**Module 4 – Introduction to DBMS**

Name: Pal Gajera

**Theory Questions:**

**Introduction to SQL**

1. What is SQL, and why is it essential in database management?

**ANS**

SQL (Structured Query Language) is used to store, retrieve, manipulate, and manage data in relational databases. It is essential for database management as it helps organize data efficiently, ensures data integrity, supports user access control, and enables complex queries for better data analysis.

2. Explain the difference between DBMS and RDBMS.

**ANS**

**DBMS (Database Management System)** stores data as files and lacks relationships between data, making it suitable for small applications.

**RDBMS (Relational Database Management System)** stores data in structured tables with relationships, supports multiple users, reduces redundancy, and ensures data integrity, making it ideal for large-scale applications.

3. Describe the role of SQL in managing relational databases.

**ANS**

SQL manages relational databases by allowing users to:

* **Define Data** (CREATE, ALTER, DROP) – Structure databases.
* **Query Data** (SELECT) – Retrieve specific information.
* **Manipulate Data** (INSERT, UPDATE, DELETE) – Modify records.
* **Control Access** (GRANT, REVOKE) – Manage permissions.
* **Ensure Integrity** (constraints like PRIMARY KEY, FOREIGN KEY) – Maintain data consistency.
* **Manage Transactions** (COMMIT, ROLLBACK) – Ensure reliable operations.

4. What are the key features of SQL?

**ANS**

Key features of SQL:

* **Data Definition** – Create, alter, and delete database structures.
* **Data Manipulation** – Insert, update, and delete records.
* **Data Querying** – Retrieve data using SELECT.
* **Data Integrity** – Enforce rules with constraints (PRIMARY KEY, FOREIGN KEY).
* **Transaction Control** – Manage transactions (COMMIT, ROLLBACK).
* **Security** – Control access with GRANT and REVOKE.

**SQL Syntax**

1. What are the basic components of SQL syntax?

**ANS**

**DDL (Data Definition Language)** – CREATE, ALTER, DROP (structure management).

**DML (Data Manipulation Language)** – INSERT, UPDATE, DELETE (data handling).

**DQL (Data Query Language)** – SELECT (data retrieval).

**DCL (Data Control Language)** – GRANT, REVOKE (access control).

**TCL (Transaction Control Language)** – COMMIT, ROLLBACK (transaction management).

2. Write the general structure of an SQL SELECT statement.

**ANS**

SELECT column1, column2 FROM table\_name WHERE condition;

3. Explain the role of clauses in SQL statements.

**ANS**

**SELECT** – Specifies columns to retrieve.

**FROM** – Defines the source table.

**WHERE** – Filters data based on conditions.

**ORDER BY** – Sorts results in ascending or descending order.

**GROUP BY** – Groups rows with similar values.

**HAVING** – Filters grouped data.

**JOIN** – Combines data from multiple tables.

**LIMIT** – Restricts the number of rows returned.

**SQL Constraints**

1. What are constraints in SQL? List and explain the different types of constraints.

**ANS**

**PRIMARY KEY** – Uniquely identifies each record, no NULL values.

**FOREIGN KEY** – Establishes a relationship between tables.

**UNIQUE** – Ensures values in a column are distinct.

**CHECK** – Enforces conditions on column values.

**NOT NULL** – Prevents empty values in a column.

**DEFAULT** – Assigns a default value if none is provided.

**AUTO\_INCREMENT** – Automatically generates unique numbers.

2. How do PRIMARY KEY and FOREIGN KEY constraints differ?

**ANS**

**PRIMARY KEY** uniquely identifies each record in a table, does not allow NULL values, and must be unique.

**FOREIGN KEY** establishes a relationship between two tables, can have duplicate values, and allows NULL values.

3. What is the role of NOT NULL and UNIQUE constraints?

**ANS**

**NOT NULL** – Ensures a column always has a value, preventing incomplete data.

**UNIQUE** – Ensures no duplicate values exist in a column, maintaining data uniqueness.

**Main SQL Commands and Sub-commands**

1. Define the SQL Data Definition Language (DDL).

**ANS**

DDL consists of commands (CREATE, ALTER, DROP, TRUNCATE) used to define, modify, and delete database structures.

2. Explain the CREATE command and its syntax.

**ANS**

CREATE TABLE table\_name (

column1 datatype constraints,

column2 datatype constraints

);

3. What is the purpose of specifying data types and constraints during table creation?

**ANS**

**Data Types** define the kind of data stored (e.g., INT, VARCHAR).

**Constraints** ensure data integrity (e.g., PRIMARY KEY, NOT NULL).

**ALTER Command**

1. What is the use of the ALTER command in SQL?

**ANS**

The ALTER command is used to modify an existing table's structure, such as adding, modifying, or deleting columns, and updating constraints.

2. How can you add, modify, and drop columns from a table using ALTER?

**ANS**

**Add a Column:**

ALTER TABLE table\_name ADD column\_name datatype;

**Modify a Column:**

ALTER TABLE table\_name MODIFY column\_name new\_datatype;

**Drop a Column:**

ALTER TABLE table\_name DROP COLUMN column\_name;

**DROP Command**

1. What is the function of the DROP command in SQL?

**ANS**

The DROP command is used to permanently delete a database object, such as a table, index, or database, removing all its data and structure.

**Syntax:**

DROP TABLE table\_name;

2. What are the implications of dropping a table from a database?

**ANS**

Permanently deletes the table and its data.

Cannot be undone unless a backup exists.

Removes all associated constraints, indexes, and relationships.

May cause issues if other tables reference it via foreign keys.

**Data Manipulation Language (DML)**

1. Define the INSERT, UPDATE, and DELETE commands in SQL.

**ANS**

**INSERT** – Adds new records to a table.

INSERT INTO table\_name (column1, column2) VALUES (value1, value2);

**UPDATE** – Modifies existing records in a table.

UPDATE table\_name SET column1 = value1 WHERE condition;

**DELETE** – Removes records from a table.

DELETE FROM table\_name WHERE condition;

2. What isthe importance of the WHERE clause in UPDATE and DELETE operations?

**ANS**

Ensures that only specific rows are updated or deleted.

Prevents unintended changes to all records.

Helps maintain data integrity and accuracy.

**Data Query Language (DQL)**

1. What is the SELECT statement, and how is it used to query data?

**ANS**

The SELECT statement is used to retrieve data from one or more tables in a database.

**Syntax:**

SELECT column1, column2 FROM table\_name WHERE condition;

2. Explain the use of the ORDER BY and WHERE clauses in SQL queries.

**ANS**

**WHERE** – Filters records based on a condition.

SELECT \* FROM employees WHERE age > 30;

**ORDER BY** – Sorts results in ascending (ASC) or descending (DESC) order.

SELECT \* FROM employees ORDER BY name ASC;

**Data Control Language (DCL)**

1. What is the purpose of GRANT and REVOKE in SQL?

**ANS**

**GRANT** – Assigns specific privileges to users or roles.

**REVOKE** – Removes previously granted privileges.

2. How do you manage privileges using these commands?

**ANS**

**Granting Privileges:**

GRANT SELECT, INSERT ON table\_name TO user\_name;

**Revoking Privileges:**

REVOKE INSERT ON table\_name FROM user\_name;

**Transaction Control Language (TCL)**

1. What is the purpose of the COMMIT and ROLLBACK commands in SQL?

**ANS**

**COMMIT** – Saves all changes made in a transaction permanently.

COMMIT;

**ROLLBACK** – Reverts changes made in a transaction if an error occurs.

ROLLBACK;

2. Explain how transactions are managed in SQL databases.

**ANS**

Transactions ensure data consistency and integrity.

**BEGIN TRANSACTION** starts a transaction.

**COMMIT** saves changes, while **ROLLBACK** undoes them.

**SAVEPOINT** allows partial rollbacks within a transaction.  
This ensures safe and controlled data modifications.

**SQL Joins**

1.Explain the concept of JOIN in SQL. What is the difference between INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN?

**ANS**

**INNER JOIN** – Returns only matching records from both tables.

**LEFT JOIN** – Returns all records from the left table and matching records from the right.

**RIGHT JOIN** – Returns all records from the right table and matching records from the left.

**FULL OUTER JOIN** – Returns all records from both tables, with NULLs where there is no match.

2. How are joins used to combine data from multiple tables?

**ANS**

Joins establish relationships between tables using foreign keys, enabling queries that retrieve and merge relevant data efficiently.

