SQL Query Code (Lab Excercise)

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.

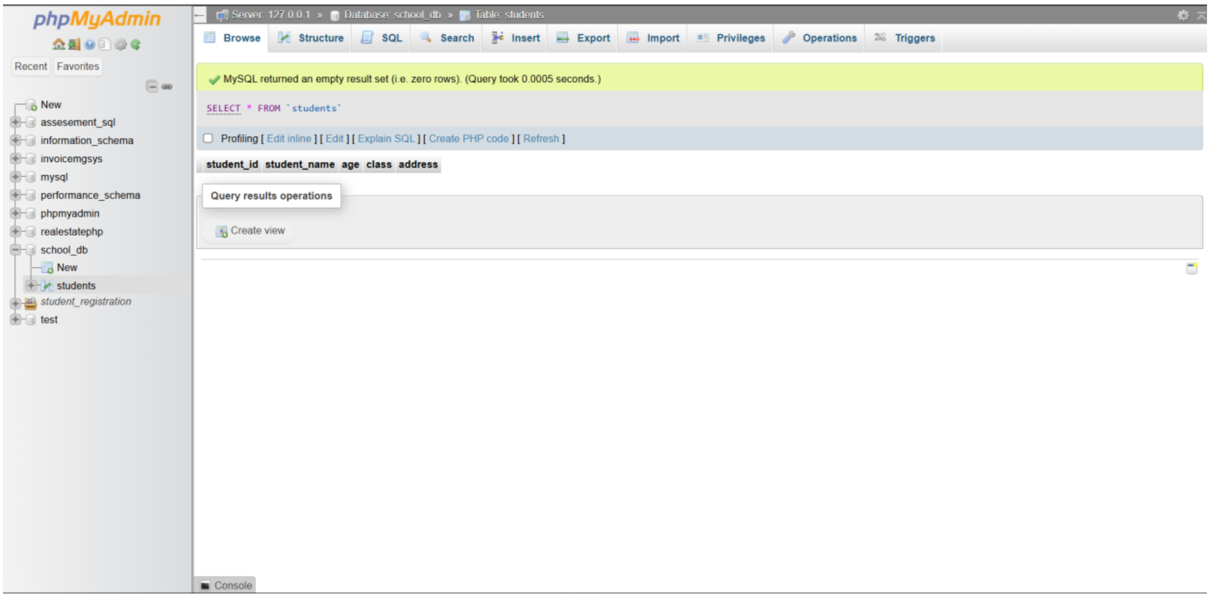
CODE:

CREATE DATABASE school_db;

USE school_db;

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

SCREENSHOT:



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

CODE:

INSERT INTO students (student_id, student_name, age, class, address) VALUES

(1, 'Rohan Patel', 15, '10A', 'Rajkot'),

(2, 'Meera Shah', 14, '9B', 'Ahmedabad'),

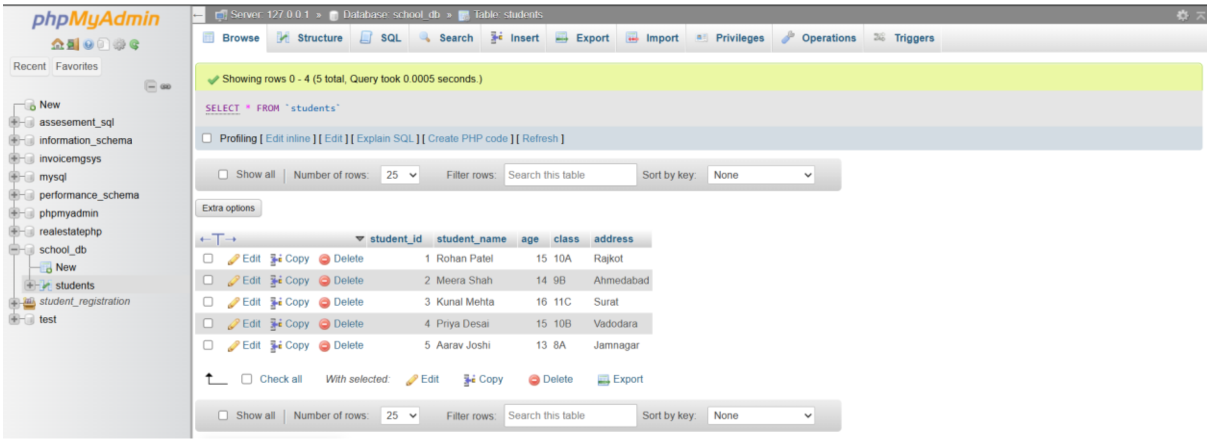
(3, 'Kunal Mehta', 16, '11C', 'Surat'),

(4, 'Priya Desai', 15, '10B', 'Vadodara'),

(5, 'Aarav Joshi', 13, '8A', 'Jamnagar');

SELECT * FROM students;

SCREENSHOT:



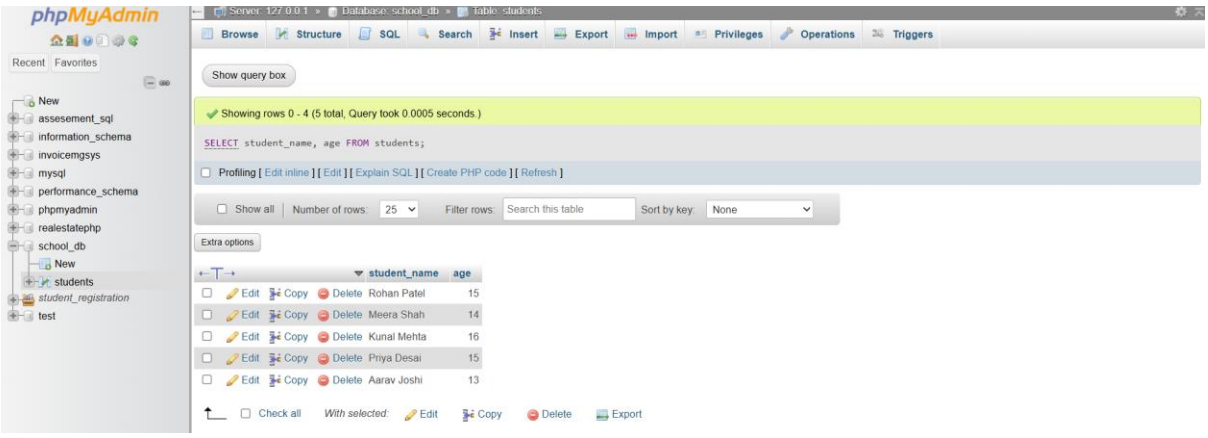
SQL Syntax

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

Code:

SELECT student_name, age FROM students;

SCREENSHOT:

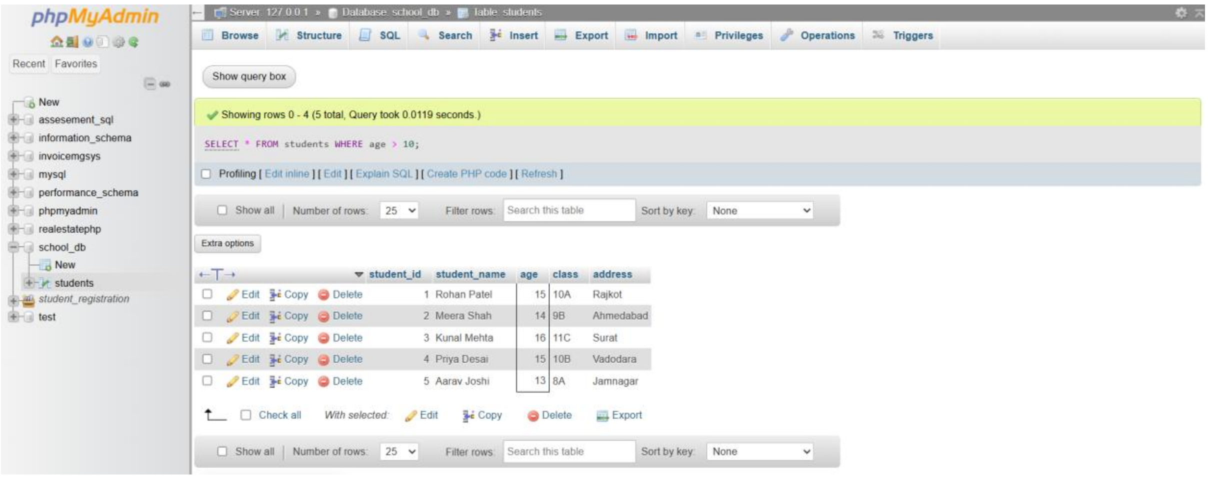


Lab 2: Write SQL queries to retrieve all students whose age is greater than 10.

CODE:

```
SELECT * FROM students

WHERE age > 10;
```

SCREENSHOT:



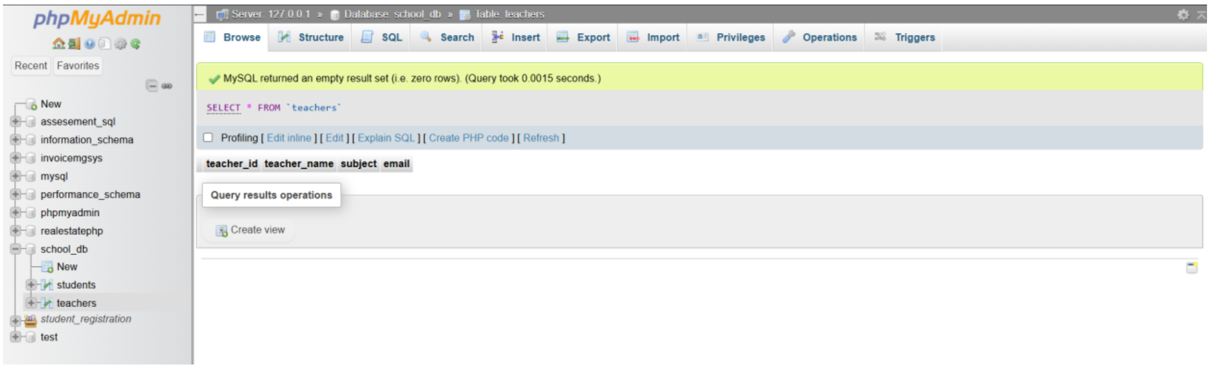
SQL Constraints

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

CODE:

```
CREATE TABLE teachers (

    teacher_id INT PRIMARY KEY,

    teacher_name VARCHAR(100) NOT NULL,

    subject VARCHAR(100) NOT NULL,

    email VARCHAR(150) UNIQUE

);
```

SCREENSHOT:



