Introduction to DBMS

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.

Answer:

Query 1: Create Database

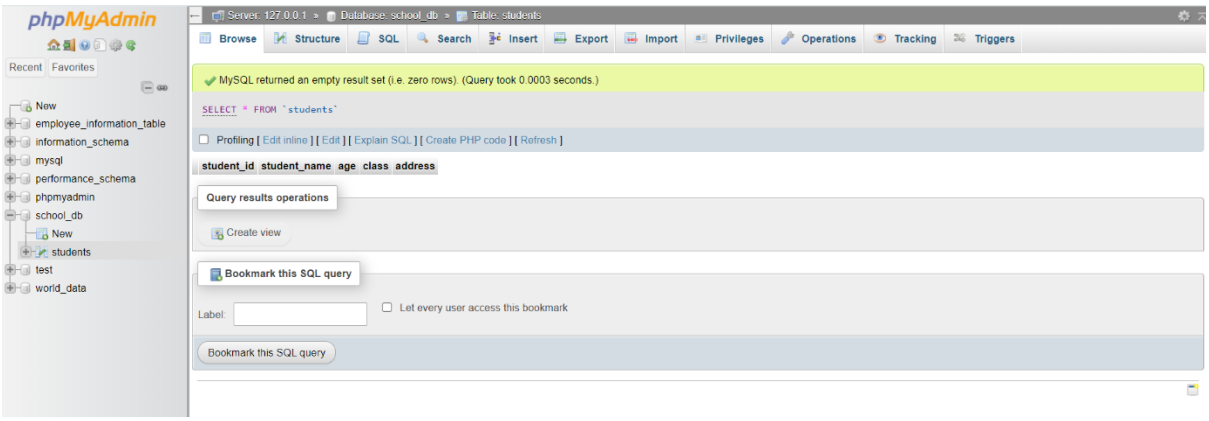
```
CREATE DATABASE school_db;
```

Query 2: Use Database

```
USE school_db;
```

Query 3: Create Table students

```
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(50),
    age INT,
    class VARCHAR(20),
    address VARCHAR(100)
);
```



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

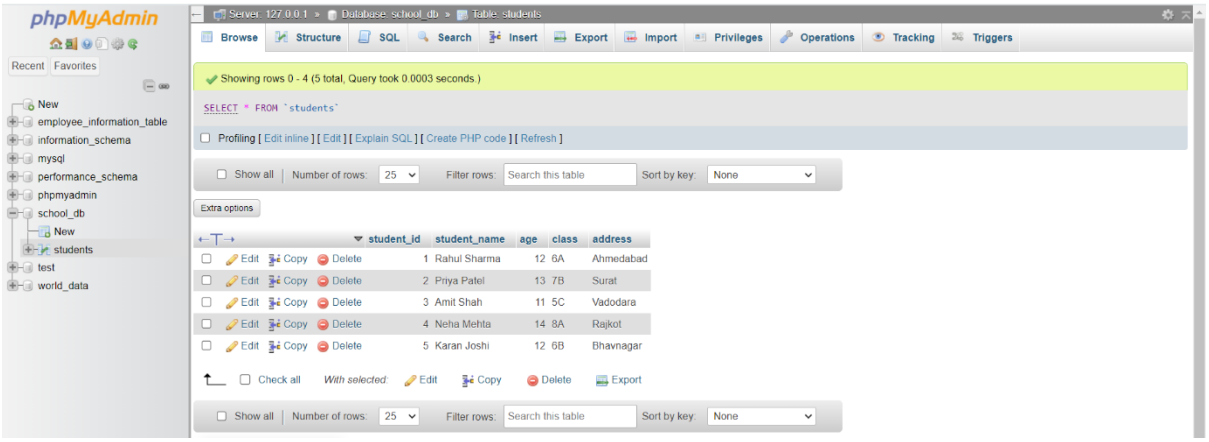
Answer:

Query 1: Insert 5 Records

```
INSERT INTO students (student_id, student_name, age, class, address) VALUES
(1, 'Rahul Sharma', 12, '6A', 'Ahmedabad'),
(2, 'Priya Patel', 13, '7B', 'Surat'),
(3, 'Amit Shah', 11, '5C', 'Vadodara'),
(4, 'Neha Mehta', 14, '8A', 'Rajkot'),
(5, 'Karan Joshi', 12, '6B', 'Bhavnagar');
```

Query 2: Retrieve All Records

```
SELECT * FROM students;
```



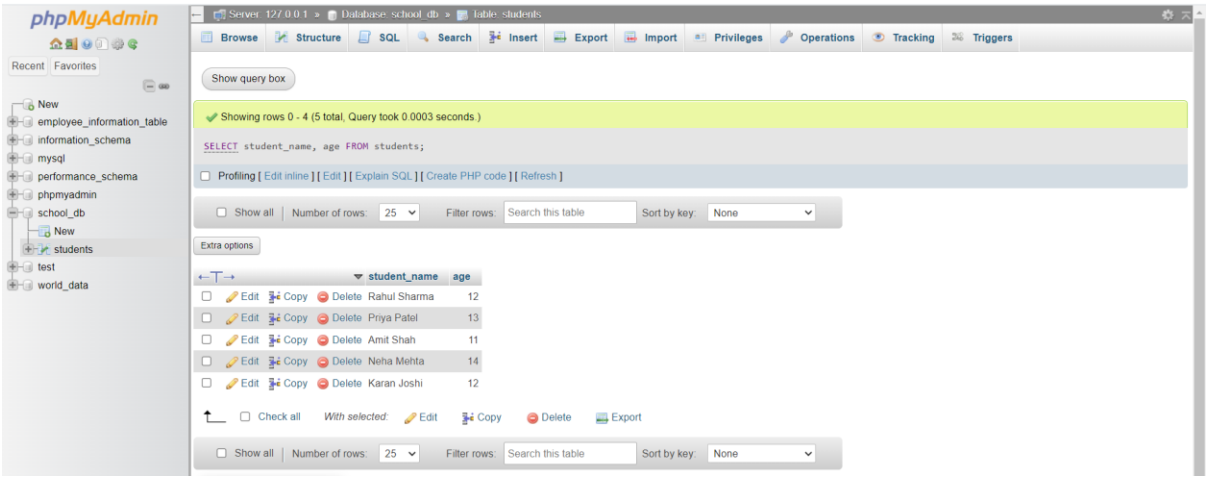
2. SQL Syntax

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

Answer:

Query: Retrieve Specific Columns (student_name and age)

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

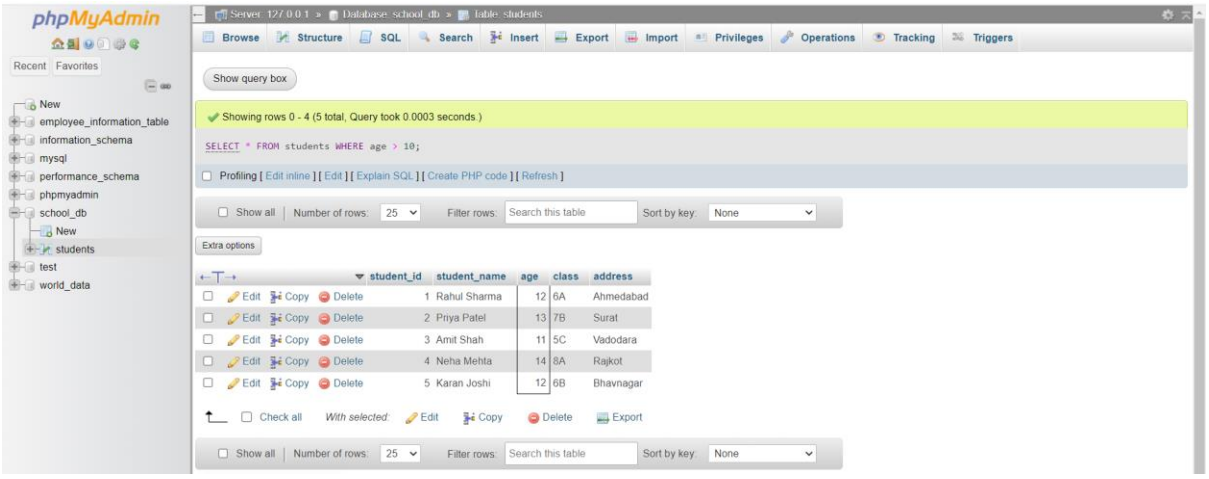


Lab 2: Write SQL queries to retrieve all students whose age is greater than 10.

Answer:

Query: Retrieve All Students Whose Age > 10

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



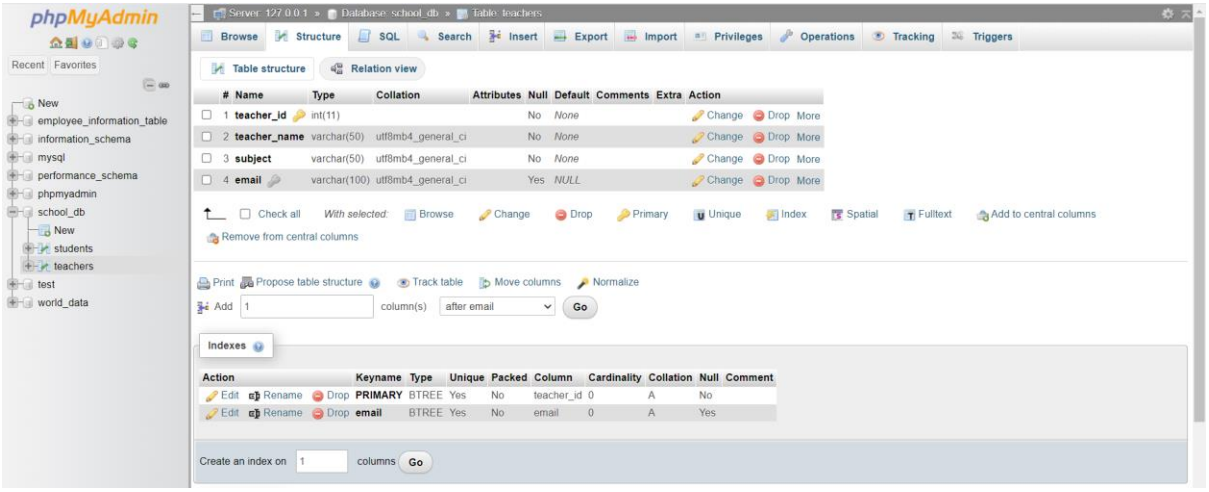
3. SQL Constraints

Lab 1: Create a table teachers with the following columns: teacher_id (Primary Key), teacher_name (NOT NULL), subject (NOT NULL), and email (UNIQUE).

Answer:

Query: Create teachers Table

```
CREATE TABLE teachers (
    teacher_id INT PRIMARY KEY,
    teacher_name VARCHAR(50) NOT NULL,
    subject VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE
);
```

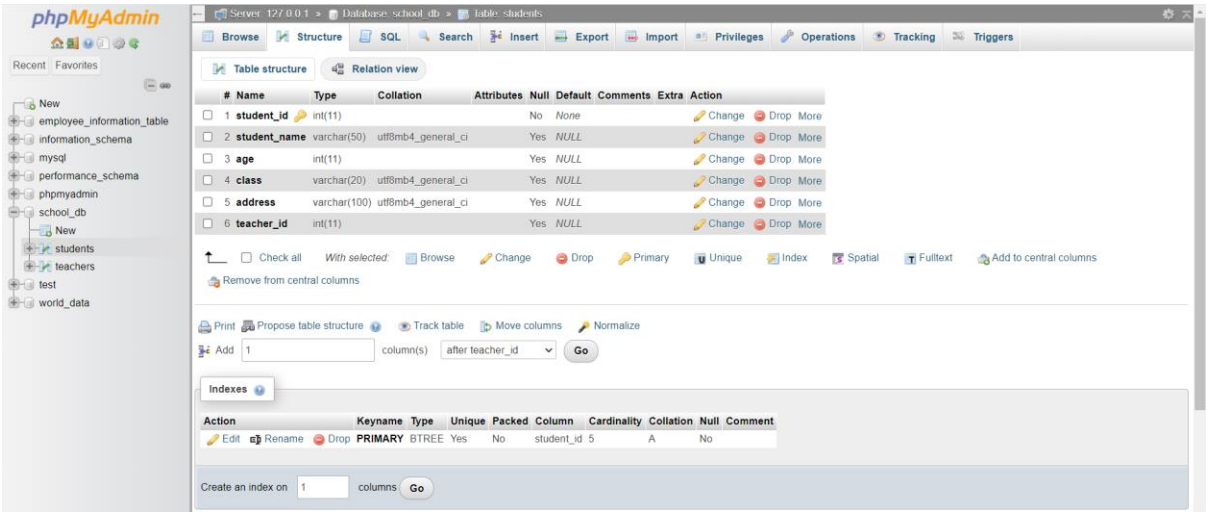


Lab 2: Implement a FOREIGN KEY constraint to relate the teacher_id from the teachers table with the students table.