**SQL Group By**

1. What is the GROUP BY clause in SQL? How is it used with aggregate functions?

**ANS**

**GROUP BY** groups rows with the same values in specified columns.

Often used with aggregate functions like COUNT(), SUM(), AVG(), MAX(), and MIN().

**Example:**

SELECT department, COUNT(\*) FROM employees GROUP BY department;

2. Explain the difference between GROUP BY and ORDER BY.

**ANS**

**GROUP BY** groups data and is used with aggregate functions.

**ORDER BY** sorts results in ascending (ASC) or descending (DESC) order.

**Example:**

SELECT \* FROM employees ORDER BY salary DESC;

**SQL Stored Procedure**

1. What is a stored procedure in SQL, and how does it differ from a standard SQL query?

**ANS**

A **stored procedure** is a precompiled SQL script stored in the database that can be executed multiple times.

**Example:**

CREATE PROCEDURE GetEmployees()

AS

SELECT \* FROM employees;

**Stored Procedure**: Predefined, reusable, and optimized for performance.

**Standard Query**: A single-use SQL statement executed manually.

2. Explain the advantages of using stored procedures.

**ANS**

**Performance** – Precompiled, reducing execution time.

**Reusability** – Can be executed multiple times without rewriting.

**Security** – Restricts direct table access and supports user permissions.

**Reduced Network Traffic** – Executes on the server, reducing data transfer.

**SQL View**

1. What is a view in SQL, and how is it different from a table?

**ANS**

A **view** is a virtual table based on an SQL query, showing selected data from one or more tables. Unlike a table, it doesn't store data but retrieves it dynamically.

**Example:**

CREATE VIEW EmployeeView AS

SELECT name, department FROM employees;

2. Explain the advantages of using views in SQL databases.

**ANS**

**Data Security** – Restricts access to sensitive columns.

**Simplifies Queries** – Stores complex queries for easy reuse.

**Data Consistency** – Ensures a consistent representation of data.

**Performance Optimization** – Improves readability and reduces redundancy.

**SQL Triggers**

1. What is a trigger in SQL? Describe its types and when they are used.

**ANS**

A **trigger** is an automatic database action executed in response to specific events (INSERT, UPDATE, DELETE) on a table.

**Types of Triggers:**

**BEFORE Trigger** – Executes before the event occurs.

**AFTER Trigger** – Executes after the event occurs.

**INSTEAD OF Trigger** – Replaces the execution of an event, mainly for views.

**Example:**

CREATE TRIGGER before\_insert\_employee

BEFORE INSERT ON employees

FOR EACH ROW

SET NEW.created\_at = NOW();

2. Explain the difference between INSERT, UPDATE, and DELETE triggers.

**ANS**

**INSERT Trigger** – Activates when a new row is added.

**UPDATE Trigger** – Activates when a row is modified.

**DELETE Trigger** – Activates when a row is removed.

**Introduction to PL/SQL**

1. What is PL/SQL, and how does it extend SQL's capabilities?

**ANS**

**What is PL/SQL & How It Extends SQL?**

PL/SQL (Procedural Language/SQL) is Oracle’s extension of SQL that adds procedural programming features like **variables, loops, conditions, and exception handling**.

**How It Extends SQL:**

* Supports **control structures** (IF, LOOP) for complex logic.
* Allows **multiple SQL statements** to execute as a block.
* Provides **exception handling** for error management.
* Enables **stored procedures, functions, and triggers** for reusability and automation.

**Example:**

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Hello, PL/SQL!');

END;

This makes SQL more powerful for advanced database programming.

2. List and explain the benefits of using PL/SQL.

**ANS**

**Benefits of Using PL/SQL**

1. **Improved Performance** – Reduces network traffic by executing multiple SQL statements as a block.
2. **Code Reusability** – Supports stored procedures, functions, and packages for modular programming.
3. **Better Security** – Restricts direct access to tables and allows role-based permissions.
4. **Exception Handling** – Provides robust error management with structured exception handling.
5. **Support for Triggers & Cursors** – Enhances automation and data manipulation.
6. **Portability** – Can run on any Oracle database without modification.
7. **Integration with SQL** – Combines procedural logic with SQL queries for powerful data processing.

These features make PL/SQL ideal for **efficient, secure, and scalable** database applications.

**PL/SQL Control Structures**

1. What are control structures in PL/SQL? Explain the IF-THEN and LOOP controlstructures.

**ANS**

**Control Structures in PL/SQL**

Control structures control the flow of execution in PL/SQL. The main types are **conditional statements (IF-THEN)** and **loops (LOOP, WHILE, FOR).**

**1. IF-THEN Control Structure (Conditional Execution)**

* Executes statements based on a condition.

**Example:**

IF salary > 10000 THEN

DBMS\_OUTPUT.PUT\_LINE('High Salary');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Low Salary');

END IF;

**2. LOOP Control Structures (Repetitive Execution)**

* **LOOP** – Runs repeatedly until an EXIT condition is met.
* **WHILE LOOP** – Runs **while** a condition is TRUE.

**Example (LOOP):**

LOOP

DBMS\_OUTPUT.PUT\_LINE('Iteration: ' || count);

count := count + 1;

EXIT WHEN count > 5;

END LOOP;

These structures help automate and control program execution efficiently.

2. How do control structures in PL/SQL help in writing complex queries?

**ANS**

**How Control Structures Help in Writing Complex Queries**

Control structures in PL/SQL improve query logic by enabling:

1. **Conditional Execution** – Using IF-THEN, queries can adapt dynamically based on conditions.
2. **Looping for Repetitive Tasks** – LOOP, WHILE, and FOR loops help automate repetitive operations, such as processing multiple records.
3. **Exception Handling** – Ensures smooth execution by handling errors with EXCEPTION.
4. **Improved Modularity** – Complex logic can be structured into **procedures and functions**, enhancing maintainability.

These features make PL/SQL more **efficient, flexible, and scalable** for handling complex database operations.

**SQL Cursors**

1. What is a cursor in PL/SQL? Explain the difference between implicit and explicit cursors.

**ANS**

**Cursor in PL/SQL**

A **cursor** is a pointer to a result set of a query, allowing row-by-row processing.

**1. Implicit Cursor**

* Automatically created by PL/SQL for SELECT, INSERT, UPDATE, or DELETE statements.
* No need for explicit declaration or handling.
* Example:
* BEGIN
* UPDATE employees SET salary = salary \* 1.1 WHERE department\_id = 10;
* END;

**2. Explicit Cursor**

* Defined by the user for handling complex queries with multiple rows.
* Requires declaration, opening, fetching, and closing.
* Example:
* DECLARE
* CURSOR emp\_cursor IS SELECT name, salary FROM employees;
* BEGIN
* OPEN emp\_cursor;
* FETCH emp\_cursor INTO emp\_name, emp\_salary;
* CLOSE emp\_cursor;
* END;

**Key Difference:**

* **Implicit Cursors** – Managed automatically, used for single-row operations.
* **Explicit Cursors** – User-controlled, used for multi-row operations.

2. When would you use an explicit cursor over an implicit one?

**ANS**

**When to Use an Explicit Cursor Over an Implicit Cursor**

You should use an **explicit cursor** when:

1. **Handling Multiple Rows** – When a query returns multiple rows and needs row-by-row processing.
2. **Better Control** – Explicit cursors allow **opening, fetching, and closing** manually.
3. **Performance Optimization** – Useful for large datasets where fetching rows one at a time is necessary.
4. **Complex Business Logic** – When additional operations are needed between row fetches.

**Example Use Case:**  
Fetching employee details and processing them individually.

DECLARE

CURSOR emp\_cursor IS SELECT name, salary FROM employees;

BEGIN

OPEN emp\_cursor;

FETCH emp\_cursor INTO emp\_name, emp\_salary;

CLOSE emp\_cursor;

END;

Implicit cursors are preferred for **single-row queries** and simpler operations.

**Rollback and Commit Savepoint**

1. Explain the concept of SAVEPOINT in transaction management. How do ROLLBACK and COMMIT interact with savepoints?

**ANS**

**SAVEPOINT in Transaction Management**

A **SAVEPOINT** marks a specific point within a transaction, allowing partial rollbacks. It helps maintain control over transaction flow.