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

CODE:

ALTER TABLE students

ADD teacher_id INT;

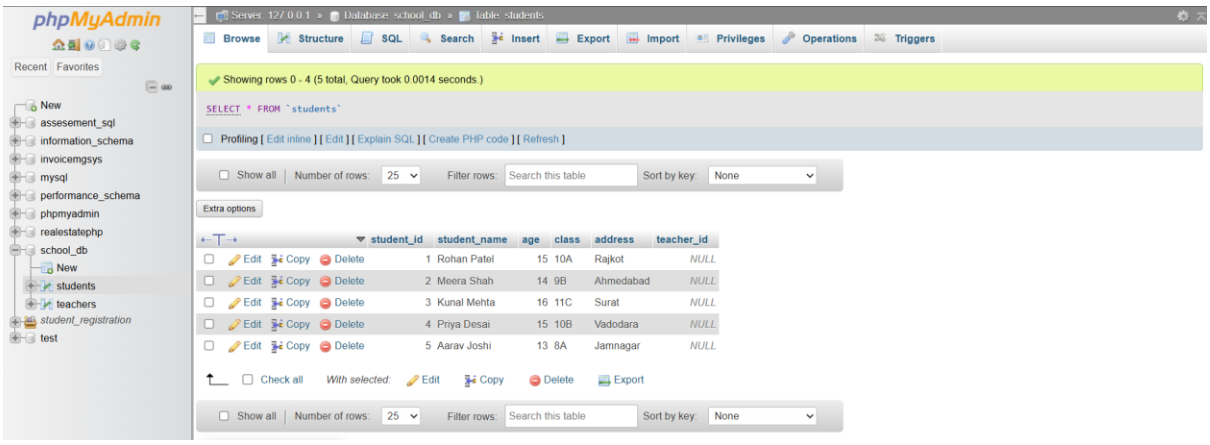
ALTER TABLE students

ADD CONSTRAINT fk_teacher

FOREIGN KEY (teacher_id)

REFERENCES teachers(teacher_id);

SCREENSHOT:



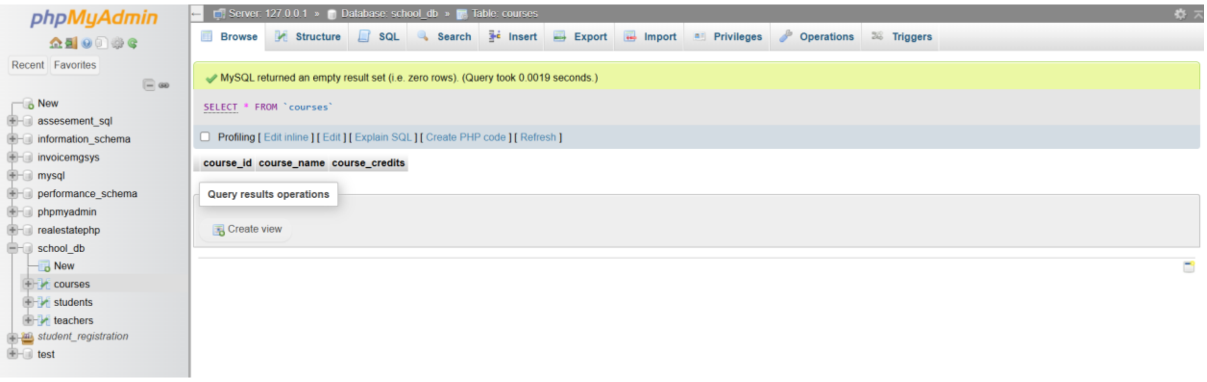
Main SQL Commands and Sub-commands (DDL)

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

CODE:

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

SCREENSHOT:

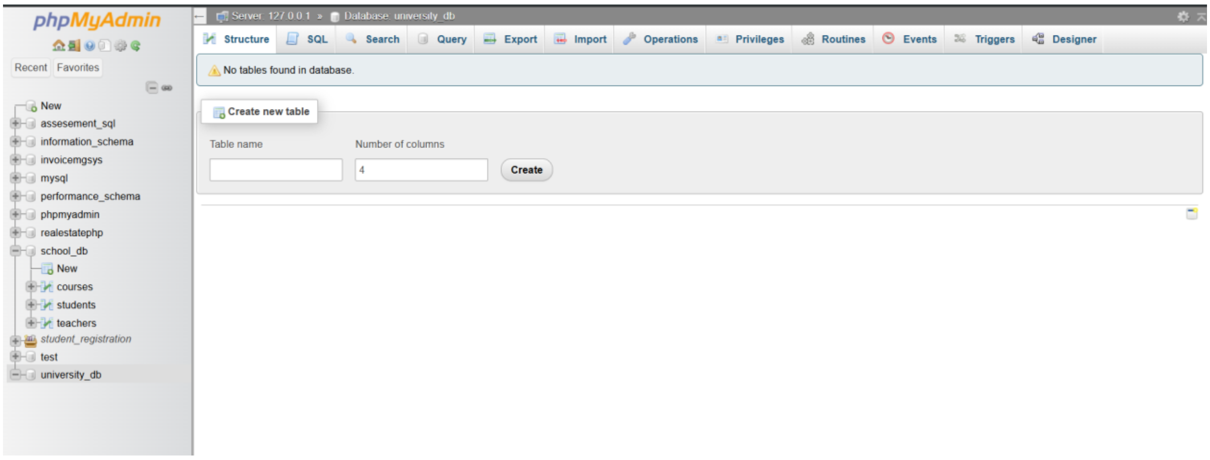


Lab 2: Use the CREATE command to create a database university_db.

CODE:

```
CREATE DATABASE university_db;
```

SCREENSHOT:



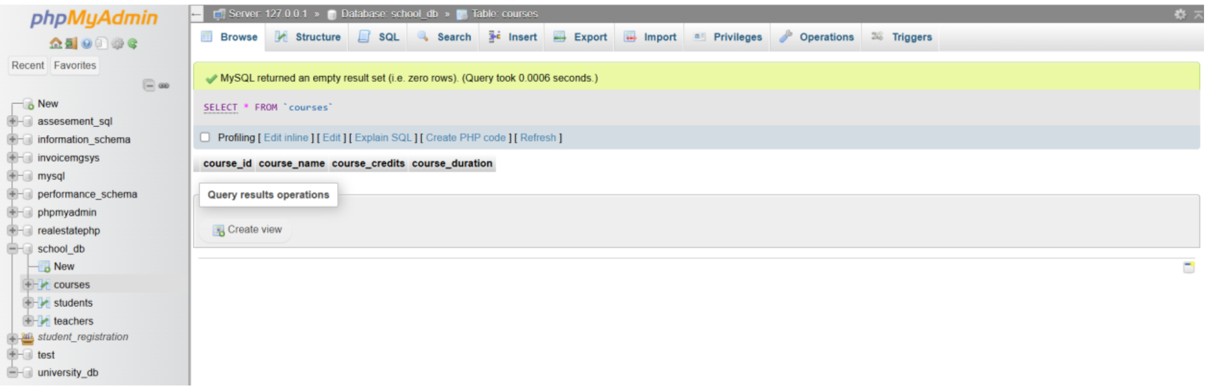
ALTER Command

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

CODE:

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

SCREENSHOT:

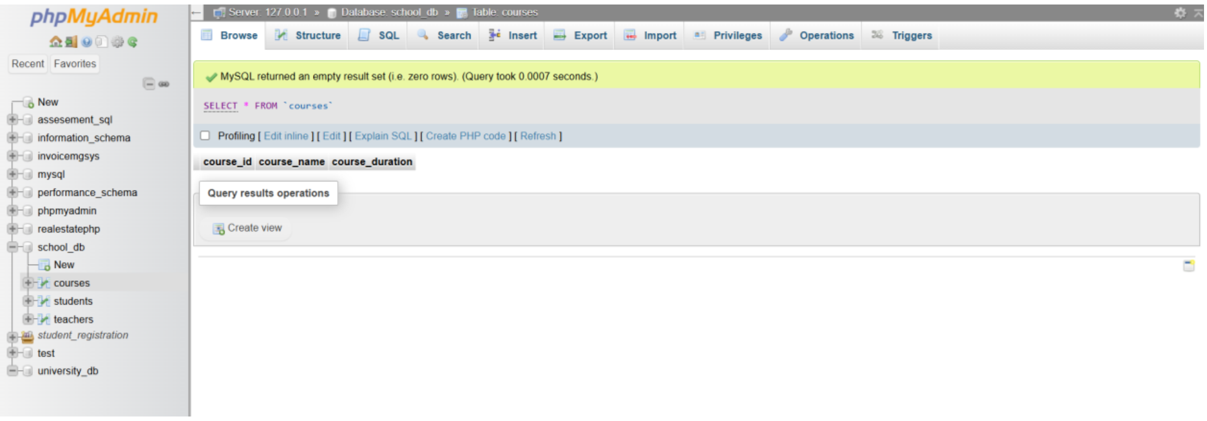


Lab 2: Drop the course_credits column from the courses table.

CODE:

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

SCREENSHOT:



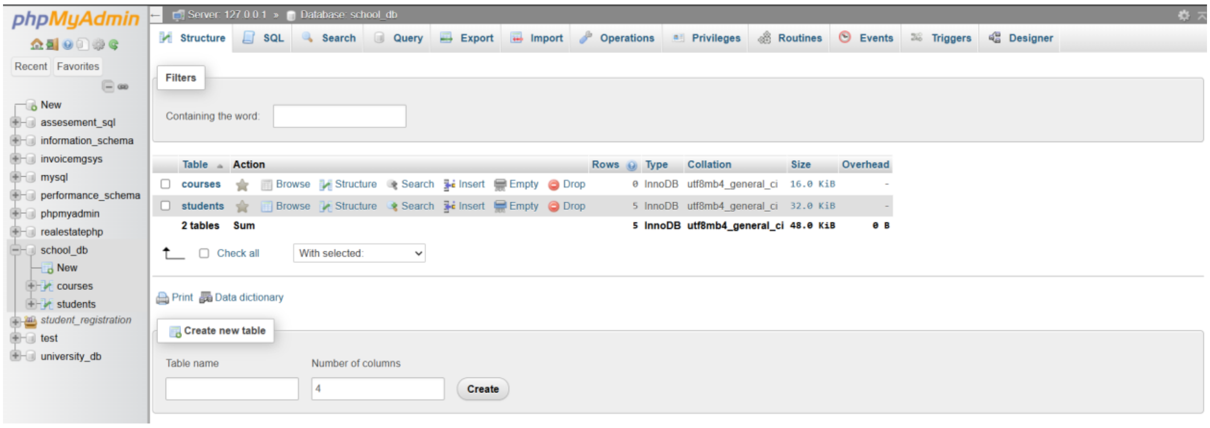
DROP Command

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

CODE:

```
ALTER TABLE students
DROP FOREIGN KEY fk_teacher;

USE school_db;
DROP TABLE teachers
```