Answer:

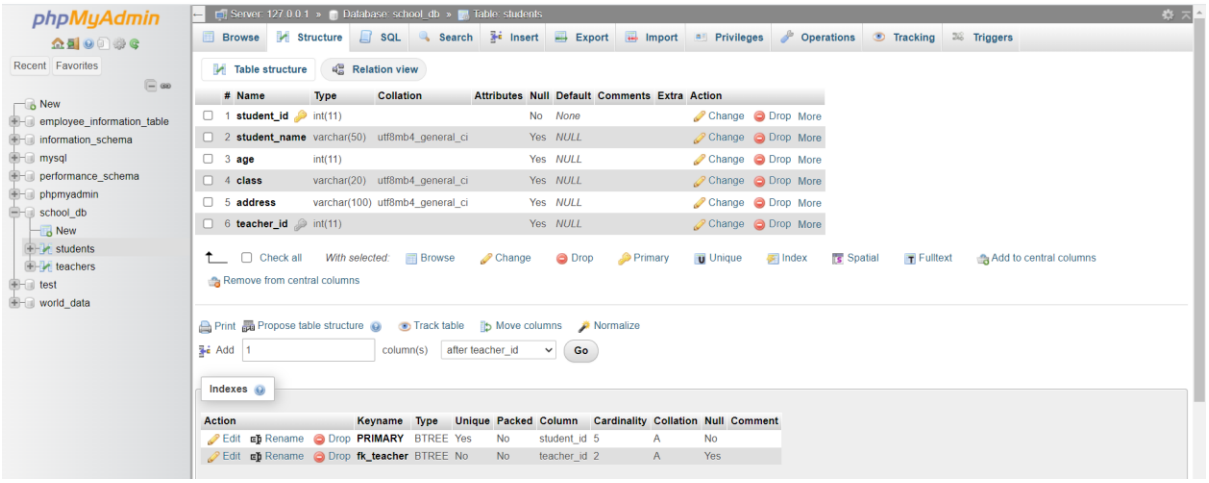
Query 1: Add teacher_id column to students table

```
ALTER TABLE students ADD teacher_id INT;
```



Query 2: Add FOREIGN KEY Constraint

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



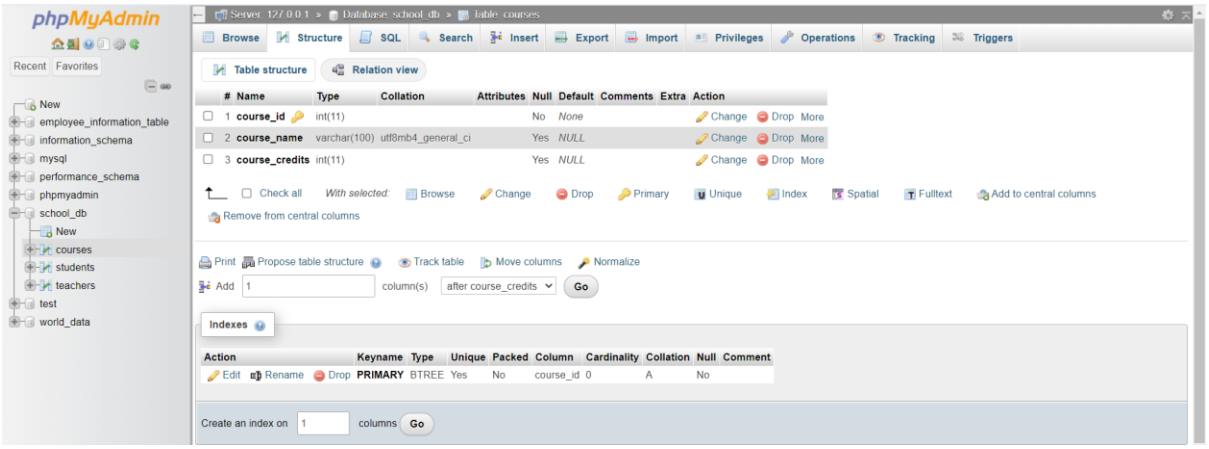
4. Main SQL Commands and Sub-commands (DDL)

Lab 1: Create a table courses with columns: course_id, course_name, and course_credits. Set the course_id as the primary key.

Answer:

Query: Create a table courses

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

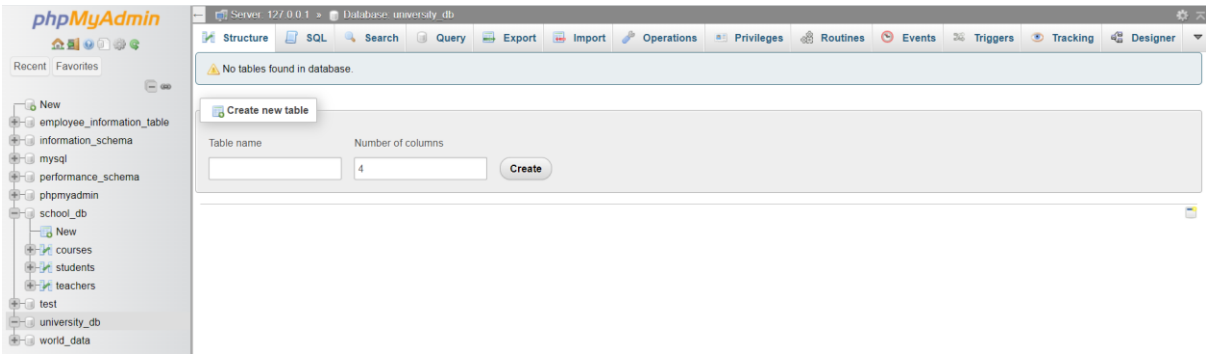


Lab 2: Use the CREATE command to create a database university_db.

Answer:

Query: Create a Database university_db

```
CREATE DATABASE university_db;
```



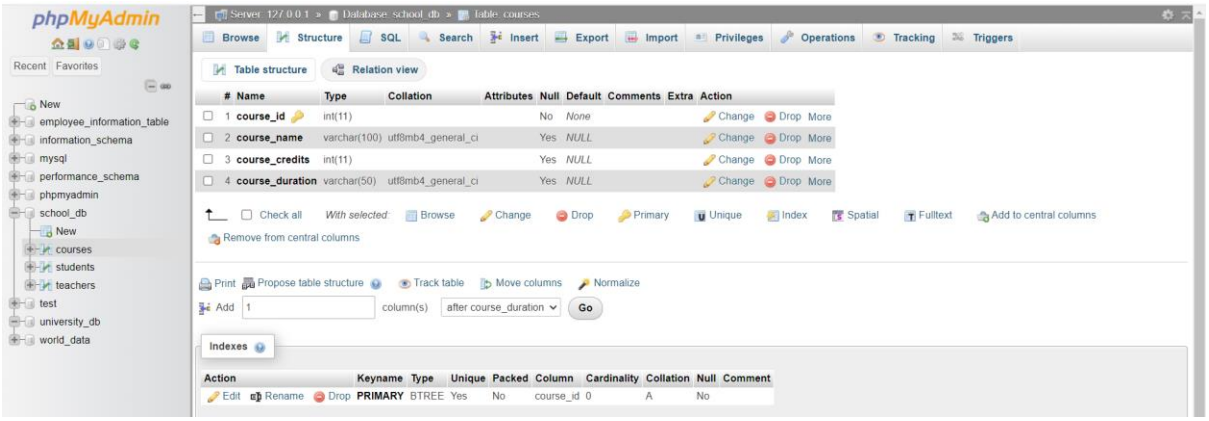
5. ALTER Command

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

Answer:

Query: Add a New Column (course_duration) to courses Table

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

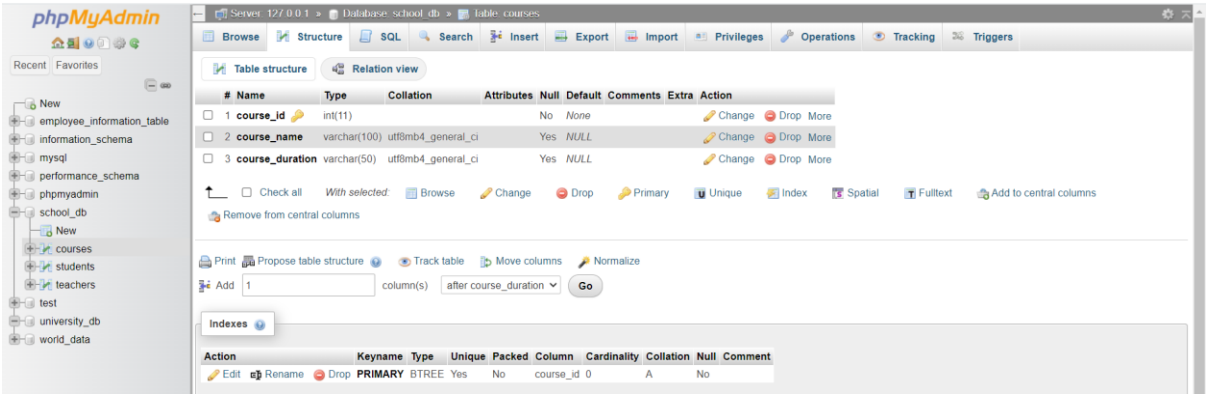


Lab 2: Drop the course_credits column from the courses table.

Answer:

Query: Drop the Column (course_credits) from courses Table

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



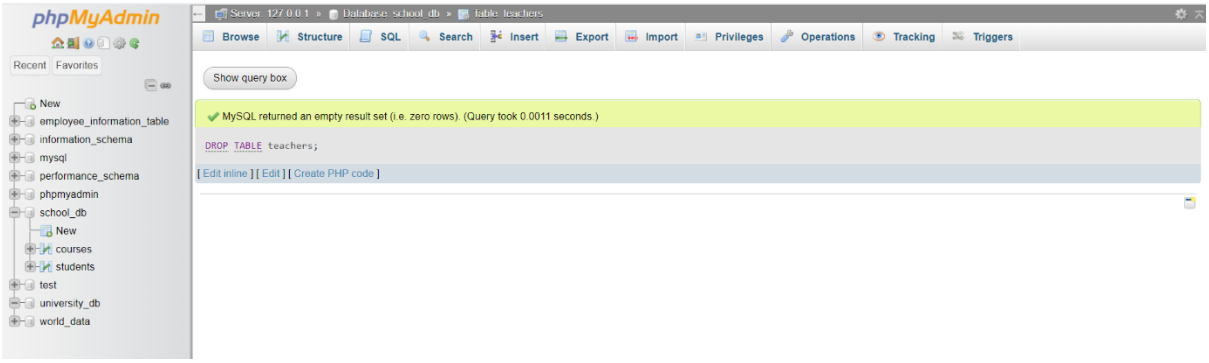
6. DROP Command

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

Answer:

Query: Drop the teachers table

```
DROP TABLE teachers;
```

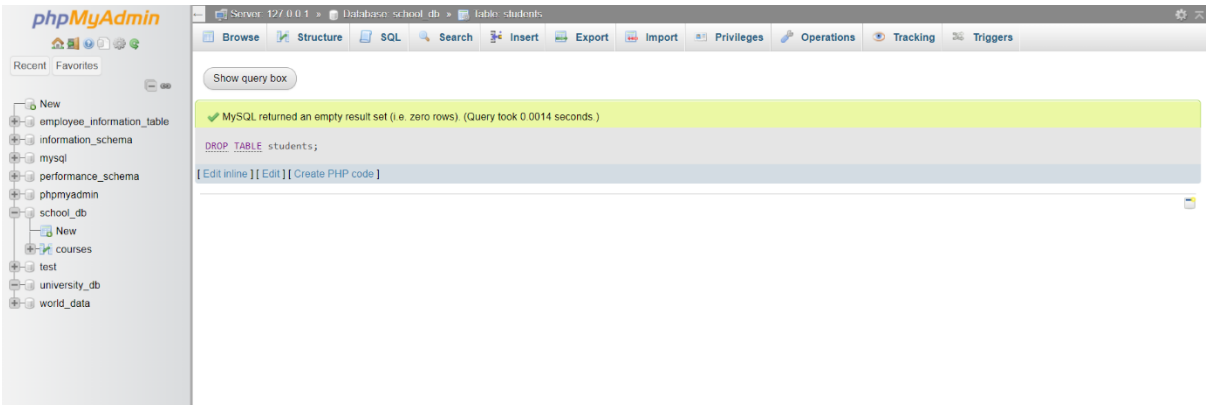


Lab 2: Drop the students table from the school_db database and verify that the table has been removed.

Answer:

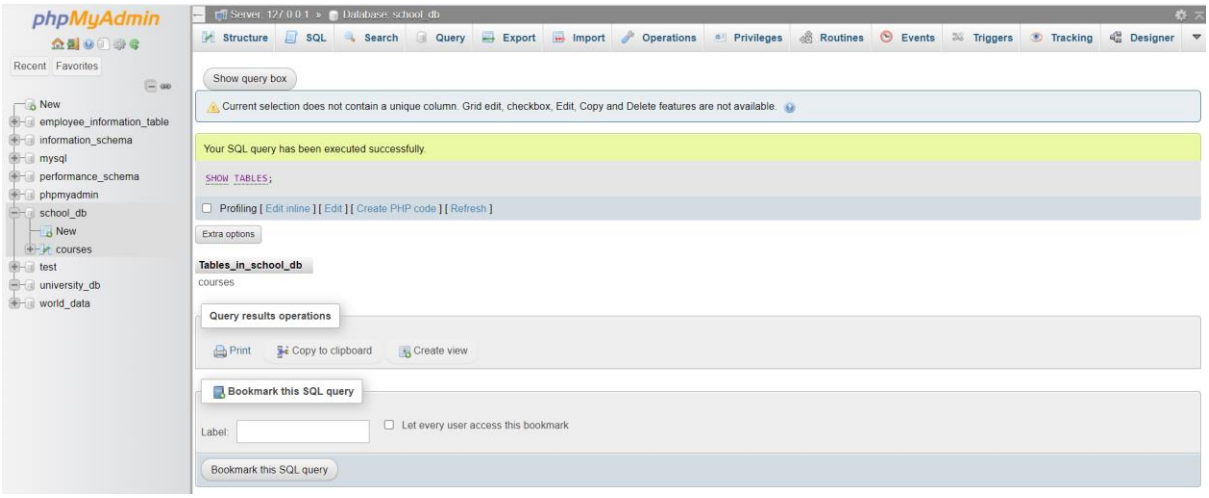
Query 1: Drop the students table and verify removal

```
DROP TABLE students;
```



Query 2: Verify that the table is removed

SHOW TABLES;