**Interaction with ROLLBACK and COMMIT**

* **COMMIT**: Saves all changes permanently, removing all savepoints.
* **ROLLBACK TO SAVEPOINT**: Undoes only the changes made after a specific savepoint, keeping the earlier part of the transaction intact.
* **ROLLBACK (without SAVEPOINT)**: Reverts the entire transaction.

**Example:**

BEGIN;

UPDATE employees SET salary = salary \* 1.1 WHERE department\_id = 10;

SAVEPOINT sp1;

UPDATE employees SET salary = salary \* 1.2 WHERE department\_id = 20;

ROLLBACK TO sp1; -- Only reverses the second update

COMMIT;

This keeps **department 10’s update** but **reverts department 20’s update**.

2. When is it useful to use savepoints in a database transaction?

**ANS**

**When to Use Savepoints in a Database Transaction**

Savepoints are useful in the following scenarios:

1. **Partial Rollback** – When only a part of a transaction needs to be undone without rolling back the entire transaction.
2. **Error Handling** – If an error occurs after a savepoint, you can roll back to that point instead of restarting the entire transaction.
3. **Long Transactions** – Helps manage complex transactions by breaking them into smaller, manageable sections.
4. **Conditional Execution** – Useful when different parts of a transaction may need to be committed or rolled back based on specific conditions.

**Example:**

BEGIN;

UPDATE accounts SET balance = balance - 500 WHERE id = 1;

SAVEPOINT sp1;

UPDATE accounts SET balance = balance + 500 WHERE id = 2;

IF balance < 0 THEN ROLLBACK TO sp1; END IF;

COMMIT;

Here, **sp1** ensures that if the second update causes an issue, only that part is rolled back.

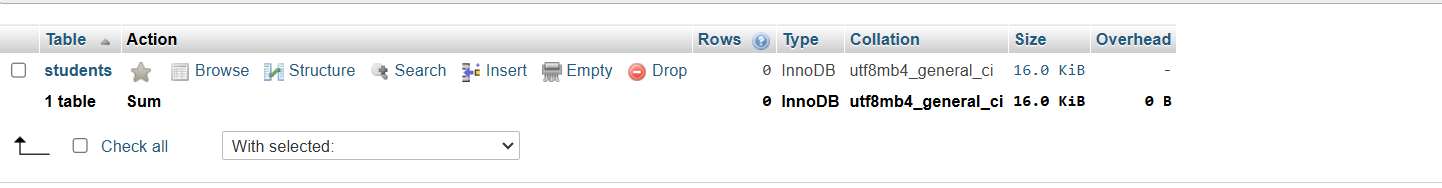
**LAB EXERCISES**

**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.

**ANS**

CREATE TABLE students ( student\_id INT PRIMARY KEY AUTO\_INCREMENT, student\_name VARCHAR(100) NOT NULL, age INT NOT NULL, class VARCHAR(20) NOT NULL, address TEXT );



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

**ANS**

INSERT INTO students (student\_name, age, class, address) VALUES

('John Doe', 15, '10A', '123 Main St'),

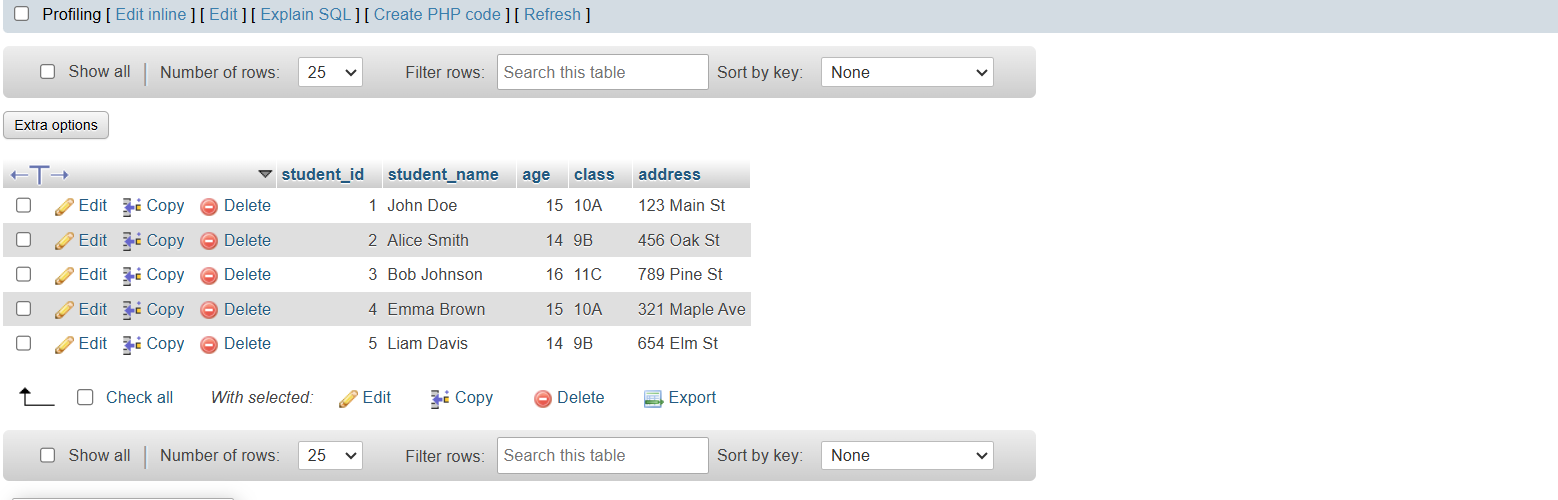
('Alice Smith', 14, '9B', '456 Oak St'),

('Bob Johnson', 16, '11C', '789 Pine St'),

('Emma Brown', 15, '10A', '321 Maple Ave'),

('Liam Davis', 14, '9B', '654 Elm St');

SELECT \* FROM students;

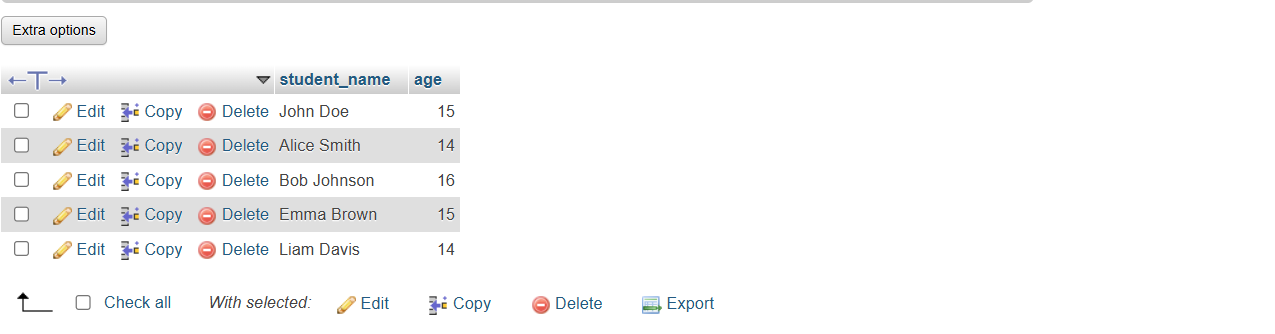


**SQL Syntax**

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

**ANS**

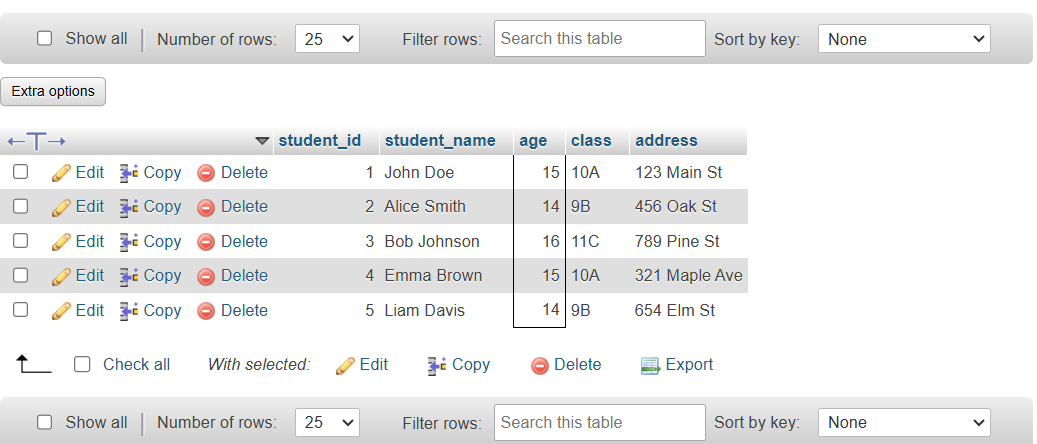
SELECT student\_name, age FROM students;



