

Module 4 – Introduction to DBMS

Practical

1.Introduction to SQL

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 the database

```
CREATE DATABASE school_db;
```

2. Select the database

```
USE school_db;
```

3. 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

1. Insert five records

```
INSERT INTO students (student_id, student_name, age, class, address) VALUES  
(1, 'Rahul Sharma', 14, '8A', 'Delhi'),
```

(2, 'Sneha Kapoor', 13, '7B', 'Mumbai'),
(3, 'Amit Verma', 15, '9C', 'Kolkata'),
(4, 'Priya Singh', 12, '6A', 'Chennai'),
(5, 'Rohan Das', 14, '8B', 'Bangalore');

2. Retrieve all records

```
SELECT * FROM students;
```

2. SQL Syntax

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

1: Retrieve specific columns (student_name and age)

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

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

2: Retrieve all students whose age is greater than 10

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

3. SQL Constraints

Lab 1: 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 2: Implement a FOREIGN KEY constraint to relate the teacher_id from the teachers table with the students table

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

If the teacher_id column already exists

Just add the foreign key constraint:

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

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

Lab 1: 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 2: Use the CREATE command to create a database university_db.

```
CREATE DATABASE university_db;
```

5. ALTER Command

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

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

Lab 2: Drop the `course_credits` column from the `courses` table.

```
ALTER TABLE courses
```

```
DROP COLUMN course_credits;
```

6. DROP Command

Lab 1: Drop the `teachers` table from the `school_db` database.

If you are already inside the `school_db` database:

```
DROP TABLE teachers;
```

If not, first switch to the database:

```
USE school_db;
```

```
DROP TABLE teachers;
```

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

Step 1: Drop the table

```
DROP TABLE students;
```

Step 2: Verify that the table is removed

You can check the tables in the database:

```
SHOW TABLES;
```

If `students` does NOT appear in the list, the table has been successfully removed.

7. Data Manipulation Language (DML)

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

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

```
VALUES
```

```
(101, 'Computer Science Basics', '3 Months'),
```

```
(102, 'Mathematics Fundamentals', '2 Months'),
```

```
(103, 'Physics Introduction', '4 Months');
```

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

```
UPDATE courses
```

```
SET course_duration = '3 Months'
```

```
WHERE course_id = 102;
```

Lab 3: 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)

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

Retrieve all courses

```
SELECT * FROM courses;
```

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

: Sort courses by course_duration in descending order

```
SELECT * FROM courses
```

```
ORDER BY course_duration DESC;
```

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

Limit results to show only the top two courses

```
SELECT * FROM courses
```

```
LIMIT 2;
```

9. Data Control Language (DCL)

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

Step 1: Create users

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

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

(You can replace the passwords with your own.)

Step 2: Grant SELECT permission on courses table to user1

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

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

Step 1: Revoke INSERT permission from user1

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

Step 2: Grant INSERT permission to user2

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

10. Transaction Control Language (TCL)

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

```
START TRANSACTION;
```

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

```
VALUES
```

```
(201, 'Chemistry Basics', '3 Months'),
```

```
(202, 'Biology Introduction', '2 Months');
```

```
COMMIT;
```

Explanation:

COMMIT permanently saves the changes to the database.

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

```
START TRANSACTION;
```

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

```
VALUES
```

```
(203, 'History Fundamentals', '4 Months'),
```

```
(204, 'English Literature', '3 Months');
```

```
ROLLBACK;
```

Effect:

The last two inserted rows (203, 204) will be undone.

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

```
START TRANSACTION;
```

-- Create a savepoint before making updates

SAVEPOINT before_update;

-- Perform updates

UPDATE courses

SET course_duration = '5 Months'

WHERE course_id = 201;

UPDATE courses

SET course_duration = '6 Months'

WHERE course_id = 202;

-- Roll back only the last updates up to the savepoint

ROLLBACK TO before_update;

COMMIT;

Explanation:

The SAVEPOINT marks a point in the transaction.

ROLLBACK TO undoes changes after the savepoint but keeps everything before it.

Finally, COMMIT saves the remaining changes.