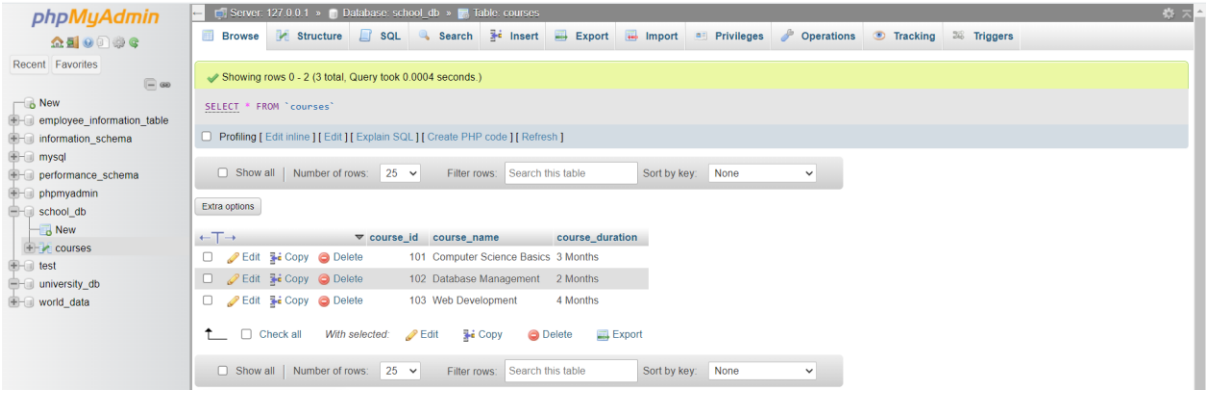
7. Data Manipulation Language (DML)

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

Answer:

Query: Insert 3 Records into courses Table

```
INSERT INTO courses (course_id, course_name, course_duration)
VALUES
(101, 'Computer Science Basics', '3 Months'),
(102, 'Database Management', '2 Months'),
(103, 'Web Development', '4 Months');
```



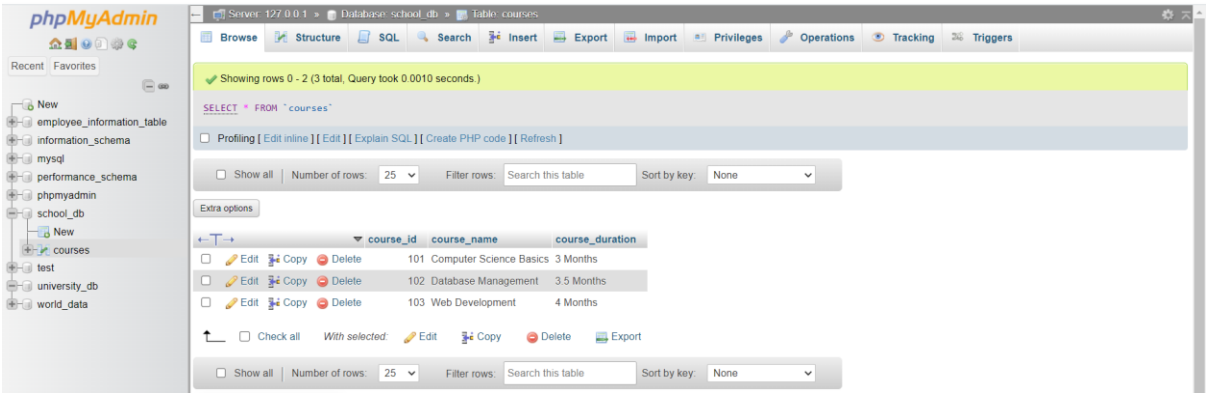
Lab 2: Update the course duration of a specific course using the UPDATE command.

Answer:

Query: Update Course Duration of a Specific Course

Example: Update duration of course_id = 102

```
UPDATE courses SET course_duration = '3.5 Months' WHERE course_id = 102;
```



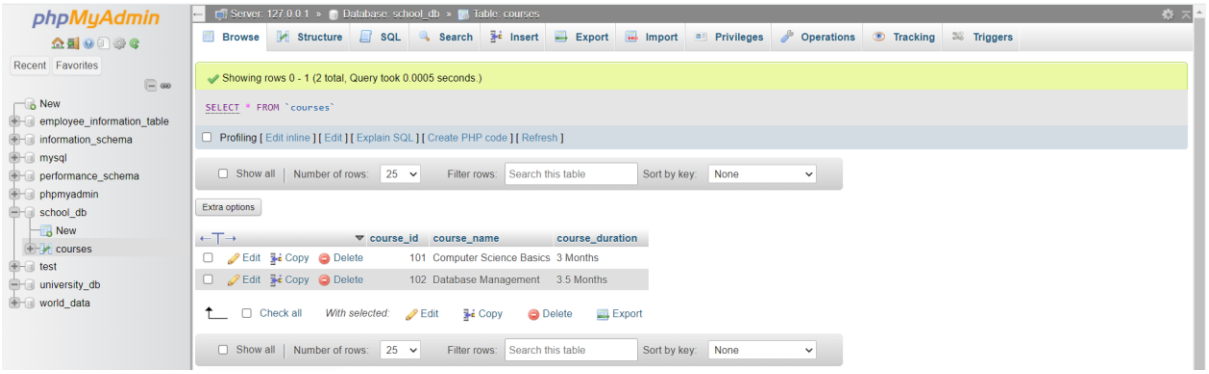
Lab 3: Delete a course with a specific course_id from the courses table using the DELETE command.

Answer:

Query: Delete a course using DELETE Command

Example: Delete course with course_id = 103

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



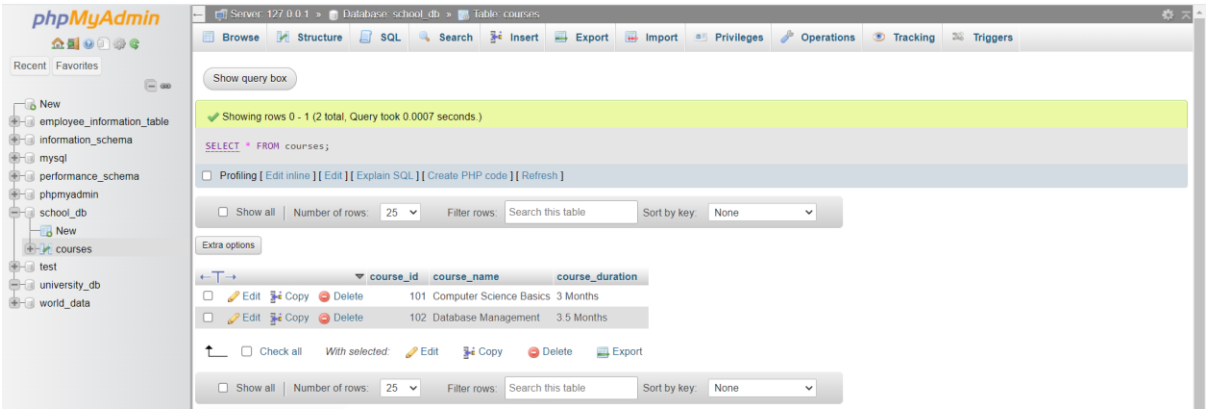
8. Data Query Language (DQL)

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

Answer:

Query: Retrieve all courses

```
SELECT * FROM courses;
```

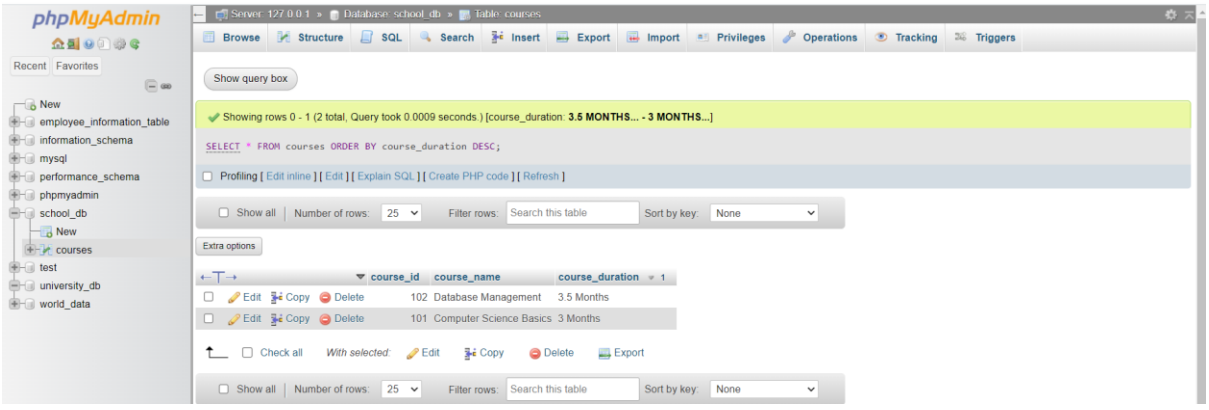


Lab 2: Sort the courses based on course_duration in descending order using ORDER BY.

Answer:

Query: Sort Courses by course_duration (Descending Order)

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

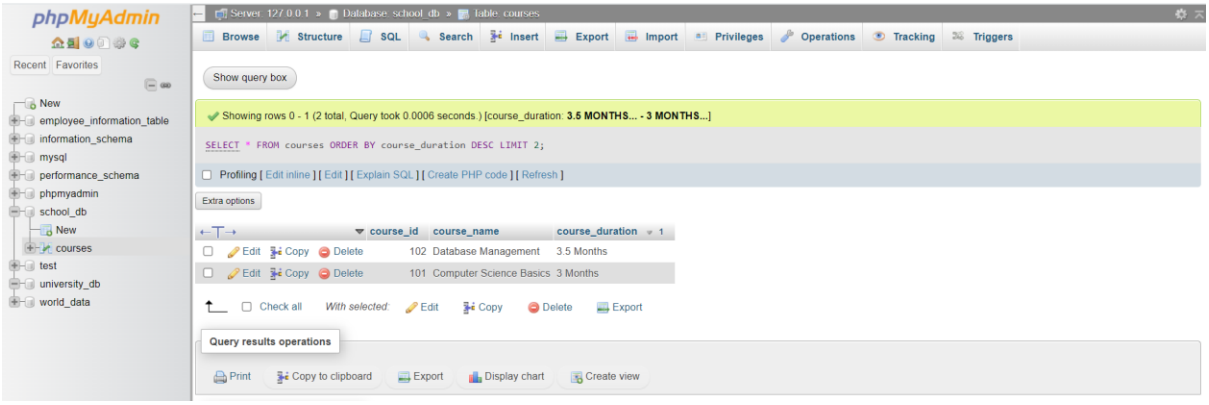


Lab 3: Limit the results of the SELECT query to show only the top two courses using LIMIT.

Answer:

Query: Show only top 2 courses using LIMIT

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

Answer:

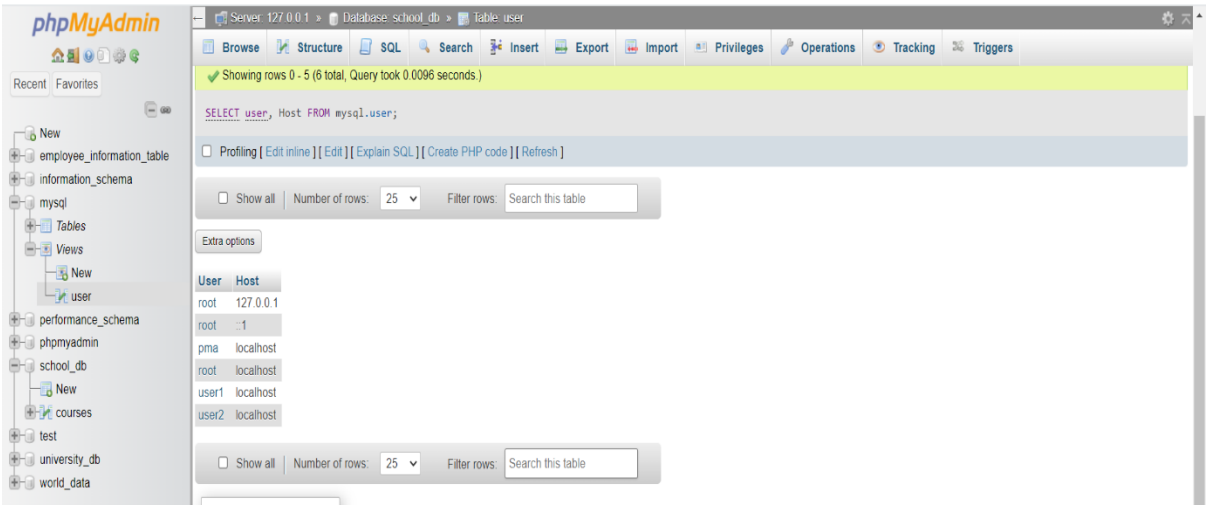
Query 1: Create two new users user1 and user2

Step 1: Create user1

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

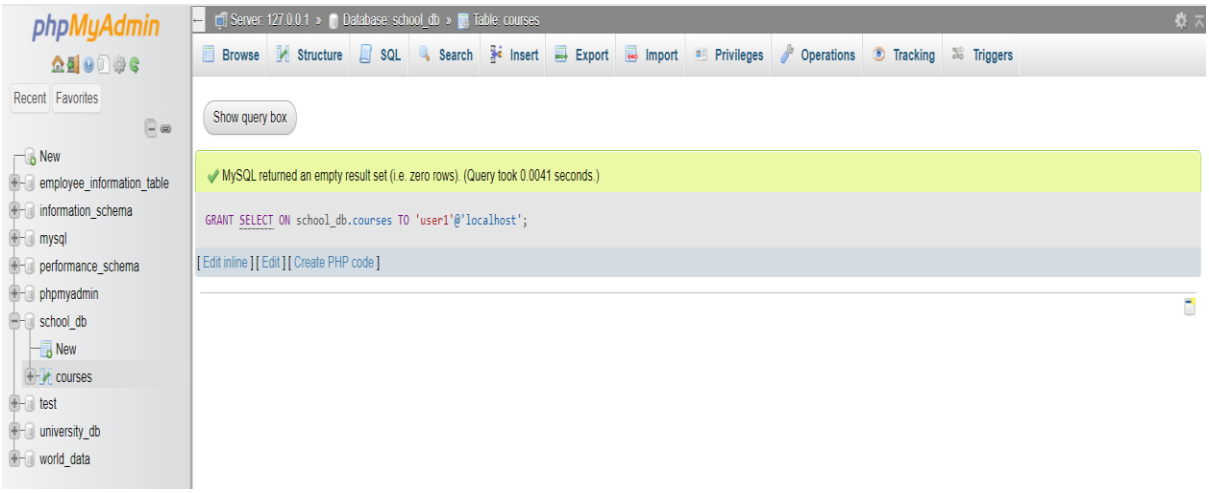
Step 2: Create user2

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



Query 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.