SCREENSHOT:

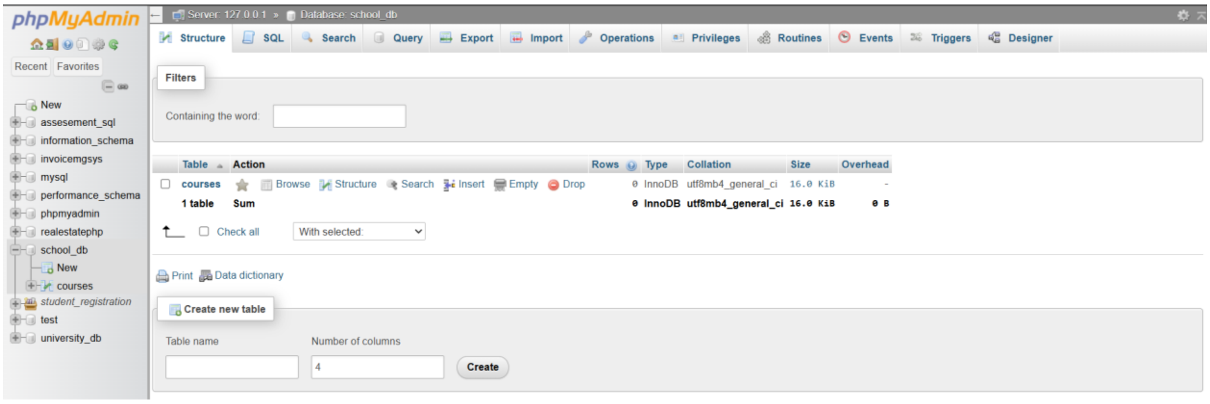


Lab 2 : Drop the students table from the school_db database and verify that the table has been removed.

CODE:

DROP TABLE students;

SCREENSHOT:



Data Manipulation Language (DML)

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

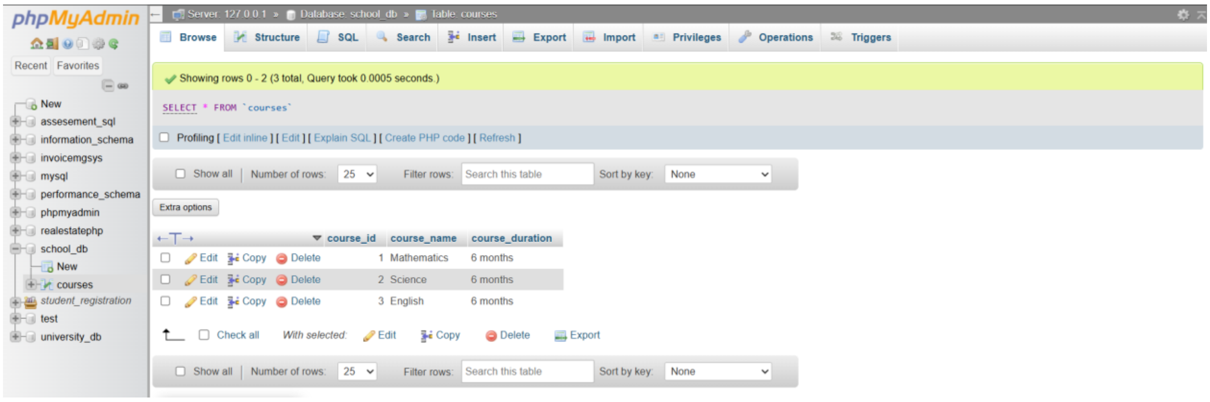
CODE:

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (1, 'Mathematics', '6 months');

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (2, 'Science', '6 months');

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (3, 'English', '6 months');

SCREENSHOT:

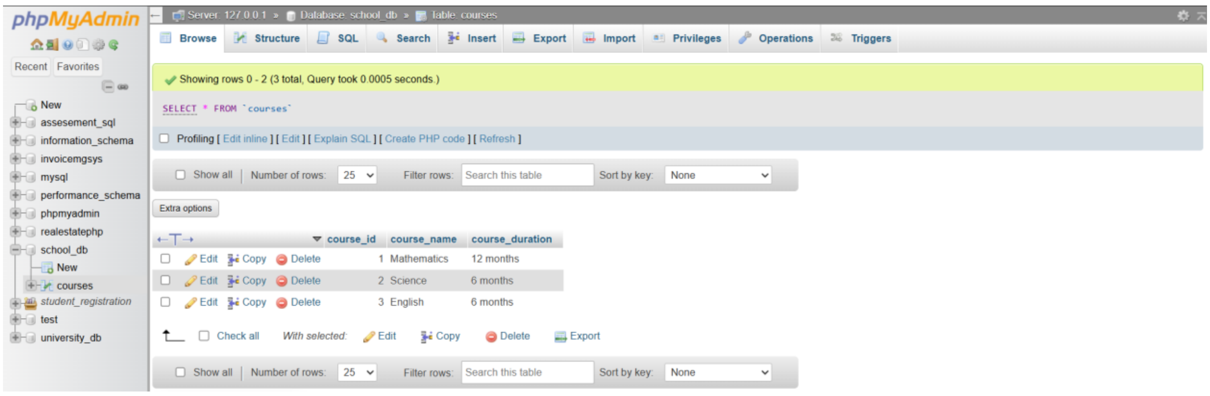


Lab 2: Update the course duration of a specific course using the UPDATE command.

CODE:

```
UPDATE courses
SET course_duration = '12 months'
WHERE course_name = 'Mathematics';
```

SCREENSHOT:

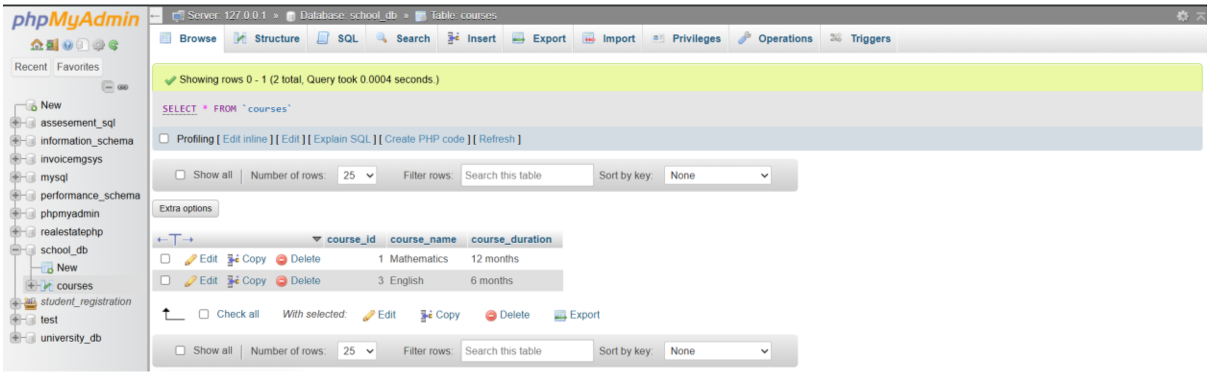


Lab 3: Delete a course with a specific course_id from the courses table using the DELETE command.

CODE:

```
DELETE FROM courses
WHERE course_id = 2;
```

SCREENSHOT:



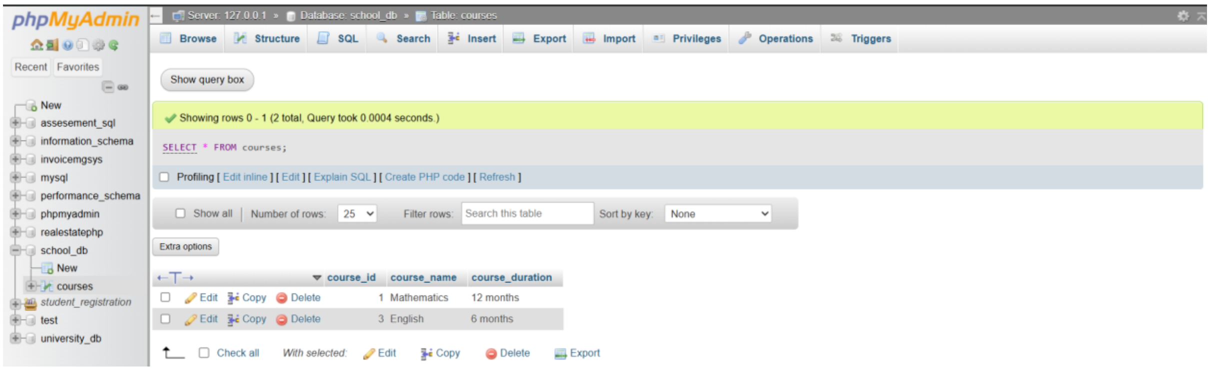
Data Query Language (DQL)

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

CODE:

SELECT * FROM courses;

SCREENSHOT:

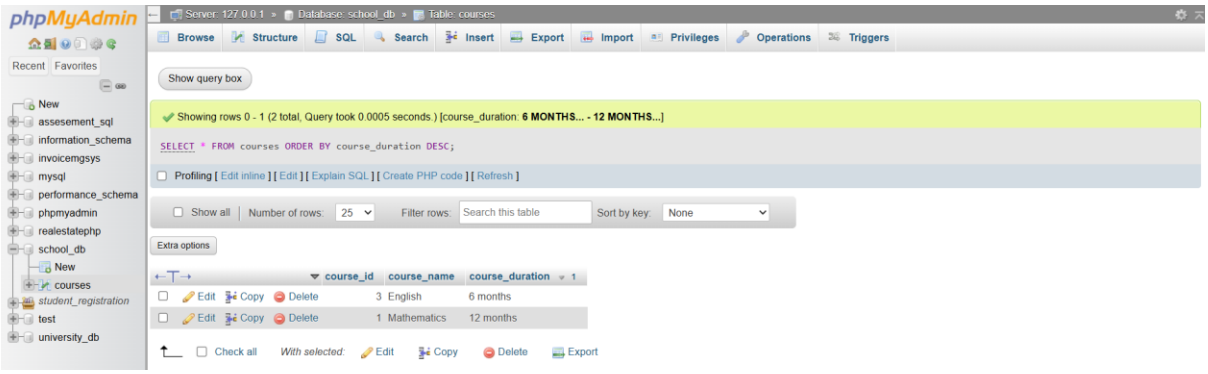


Lab 2: Sort the courses based on course_duration in descending order using ORDER BY.

