Module – 4

(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: Queries :-

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

-- Use the created database
USE school_db;

-- Create the students table
CREATE TABLE students (
    student_id INT AUTO_INCREMENT PRIMARY KEY, -- Unique student ID
    student_name VARCHAR(100) NOT NULL,        -- Student name
    age INT,                                   -- Age of student
    class VARCHAR(50),                         -- Class/Grade
    address VARCHAR(255)                       -- Address of student
);
```

AUTO_INCREMENT makes sure student_id is generated automatically for each new record.
PRIMARY KEY ensures every student has a unique ID.

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

Ans: Queries :-

```sql
-- Insert 5 records into the students table
INSERT INTO students (student_name, age, class, address)
VALUES
('Rahul Sharma', 15, '10A', 'Delhi'),
('Priya Mehta', 14, '9B', 'Mumbai'),
('Amit Patel', 16, '11C', 'Ahmedabad'),
('Sneha Gupta', 13, '8A', 'Kolkata'),
('Karan Verma', 15, '10B', 'Chennai');

-- 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

Ans: Queries :-

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

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

Ans: Queries :-

```sql
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).

Ans: Queries :-

```sql
-- Create teachers table
CREATE TABLE teachers (
    teacher_id INT AUTO_INCREMENT PRIMARY KEY,  -- Unique ID for each teacher
    teacher_name VARCHAR(100) NOT NULL,         -- Teacher's name (cannot be NULL)
    subject VARCHAR(50) NOT NULL,               -- Subject taught (cannot be NULL)
    email VARCHAR(100) UNIQUE                   -- Email must be unique
);
```

teacher_id → Primary Key + Auto Increment (unique for each teacher).

teacher_name & subject → NOT NULL

email → UNIQUE so no two teachers can share the same email.

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

Ans: Queries :-

```sql
-- First, alter the students table to add teacher_id
ALTER TABLE students
ADD teacher_id INT;

-- Now, add the FOREIGN KEY constraint to link with teachers table
ALTER TABLE students
ADD 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.

Ans:

```sql
-- Create courses table
CREATE TABLE courses (
    course_id INT AUTO_INCREMENT PRIMARY KEY,  -- Unique ID for each course
    course_name VARCHAR(100) NOT NULL,         -- Name of the course
    course_credits INT NOT NULL                -- Number of credits for the course
);
```

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

Ans: Create university_db:-

```sql
-- Create a new database
CREATE DATABASE university_db;
```

Use of db:-

```sql
USE university_db;
```

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

Ans: Queries:-

```sql
-- Add a new column course_duration to the courses table
ALTER TABLE courses
ADD course_duration VARCHAR(50);
```

(10). Drop the course_credits column from the courses table.

Ans: Queries:-

```
-- Remove 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.

Ans: Queries:-

```
-- 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.

Ans: Queries:-

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

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

DROP TABLE students; permanently deletes the table and all its data.

SHOW TABLES; lists all tables in the current database. If students is not listed, it has been successfully removed.

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

Ans: Queries:-

```sql
-- Insert 3 records into the courses table
INSERT INTO courses (course_name, course_duration)
VALUES
('Computer Science', '6 months'),
('Mathematics', '1 year'),
('Physics', '8 months');

-- Retrieve all records to verify
SELECT * FROM courses;
```

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

Ans: Queries:-

```sql
-- Update course duration for Mathematics
UPDATE courses
SET course_duration = '18 months'
WHERE course_name = 'Mathematics';

-- Verify the update
SELECT * FROM courses;
```

Explanation:

- SET course_duration = '18 months' specifies the new value.
- WHERE course_name = 'Mathematics' ensures only the intended course is updated.
- Always include a WHERE clause to avoid updating all rows unintentionally.

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

Ans: Queries:-

```sql
-- Delete a course with a specific course_id, e.g., course_id = 2
DELETE FROM courses
WHERE course_id = 2;

-- Verify the deletion
SELECT * FROM courses;
```

Explanation:

- DELETE FROM courses removes rows from the table.
- WHERE course_id = 2 ensures only the course with that ID is deleted.
- Always include a WHERE clause to avoid deleting all records.