11. SQL Joins

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

Step 1: Create `departments` table

```
CREATE TABLE departments (  
    dept_id INT PRIMARY KEY,  
    dept_name VARCHAR(100)  
);
```

Step 2: 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)  
);
```

Step 3: Example data (optional)

```
INSERT INTO departments (dept_id, dept_name)  
VALUES (1, 'HR'), (2, 'Finance'), (3, 'IT');  
  
INSERT INTO employees (emp_id, emp_name, dept_id)  
VALUES  
(101, 'Alice', 1),  
(102, 'Bob', 3),  
(103, 'Charlie', 2);
```

Step 4: INNER JOIN to show employees with their departments

```
SELECT employees.emp_id, employees.emp_name, departments.dept_name
FROM employees
INNER JOIN departments
ON employees.dept_id = departments.dept_id;
```

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

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

Result:

All departments will be shown.

Departments with no employees will show NULL in emp_name.

12. SQL Group By

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

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

Result:

This will show each dept_id along with how many employees belong to that department.

If you want department names included:

sql

Copy code

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

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

salary INT

Query:

sql

Copy code

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

Or with department names:

sql

Copy code

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

13. SQL Stored Procedure

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

DELIMITER \$\$

```
CREATE PROCEDURE GetEmployeesByDepartment(IN dep_id INT)
```

```
BEGIN
```

```
    SELECT emp_id, emp_name, dept_id
```

```
    FROM employees
```

```
    WHERE dept_id = dep_id;
```

```
END $$
```

DELIMITER ;

✓ How to call the procedure:

sql

Copy code

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

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

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:

sql

Copy code

```
CALL GetCourseDetails(101);
```

14. SQL View

Lab 1: 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
JOIN departments d
ON e.dept_id = d.dept_id;
```

Lab 2: 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
JOIN departments d
ON e.dept_id = d.dept_id
WHERE e.salary >= 50000;
```

15. SQL Triggers

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

Trigger to Log New Employee Insertions

Step 1: Create a log table (if not already created)

This table will store log details when a new employee is added.

```
CREATE TABLE employee_logs (
    log_id INT AUTO_INCREMENT PRIMARY KEY,
    emp_id INT,
    emp_name VARCHAR(100) ,
```

```
        action_type VARCHAR(50),  
        log_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
    );
```

Step 2: Create the trigger

```
DELIMITER $$
```

```
CREATE TRIGGER log_employee_insert  
AFTER INSERT ON employees  
FOR EACH ROW  
BEGIN  
    INSERT INTO employee_logs (emp_id, emp_name, action_type)  
    VALUES (NEW.emp_id, NEW.emp_name, 'INSERT');  
END $$
```

```
DELIMITER ;
```

This trigger runs **automatically** whenever a new employee is added.

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

Trigger to Update last_modified Timestamp on Update

Step 1: Ensure employees table has a last_modified column

```
ALTER TABLE employees
```

```
ADD last_modified TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE  
CURRENT_TIMESTAMP;
```

(If this column already exists, skip this step.)

Step 2: Create the trigger

```
DELIMITER $$
```

```
CREATE TRIGGER update_last_modified  
BEFORE UPDATE ON employees  
FOR EACH ROW  
BEGIN  
    SET NEW.last_modified = CURRENT_TIMESTAMP;  
END $$
```

```
DELIMITER ;
```

This trigger updates `last_modified` every time an employee record is modified.

16. Introduction to PL/SQL

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

: PL/SQL Block to Print Total Number of Employees

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

/

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

```
DECLARE

    total_sales NUMBER;

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

Assuming we are checking for dept_id = 2:

```
DECLARE

    v_dept_id employees.dept_id%TYPE;

    v_emp_name employees.emp_name%TYPE;

BEGIN

    -- Example: check employee with emp_id = 101

    SELECT dept_id, emp_name INTO v_dept_id, v_emp_name
```

```

FROM employees

WHERE emp_id = 101;

IF v_dept_id = 2 THEN

    DBMS_OUTPUT.PUT_LINE(v_emp_name || ' belongs to the Finance department.');
```

ELSE

```

    DBMS_OUTPUT.PUT_LINE(v_emp_name || ' does not belong to the Finance
department.');
```

END IF;

```

END;

/
```

This block checks the department of a specific employee and prints a message.

