;

    DBMS_OUTPUT.PUT_LINE('Rolled back to the savepoint, inserts undone.');
END;
```

After running the block, check the courses table:

```
SELECT * FROM courses WHERE course_id IN (6, 7);
```

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

```
BEGIN
```

```
    -- Insert the first record and commit it
    INSERT INTO courses (course_id, course_name, course_duration)
    VALUES (8, 'Geology', '2 years');

    COMMIT;
    DBMS_OUTPUT.PUT_LINE('First record inserted and committed.');

    -- Create a savepoint before further changes
    SAVEPOINT before_more_inserts;

    -- Insert additional records
    INSERT INTO courses (course_id, course_name, course_duration)
    VALUES (9, 'Anthropology', '3 years');

    INSERT INTO courses (course_id, course_name, course_duration)
    VALUES (10, 'Sociology', '2 years');

    DBMS_OUTPUT.PUT_LINE('Additional records inserted.');

    -- Rollback to the savepoint, undoing the last two inserts
    ROLLBACK TO SAVEPOINT before_more_inserts;

    DBMS_OUTPUT.PUT_LINE('Rolled back to savepoint. Additional inserts undone.');

    -- Optionally commit the final state
    COMMIT;
END;
```

Check the contents of the courses table:

```
SELECT * FROM courses WHERE course_id IN (8, 9, 10);
```