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

**ANS**

SELECT \* FROM students WHERE age > 10;

****

**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).

**ANS**

CREATE TABLE teachers (

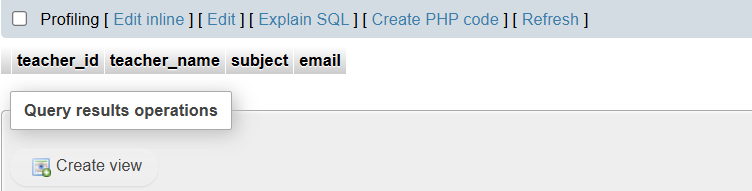
teacher\_id INT PRIMARY KEY AUTO\_INCREMENT,

teacher\_name VARCHAR(100) 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.

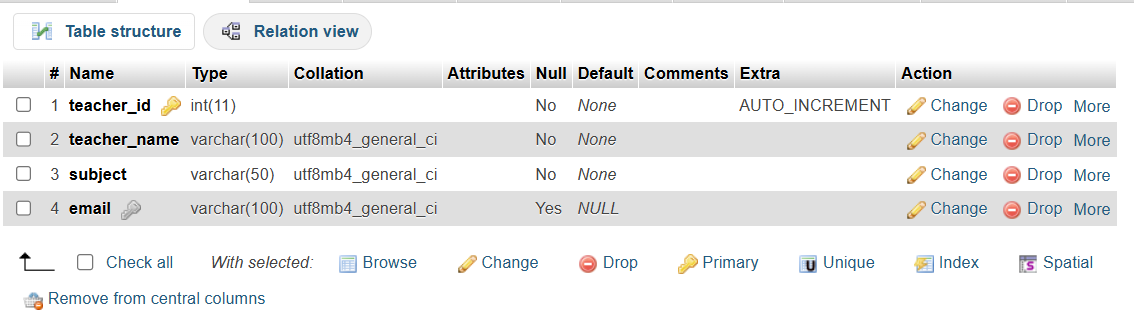
**ANS**

ALTER TABLE students

ADD COLUMN teacher\_id INT,

ADD CONSTRAINT fk\_teacher

FOREIGN KEY (teacher\_id) REFERENCES teachers(teacher\_id);



**Main SQL Commands and Sub-commands**

Lab 1: Create a table courses with columns: course\_id, course\_name, and course\_credits. Set the course\_id as the primary key.

**ANS**

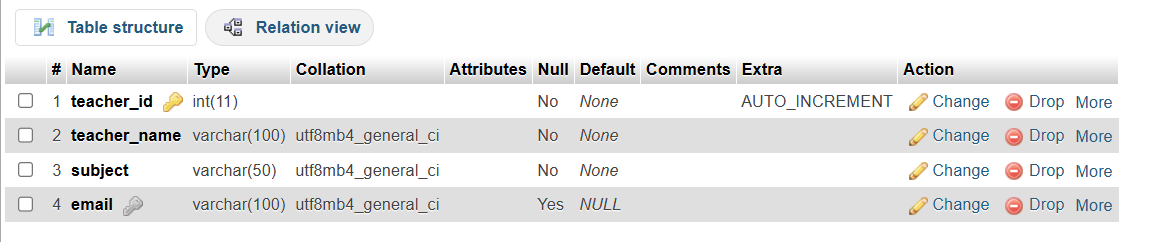
CREATE TABLE courses (

course\_id INT PRIMARY KEY AUTO\_INCREMENT,

course\_name VARCHAR(100) NOT NULL,

course\_credits INT NOT NULL

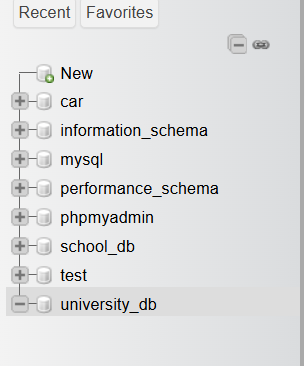
);



Lab 2: Use the CREATE command to create a database university\_db

**ANS**

CREATE DATABASE university\_db;



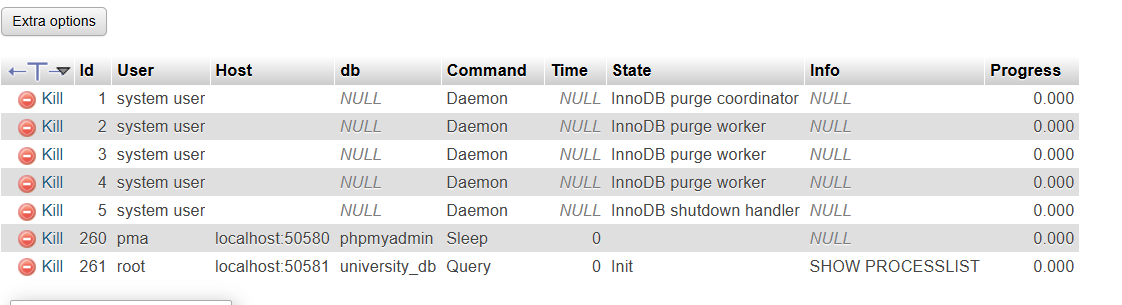
**ALTER Command**

Lab 1: Modify the courses table by adding a column course\_duration using the ALTER command.

**ANS**

ALTER TABLE courses

ADD COLUMN course\_duration INT;



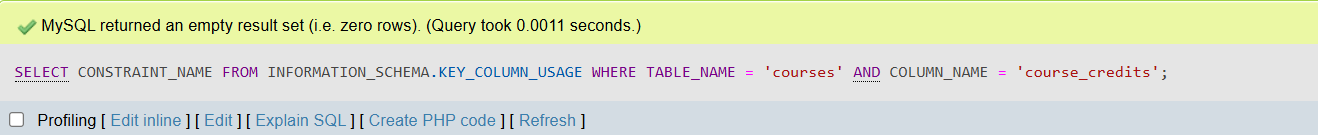
Lab 2: Drop the course\_credits column from the courses table

**ANS**

SELECT CONSTRAINT\_NAME

FROM INFORMATION\_SCHEMA.KEY\_COLUMN\_USAGE

WHERE TABLE\_NAME = 'courses' AND COLUMN\_NAME = 'course\_credits';

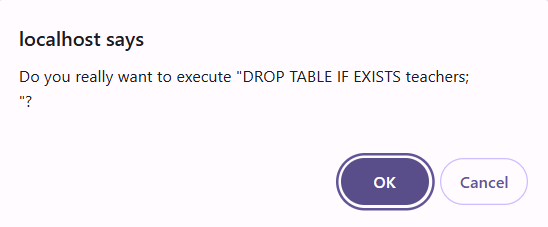


**DROP Command**

Lab 1: Drop the teachers table from the school\_db database.