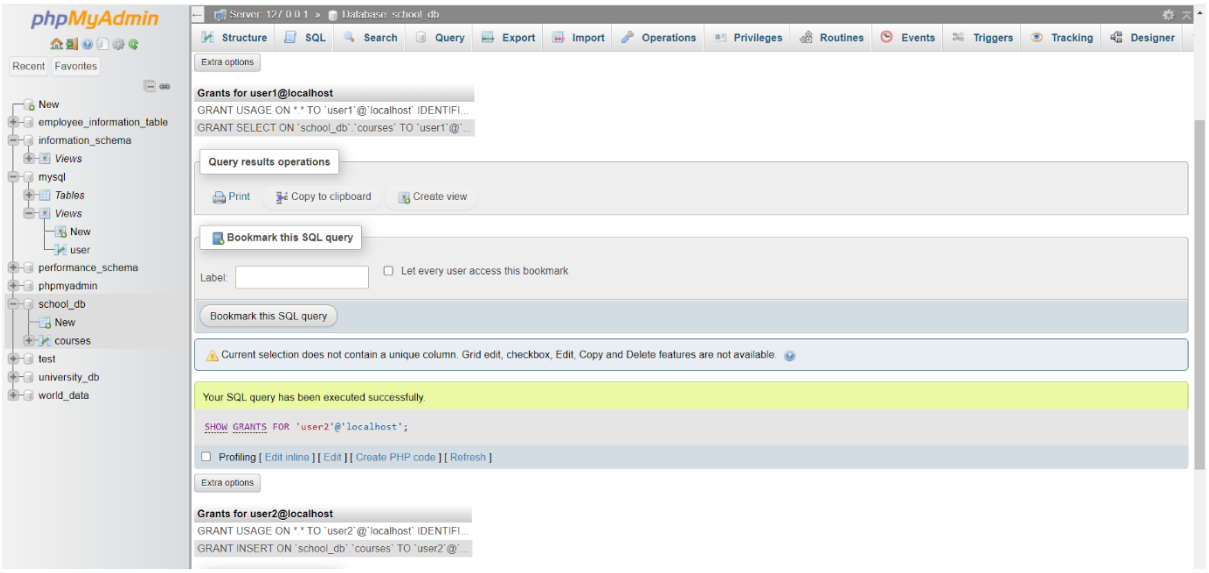
Answer:

Query 1: Revoke INSERT permission from user1

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

Query 2: Grant INSERT permission to user2

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



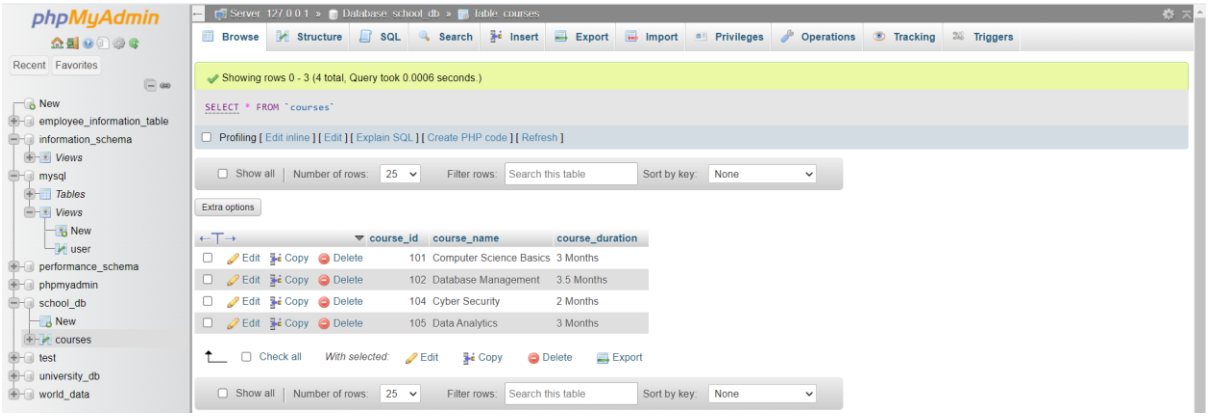
10. Transaction Control Language (TCL)

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

Answer:

Query: Insert a few rows + COMMIT

```
START TRANSACTION;

INSERT INTO courses (course_id, course_name, course_duration)
VALUES
(104, 'Cyber Security', '2 Months'),
(105, 'Data Analytics', '3 Months');

COMMIT;
```



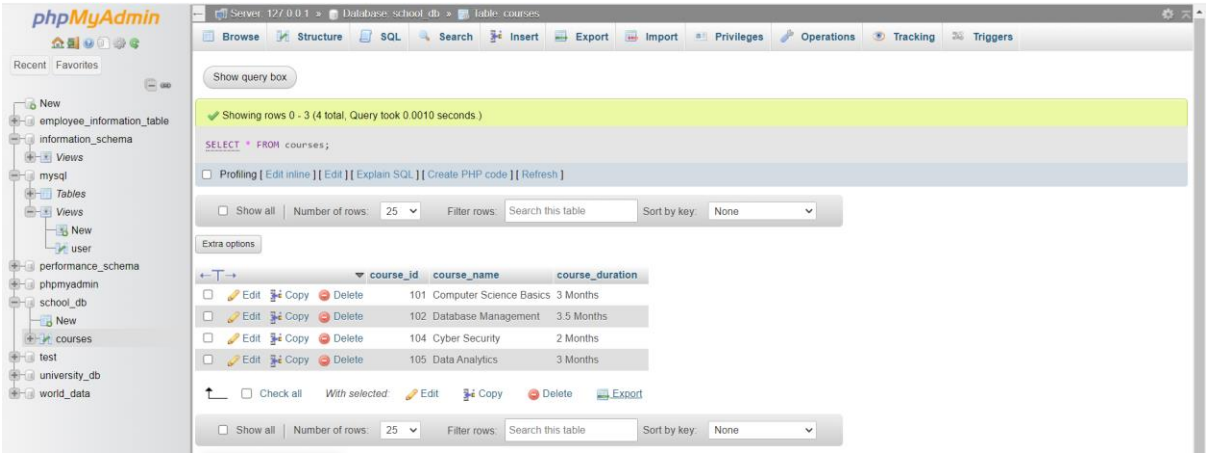
- Rows **104 and 105** will appear permanently.

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

Answer:

Query: Insert additional rows + ROLLBACK

```
START TRANSACTION;

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (106, 'Cloud Computing', '5 Months');

ROLLBACK;
```



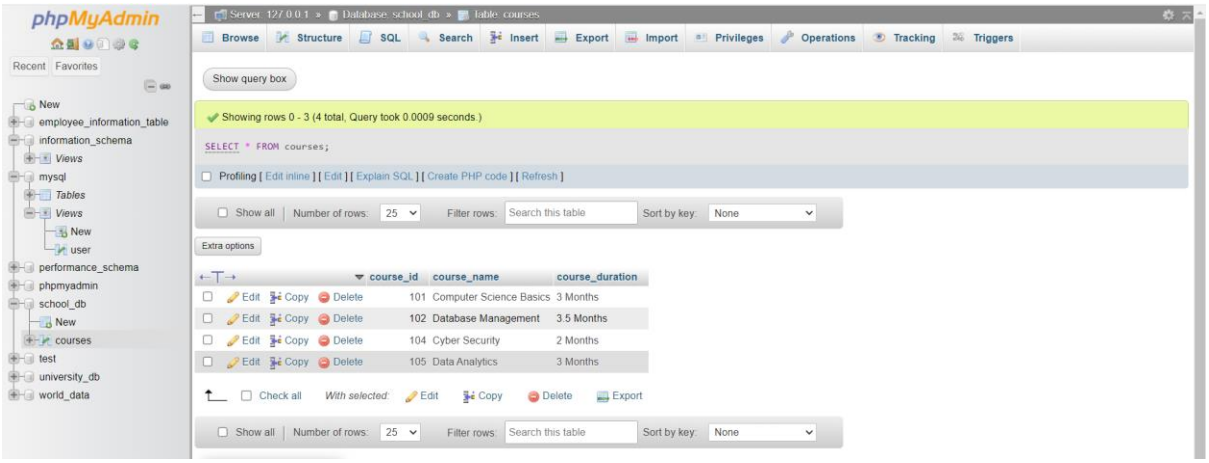
- **Course_id 106** will **NOT appear** because rollback cancelled the insert.

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

Answer:

Query: SAVEPOINT + ROLLBACK TO SAVEPOINT

```
START TRANSACTION;

-- Create savepoint
SAVEPOINT before_update;

-- Update a course
UPDATE courses SET course_duration = '6 Months' WHERE course_id = 101;

-- Rollback only this update
ROLLBACK TO before_update;

-- Commit remaining changes
COMMIT;
```



- Course_id 101 duration will remain ‘3 Months’ (rollback undo the update)

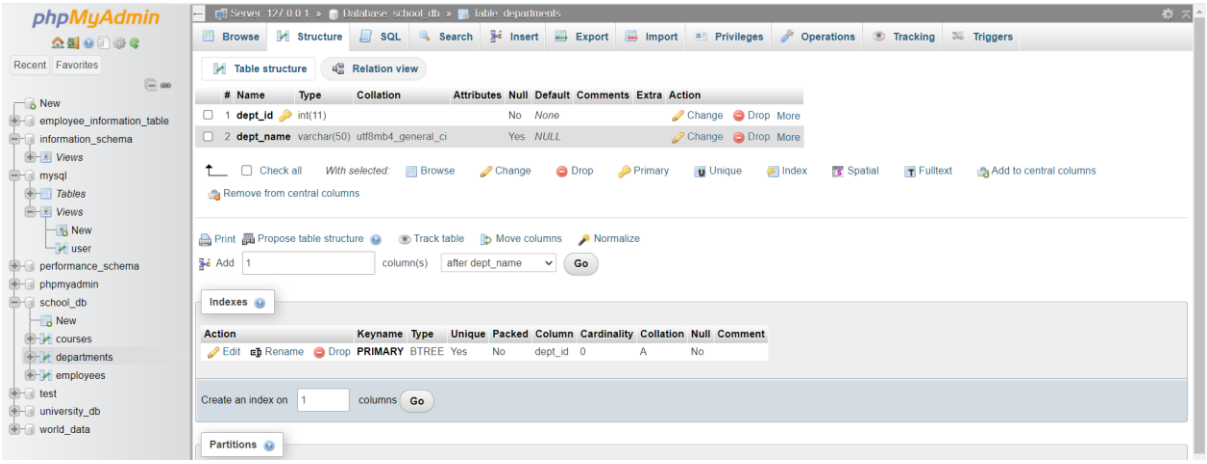
11. SQL Joins

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

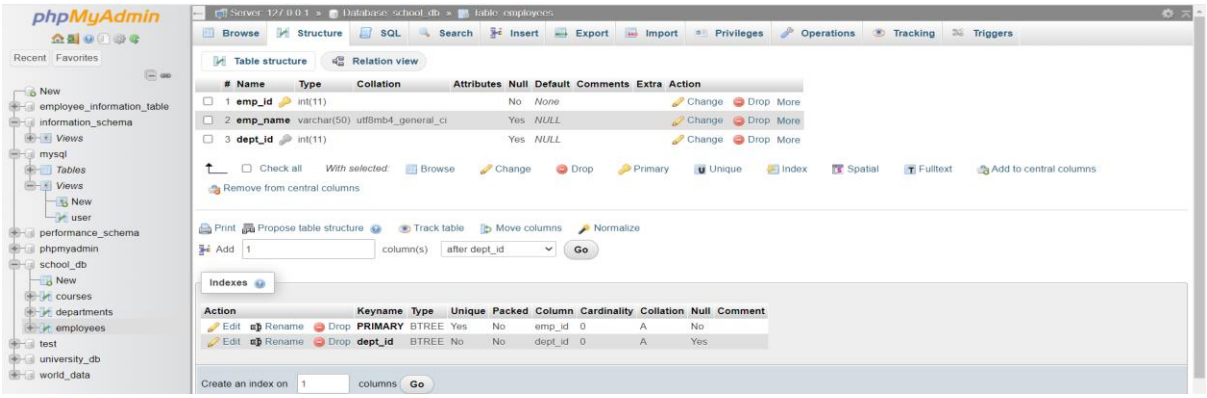
Answer:

Query 1: Create two tables

```
1. CREATE TABLE departments (
       dept_id INT PRIMARY KEY,
       dept_name VARCHAR(50)
   );
```




```
2. CREATE TABLE employees (
       emp_id INT PRIMARY KEY,
       emp_name VARCHAR(50),
       dept_id INT,
       FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
   );
```