. Lab 2: 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;

/
```

: This block iterates through all employee records and prints their names.

18. SQL Cursors

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

PL/SQL Block Using an Explicit Cursor for Employee Details

DECLARE

-- Declare cursor

CURSOR emp_cursor IS

SELECT emp_id, emp_name, dept_id, salary

FROM employees;

-- Declare variables to hold cursor values

v_emp_id employees.emp_id%TYPE;

v_emp_name employees.emp_name%TYPE;

v_dept_id employees.dept_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_emp_id, v_emp_name, v_dept_id, v_salary;

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

-- Display employee details

DBMS_OUTPUT.PUT_LINE('ID: ' || v_emp_id || ', Name: ' || v_emp_name ||
' , Dept ID: ' || v_dept_id || ', Salary: ' || v_salary);

END LOOP;

```
-- Close the cursor

CLOSE emp_cursor;

END;

/
```

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

PL/SQL Block Using a Cursor to Retrieve All Courses

```
DECLARE

    -- Declare cursor

    CURSOR course_cursor IS

        SELECT course_id, course_name, course_duration

        FROM courses;

    -- Variables to hold cursor values

    v_course_id courses.course_id%TYPE;

    v_course_name courses.course_name%TYPE;

    v_course_duration courses.course_duration%TYPE;

BEGIN

    OPEN course_cursor;

    LOOP

        FETCH course_cursor INTO v_course_id, v_course_name,
v_course_duration;

        EXIT WHEN course_cursor%NOTFOUND;

        DBMS_OUTPUT.PUT_LINE('Course ID:' || v_course_id ||
```

```

        ', Name: ' || v_course_name ||
        ', Duration: ' || v_course_duration);

    END LOOP;

    CLOSE course_cursor;

END;

/

```

19. Rollback and Commit Savepoint

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

Transaction with Savepoint, Insert, and Rollback

-- Start the transaction

```
START TRANSACTION;
```

-- Create a savepoint

```
SAVEPOINT before_insert;
```

-- Insert some records into courses table

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

```
VALUES (301, 'Art History', '2 Months');
```

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

```
VALUES (302, 'Philosophy 101', '3 Months');
```

-- Rollback to savepoint (undo the inserts after the savepoint)

```
ROLLBACK TO before_insert;
```

-- Commit the transaction (nothing inserted after rollback)

```
COMMIT;
```

Effect:

The inserted rows (301, 302) are **undone**, but the transaction itself is still valid.

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

Commit Part of a Transaction, Then Rollback Remaining Changes

-- Start the transaction

```
START TRANSACTION;
```

-- Insert first set of records

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

```
VALUES (303, 'Economics Basics', '3 Months');
```

-- Commit this part

```
COMMIT;
```

-- Create a savepoint before second set of inserts

```
SAVEPOINT before_second_insert;
```

-- Insert additional records

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

```
VALUES (304, 'Political Science', '4 Months');
```

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

```
VALUES (305, 'Sociology 101', '3 Months');
```

```
-- Rollback to savepoint (undo second set)
```

```
ROLLBACK TO before_second_insert;
```

```
-- Commit remaining changes (only the first insert remains)
```

```
COMMIT;
```

Effect:

303 (Economics Basics) is **saved permanently**.

304 and 305 are **rolled back**.

❖ EXTRA LAB PRACTISE FOR DATABASE CONCEPT

1.Introduction to SQ

Lab 3: Create a database called library_db and a table books with columns: book_id, title, author, publisher, year_of_publication, and price. Insert five records into the table.

Create library_db and books table, then insert records

Step 1: Create the database

```
CREATE DATABASE library_db;
```

Step 2: Use the database

```
USE library_db;
```

Step 3: Create books table

```
CREATE TABLE books (  
    book_id INT PRIMARY KEY,  
    title VARCHAR(100),  
    author VARCHAR(100),  
    publisher VARCHAR(100),  
    year_of_publication YEAR,  
    price DECIMAL(8,2)  
);
```

Step 4: Insert five records into books table

```
INSERT INTO books (book_id, title, author, publisher, year_of_publication,  
price)  
VALUES  
(1, 'The Great Gatsby', 'F. Scott Fitzgerald', 'Scribner', 1925, 350.00),  
(2, '1984', 'George Orwell', 'Secker & Warburg', 1949, 300.00),  
(3, 'To Kill a Mockingbird', 'Harper Lee', 'J.B. Lippincott & Co.', 1960, 400.00),  
(4, 'Pride and Prejudice', 'Jane Austen', 'T. Egerton', 1813, 250.00),  
(5, 'The Hobbit', 'J.R.R. Tolkien', 'George Allen & Unwin', 1937, 450.00);
```

Lab 4: Create a table members in library_db with columns: member_id, member_name, date_of_membership, and email. Insert five records into this table.

Create members table and insert records

Step 1: Create members table

```
CREATE TABLE members (  
    member_id INT PRIMARY KEY,
```

```
member_name VARCHAR(100),  
date_of_membership DATE,  
email VARCHAR(100) UNIQUE  
);
```

Step 2: Insert five records into `members` table