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

Ans: Queries:-

```sql
-- Retrieve all courses
SELECT * FROM courses;
```

Explanation:

- SELECT * returns **all columns** and **all rows** from the courses table.
- This will display course_id, course_name, and course_duration for every course.

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

Ans: Queries:-

```sql
-- Retrieve all courses sorted by course_duration in descending order
SELECT *
FROM courses
ORDER BY course_duration DESC;
```

Explanation:

- ORDER BY course_duration sorts the results based on the course_duration column.
- DESC ensures the sorting is in **descending order** (from longest to shortest).
- Use ASC instead if you want ascending order.

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

Ans: Queries:-

```
-- Retrieve only the top 2 courses
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.

Ans: Queries:-

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

-- Grant SELECT permission on the courses table to user1
GRANT SELECT ON university_db.courses TO 'user1'@'localhost';

-- Apply the privileges
FLUSH PRIVILEGES;
```

Explanation:

- CREATE USER creates the new database users.

- GRANT SELECT ON university_db.courses allows user1 to **read data** from the courses table only.
- FLUSH PRIVILEGES ensures the changes take effect immediately.

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

Ans: Queries:-

```
-- Revoke INSERT permission from user1 on the courses table
REVOKE INSERT ON university_db.courses FROM 'user1'@'localhost';

-- Grant INSERT permission to user2 on the courses table
GRANT INSERT ON university_db.courses TO 'user2'@'localhost';

-- Apply the changes
FLUSH PRIVILEGES;
```

Explanation:

- REVOKE INSERT removes the ability to insert new records.
- GRANT INSERT gives user2 the ability to add new rows to the table.
- FLUSH PRIVILEGES ensures the changes take effect immediately.

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

Ans: Queries:-

```
-- Start inserting rows
INSERT INTO courses (course_name, course_duration)
VALUES
('Biology', '10 months'),
('Chemistry', '12 months'),
('English', '8 months');

-- Commit the changes to make them permanent
COMMIT;

-- Verify the inserted records
SELECT * FROM courses;
```

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

Ans: Queries:-

```
-- Start a transaction (optional in some databases, but good practice)
START TRANSACTION;

-- Insert additional rows
INSERT INTO courses (course_name, course_duration)
VALUES
('History', '9 months'),
('Geography', '7 months');

-- Undo the last insert operation
ROLLBACK;

-- Verify that the last insert was undone
SELECT * FROM courses;
```

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

Ans: Queries:-

```
-- Start a transaction
START TRANSACTION;

-- Update some courses
UPDATE courses
SET course_duration = '14 months'
WHERE course_name = 'Biology';

UPDATE courses
SET course_duration = '15 months'
WHERE course_name = 'Chemistry';

-- Create a SAVEPOINT before the second update
SAVEPOINT before_chemistry_update;
```

```sql
-- Update Chemistry course duration again
UPDATE courses
SET course_duration = '18 months'
WHERE course_name = 'Chemistry';

-- Roll back to the SAVEPOINT to undo the last update only
ROLLBACK TO SAVEPOINT before_chemistry_update;

-- Commit the remaining changes
COMMIT;

-- Verify the final state
SELECT * FROM courses;
```

Explanation:

- START TRANSACTION begins a transaction.
- SAVEPOINT before_chemistry_update marks a point in the transaction.
- ROLLBACK TO SAVEPOINT undoes changes made **after the savepoint** without affecting earlier updates.
- COMMIT makes all remaining changes permanent.

This allows selective undo of changes while keeping earlier updates intact.