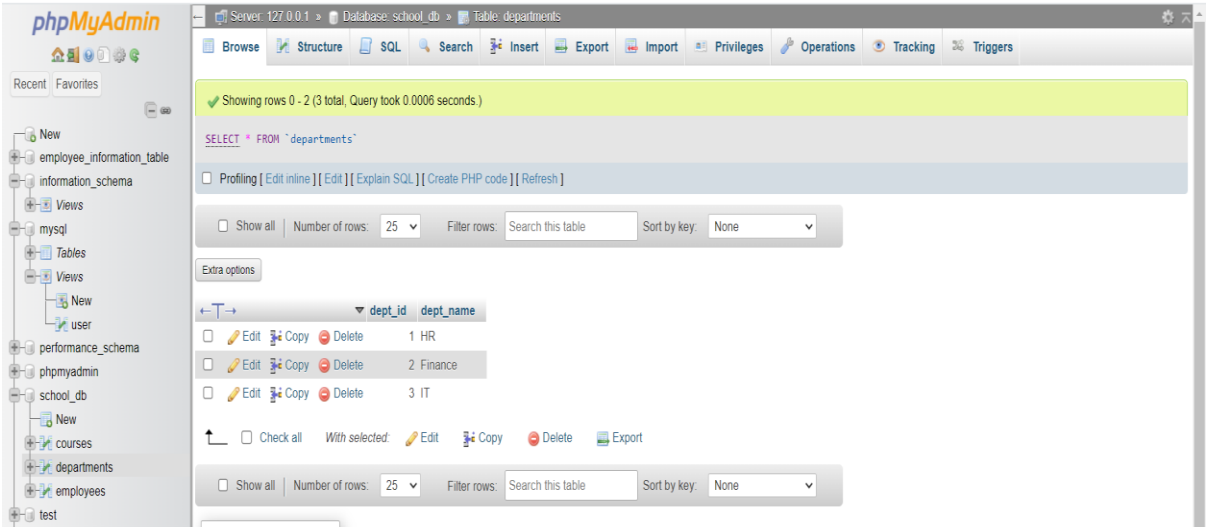
Query 2: Insert sample records

1. INSERT INTO departments VALUES

   (1, 'HR'),

   (2, 'Finance'),

   (3, 'IT');

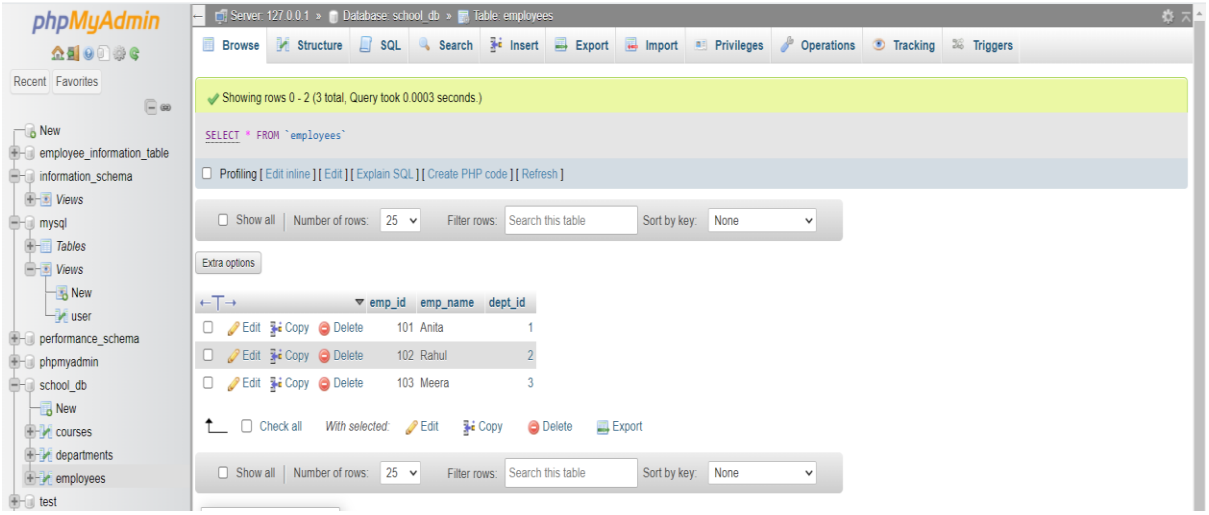


2. INSERT INTO employees VALUES

   (101, 'Anita', 1),

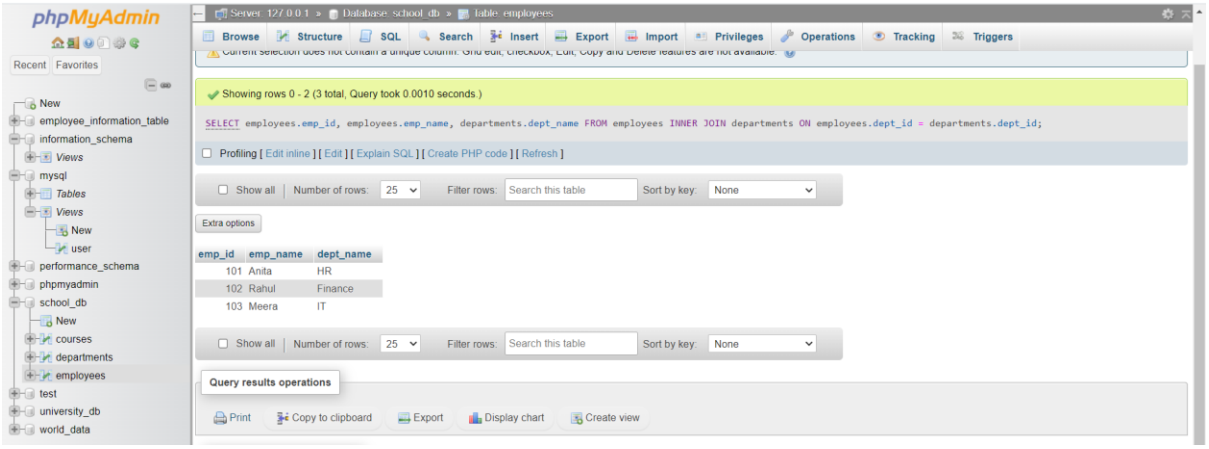
   (102, 'Rahul', 2),

   (103, 'Meera', 3);



Query 3: Perform INNER JOIN

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



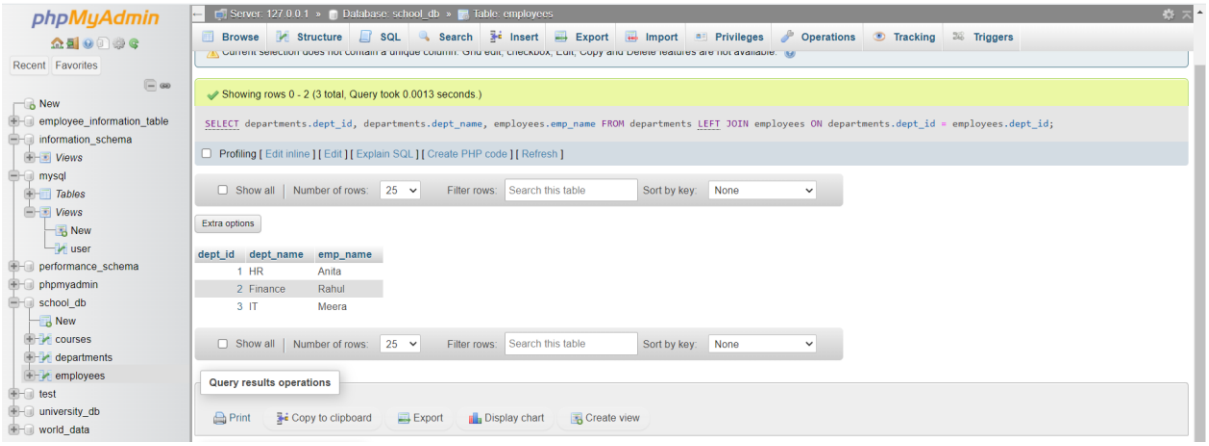
- INNER JOIN shows **only matching** employee – department records.

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

Answer:

Query: LEFT JOIN (Show all departments even without employees)

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



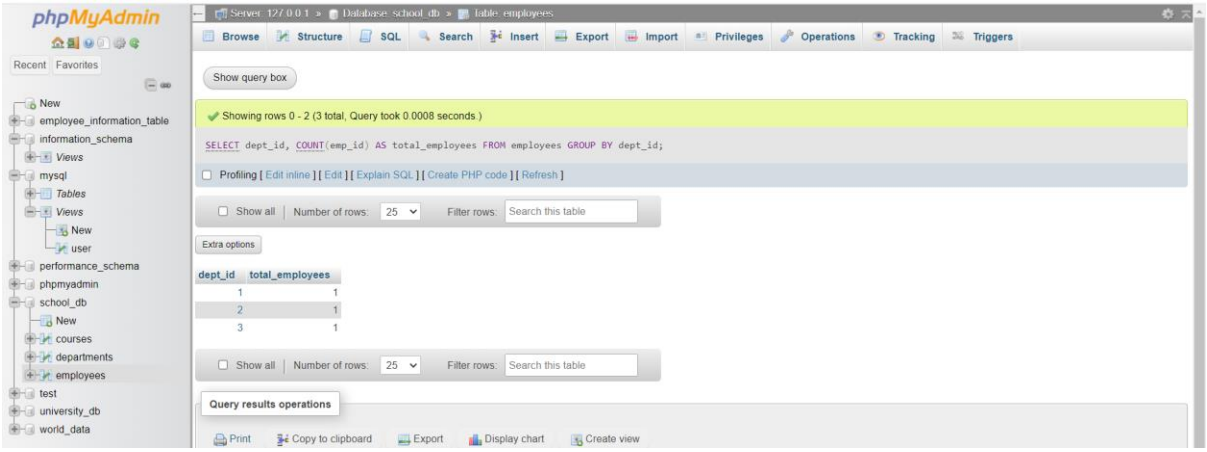
12. SQL Group By

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

Answer:

Query: GROUP BY – Count Employees in Each Department

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



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

Answer:

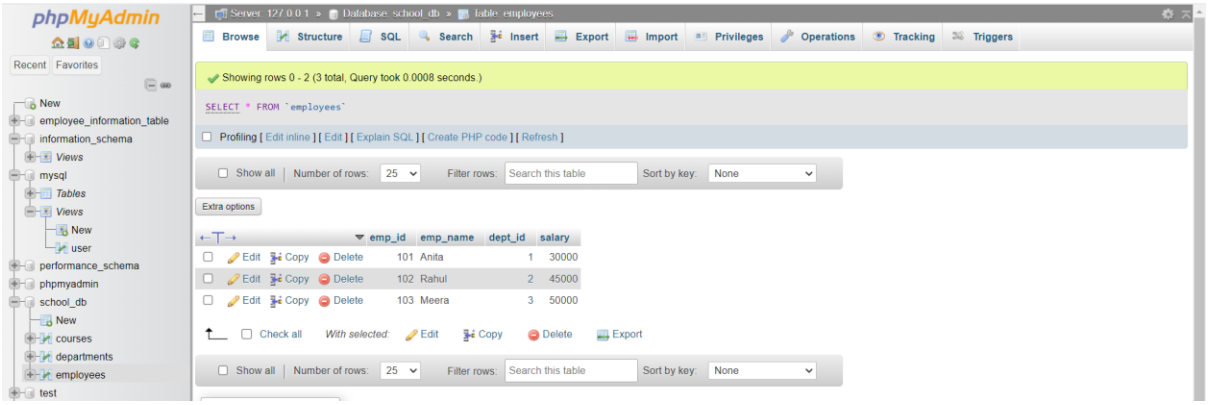
Important note:

- **To use Lab 2, your employees table must have a salary column, e.g.:**

  ```
  ALTER TABLE employees ADD salary INT;
  ```

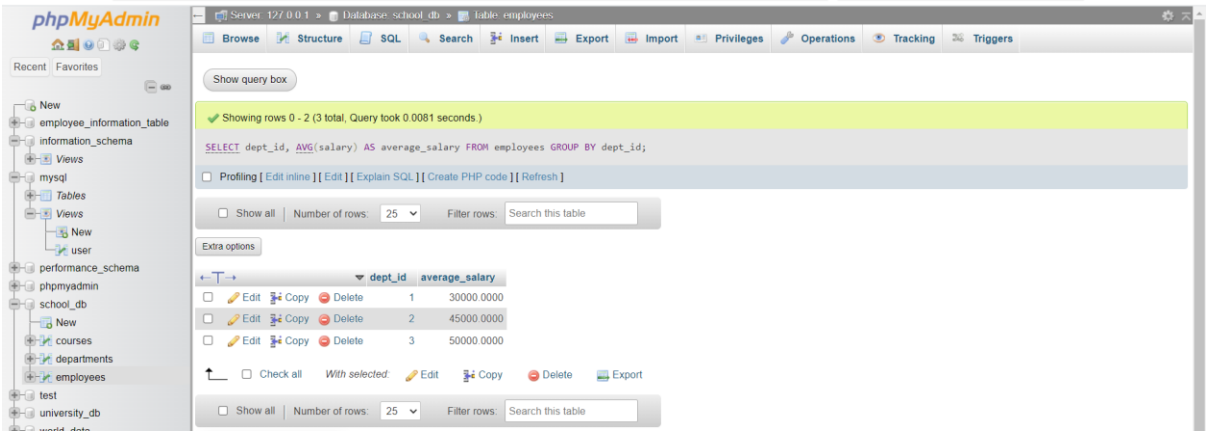
- **And insert/update salaries:**

  ```
  UPDATE employees SET salary = 30000 WHERE emp_id = 101;
  UPDATE employees SET salary = 45000 WHERE emp_id = 102;
  UPDATE employees SET salary = 50000 WHERE emp_id = 103;
  ```



Query: Find Average Salary per Department (Using AVG)

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



13. SQL Stored Procedure

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

Answer:

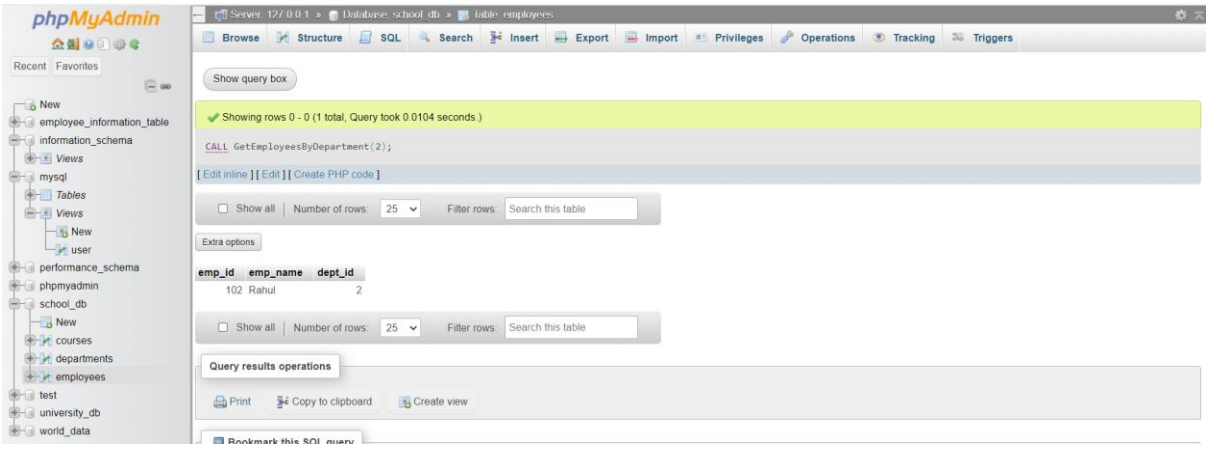
Query: Stored Procedure to Retrieve Employees by Department

- **Stored Procedure**

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

- **How to Execute the Procedure**

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



- This procedure returns employees who work in a specific department.