```
INSERT INTO members (member_id, member_name, date_of_membership, email)  
VALUES  
(1, 'Alice Johnson', '2025-01-15', 'alice@example.com'),  
(2, 'Bob Smith', '2025-02-20', 'bob@example.com'),  
(3, 'Charlie Brown', '2025-03-05', 'charlie@example.com'),  
(4, 'Diana Prince', '2025-04-10', 'diana@example.com'),  
(5, 'Ethan Hunt', '2025-05-25', 'ethan@example.com');
```

2.SQL Syntax

Lab 3: Retrieve all members who joined the library before 2022. Use appropriate SQL syntax with WHERE and ORDER BY.

Retrieve all members who joined before 2022, sorted by `date_of_membership`

```
SELECT *  
FROM members  
WHERE date_of_membership < '2022-01-01'  
ORDER BY date_of_membership;
```

Explanation:

`WHERE` filters members who joined before 2022.

ORDER BY sorts the results by `date_of_membership` in ascending order.

Lab 4: Write SQL queries to display the titles of books published by a specific author. Sort the results by `year_of_publication` in descending order.

Display titles of books by a specific author, sorted by year descending

Example: Author = 'George Orwell'

```
SELECT title, year_of_publication
FROM books
WHERE author = 'George Orwell'
ORDER BY year_of_publication DESC;
```

Explanation:

WHERE filters books by the specific author.

ORDER BY ... DESC sorts the books from newest to oldest publication.

3.SQL Constraints

Lab 3: Add a CHECK constraint to ensure that the price of books in the books table is greater than 0.

Add a CHECK constraint on `books.price`

```
ALTER TABLE books
ADD CONSTRAINT chk_price_positive
CHECK (price > 0);
```

Explanation:

Ensures that no book can have a price less than or equal to 0.

Lab 4: Modify the members table to add a UNIQUE constraint on the email column, ensuring that each member has a unique email address.

Add a UNIQUE constraint on `members.email`

```
ALTER TABLE members  
  
ADD CONSTRAINT unique_email  
  
UNIQUE (email);
```

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

Lab 3: Create a table authors with the following columns: author_id, first_name, last_name, and country. Set author_id as the primary key.

Create authors table

```
CREATE TABLE authors (  
  
    author_id INT PRIMARY KEY,  
  
    first_name VARCHAR(50),  
  
    last_name VARCHAR(50),  
  
    country VARCHAR(50)  
  
);
```

Explanation:

`author_id` is the primary key, ensuring each author has a unique ID.

Lab 4: Create a table publishers with columns: publisher_id, publisher_name, contact_number, and address. Set publisher_id as the primary key and contact_number as unique.

Create publishers table

```
CREATE TABLE publishers (  
    publisher_id INT PRIMARY KEY,  
    publisher_name VARCHAR(100) ,  
    contact_number VARCHAR(15) UNIQUE,  
    address VARCHAR(200)  
);
```

Explanation:

publisher_id is the primary key.

contact_number is set as UNIQUE to prevent duplicate contact numbers.

5.ALTER Command

Lab 3: Add a new column genre to the books table. Update the genre for all existing records.

Add genre column to books table and update existing records

Step 1: Add the new column

```
ALTER TABLE books  
ADD genre VARCHAR(50) ;
```

Step 2: Update the genre for existing records

```
UPDATE books  
SET genre = 'Fiction'
```

```
WHERE book_id IN (1, 2, 3, 4, 5);
```

Lab 4: Modify the members table to increase the length of the email column to 100 characters

Increase length of `email` column in `members` table

```
ALTER TABLE members  
MODIFY email VARCHAR(100);
```

Explanation:

This increases the maximum allowed length of the `email` column to 100 characters.

6.DROP Command

Lab 3: Drop the publishers table from the database after verifying its structure.

Drop the `publishers` table after verifying its structure

Step 1: Verify the table structure

```
DESCRIBE publishers;
```

or

```
SHOW CREATE TABLE publishers;
```

Step 2: Drop the table

```
DROP TABLE publishers;
```

Lab 4: Create a backup of the members table and then drop the original members table.

Backup the `members` table and drop the original

Step 1: Create a backup table

```
CREATE TABLE members_backup AS
```

```
SELECT * FROM members;
```

Step 2: Drop the original `members` table

```
DROP TABLE members;
```

Explanation:

`members_backup` now contains all the data from `members`.

The original `members` table is removed, but the backup remains.

7.Data Manipulation Language (DML)

Lab 4: Insert three new authors into the authors table, then update the last name of one of the authors

Insert three new authors and update the last name of one

Step 1: Insert three authors