**ANS**

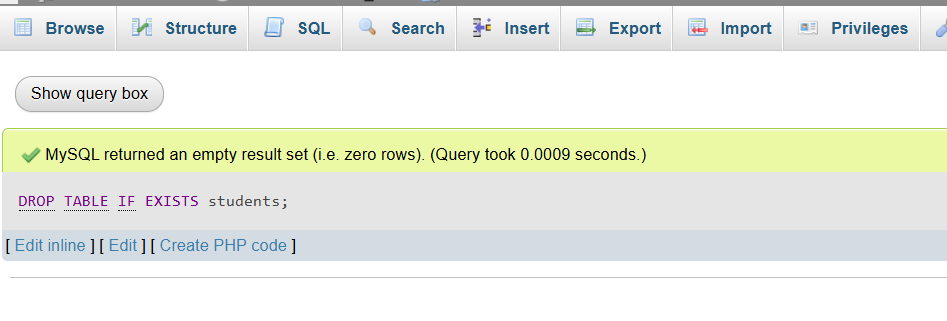
DROP TABLE IF EXISTS teachers;



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

**ANS**

DROP TABLE IF EXISTS students;



**Data Manipulation Language**

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

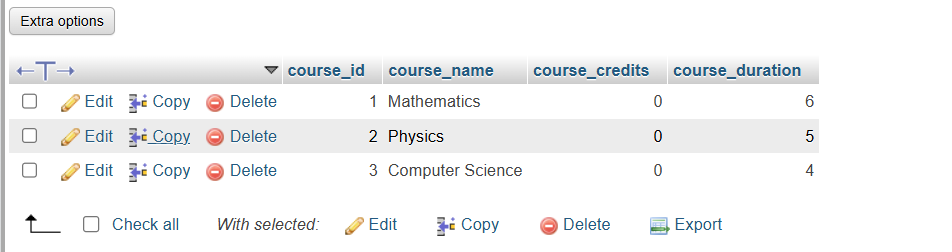
**ANS**

INSERT INTO courses (course\_id, course\_name, course\_duration) VALUES

(1, 'Mathematics', 6),

(2, 'Physics', 5),

(3, 'Computer Science', 4);



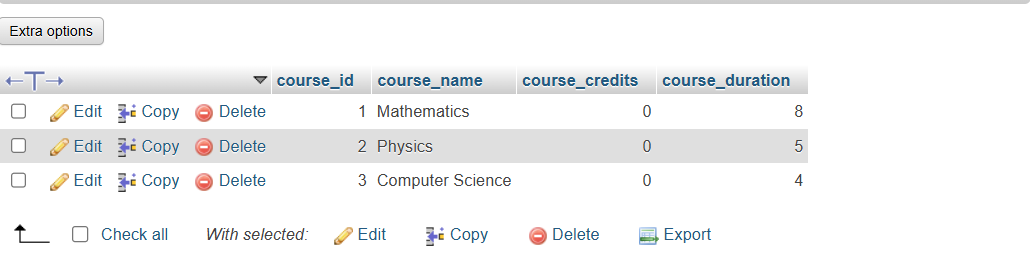
Lab 2: Update the course duration of a specific course using the UPDATE command.

**ANS**

UPDATE courses

SET course\_duration = 8

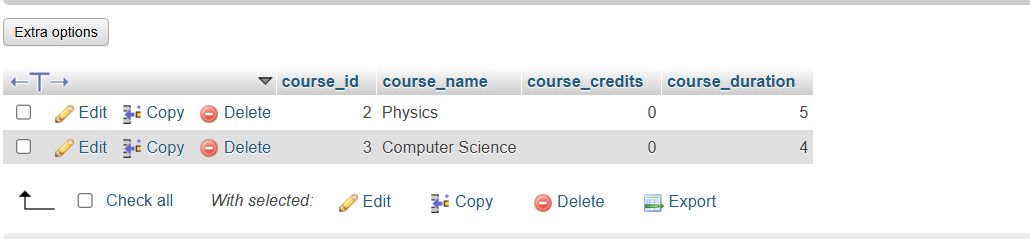
WHERE course\_name = 'Mathematics';



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

**ANS**

DELETE FROM courses WHERE course\_id = 1;

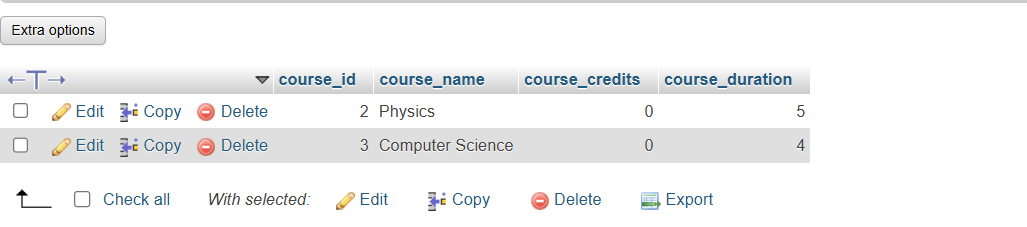
****

**Data Query Language (DQL)**

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

**ANS**

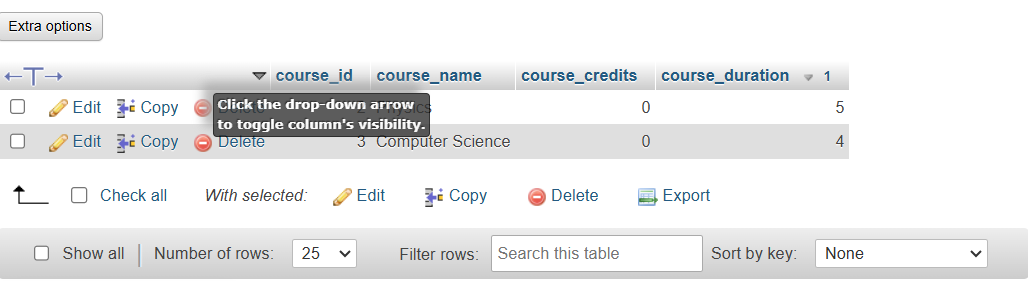
SELECT \* FROM courses;



Lab 2: Sort the courses based on course\_duration in descending order using ORDER BY.

**ANS**

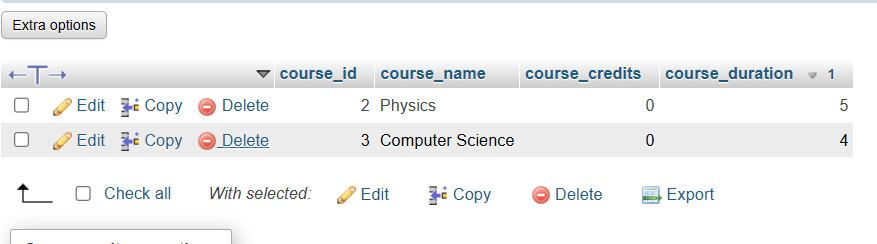
SELECT \* FROM courses ORDER BY course\_duration DESC;



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

**ANS**

SELECT \* FROM courses ORDER BY course\_duration DESC LIMIT 2;



**Data Control Language**

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

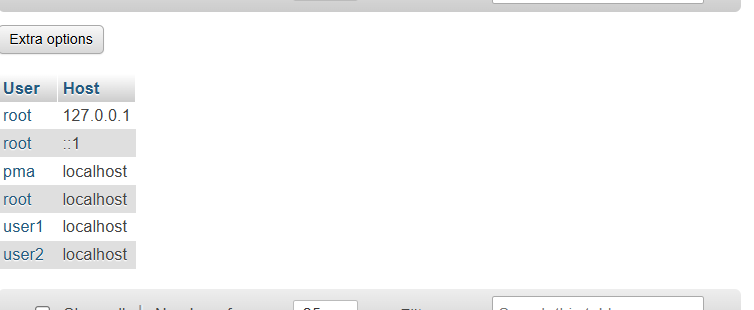
**ANS**

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

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

GRANT SELECT ON school\_db.courses TO 'user1'@'localhost';

FLUSH PRIVILEGES;



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

**ANS**

REVOKE INSERT ON school\_db.courses FROM 'user1'@'localhost';

GRANT INSERT ON school\_db.courses TO 'user2'@'localhost';

FLUSH PRIVILEGES;



**Transaction Control Language**

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

**ANS**

START TRANSACTION;

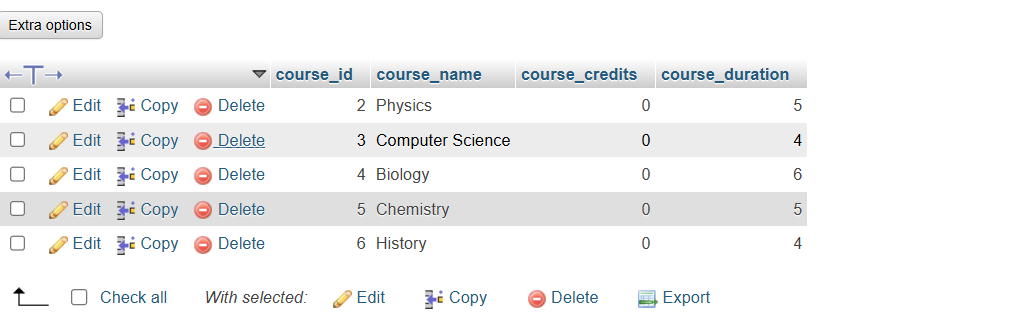
INSERT INTO courses (course\_name, course\_duration) VALUES