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

Answer:

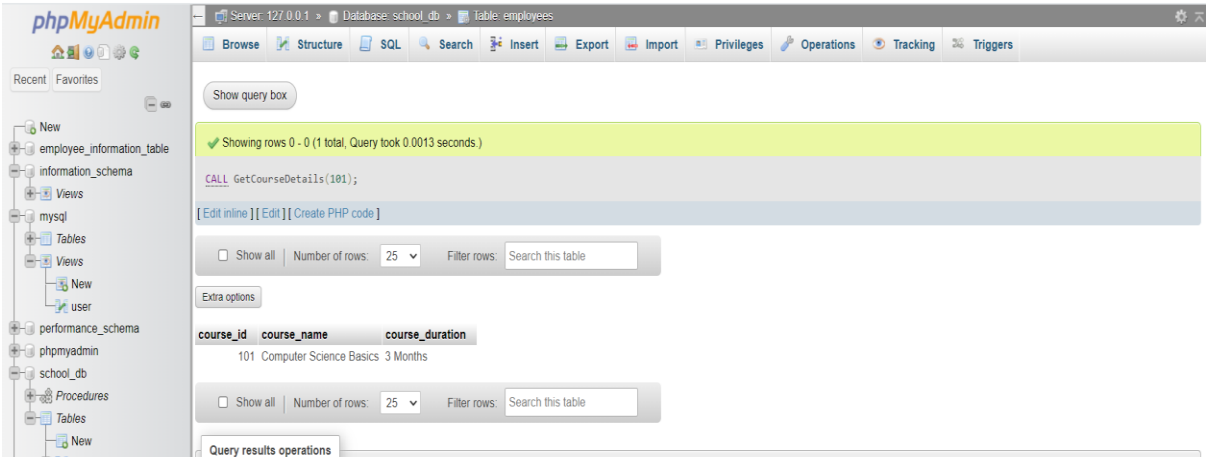
Query: Stored Procedure to Retrieve Course Details by course_id

- **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 Execute the Procedure**

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



- This procedure returns only the details of the course whose ID you pass.

14. SQL View

Lab 1: Create a view to show all employees along with their department names.
Answer:

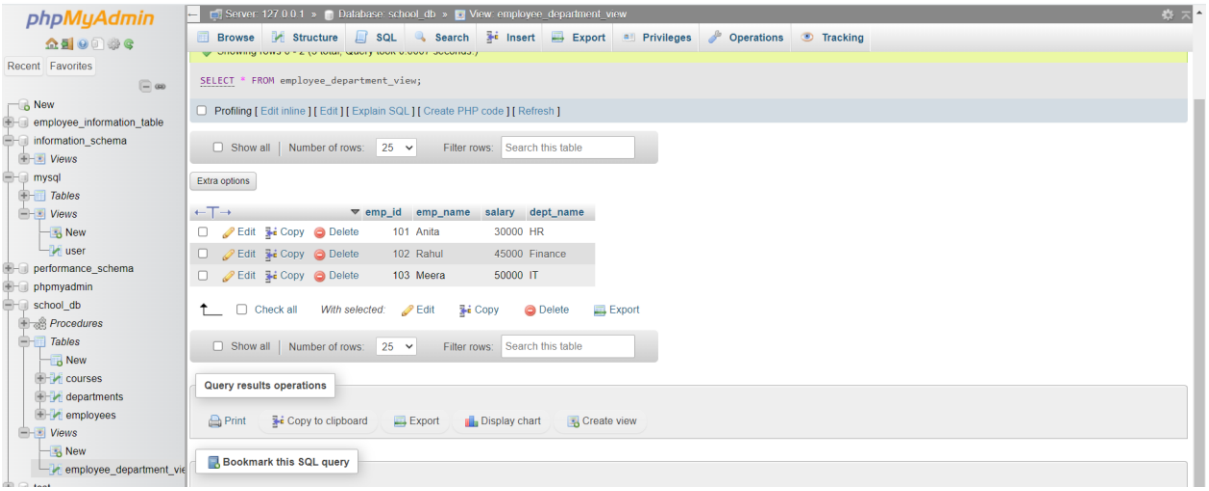
Query: Create a View to Show Employees with Department Names

- **Step 1: Create a VIEW using JOIN**

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

- **Step 2: To see the view**

```
SELECT * FROM employee_department_view;
```



Lab 2: Modify the view to exclude employees whose salaries are below $50,000.
Answer:

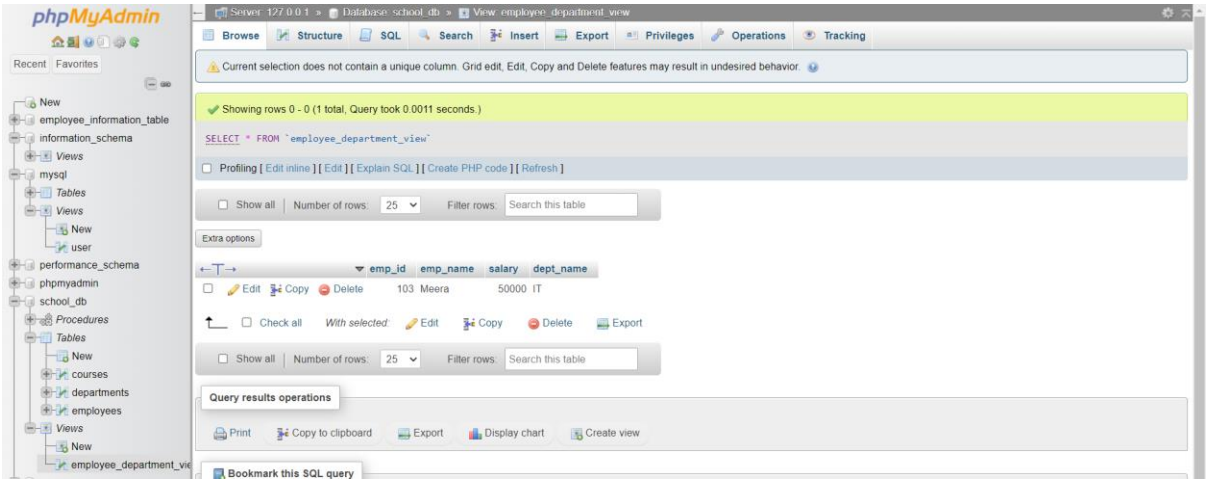
Query: Modify the View to Exclude Employees with Salary < 50,000

- **Use OR REPLACE to update the view**

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

- **To check updated view**

```
SELECT * FROM employee_department_view;
```



15. SQL Triggers

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

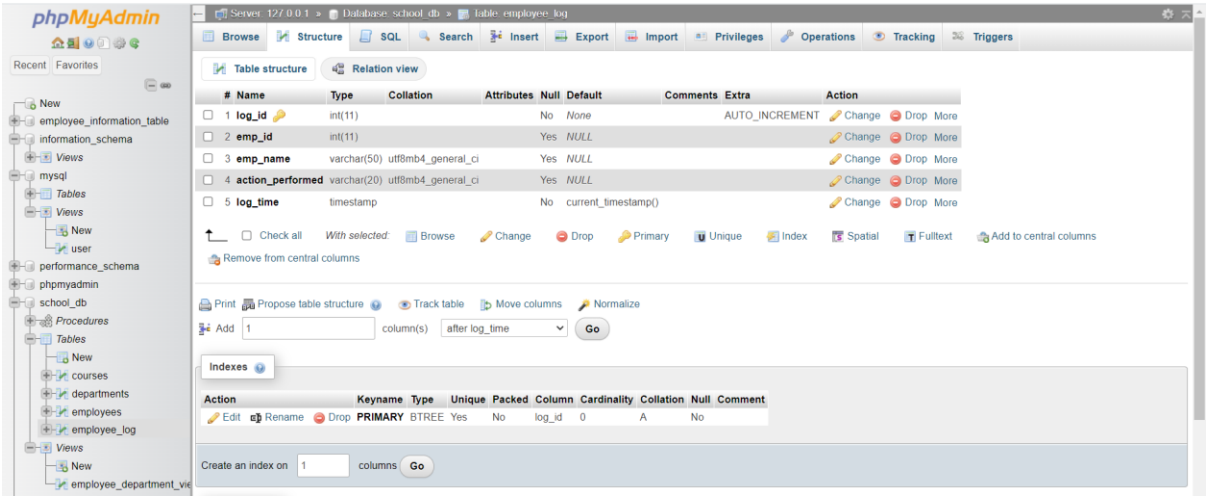
Answer:

Query: Trigger to Log New Employee Insertions

- **Step 1: Create a log table**

  You need a table to store log records.

```
CREATE TABLE employee_log (
    log_id INT AUTO_INCREMENT PRIMARY KEY,
    emp_id INT,
    emp_name VARCHAR(50),
    action_performed VARCHAR(20),
    log_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```



- **Step 2: Create the trigger**

  This trigger will run AFTER a new employee is inserted.

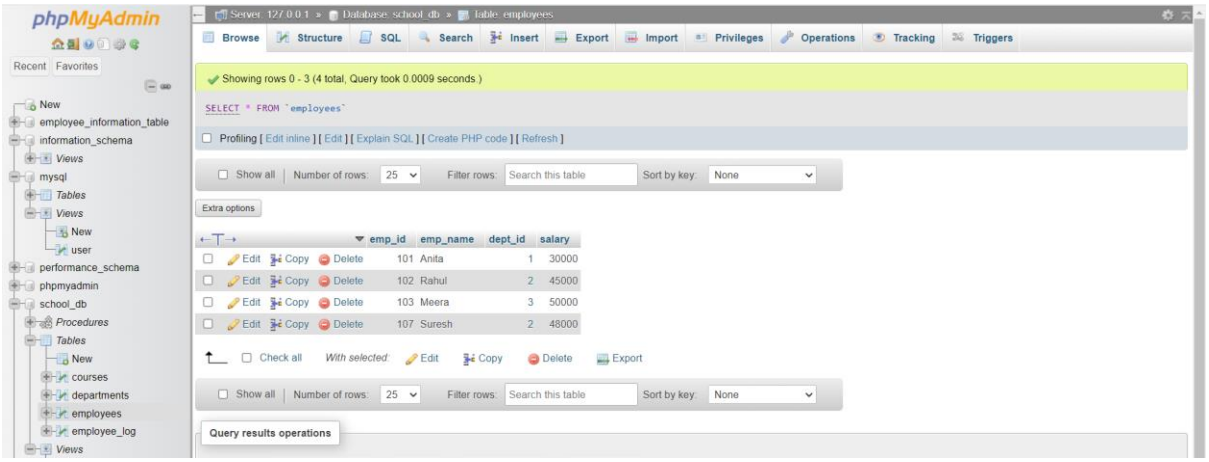
```
DELIMITER $$
CREATE TRIGGER log_new_employee
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
```

```
    INSERT INTO employee_log (emp_id, emp_name, action_performed)
    VALUES (NEW.emp_id, NEW.emp_name, 'INSERT');
END $$
DELIMITER ;
```

- **Sample Output**

(After inserting a new employee)

```
INSERT INTO employees (emp_id, emp_name, dept_id, salary)
VALUES (107, 'Suresh', 2, 48000);
```



- Running:

```
SELECT * FROM employee_log;
```

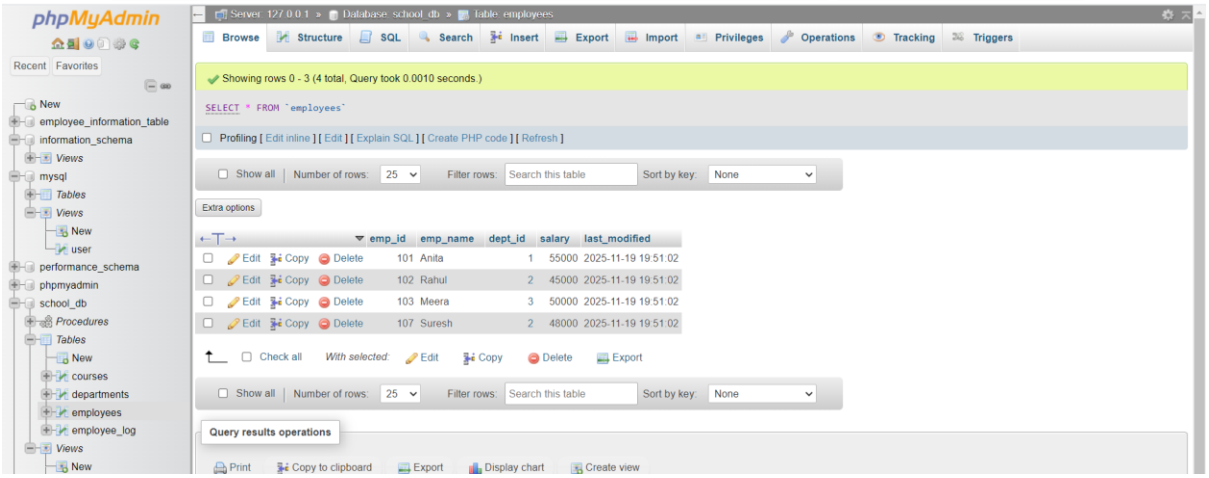
Lab 2: Create a trigger to update the last_modified timestamp whenever an employee record is updated.

Answer:

Query: Trigger to Automatically Update last_modified Timestamp

- **Step 1: Add a column to employees table**

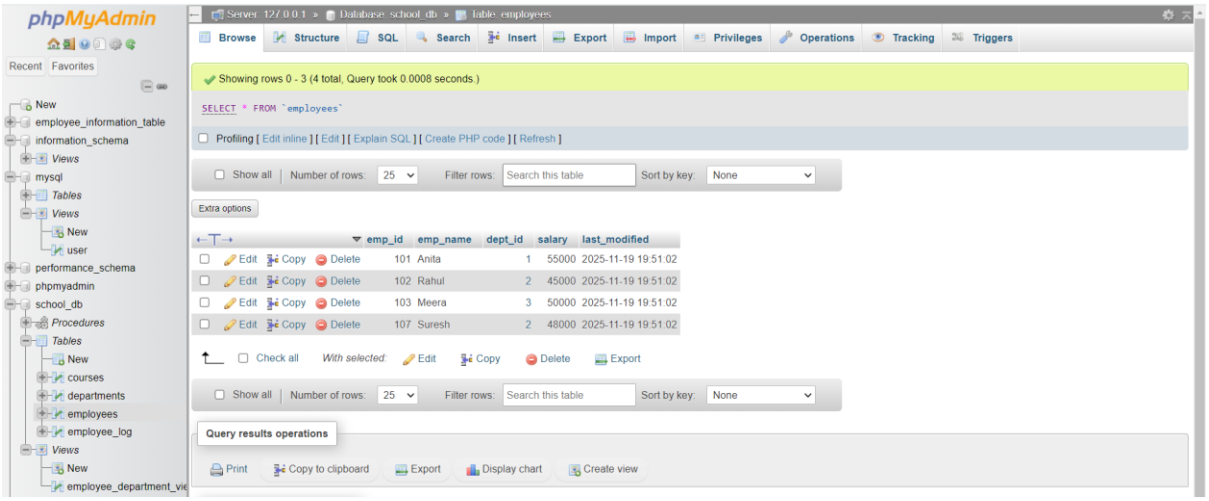
```
ALTER TABLE employees
ADD last_modified TIMESTAMP DEFAULT CURRENT_TIMESTAMP;
```



- **Step 2: Create the UPDATE trigger**

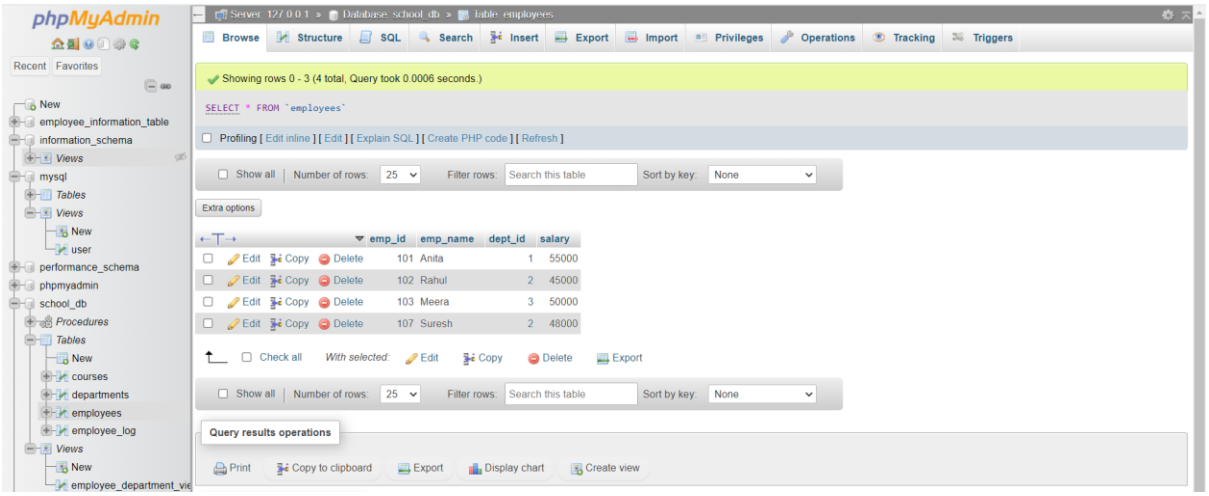
This trigger updates the timestamp whenever an employee record changes.

```
DELIMITER $$

CREATE TRIGGER update_last_modified

BEFORE UPDATE ON employees

FOR EACH ROW

BEGIN

   SET NEW.last_modified = CURRENT_TIMESTAMP;

END $$

DELIMITER ;
```



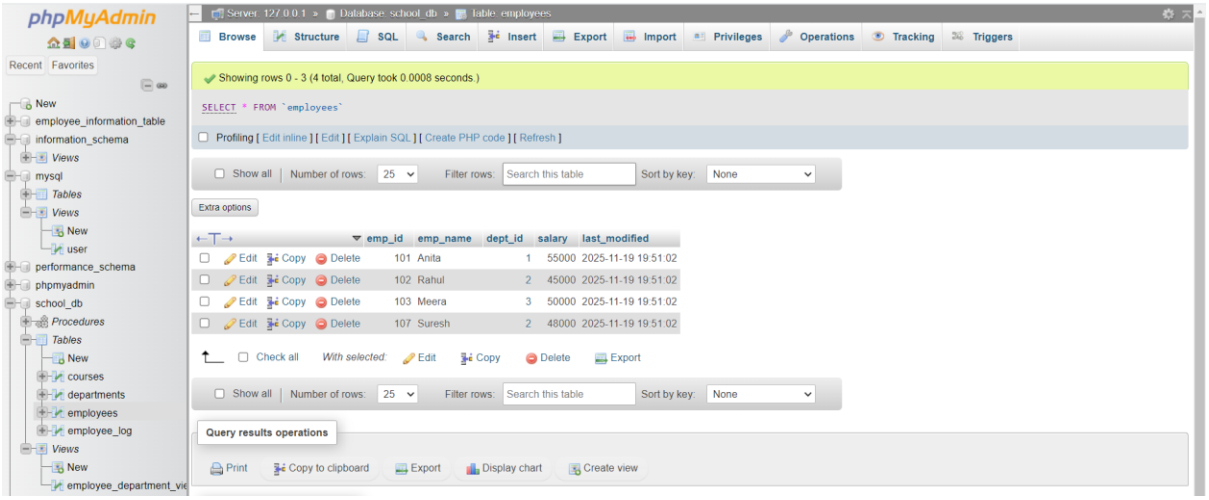
- **Example Update**

```
UPDATE employees
SET salary = 55000
WHERE emp_id = 101;
```



- **Output (View the updated timestamp)**

```
SELECT emp_id, emp_name, salary, last_modified FROM employees;
```



- Automatically updated — no need to manually set last_modified.

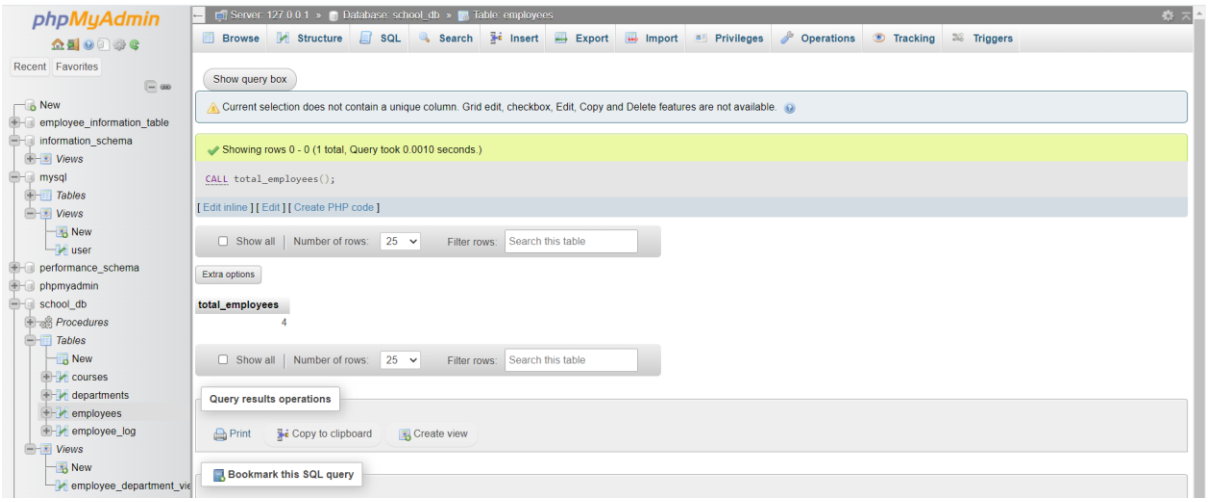
16. Introduction to PL/SQL

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

Answer:

Query: Print total number of employees

```
SELECT COUNT(*) AS total_employees FROM employees;
```

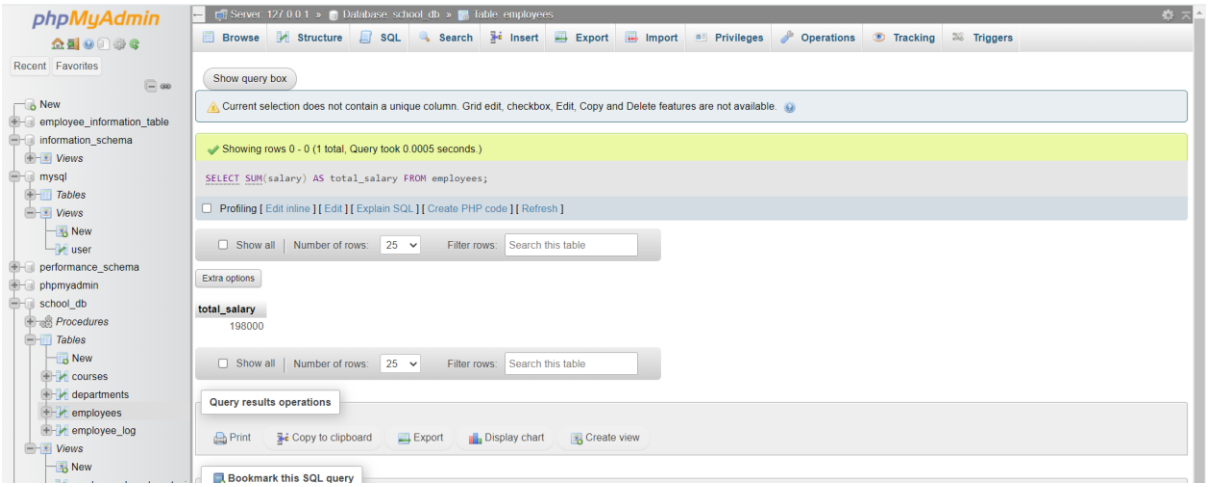


Lab 2: Create a PL/SQL block that calculates the total salary from an orders table.

Answer:

Query: Calculate Total Salary

```
SELECT SUM(salary) AS total_salary FROM employees;
```



17. PL/SQL Control Structures

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

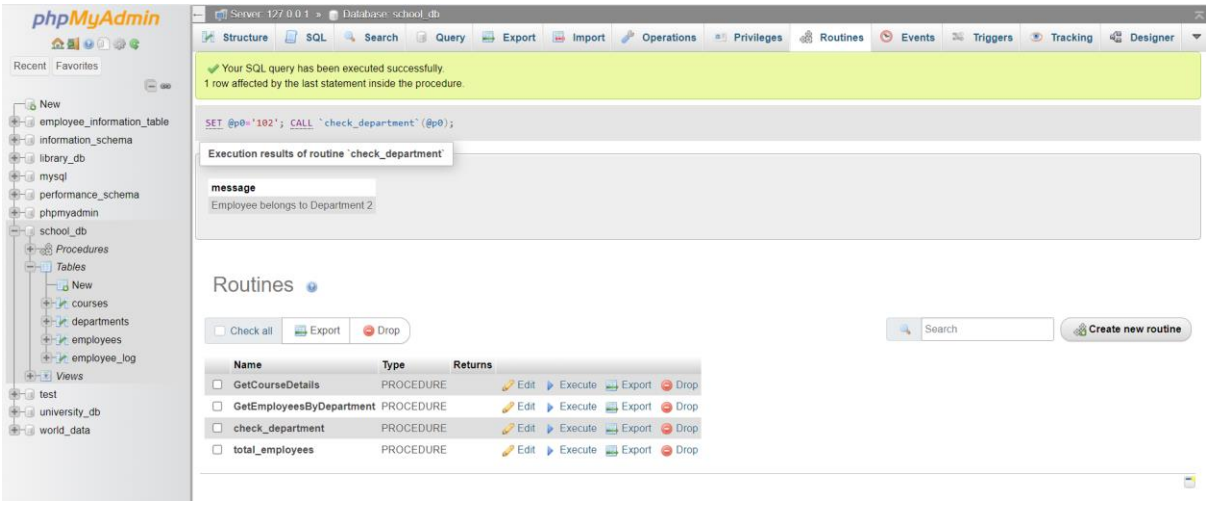
Answer:

Query 1: Stored Procedure