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

Ans: Queries:-

```sql
-- Create departments table
CREATE TABLE departments (
    department_id INT AUTO_INCREMENT PRIMARY KEY,
    department_name VARCHAR(100) NOT NULL
);

-- Create employees table
CREATE TABLE employees (
    employee_id INT AUTO_INCREMENT PRIMARY KEY,
    employee_name VARCHAR(100) NOT NULL,
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

```sql
-- Insert sample data into departments
INSERT INTO departments (department_name) VALUES
('HR'),
('Finance'),
('IT');

-- Insert sample data into employees
INSERT INTO employees (employee_name, department_id) VALUES
('Alice', 1),
('Bob', 2),
('Charlie', 3),
('David', 1);

-- Perform INNER JOIN to display employees with their departments
SELECT e.employee_id, e.employee_name, d.department_name
FROM employees e
INNER JOIN departments d
ON e.department_id = d.department_id;
```

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

Ans: Queries:-

```sql
-- Perform LEFT JOIN to show all departments and their employees (if any)
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;
```

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

Ans: Queries:-

```sql
-- Count employees in each department
SELECT d.department_name, COUNT(e.employee_id) AS employee_count
FROM departments d
LEFT JOIN employees e
ON d.department_id = e.department_id
GROUP BY d.department_name;
```

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

Ans: Queries:

```sql
-- Find average salary of employees in each department
SELECT d.department_name, AVG(e.salary) AS average_salary
FROM departments d
LEFT JOIN employees e
ON d.department_id = e.department_id
GROUP BY d.department_name;
```

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

Ans: Queries:

```sql
-- Create the stored procedure
DELIMITER //

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

DELIMITER ;
```

Explanation:

- IN dept_id INT defines an **input parameter** for the procedure.
- SELECT retrieves employees whose department_id matches the input.
- DELIMITER // is used to allow the procedure body to contain semicolons.

To call the procedure for, say, department 2:

```sql
CALL GetEmployeesByDepartment(2);
```

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

Ans: Queries:

```
-- 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 ;
```

Explanation:

- IN c_id INT defines an **input parameter** for the procedure.
- The SELECT statement retrieves the course details matching the input course_id.
- DELIMITER // is used to allow semicolons inside the procedure body.

To call the procedure for, say, course_id = 1:

```
CALL GetCourseDetails(1);
```

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

Ans: Queries:

```
-- Create a view for employees with their department names
CREATE VIEW EmployeeDepartmentView 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;
```

Explanation:

- CREATE VIEW EmployeeDepartmentView AS defines a view that can be queried like a table.
- INNER JOIN ensures only employees assigned to a department are included.
- You can now retrieve the data using:

```
SELECT * FROM EmployeeDepartmentView;
```

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

Ans: Queries:

```
-- Create a log table to store changes
CREATE TABLE employee_log (
    log_id INT AUTO_INCREMENT PRIMARY KEY,
    employee_id INT,
    employee_name VARCHAR(100),
    department_id INT,
    action_time DATETIME,
    action_type VARCHAR(50)
);
```

```
-- Create a trigger to log inserts into employees table
DELIMITER //

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

DELIMITER ;
```

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

Ans: Queries:

Step 1: Create a log table

```
CREATE TABLE employee_log (
    log_id        NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    employee_id   NUMBER,
    action        VARCHAR2(50),
    action_date   DATE DEFAULT SYSDATE,
    details       VARCHAR2(4000)
);
```

employee_id: stores the ID of the new employee.

action: describes the type of operation (e.g., 'INSERT').

action_date: timestamp when the action occurred.

details: optional info about the employee added.

Step 2: Create the trigger

```
CREATE OR REPLACE TRIGGER trg_employee_insert
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
    INSERT INTO employee_log(employee_id, action, details)
    VALUES (:NEW.employee_id, 'INSERT',
            'New employee added: ' || :NEW.first_name || ' ' || :NEW.last_name);
END;
/
```

AFTER INSERT ON employees: fires after a new row is inserted in employees.

NEW: refers to the new row being inserted.

The trigger inserts a record into employee_log automatically.

How it works

1. You add a new employee:

```
INSERT INTO employees(employee_id, first_name, last_name, department_id, salary)
VALUES (101, 'John', 'Doe', 10, 50000);
```

2. The trigger trg_employee_insert fires automatically.

3. A corresponding log entry is inserted into employee_log.

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

Ans: **Step 1: Ensure your table has a last_modified column**