('Biology', 6),

('Chemistry', 5),

('History', 4);

COMMIT;



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

**ANS**

START TRANSACTION;

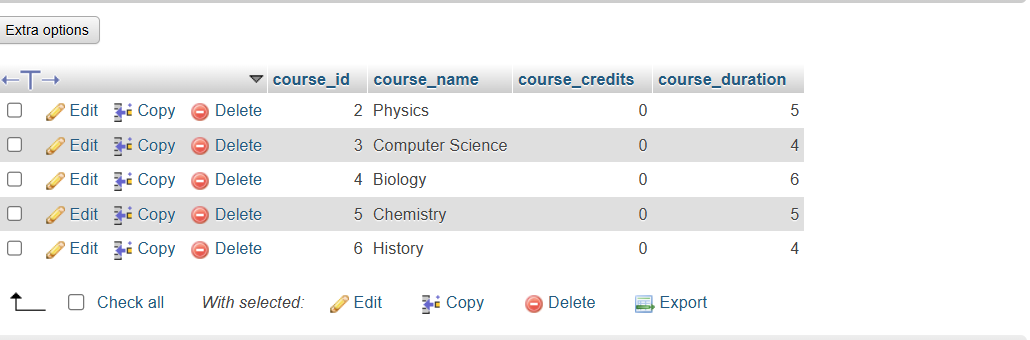
INSERT INTO courses (course\_name, course\_duration) VALUES

('Geography', 3),

('Philosophy', 4),

('Psychology', 5);

ROLLBACK;



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

**ANS**

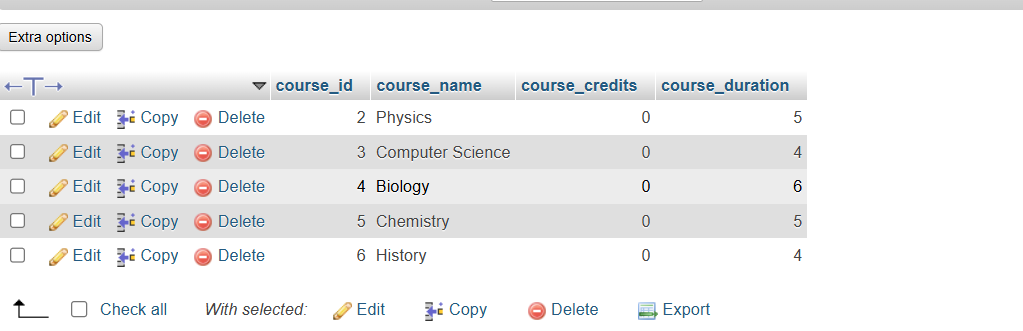
START TRANSACTION;

SAVEPOINT before\_update;

UPDATE courses SET course\_duration = 7 WHERE course\_name = 'Mathematics';

ROLLBACK TO SAVEPOINT before\_update;

COMMIT;



**SQL Joins**

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

**ANS**

CREATE TABLE departments (

department\_id INT PRIMARY KEY,

department\_name VARCHAR(100) NOT NULL

);

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

employee\_name VARCHAR(100) NOT NULL,

department\_id INT,

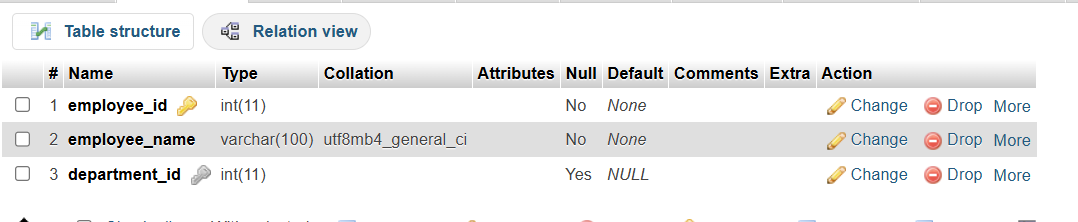
FOREIGN KEY (department\_id) REFERENCES departments(department\_id)

);

SELECT employees.employee\_id, employees.employee\_name, departments.department\_name

FROM employees

INNER JOIN departments ON employees.department\_id = departments.department\_id;



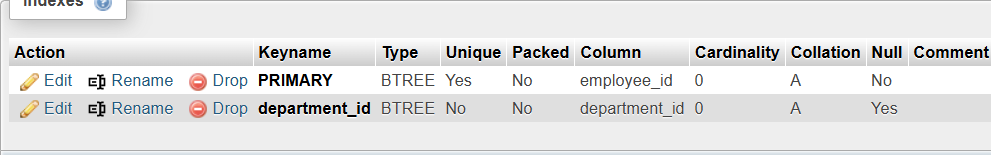
Lab 2: Use a LEFT JOIN to show all departments, even those without employees.

**ANS**

SELECT departments.department\_id, departments.department\_name, employees.employee\_id, employees.employee\_name

FROM departments

LEFT JOIN employees ON departments.department\_id = employees.department\_id;



**SQL Group By**

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

**ANS**

SELECT department\_id, COUNT(employee\_id) AS employee\_count

FROM employees

GROUP BY department\_id;



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

**ANS**

SELECT department\_id, AVG(salary) AS average\_salary

FROM employees

GROUP BY department\_id;



**SQL Stored Procedure**

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

**ANS**

DELIMITER $$

CREATE PROCEDURE GetEmployeesByDepartment(IN dept\_id INT)

BEGIN

SELECT \* FROM employees WHERE department\_id = dept\_id;

END $$

DELIMITER ;



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

**ANS**

DELIMITER $$

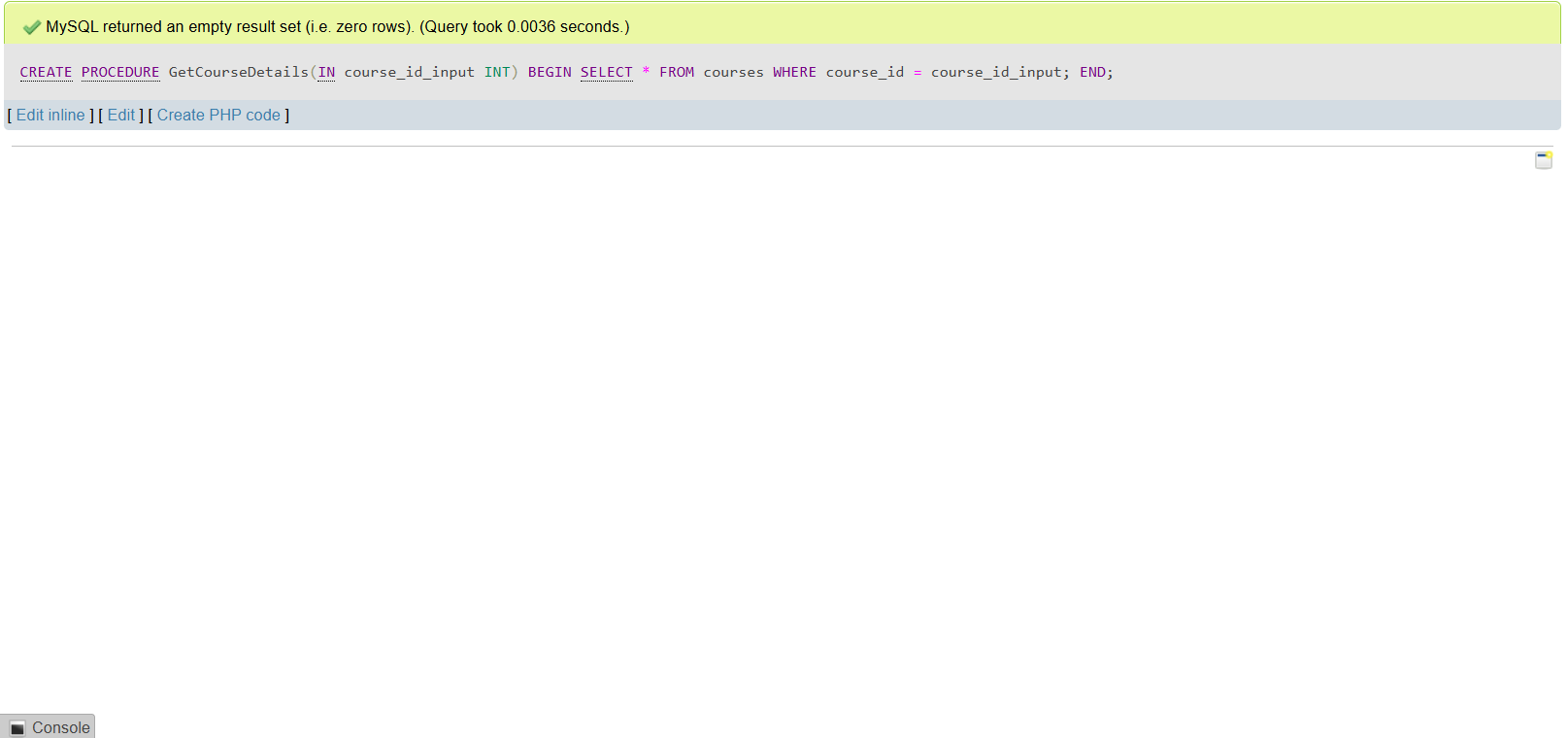
CREATE PROCEDURE GetCourseDetails(IN course\_id\_input INT)