```
DELIMITER $$

CREATE PROCEDURE check_department(IN p_emp_id INT)
BEGIN
   DECLARE v_dept INT;

   SELECT dept_id INTO v_dept
   FROM employees
   WHERE emp_id = p_emp_id;

   IF v_dept = 2 THEN
      SELECT 'Employee belongs to Department 2' AS message;
   ELSE
      SELECT 'Employee does NOT belong to Department 2' AS message;
   END IF;
END $$

DELIMITER ;
```

Query 2: Call the Procedure

```
CALL check_department(102);
```



Lab 2: Use a FOR LOOP to iterate through employee records and display their names.

Answer:

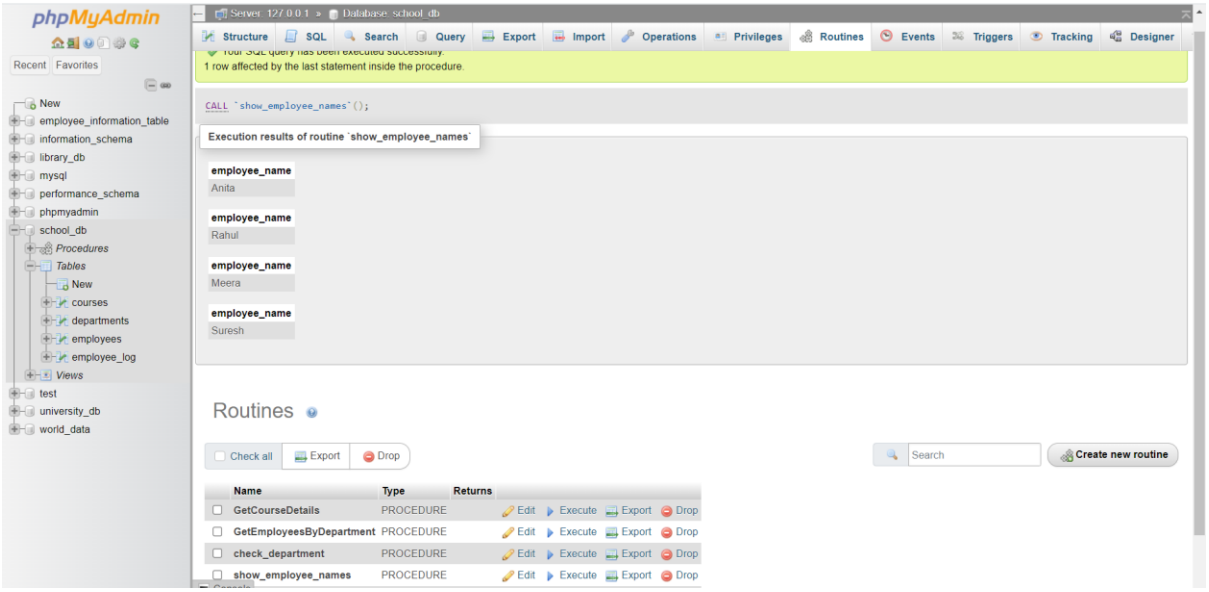
Query 1: Stored Procedure with FOR-LIKE LOOP (MySQL uses WHILE)

```
DELIMITER $$

CREATE PROCEDURE show_employee_names()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE v_name VARCHAR(50);

    DECLARE emp_cursor CURSOR FOR
        SELECT emp_name FROM employees;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    OPEN emp_cursor;

    read_loop: LOOP
        FETCH emp_cursor INTO v_name;
        IF done = 1 THEN
```

```
            LEAVE read_loop;
        END IF;

        SELECT v_name AS employee_name;
    END LOOP;

    CLOSE emp_cursor;
END $$

DELIMITER ;
```

Query 2: Call the procedure

```
CALL show_employee_names();
```



18. SQL Cursors

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

Answer:

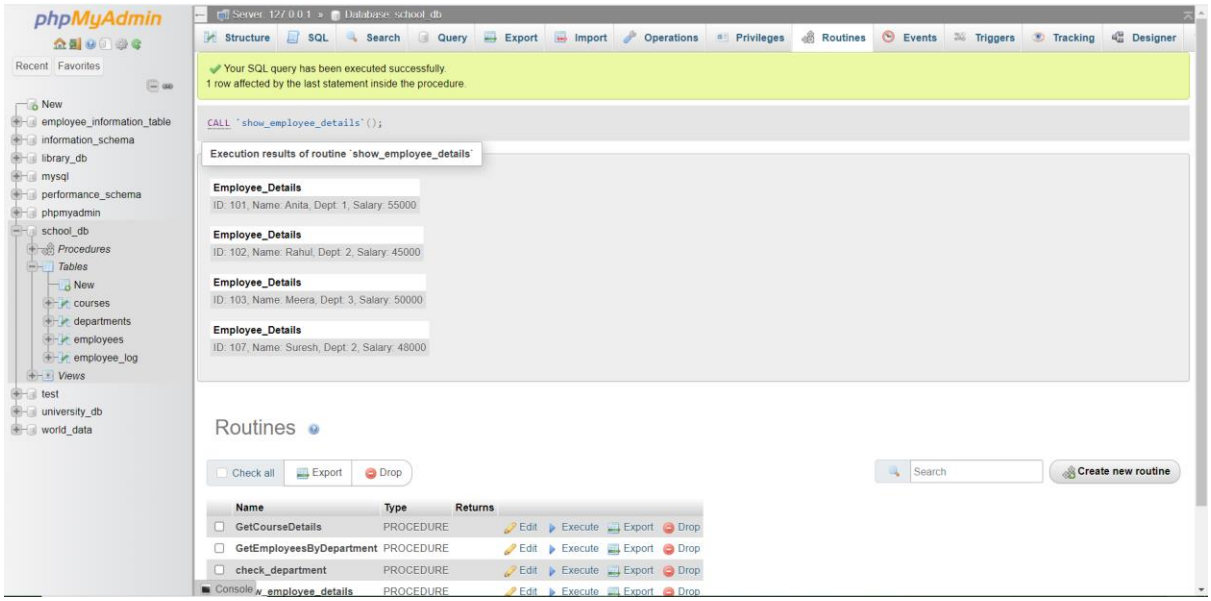
Query 1: Stored Procedure

```
DELIMITER $$


CREATE PROCEDURE show_employee_details()
BEGIN
   DECLARE done INT DEFAULT 0;
   DECLARE v_id INT;
   DECLARE v_name VARCHAR(50);
   DECLARE v_dept INT;
   DECLARE v_salary INT;

   DECLARE emp_cursor CURSOR FOR
      SELECT emp_id, emp_name, dept_id, salary FROM employees;

   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

   OPEN emp_cursor;

   read_loop: LOOP
      FETCH emp_cursor INTO v_id, v_name, v_dept, v_salary;
      IF done = 1 THEN
         LEAVE read_loop;
      END IF;

      SELECT CONCAT(
         'ID: ', v_id,
```

```
            ', Name: ', v_name,
            ', Dept: ', v_dept,
            ', Salary: ', v_salary
        ) AS Employee_Details;
    END LOOP;

    CLOSE emp_cursor;
END$$

DELIMITER ;
```

Query 2: Run the procedure

```
CALL show_employee_details();
```



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

Answer:

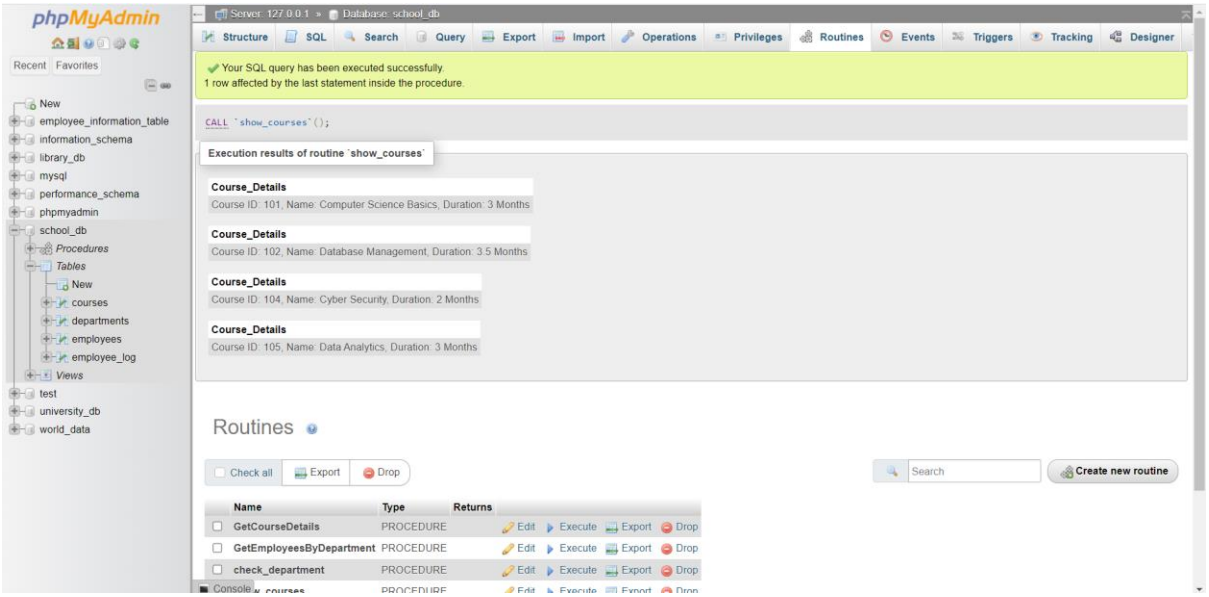
Query 1: Stored Procedure

```
DELIMITER $$

CREATE PROCEDURE show_courses()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE v_id INT;
    DECLARE v_name VARCHAR(100);
    DECLARE v_duration VARCHAR(50);

    DECLARE course_cursor CURSOR FOR
        SELECT course_id, course_name, course_duration FROM courses;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    OPEN course_cursor;

    course_loop: LOOP
        FETCH course_cursor INTO v_id, v_name, v_duration;
        IF done = 1 THEN
            LEAVE course_loop;
        END IF;

        SELECT CONCAT(
            'Course ID: ', v_id,
            ', Name: ', v_name,
            ', Duration: ', v_duration
```

```
    ) AS Course_Details;
  END LOOP;

  CLOSE course_cursor;
END$$

DELIMITER ;
```

Query 2: Run the procedure

```
CALL show_courses();
```



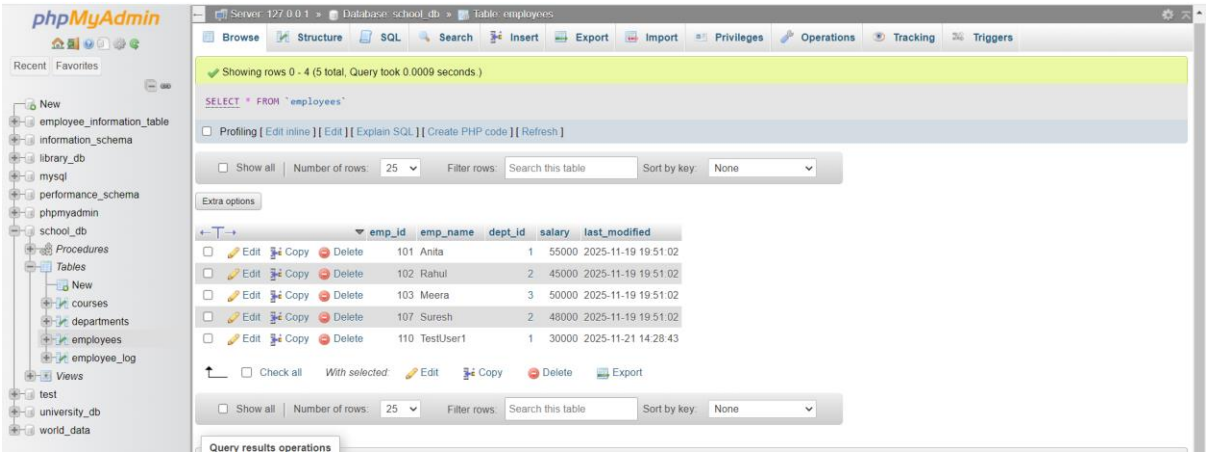
19. Rollback and Commit Savepoint

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

Answer:

Query: Create Savepoint → Insert Records → Rollback to Savepoint

```
-- Start the transaction
START TRANSACTION;

-- Insert 1st record
INSERT INTO employees (emp_id, emp_name, dept_id, salary)
VALUES (110, 'TestUser1', 1, 30000);

-- Create savepoint
SAVEPOINT sp1;

-- Insert 2nd record
INSERT INTO employees (emp_id, emp_name, dept_id, salary)
VALUES (111, 'TestUser2', 2, 40000);

-- Rollback to savepoint (means the 2nd insert will be removed)
ROLLBACK TO sp1;

-- Commit remaining changes
COMMIT;
```



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

Answer:

Query: Commit part of a transaction → Rollback remaining changes

```
-- Start transaction
START TRANSACTION;

-- Insert two new employees
INSERT INTO employees (emp_id, emp_name, dept_id, salary)
VALUES (120, 'PartialUser1', 1, 35000);

INSERT INTO employees (emp_id, emp_name, dept_id, salary)
VALUES (121, 'PartialUser2', 2, 36000);

-- Create savepoint
SAVEPOINT sp2;

-- Insert another record
INSERT INTO employees (emp_id, emp_name, dept_id, salary)
VALUES (122, 'RollbackUser', 3, 37000);

-- Commit changes BEFORE savepoint (120, 121)
```

```
RELEASE SAVEPOINT sp2;

COMMIT;


-- Now rollback remaining uncommitted changes

ROLLBACK;
```