CODE:

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

SCREENSHOT:

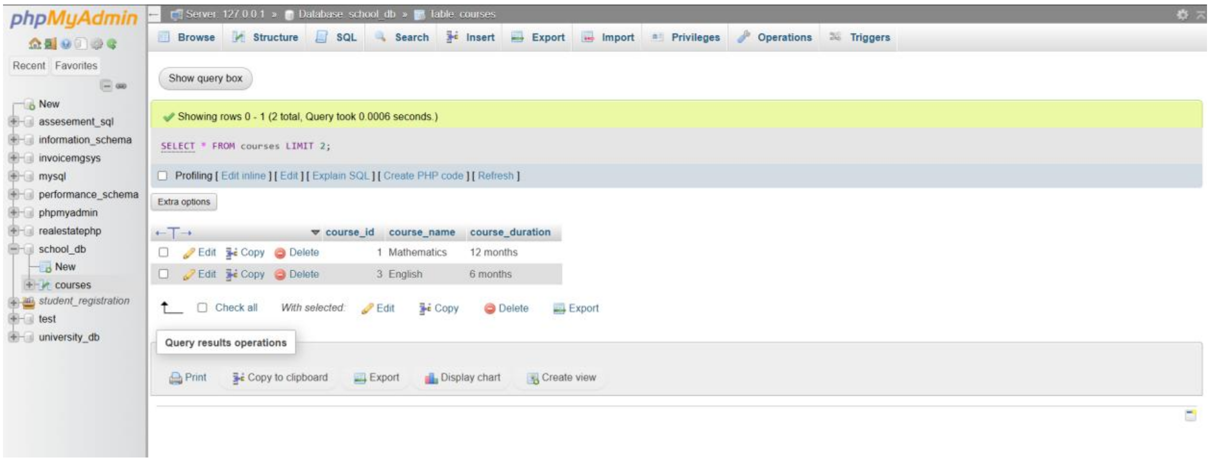


Lab 3: Limit the results of the SELECT query to show only the top two courses using LIMIT.

CODE:

SELECT * FROM courses LIMIT 2;

SCREENSHOT:



Data Control Language (DCL)

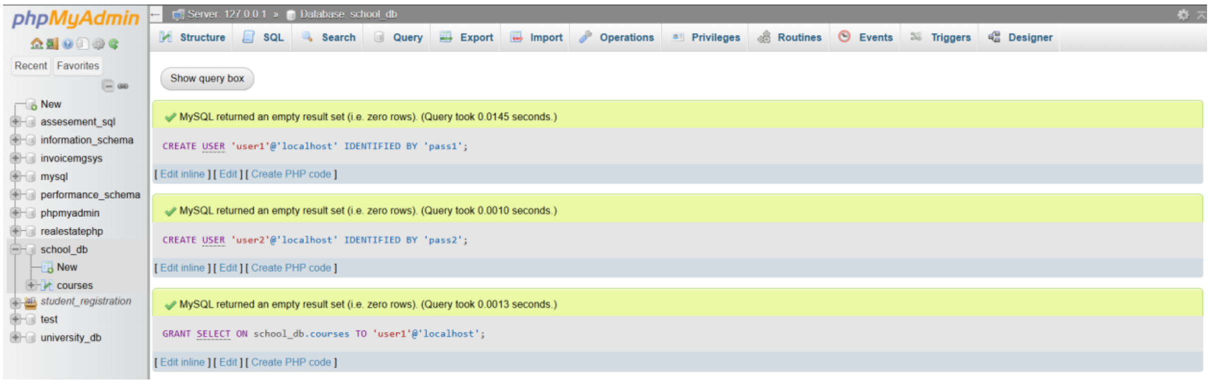
Lab 1: Create two new users user1 and user2 and grant user1 permission to SELECT from the courses table.

CODE:

CREATE USER 'user1'@'localhost' IDENTIFIED BY 'pass1';
CREATE USER 'user2'@'localhost' IDENTIFIED BY 'pass2';

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

SCREENSHOT:



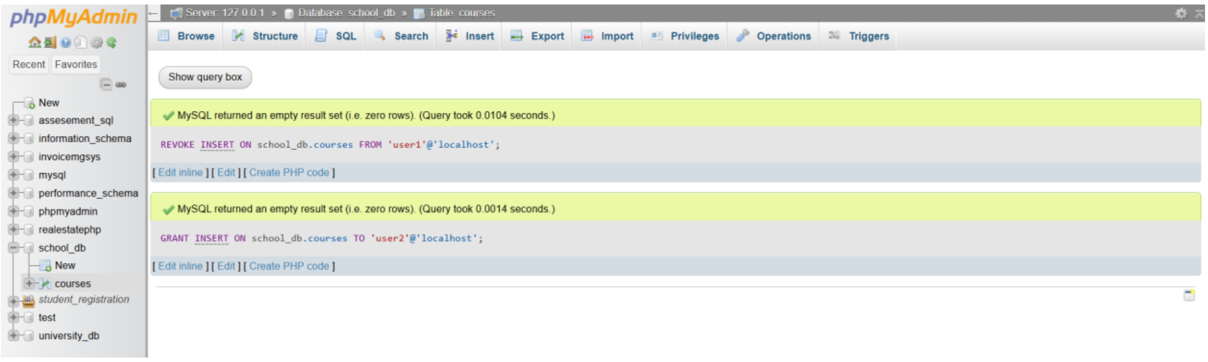
Lab 2:Revoke the INSERT permission from user1 and give it to user2.

CODE:

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

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

SCREENSHOT:



Transaction Control Language (TCL)

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

CODE:

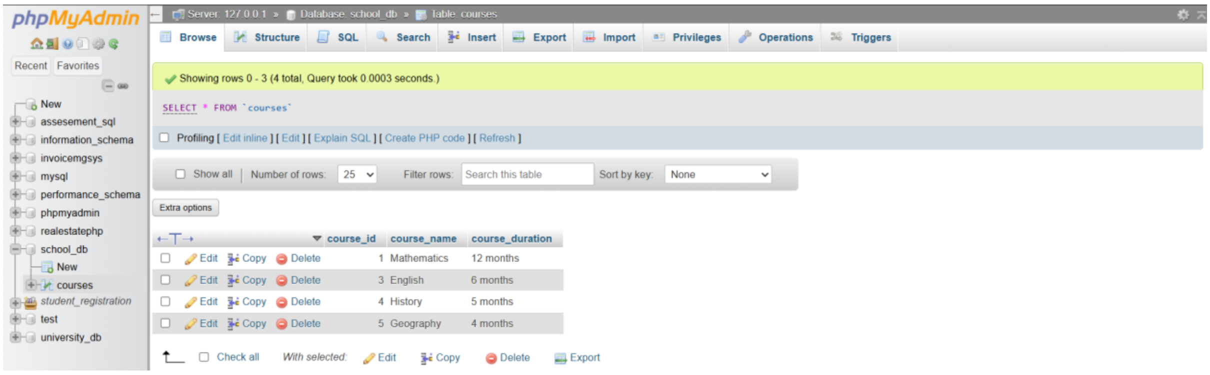
START TRANSACTION;

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (4, 'History', '5 months');

INSERT INTO courses (course_id, course_name, course_duration)
VALUES (5, 'Geography', '4 months');

COMMIT;

SCREENSHOT:

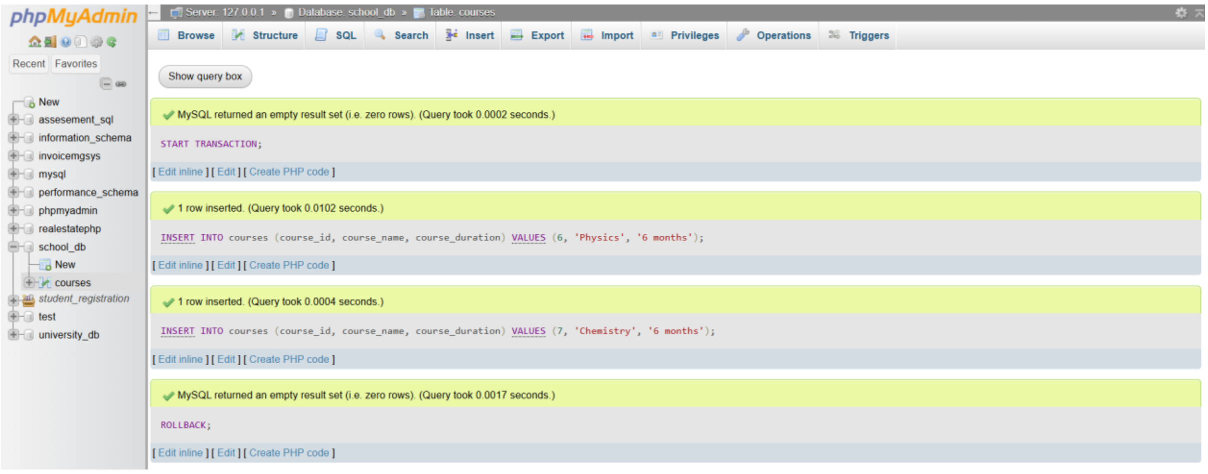


Lab 2: Insert additional rows, then use ROLLBACK to undo the last insert operation.

CODE:

START TRANSACTION;
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (6, 'Physics', '6 months');
INSERT INTO courses (course_id, course_name, course_duration)
VALUES (7, 'Chemistry', '6 months');
ROLLBACK;

SCREENSHOT:

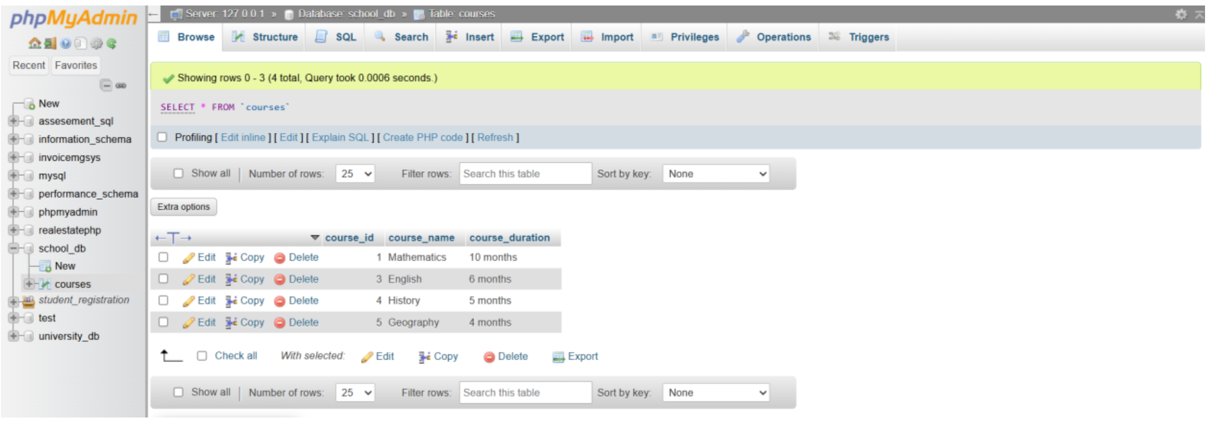


Lab 3: Create a SAVEPOINT before updating the courses table, and use it to roll back specific changes.