```
INSERT INTO authors (author_id, first_name, last_name, country)
```

```
VALUES
```

```
(1, 'George', 'Orwell', 'United Kingdom'),
```

```
(2, 'Harper', 'Lee', 'United States'),
```

```
(3, 'Jane', 'Austen', 'United Kingdom');
```

Step 2: Update the last name of one author

Example: Change Jane Austen's last name to `Doe`:

```
UPDATE authors
```

```
SET last_name = 'Doe'  
  
WHERE author_id = 3;
```

Lab 5: Delete a book from the books table where the price is higher than \$1

Delete books where price is higher than \$100

```
DELETE FROM books  
  
WHERE price > 100;
```

Explanation:

This removes all book records with a price greater than \$100 from the `books` table.

8.UPDATE Command

Lab 3: Update the year_of_publication of a book with a specific book_id.

Update `year_of_publication` for a specific book

Example: Update the book with `book_id = 1`:

```
UPDATE books  
  
SET year_of_publication = 2020  
  
WHERE book_id = 1;
```

Explanation:

Updates the publication year of the specified book.

Lab 4: Increase the price of all books published before 2015 by 10%

Increase price of all books published before 2015 by 10%

```
UPDATE books  
  
SET price = price * 1.10  
  
WHERE year_of_publication < 2015;
```

Explanation:

Multiplies the current price by 1.10 (increasing by 10%) for all books published before 2015.

9.DELETE Command

Lab 3: Remove all members who joined before 2020 from the members table

Remove members who joined before 2020

```
DELETE FROM members  
  
WHERE date_of_membership < '2020-01-01';
```

Explanation:

Deletes all rows where `date_of_membership` is before 2020.

Lab 4: Delete all books that have a NULL value in the author column.

Delete books with NULL in the `author` column

```
DELETE FROM books  
  
WHERE author IS NULL;
```

Explanation:

Removes all book records where the `author` field is NULL.

10.Data Query Language (DQL)

Lab 4: Write a query to retrieve all books with price between \$50 and \$100.

SELECT *

FROM books

WHERE price BETWEEN 50 AND 100;

Explanation:

BETWEEN 50 AND 100 includes books priced greater than or equal to 50 and less than or equal to 100.

Lab 5: Retrieve the list of books sorted by author in ascending order and limit the results to the top 3 entries

SELECT *

FROM books

ORDER BY author ASC

LIMIT 3;

Explanation:

Sorts the books alphabetically by author.

LIMIT 3 shows only the first 3 records in the sorted list.

11.Data Control Language (DCL)

Lab 3: Grant SELECT permission to a user named librarian on the books table.

```
GRANT SELECT ON library_db.books TO 'librarian'@'localhost';
```

Explanation:

Allows the user librarian to only read data from the books table.

Lab 4: Grant INSERT and UPDATE permissions to the user admin on the members table.

```
GRANT INSERT, UPDATE ON library_db.members TO 'admin'@'localhost';
```

Explanation:

Gives the user admin the ability to add new members and modify existing member records.

12.REVOKE Command

Lab 3: Revoke the INSERT privilege from the user librarian on the books table.

```
REVOKE INSERT ON library_db.books FROM 'librarian'@'localhost';
```

Explanation:

Removes the ability for librarian to insert new records into the books table.

Lab 4: Revoke all permissions from user admin on the members table.

REVOKE ALL PRIVILEGES ON library_db.members FROM 'admin'@'localhost';

Explanation:

Removes all previously granted privileges for admin on the members table, including SELECT, INSERT, UPDATE, DELETE, etc.

13.Transaction Control Language (TCL)

Lab 3: Use COMMIT after inserting multiple records into the books table, then make another insertion and perform a ROLLBACK.

-- Start the transaction

START TRANSACTION;

-- Insert multiple records

INSERT INTO books (book_id, title, author, publisher, year_of_publication, price)

VALUES

(401, 'Data Science Basics', 'Alice Smith', 'TechPub', 2021, 500.00),

(402, 'Machine Learning 101', 'Bob Johnson', 'TechPub', 2022, 600.00);

-- Commit these insertions

COMMIT;

-- Insert another record

INSERT INTO books (book_id, title, author, publisher, year_of_publication, price)

VALUES (403, 'Deep Learning Guide', 'Charlie Brown', 'TechPub', 2023, 700.00);

-- Rollback the last insertion

ROLLBACK;

Effect:

Records 401 and 402 are saved permanently.

Record 403 is undone.

Lab 4: Set a SAVEPOINT before making updates to the members table, perform some updates, and then roll back to the SAVEPOINT.

-- Start the transaction

START TRANSACTION;

-- Set a savepoint before updates

SAVEPOINT before_update;

-- Perform updates on members table

UPDATE members

SET email = 'alice_new@example.com'

WHERE member_id = 1;

UPDATE members

SET email = 'bob_new@example.com'

WHERE member_id = 2;

-- Rollback to savepoint (undo the updates)

ROLLBACK TO before_update;

-- Commit remaining changes (if any)

COMMIT;

Effect:

Any updates done after the savepoint are undone.

Transaction remains valid for other operations.

14.SQL Joins

Lab 3: Perform an INNER JOIN between books and authors tables to display the title of books and their respective authors' names.

INNER JOIN to display book titles with their authors

SELECT b.title, a.first_name, a.last_name

FROM books b

INNER JOIN authors a

ON b.author = CONCAT(a.first_name, ' ', a.last_name);

Explanation:

Only books with a matching author in the `authors` table will be displayed.

`CONCAT(a.first_name, ' ', a.last_name)` assumes the `books.author` column stores full names. Adjust as needed.

Lab 4: Use a FULL OUTER JOIN to retrieve all records from the books and authors tables, including those with no matching entries in the other table.

FULL OUTER JOIN to include all books and authors

Note: MySQL does not support `FULL OUTER JOIN` directly. You can simulate it using `UNION` of `LEFT` and `RIGHT JOINS`.

```
SELECT b.title, a.first_name, a.last_name
FROM books b
LEFT JOIN authors a
ON b.author = CONCAT(a.first_name, ' ', a.last_name)
```

UNION

```
SELECT b.title, a.first_name, a.last_name
FROM books b
RIGHT JOIN authors a
ON b.author = CONCAT(a.first_name, ' ', a.last_name);
```

Explanation:

Includes all books and all authors, even if there is no match in the other table.

Rows with no match will display `NULL` for missing columns.

15.SQL Group By

Lab 3: Group books by genre and display the total number of books in each genre.

```
SELECT genre, COUNT(*) AS total_books
```

```
FROM books
```

```
GROUP BY genre;
```

Explanation:

Groups all books by their genre.

COUNT(*) returns the number of books in each genre.

Lab 4: Group members by the year they joined and find the number of members who joined each year.

```
SELECT YEAR(date_of_membership) AS join_year, COUNT(*) AS total_members
```

```
FROM members
```

```
GROUP BY YEAR(date_of_membership)
```

```
ORDER BY join_year;
```

Explanation:

YEAR(date_of_membership) extracts the year from the membership date.

Groups members by join year and counts them.

ORDER BY sorts the results by year.

16.SQL Stored Procedure

Lab 3: Write a stored procedure to retrieve all books by a particular author.

DELIMITER \$\$

```
CREATE PROCEDURE GetBooksByAuthor(IN author_name VARCHAR(100))
```

```
BEGIN
```

```
    SELECT book_id, title, genre, price
```

```
    FROM books
```

```
    WHERE author = author_name;
```

```
END $$
```

DELIMITER ;

How to call the procedure:

sql

Copy code

```
CALL GetBooksByAuthor('George Orwell');
```

Explanation

Takes the author's name as input and returns all books written by that author.

Lab 4: Write a stored procedure that takes book_id as an argument and returns the price of the book.

DELIMITER \$\$

CREATE PROCEDURE GetBookPrice(IN b_id INT)

BEGIN

SELECT title, price

FROM books

WHERE book_id = b_id;

END \$\$

DELIMITER ;

How to call the procedure:

sql

Copy code

CALL GetBookPrice(1);

Explanation:

Accepts a book_id as input and returns the title and price of that book.

17.SQL View

Lab 3: Create a view to show only the title, author, and price of books from the books table.

```
CREATE VIEW books_view AS
```

```
SELECT title, author, price
```

```
FROM books;
```

Explanation:

The view books_view will only display the selected columns (title, author, price) from the books table.

Lab 4: Create a view to display members who joined before 2020.

```
CREATE VIEW members_before_2020 AS
```

```
SELECT member_id, member_name, date_of_membership, email
```

```
FROM members
```

```
WHERE date_of_membership < '2020-01-01';
```

Explanation:

The view members_before_2020 filters members by date_of_membership before 2020.

18.SQL Trigger

Lab 3: Create a trigger to automatically update the last_modified timestamp of the books table whenever a record is updated.

Step 1: Ensure last_modified column exists

```
ALTER TABLE books
```

```
ADD last_modified TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE  
CURRENT_TIMESTAMP;
```

(Skip if already present.)

Step 2: Create the trigger

```
DELIMITER $$
```

```
CREATE TRIGGER update_books_timestamp
```

```
BEFORE UPDATE ON books
```

```
FOR EACH ROW
```

```
BEGIN
```

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

```
END $$
```

```
DELIMITER ;
```

Explanation:

Automatically updates the last_modified column whenever a row in books is updated.

Lab 4: Create a trigger that inserts a log entry into a log_changes table whenever a DELETE operation is performed on the books table.

Step 1: Create a log table

```
CREATE TABLE log_changes (  
    log_id INT AUTO_INCREMENT PRIMARY KEY,  
    book_id INT,  
    title VARCHAR(100),  
    deleted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
);
```

Step 2: Create the trigger

```
DELIMITER $$  
  
CREATE TRIGGER log_book_deletion  
AFTER DELETE ON books  
FOR EACH ROW  
BEGIN  
    INSERT INTO log_changes (book_id, title)  
    VALUES (OLD.book_id, OLD.title);  
END $$  
  
DELIMITER ;
```

Explanation:

Whenever a book is deleted, its `book_id` and `title` are recorded in the `log_changes` table along with the timestamp.

19. Introduction to PL/SQ

3: Write a PL/SQL block to insert a new book into the books table and display a confirmation message.

```
DECLARE

BEGIN

    -- Insert a new book

    INSERT INTO books (book_id, title, author, publisher, year_of_publication, price)

    VALUES (501, 'Introduction to AI', 'Alice Smith', 'TechPub', 2025, 550.00);

    -- Display confirmation message

    DBMS_OUTPUT.PUT_LINE('Book "Introduction to AI" has been successfully added.');
```

END;

/

Explanation:

Inserts a new record into the books table.

Prints a confirmation message using DBMS_OUTPUT.PUT_LINE.

Lab 4: Write a PL/SQL block to display the total number of books in the books table.

```
DECLARE
```

```
total_books NUMBER;

BEGIN

-- Count total books

SELECT COUNT(*) INTO total_books

FROM books;


-- Display the total

DBMS_OUTPUT.PUT_LINE('Total number of books: ' || total_books);

END;

/
```

Explanation:

Uses SELECT COUNT(*) INTO to store the total number of books in a variable.

Prints the total using DBMS_OUTPUT.PUT_LINE.

20.PL/SQL Syntax

Lab 3: Write a PL/SQL block to declare variables for book_id and price, assign values, and display the results.

```
DECLARE

v_book_id books.book_id%TYPE;

v_price books.price%TYPE;
```

```

BEGIN

    -- Assign values

    v_book_id := 601;

    v_price := 499.99;


    -- Display the values

    DBMS_OUTPUT.PUT_LINE('Book ID: ' || v_book_id);

    DBMS_OUTPUT.PUT_LINE('Price: $' || v_price);

END;

/

```

Explanation:

Declares variables for book_id and price.

Assigns values and prints them using DBMS_OUTPUT.PUT_LINE.

Lab 4: Write a PL/SQL block using constants and perform arithmetic operations on book prices.

```

DECLARE

    c_tax_rate CONSTANT NUMBER := 0.10; -- 10% tax

    v_price books.price%TYPE := 500;    -- Example price

    v_total_price NUMBER;

BEGIN

```

```
-- Calculate total price with tax

v_total_price := v_price + (v_price * c_tax_rate);


-- Display results

DBMS_OUTPUT.PUT_LINE('Original Price: $' || v_price);

DBMS_OUTPUT.PUT_LINE('Price after 10% tax: $' || v_total_price);

END;

/
```

Explanation:

Declares a constant `c_tax_rate`.

Performs arithmetic to calculate total price including tax.

Displays both original and final prices.

21.PL/SQL Control Structures

Lab 3: Write a PL/SQL block using IF-THEN-ELSE to check if a book's price is above \$100 and print a message accordingly.

```
DECLARE

v_price books.price%TYPE;
```

```

v_title books.title%TYPE;

BEGIN

-- Example: check book with book_id = 1

SELECT price, title INTO v_price, v_title

FROM books

WHERE book_id = 1;


IF v_price > 100 THEN

    DBMS_OUTPUT.PUT_LINE('The book "' || v_title || '" is priced above $100.');
```

ELSE

```

    DBMS_OUTPUT.PUT_LINE('The book "' || v_title || '" is priced $100 or below.');
```

END IF;

```

END;

/
```

Explanation:

Fetches the price and title of a specific book.

Uses IF-THEN-ELSE to print a message based on the price.

Lab 4: Use a FOR LOOP in PL/SQL to display the details of all books one by one.

```

BEGIN

FOR book_rec IN (SELECT book_id, title, author, price FROM books) LOOP

    DBMS_OUTPUT.PUT_LINE('Book ID: ' || book_rec.book_id ||
```

```
        ', Title: ' || book_rec.title ||  
        ', Author: ' || book_rec.author ||  
        ', Price: $' || book_rec.price);  
  
    END LOOP;  
  
END;  
  
/
```

Explanation:

Iterates through all rows in the books table using a cursor FOR LOOP.

Prints each book's ID, title, author, and price.

22.SQL Cursors

Lab 3: Write a PL/SQL block using an explicit cursor to fetch and display all records from the members table.

```
DECLARE  
  
    -- Declare cursor  
  
    CURSOR members_cursor IS  
  
        SELECT member_id, member_name, date_of_membership, email  
  
        FROM members;  
  
  
    -- Variables to hold cursor values
```

```

v_member_id members.member_id%TYPE;

v_member_name members.member_name%TYPE;

v_date_of_membership members.date_of_membership%TYPE;

v_email members.email%TYPE;

BEGIN

-- Open the cursor

OPEN members_cursor;

LOOP

    FETCH members_cursor INTO v_member_id, v_member_name, v_date_of_membership, v_email;

    EXIT WHEN members_cursor%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE('ID: ' || v_member_id ||

        ', Name: ' || v_member_name ||

        ', Joined: ' || v_date_of_membership ||

        ', Email: ' || v_email);

END LOOP;

-- Close the cursor

CLOSE members_cursor;

END;

/

```

Explanation:

Declares an explicit cursor, fetches each row into variables, and displays the details.

Lab 4: Create a cursor to retrieve books by a particular author and display their titles.

DECLARE

v_author_name books.author%TYPE := 'George Orwell'; -- Example author

-- Declare cursor

CURSOR books_cursor IS

SELECT title

FROM books

WHERE author = v_author_name;

-- Variable to hold book title

v_title books.title%TYPE;

BEGIN

OPEN books_cursor;

LOOP

FETCH books_cursor INTO v_title;

EXIT WHEN books_cursor%NOTFOUND;

DBMS_OUTPUT.PUT_LINE('Book Title: ' || v_title);

END LOOP;

```
CLOSE books_cursor;  
END;  
/
```

Explanation:

Fetches all books written by a specific author and displays their titles.

The cursor iterates through the filtered result set.

23.Rollback and Commit Savepoint

Q: Perform a transaction that includes inserting a new member, setting a SAVEPOINT, and rolling back to the savepoint after making updates.

-- Start the transaction

```
START TRANSACTION;
```

-- Insert a new member

```
INSERT INTO members (member_id, member_name, date_of_membership, email)
```

```
VALUES (101, 'John Doe', '2025-12-01', 'john.doe@example.com');
```

-- Set a savepoint

```
SAVEPOINT before_updates;
```

-- Perform some updates

UPDATE members

SET email = 'john.new@example.com'

WHERE member_id = 101;

-- Rollback to savepoint (undo the updates but keep the insertion)

ROLLBACK TO before_updates;

-- Commit the transaction (insertion is saved)

COMMIT;

Effect:

The new member is inserted permanently.

Any updates made after the savepoint are undone.

Q: Use COMMIT after successfully inserting multiple books into the books table, then use ROLLBACK to undo a set of changes made after a savepoint.

START TRANSACTION;

-- Insert multiple books

INSERT INTO books (book_id, title, author, publisher, year_of_publication, price)

VALUES

(601, 'Python Basics', 'Alice Smith', 'TechPub', 2023, 400.00),

(602, 'Advanced SQL', 'Bob Johnson', 'TechPub', 2022, 500.00);

-- Commit these insertions

COMMIT;

-- Set a savepoint before further changes

SAVEPOINT before_extra_changes;

-- Insert additional books

INSERT INTO books (book_id, title, author, publisher, year_of_publication, price)

VALUES (603, 'Data Analytics', 'Charlie Brown', 'TechPub', 2023, 600.00);

-- Rollback to savepoint (undo the last insertion)

ROLLBACK TO before_extra_changes;

-- Commit the remaining changes

COMMIT;

Effect:

Books 601 and 602 are saved permanently.

Book 603 is undone.