BEGIN

SELECT \* FROM courses WHERE course\_id = course\_id\_input;

END $$

DELIMITER ;



**SQL View**

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

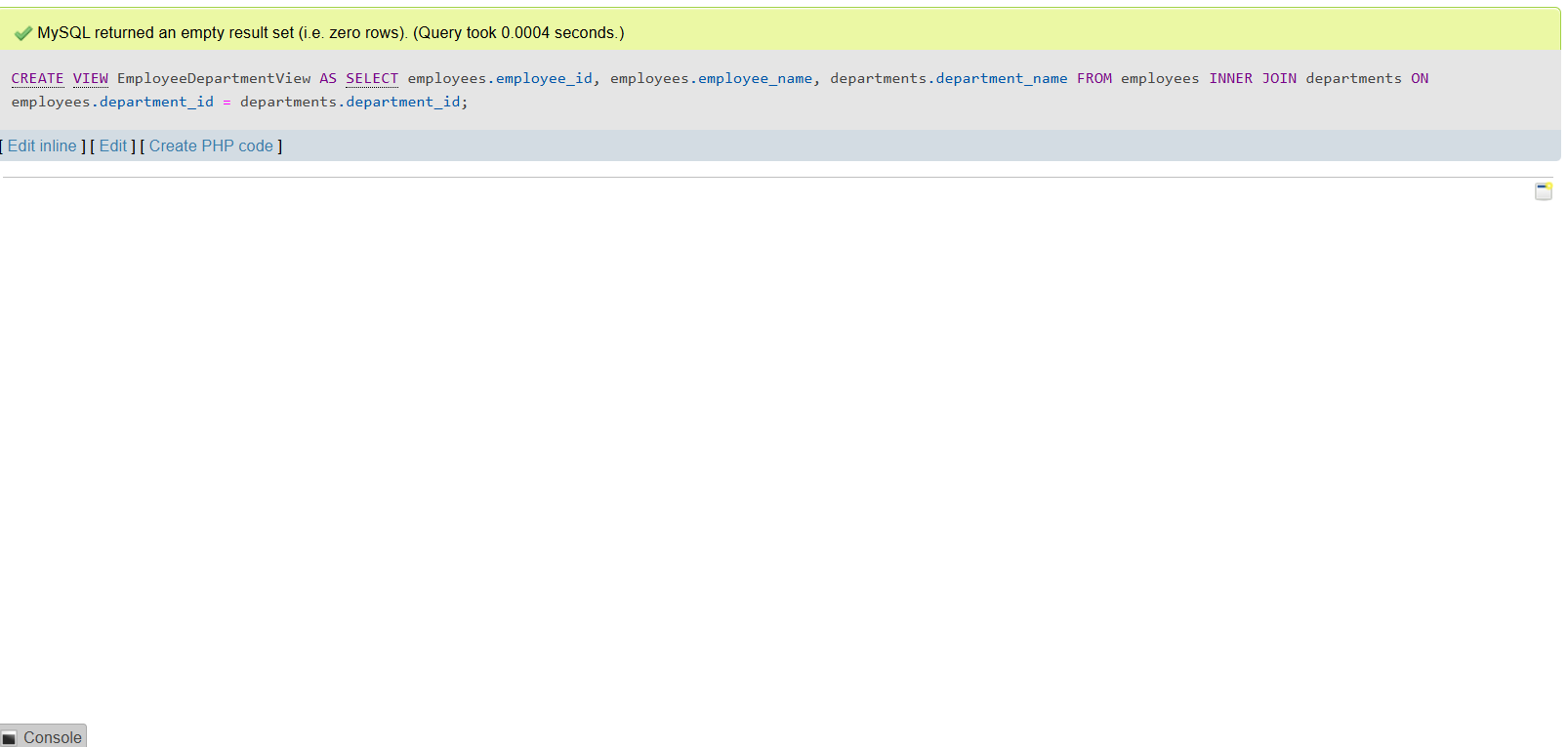
**ANS**

CREATE VIEW EmployeeDepartmentView AS

SELECT employees.employee\_id, employees.employee\_name, departments.department\_name

FROM employees

INNER JOIN departments ON employees.department\_id = departments.department\_id;



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

**ANS**

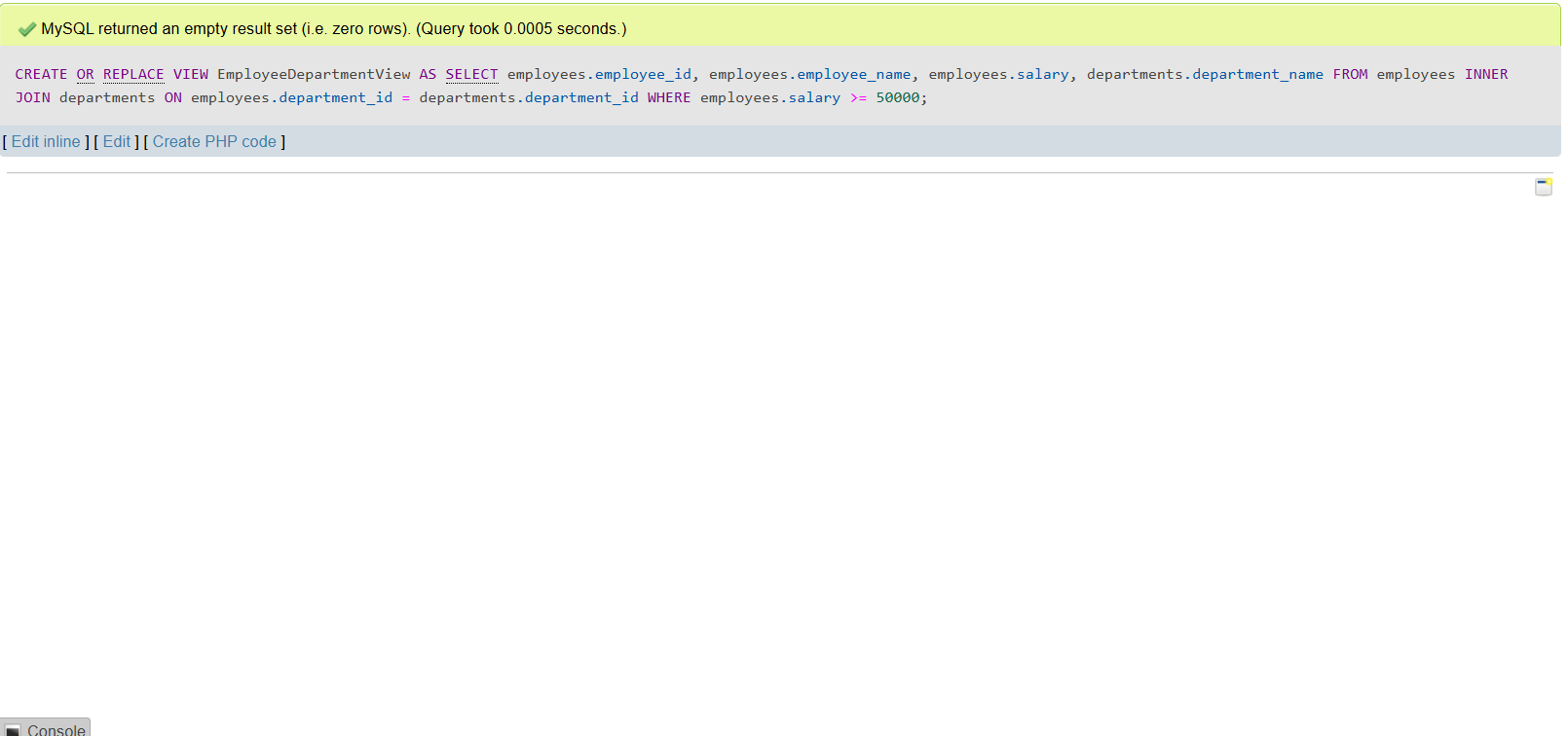
CREATE OR REPLACE VIEW EmployeeDepartmentView AS

SELECT employees.employee\_id, employees.employee\_name, employees.salary, departments.department\_name

FROM employees

INNER JOIN departments ON employees.department\_id = departments.department\_id

WHERE employees.salary >= 50000;



**SQL Triggers**

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

**ANS**

CREATE TABLE employee\_audit (

audit\_id INT AUTO\_INCREMENT PRIMARY KEY,

emp\_id INT,

emp\_name VARCHAR(100),

change\_type VARCHAR(50),

change\_time TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

DELIMITER $$

CREATE TRIGGER log\_employee\_addition

AFTER INSERT ON employees

FOR EACH ROW

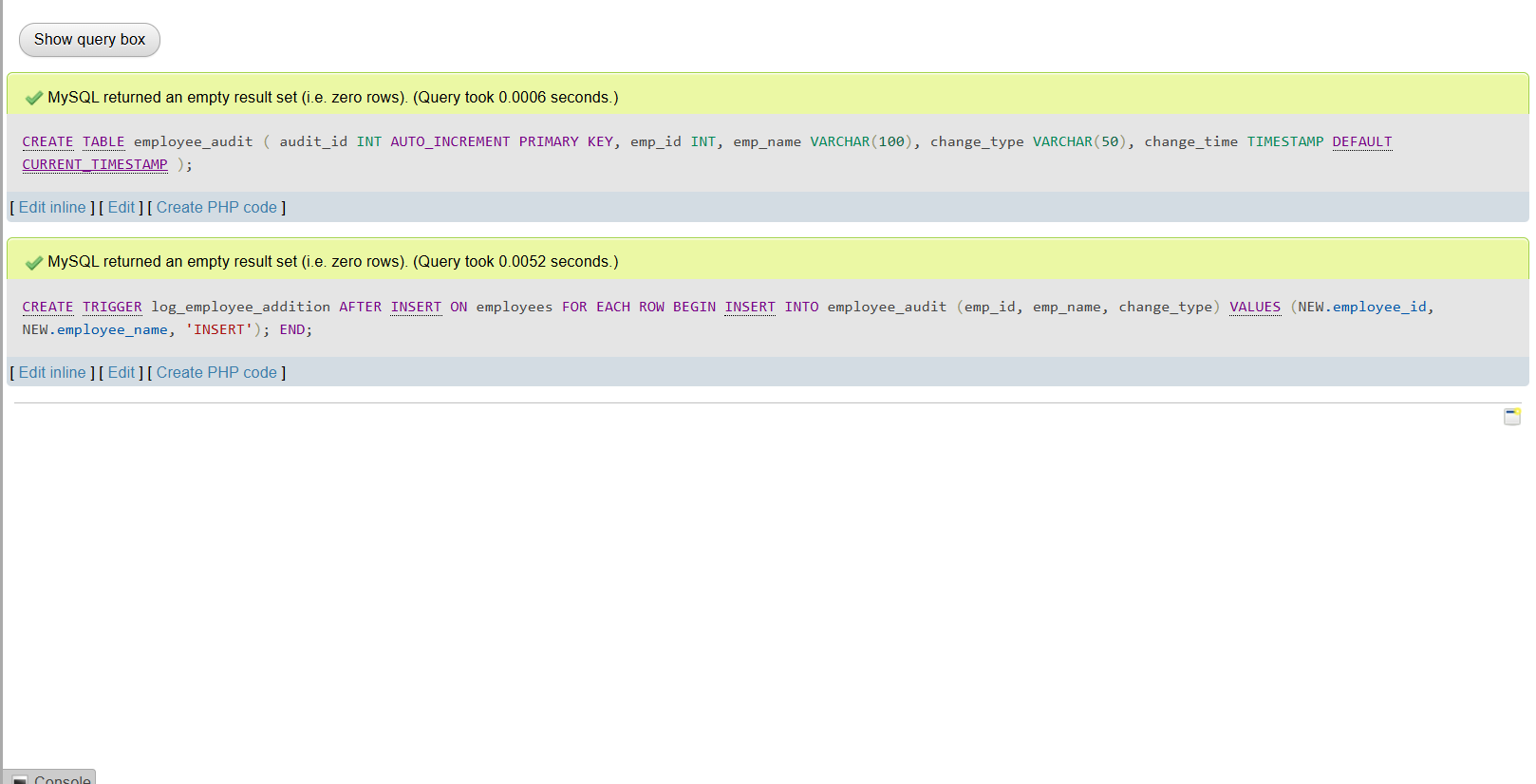
BEGIN

INSERT INTO employee\_audit (emp\_id, emp\_name, change\_type)

VALUES (NEW.employee\_id, NEW.employee\_name, 'INSERT');

END $$

DELIMITER ;



**Introduction to PL/SQL**

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

table.

**ANS**

DECLARE

total\_emp NUMBER;

BEGIN

SELECT COUNT(\*) INTO total\_emp FROM employees;

DBMS\_OUTPUT.PUT\_LINE('Total Employees: ' || total\_emp);

END;

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

**ANS**

DECLARE

total\_sales NUMBER;

BEGIN

SELECT SUM(order\_amount) INTO total\_sales FROM orders;

DBMS\_OUTPUT.PUT\_LINE('Total Sales: ' || total\_sales);

END;

**SQL Cursors**

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

**ANS**

DECLARE

CURSOR emp\_cursor IS SELECT employee\_id, first\_name, last\_name FROM employees;

v\_id employees.employee\_id%TYPE;

v\_fname employees.first\_name%TYPE;

v\_lname employees.last\_name%TYPE;

BEGIN

OPEN emp\_cursor;

LOOP

FETCH emp\_cursor INTO v\_id, v\_fname, v\_lname;

EXIT WHEN emp\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('ID: ' || v\_id || ', Name: ' || v\_fname || ' ' || v\_lname);

END LOOP;

CLOSE emp\_cursor;

END;

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

**ANS**

DECLARE

CURSOR course\_cursor IS SELECT course\_id, course\_name FROM courses;

v\_id courses.course\_id%TYPE;

v\_name courses.course\_name%TYPE;

BEGIN

OPEN course\_cursor;

LOOP

FETCH course\_cursor INTO v\_id, v\_name;

EXIT WHEN course\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Course ID: ' || v\_id || ', Course Name: ' || v\_name);

END LOOP;

CLOSE course\_cursor;

END;

**Rollback and Commit Savepoint**

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

**ANS**

BEGIN

SAVEPOINT before\_insert;

INSERT INTO employees (employee\_id, first\_name, last\_name, salary)

VALUES (999, 'John', 'Doe', 50000);

DBMS\_OUTPUT.PUT\_LINE('Record Inserted.');

ROLLBACK TO before\_insert;

DBMS\_OUTPUT.PUT\_LINE('Rollback to Savepoint Done.');

END;

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

**ANS**

START TRANSACTION;

SAVEPOINT before\_insert;

INSERT INTO employees (employee\_id, first\_name, last\_name, salary)

VALUES (1001, 'Alice', 'Smith', 60000);

COMMIT;

SAVEPOINT second\_point;

INSERT INTO employees (employee\_id, first\_name, last\_name, salary)

VALUES (1002, 'Bob', 'Johnson', 55000);

ROLLBACK TO second\_point;

COMMIT;d