CODE:

```
START TRANSACTION;

UPDATE courses
SET course_duration = '10 months'
WHERE course_name = 'Mathematics';

SAVEPOINT before_science_update;

UPDATE courses
SET course_duration = '8 months'
WHERE course_name = 'Science';

ROLLBACK TO SAVEPOINT before_science_update;

COMMIT;
```

SCREENSHOT:



SQL Joins

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

CODE:

Step 1 — Create the tables

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

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

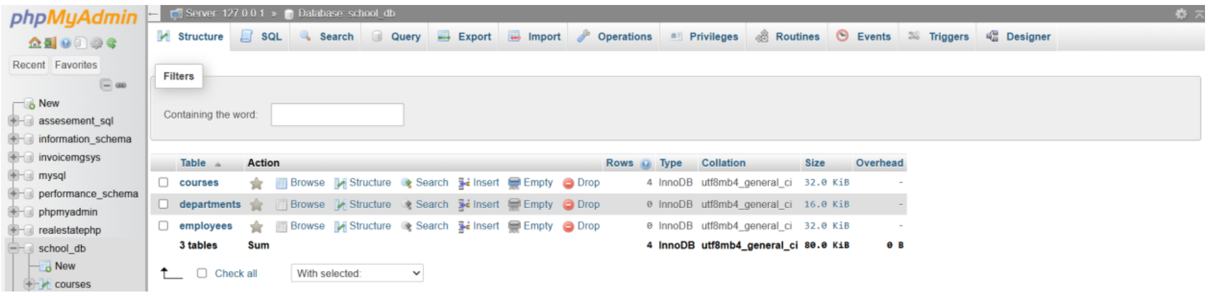
Step 2 — Insert sample data

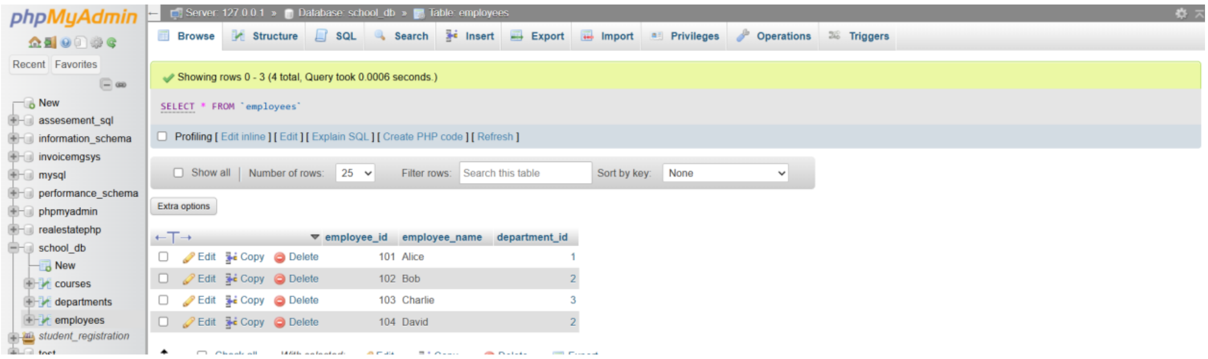
```
INSERT INTO departments (department_id, department_name)
VALUES (1, 'HR'), (2, 'Finance'), (3, 'IT');

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

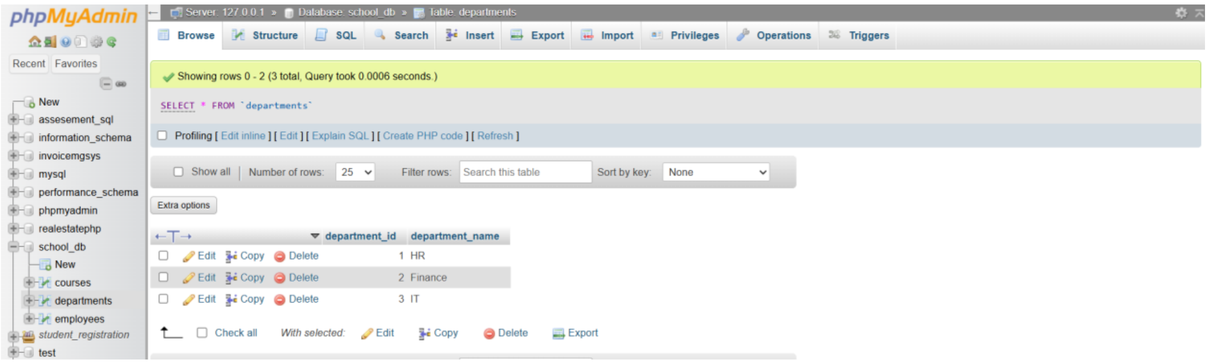
Step 3 — Perform INNER JOIN

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

SCREENSHOT:





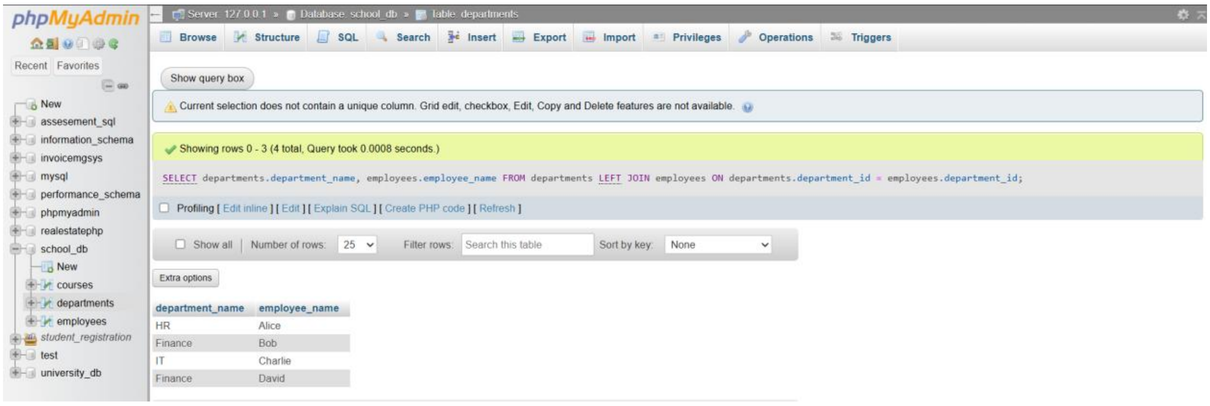


Lab 2: Use a LEFT JOIN to show all departments, even those without employees.

CODE:

SELECT departments.department_name, employees.employee_name
FROM departments
LEFT JOIN employees
ON departments.department_id = employees.department_id;

SCREENSHOT:



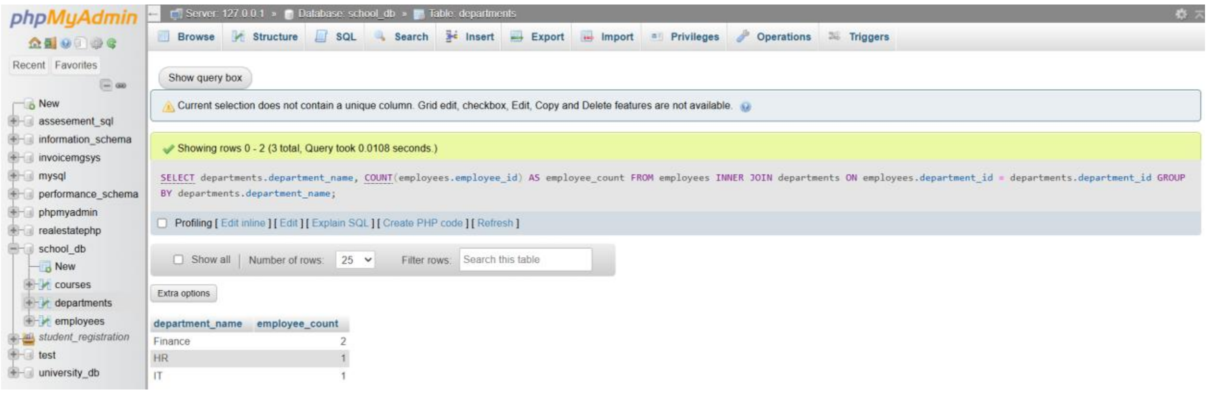
SQL Group By

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

CODE:

SELECT departments.department_name, COUNT(employees.employee_id) AS employee_count
FROM employees
INNER JOIN departments
ON employees.department_id = departments.department_id
GROUP BY departments.department_name;

SCREENSHOT:



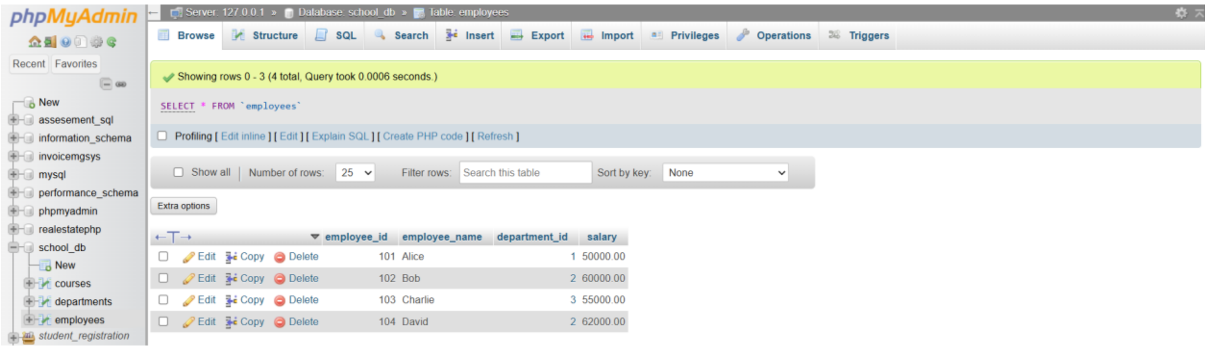
Lab 2: Use the AVG aggregate function to find the average salary of employees in each department.

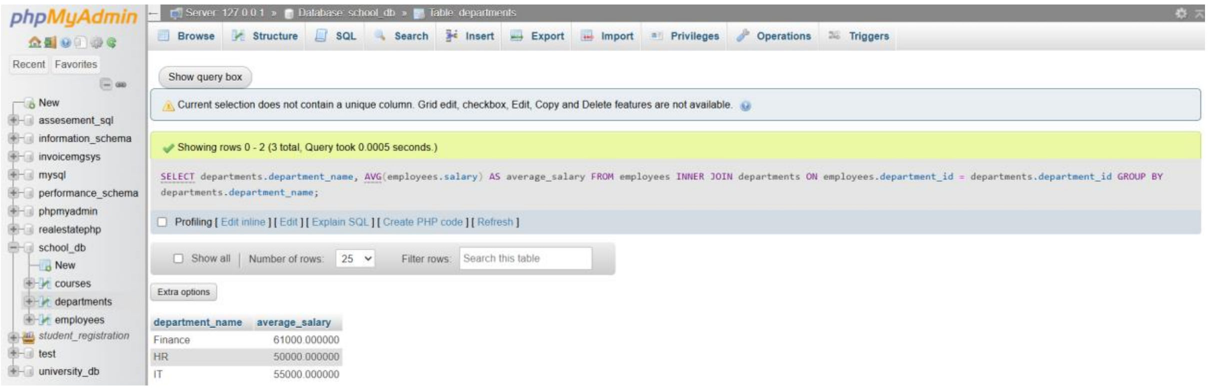
CODE:

UPDATE employees
SET salary = 50000 WHERE employee_id = 101;
UPDATE employees
SET salary = 60000 WHERE employee_id = 102;
UPDATE employees
SET salary = 55000 WHERE employee_id = 103;
UPDATE employees
SET salary = 62000 WHERE employee_id = 104;

SELECT departments.department_name, AVG(employees.salary) AS average_salary
FROM employees
INNER JOIN departments
ON employees.department_id = departments.department_id
GROUP BY departments.department_name;

SCREENSHOT:





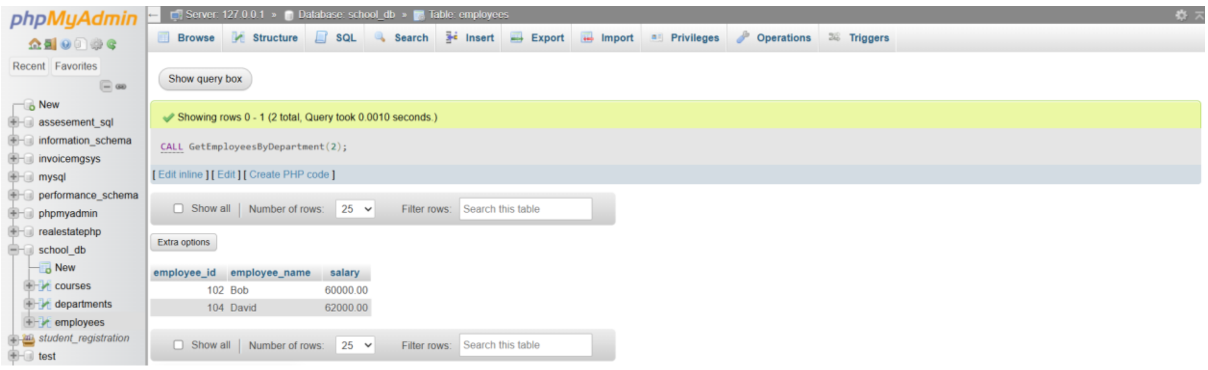
SQL Stored Procedure

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

CODE:

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

SCREENSHOT:



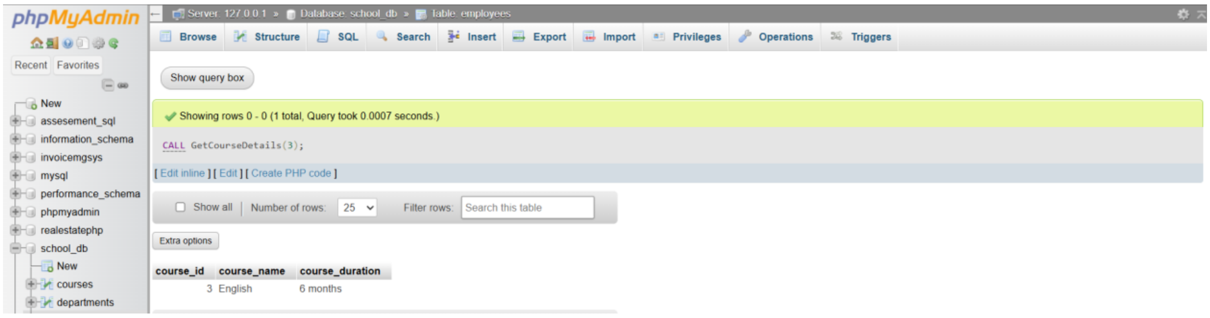
Lab 2: Write a stored procedure that accepts course_id as input and returns the course details.

CODE:

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

CALL GetCourseDetails(3);

SCREENSHOT:



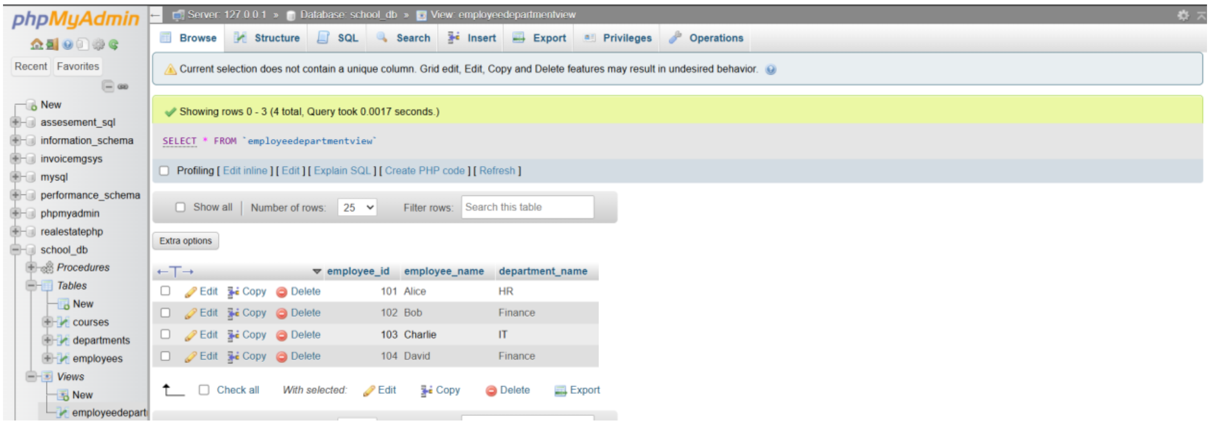
SQL View

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

CODE:

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;

SCREENSHOT:

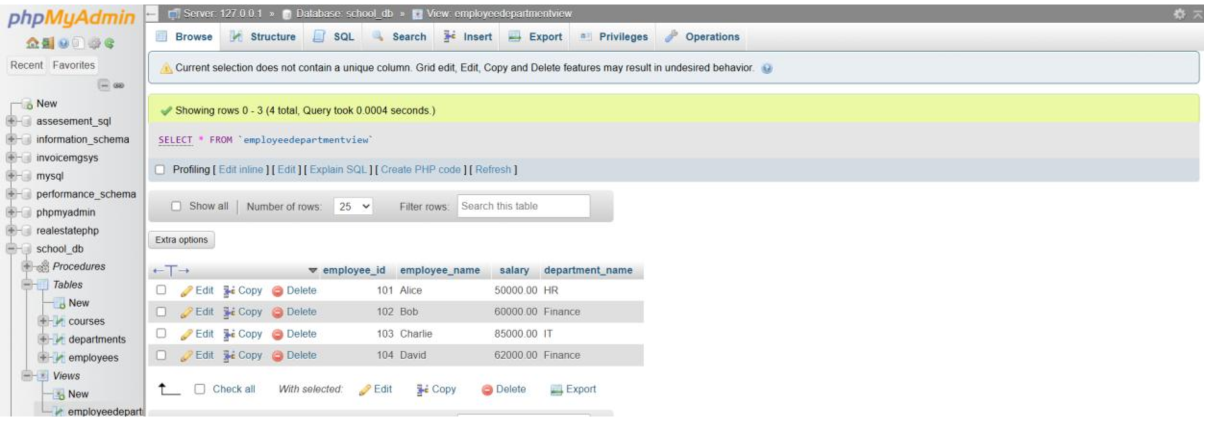


Lab 2: Modify the view to exclude employees whose salaries are below $50,000.

CODE:

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

SCREENSHOT:



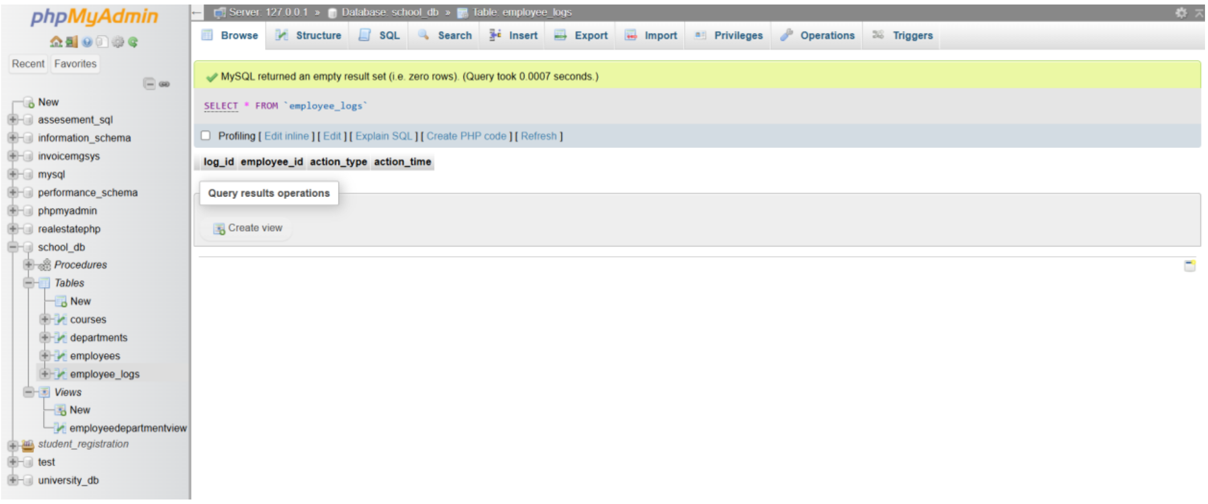
SQL Triggers

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

CODE:

```
CREATE TABLE employee_logs (
   log_id INT AUTO_INCREMENT PRIMARY KEY,
   employee_id INT,
   action_type VARCHAR(50),
   action_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

SCREENSHOT:

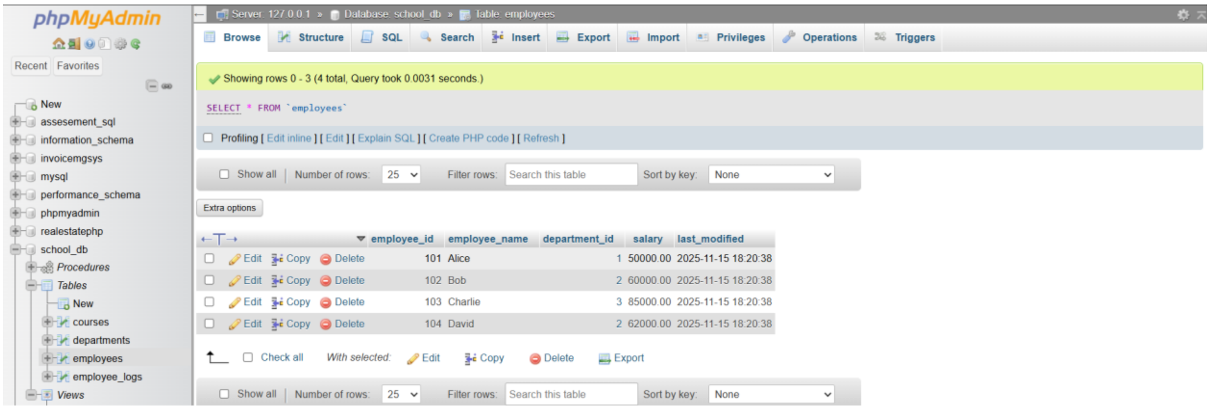


Lab 2: Create a trigger to update the last_modified timestamp whenever an employee record is updated.

CODE:

CREATE TRIGGER update_employee_timestamp
BEFORE UPDATE ON employees
FOR EACH ROW
SET NEW.last_modified = CURRENT_TIMESTAMP;

SCREEENSHOT:



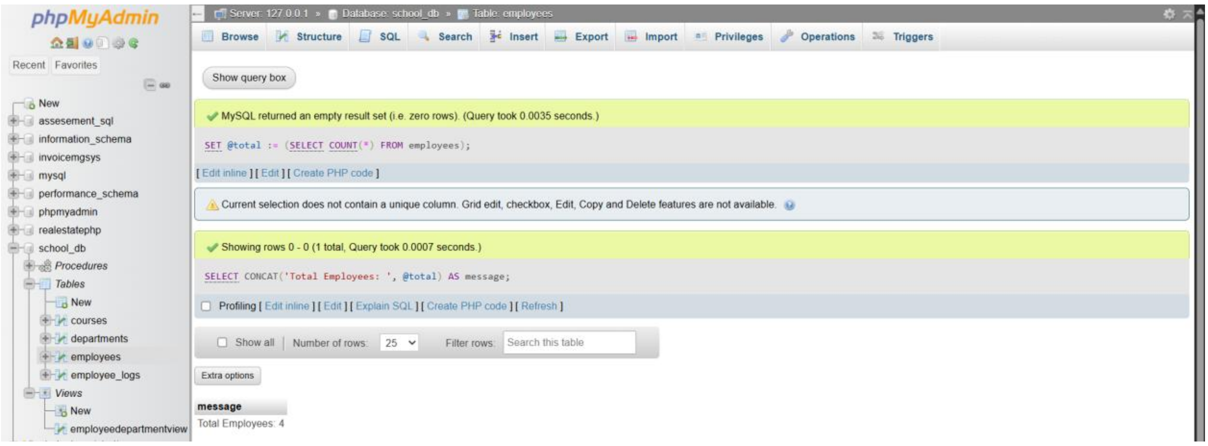
Introduction to PL/SQL

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

CODE:

SET @total := (SELECT COUNT(*) FROM employees);
SELECT CONCAT('Total Employees: ', @total) AS message;

SCREENSHOT:

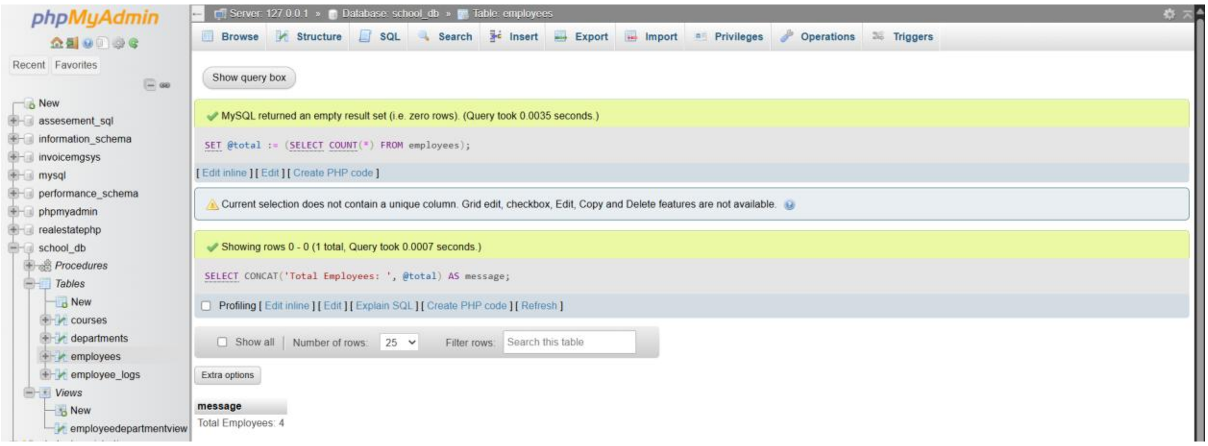


Lab 2: Create a PL/SQL block that calculates the total sales from an orders table.

CODE:

SET @sales := (SELECT SUM(total_amount) FROM orders);
SELECT CONCAT('Total Sales: ', @sales) AS message;

SCREENSHOT:



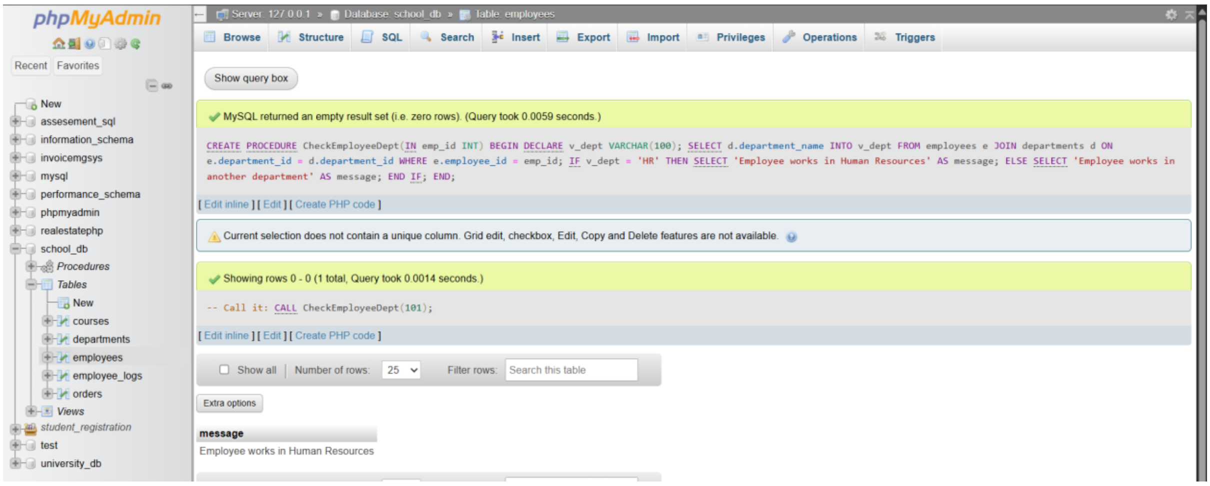
PL/SQL Control Structures

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

CODE:

```
DELIMITER $$

CREATE PROCEDURE CheckEmployeeDept(IN emp_id INT)
BEGIN
   DECLARE v_dept VARCHAR(100);
   SELECT d.department_name INTO v_dept
   FROM employees e
   JOIN departments d ON e.department_id = d.department_id
   WHERE e.employee_id = emp_id;

   IF v_dept = 'HR' THEN
      SELECT 'Employee works in Human Resources' AS message;
   ELSE
      SELECT 'Employee works in another department' AS message;
   END IF;
END $$

DELIMITER ;
CALL CheckEmployeeDept(101);
```

SCREENSHOT:

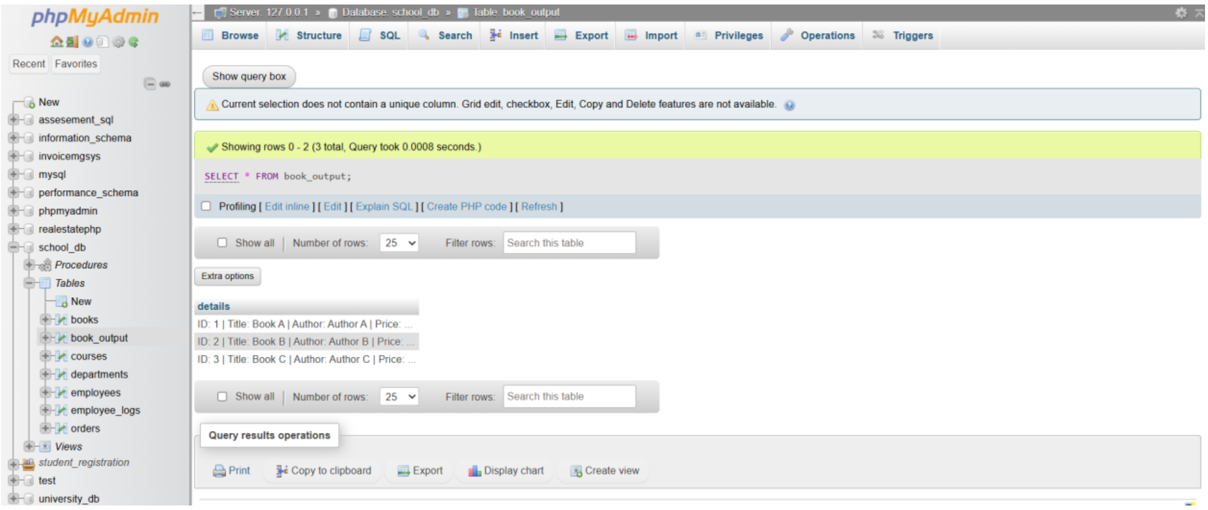


Lab 4: Use a FOR LOOP in PL/SQL to display the details of all books one by one.