```
ALTER TABLE employees
ADD last_modified DATE;
```

- This column will store the timestamp of the last update.

Step 2: Create the trigger

```
CREATE OR REPLACE TRIGGER trg_employee_update
BEFORE UPDATE ON employees
FOR EACH ROW
BEGIN
    :NEW.last_modified := SYSDATE;
END;
/
```

Explanation:

- BEFORE UPDATE ON employees: fires before an update on each row of the employees table.
- :NEW.last_modified := SYSDATE;: sets the last_modified column to the current timestamp automatically whenever the row is updated.

✅ **How it works**

1. You update an employee record:

```
UPDATE employees
SET salary = 60000
WHERE employee_id = 101;
```

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

Ans:

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

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

Explanation:

1. **DECLARE** section:
    - v_total_employees is a variable to store the count.
2. **BEGIN…END** section:
    - SELECT COUNT(*) INTO v_total_employees FROM employees; retrieves the total number of rows.
    - DBMS_OUTPUT.PUT_LINE prints the result.

Note: Make sure DBMS_OUTPUT is enabled to see the output. In SQL*Plus or SQL Developer:

```
SET SERVEROUTPUT ON;
```

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

Ans:

```
DECLARE
    v_total_sales NUMBER(12,2);  -- variable to store total sales
BEGIN
    -- Calculate the total sales
    SELECT SUM(order_amount)
    INTO v_total_sales
    FROM orders;

    -- Print the total sales
    DBMS_OUTPUT.PUT_LINE('Total sales: ' || TO_CHAR(v_total_sales, '9999990.00'));
END;
/
```

Explanation:

1. **DECLARE section**:
    - v_total_sales stores the sum of all order amounts.
2. **BEGIN section**:
    - SELECT SUM(order_amount) INTO v_total_sales FROM orders; calculates the total sales.
    - DBMS_OUTPUT.PUT_LINE prints the total sales formatted with two decimal places.

Note: Ensure DBMS_OUTPUT is enabled:

```
SET SERVEROUTPUT ON;
```

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

Ans:

```
BEGIN
    -- Loop through each employee record
    FOR emp_rec IN (SELECT first_name, last_name FROM employees) LOOP
        -- Display full name
        DBMS_OUTPUT.PUT_LINE(emp_rec.first_name || ' ' || emp_rec.last_name);
    END LOOP;
END;
/
```

Explanation:

1. **FOR emp_rec IN (SELECT …)**
    - This is a **cursor FOR LOOP**. It automatically opens, fetches, and closes the cursor.
    - emp_rec is a record holding the current row from the query.
2. **DBMS_OUTPUT.PUT_LINE**
    - Prints the employee's full name (first_name + last_name).

Note: Make sure DBMS_OUTPUT is enabled:

```
SET SERVEROUTPUT ON;
```

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

Ans: **Example Scenario:**

Suppose we have an employees table and we want to:

1. Insert two employees.
2. Commit the first insert.
3. Rollback the second insert.

PL/SQL Example:

```
BEGIN
    -- Insert first employee
    INSERT INTO employees(employee_id, first_name, last_name, department_id, salary)
    VALUES (201, 'Alice', 'Smith', 10, 50000);

    -- Create a savepoint after first insert
    SAVEPOINT sp_after_first_insert;

    -- Insert second employee
    INSERT INTO employees(employee_id, first_name, last_name, department_id, salary)
    VALUES (202, 'Bob', 'Johnson', 20, 60000);

    -- Rollback only the second insert to the savepoint
    ROLLBACK TO sp_after_first_insert;

    -- Commit the first insert
    COMMIT;
END;
/
```

Explanation:

1. **SAVEPOINT sp_after_first_insert**
    - Marks a point in the transaction after the first insert.
2. **ROLLBACK TO sp_after_first_insert**
    - Undoes changes made **after** the savepoint (here, the second insert).
    - The first insert remains intact.
3. **COMMIT**
    - Finalizes the changes **up to the savepoint**.

After this block:

- Employee Alice will be in the table.
- Employee Bob will **not** be inserted.