CODE:

```
CALL display_books();

SELECT * FROM book_output;
```

SCREENSHOT:



SQL Cursors

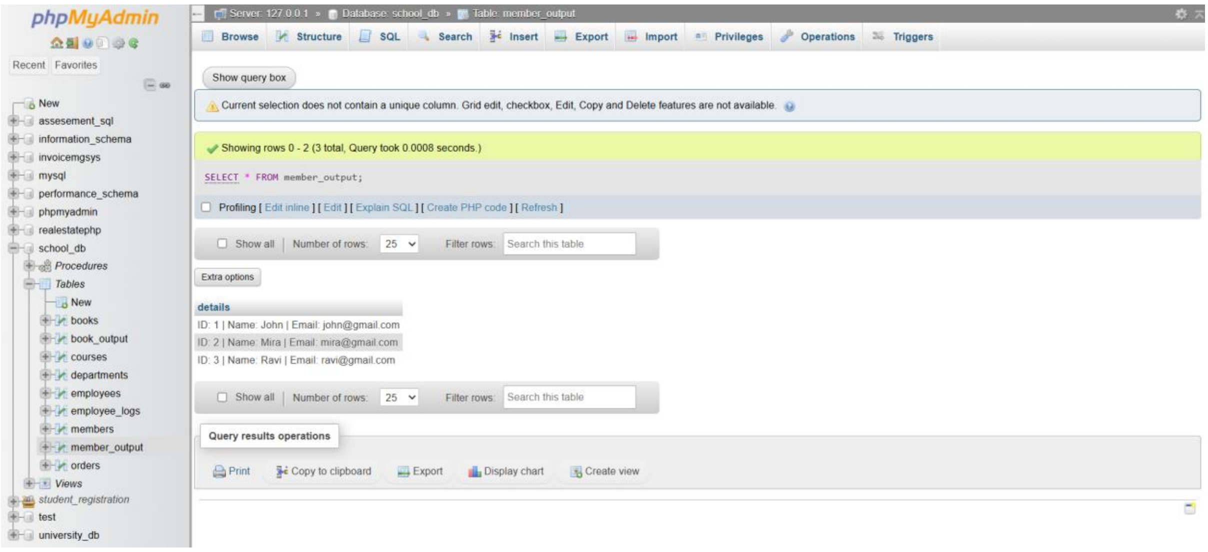
Lab 3: Write a PL/SQL block using an explicit cursor to fetch and display all records from the members table.

CODE:

```
DROP TABLE IF EXISTS member_output;

CREATE TABLE member_output (
   details VARCHAR(300)
);

DELIMITER $$

CREATE PROCEDURE display_members()
BEGIN
   DECLARE done INT DEFAULT FALSE;
   DECLARE m_id INT;
   DECLARE m_name VARCHAR(100);
   DECLARE m_email VARCHAR(100);

   DECLARE member_cursor CURSOR FOR
      SELECT member_id, member_name, email FROM members;
```

```
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    OPEN member_cursor;

    read_loop: LOOP
        FETCH member_cursor INTO m_id, m_name, m_email;

        IF done THEN
            LEAVE read_loop;
        END IF;

        INSERT INTO member_output VALUES (
            CONCAT(
                'ID: ', m_id,
                ' | Name: ', m_name,
                ' | Email: ', m_email
            )
        );
    END LOOP;

    CLOSE member_cursor;
END $$

DELIMITER ;
```

SCREENSHOT:

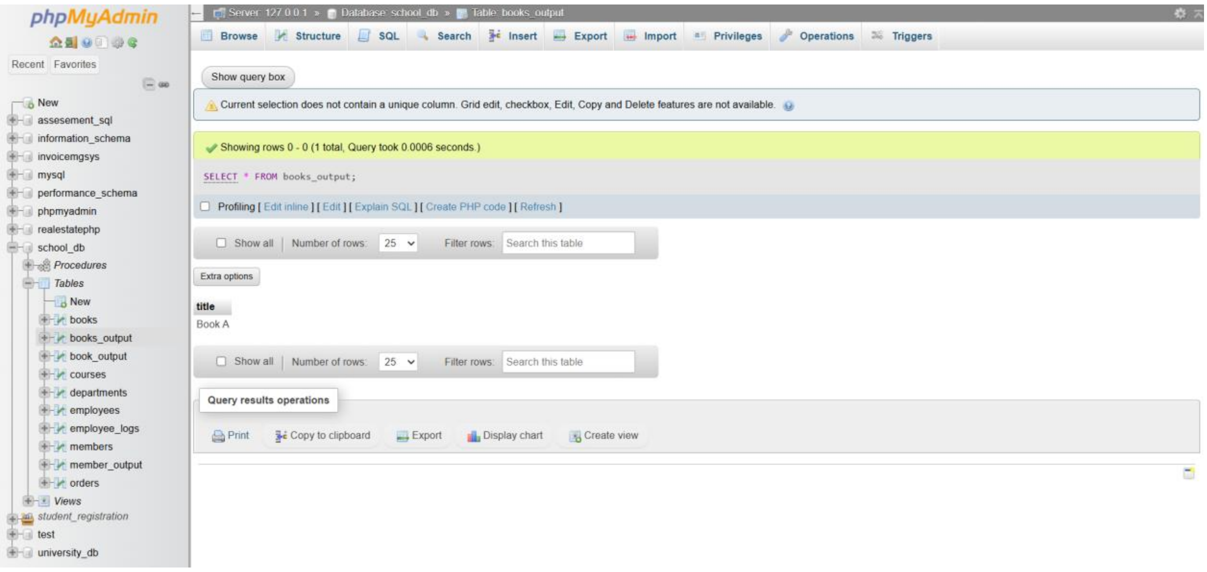


Lab 4: Create a cursor to retrieve books by a particular author and display their titles.

CODE:

```
DELIMITER $$

CREATE PROCEDURE get_books_by_author(IN p_author VARCHAR(100))
BEGIN
   DECLARE done INT DEFAULT FALSE;
   DECLARE b_title VARCHAR(200);

   DECLARE book_cursor CURSOR FOR
      SELECT title FROM books WHERE author = p_author;

   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

   OPEN book_cursor;

   read_loop: LOOP
      FETCH book_cursor INTO b_title;

      IF done THEN
         LEAVE read_loop;
      END IF;

      INSERT INTO books_output VALUES (b_title);
   END LOOP;

   CLOSE book_cursor;
END $$

DELIMITER ;
```

SCREENSHOT:



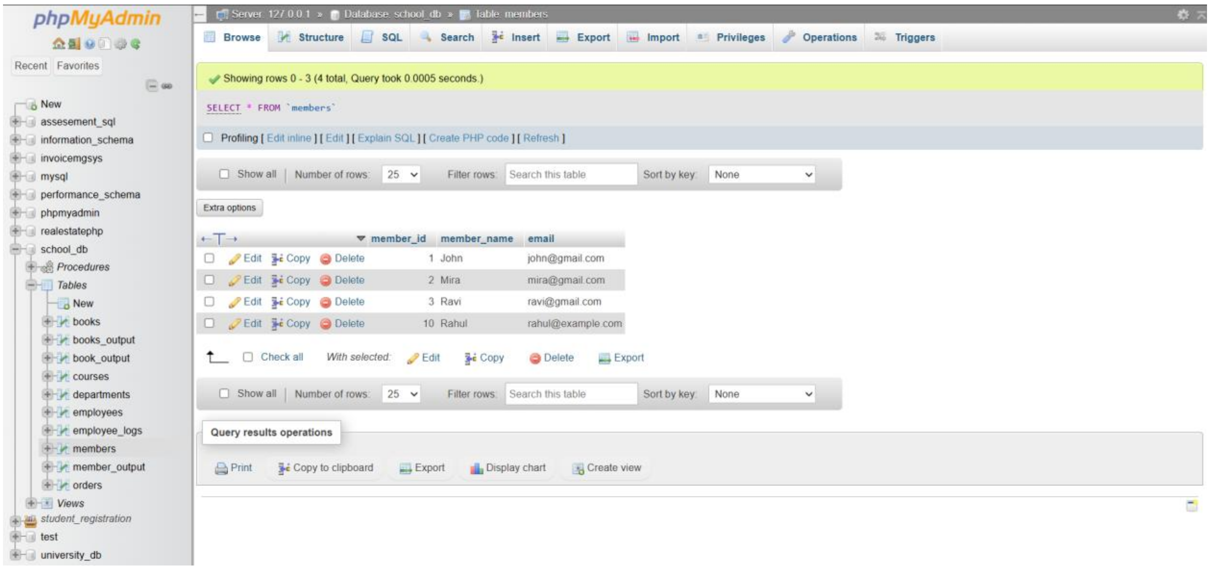
Rollback and Commit Savepoint

Lab 3: Perform a transaction that includes inserting a new member, setting a SAVEPOINT, and rolling back to the savepoint after making updates.

CODE:

```
START TRANSACTION;

INSERT INTO members (member_id, member_name, email)
VALUES (10, 'Rahul', 'rahul@example.com');

SAVEPOINT sp1;

UPDATE members
SET member_name = 'Rahul Kumar'
WHERE member_id = 10;

UPDATE members
SET email = 'rahul.k@sample.com'
WHERE member_id = 10;

ROLLBACK TO sp1;

COMMIT;
```

SCREENSHOT:



Lab 4: Use COMMIT after successfully inserting multiple books into the books table, then use ROLLBACK to undo a set of changes made after a savepoint.

CODE:

```
START TRANSACTION;

INSERT INTO books (book_id, title, author, price)
VALUES
(101, 'Book One', 'Author A', 250),
(102, 'Book Two', 'Author B', 300),
(103, 'Book Three', 'Author A', 400);

COMMIT;

START TRANSACTION;

SAVEPOINT sp1;

UPDATE books SET price = 500 WHERE book_id = 101;
UPDATE books SET title = 'Updated Book' WHERE book_id = 102;

ROLLBACK TO sp1;

COMMIT;
```

SCREENSHOT: