**Module\_5 LAB EXERCISES**

**Introduction to SQL:**

**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:** To create a new database named school\_db and a table called students with the specified columns, you would use SQL commands.

First, create the database:

CREATE DATABASE school\_db;

Next, select the newly created database to work within it:

USE school\_db;

Finally, create the students table with the defined columns and their respective data types. It is recommended to set student\_id as the primary key for unique identification.

CREATE TABLE students (  
 student\_id INT PRIMARY KEY,  
 student\_name VARCHAR(100),  
 age INT,  
 class VARCHAR(50),  
 address VARCHAR(255)  
);

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

**Ans:** To insert five records into a students table and then retrieve all records from it, the following SQL statements are used:

1. Inserting Five Records:

Assuming a students table with columns like student\_id (integer), name (varchar), and grade (integer), the INSERT INTO statement is used:

INSERT INTO students (student\_id, name, grade) VALUES  
(1, 'Alice', 95),  
(2, 'Bob', 88),  
(3, 'Charlie', 72),  
(4, 'Diana', 91),  
(5, 'Eve', 85);

2. Retrieving All Records:

To retrieve all records from the students table, the SELECT statement is used:

SELECT \* FROM students;

**SQL Syntax:**

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

**Ans:** To retrieve specific columns, such as student\_name and age, from a table named students, the SELECT statement in SQL is used.

SELECT student\_name, age  
FROM students;

This query will return a result set containing only the student\_name and age columns for all records present in the students table. If you need to retrieve all columns, you would use the asterisk (\*) wildcard instead of specifying individual column names:

SELECT \*  
FROM students;

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

**Ans:** To retrieve all students whose age is greater than 10 from a table named students, the following SQL query can be used:

SELECT \*  
FROM students  
WHERE age > 10;

**SQL Constraints:**

**1: Create a table teachers with the following columns: teacher\_id (Primary Key), teacher\_name (NOT NULL), subject (NOT NULL), and email (UNIQUE).**

**Ans:** To create a table named teachers with the specified columns and constraints, the following SQL CREATE TABLE statement can be used:

CREATE TABLE teachers (  
 teacher\_id INT PRIMARY KEY,  
 teacher\_name VARCHAR(255) NOT NULL,  
 subject VARCHAR(255) NOT NULL,  
 email VARCHAR(255) UNIQUE  
);

**2: Implement a FOREIGN KEY constraint to relate the teacher\_id from the teachers table with the students table.**

**Ans:** To implement a FOREIGN KEY constraint relating teacher\_id in the students table to the teachers table, assuming teacher\_id is the primary key in the teachers table, you can use the ALTER TABLE statement.

1. Adding the teacher\_id column to the students table (if it doesn't exist):

If your students table does not already have a teacher\_id column, you need to add it first, ensuring its data type matches the teacher\_id in the teachers table.

ALTER TABLE students  
ADD COLUMN teacher\_id INT; -- Adjust data type as needed (e.g., BIGINT, UUID)

2. Adding the FOREIGN KEY constraint:

Once the teacher\_id column exists in the students table, you can add the FOREIGN KEY constraint.

ALTER TABLE students  
ADD CONSTRAINT fk\_teacher\_student *-- Optional: Give a meaningful name to the constraint*  
FOREIGN KEY (teacher\_id)  
REFERENCES teachers(teacher\_id);

**Main SQL Commands and Sub-commands (DDL):**

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

**Ans:** To create a table named courses with the specified columns and primary key, the following SQL CREATE TABLE statement can be used:

CREATE TABLE courses (  
 course\_id INT PRIMARY KEY,  
 course\_name VARCHAR(255),  
 course\_credits INT  
);

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

**Ans:** To create a database named "university\_db" using the CREATE command in SQL, the following syntax is used:

CREATE DATABASE university\_db;

This statement instructs the database management system (DBMS) to establish a new database container named university\_db.

**ALTER Command:**

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

**Ans:** To add a column named course\_duration to the courses table using the ALTER TABLE command, the following SQL statement can be used:

ALTER TABLE courses  
ADD COLUMN course\_duration INT;

**2: Drop the course\_credits column from the courses table.**

**Ans:** To drop the credits column from the course table in a database, the ALTER TABLE statement with the DROP COLUMN clause is used.

ALTER TABLE course  
DROP COLUMN credits;

**DROP Command:**

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

**Ans:** To drop the teachers table from the school\_db database, execute the SQL command DROP TABLE teachers; within a SQL client connected to the school\_db database.

DROP TABLE teachers;

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

**Ans:** To drop the students table from the school\_db database and verify its removal, execute the following SQL commands:

USE school\_db;  
DROP TABLE students;

To verify the table's removal, query the database's information schema or list the tables within school\_db:

USE school\_db;  
SHOW TABLES;

Alternatively, query the information\_schema.tables view:

SELECT table\_name  
FROM information\_schema.tables  
WHERE table\_schema = 'school\_db' AND table\_name = 'students';

**Data Manipulation Language (DML):**

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

**Ans:** To insert three records into a courses table using the INSERT command, the following SQL statement can be used, assuming the courses table has columns for course\_id, course\_name, and credits:

INSERT INTO courses (course\_id, course\_name, credits) VALUES  
(101, 'Introduction to Programming', 3),  
(102, 'Database Management Systems', 4),  
(103, 'Web Development Fundamentals', 3);

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

**Ans:** To update the course duration for a specific course, use the SQL UPDATE command with a SET clause to specify the new duration and a WHERE clause to identify the particular course by its ID or another unique identifier. For example: UPDATE Courses SET duration = 'new\_duration' WHERE course\_id = 'specific\_course\_id'; .

Example

Imagine you have a table named Courses with a course\_id and a duration column, and you want to change the duration of the course with course\_id 'CS101' to '4 months'.

UPDATE Courses

SET duration = '4 months'

WHER course\_id 'CS101' ;

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

**Ans:** To delete a course with a specific course\_id from the courses table using the DELETE command, the following SQL statement can be used:

DELETE FROM courses  
WHERE course\_id = 'your\_course\_id';

**Data Query Language (DQL):**

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

**Ans:** To retrieve all courses from a table named courses in a database using the SQL SELECT statement, the following syntax is used:

SELECT \* FROM courses;

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

**Ans:** To sort courses by course\_duration in descending order, use the ORDER BY clause with the DESC keyword in your SQL query.

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

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

**Ans:** To limit the results of a SELECT query to show only the top two courses using the LIMIT clause, the following SQL syntax can be used:

SELECT \*  
FROM Courses  
ORDER BY CourseScore DESC

LIMIT 2;

**Data Control Language (DCL):**

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

**Ans:** To create two new users, user1 and user2, and grant user1 SELECT permission on the courses table, the following SQL commands can be used.

*-- Create user1*  
CREATE USER 'user1'@'localhost' IDENTIFIED BY 'password\_for\_user1';  
  
*-- Create user2*  
CREATE USER 'user2'@'localhost' IDENTIFIED BY 'password\_for\_user2';  
  
*-- Grant SELECT permission on the 'courses' table to user1*  
GRANT SELECT ON your\_database\_name.courses TO 'user1'@'localhost';  
  
*-- Flush privileges to apply changes*  
FLUSH PRIVILEGES;

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

**Ans:** To revoke the INSERT permission from user1 and grant it to user2 on a specific table, the following SQL commands can be used. Replace your\_table\_name with the actual name of the table.

*-- Revoke INSERT permission from user1*  
REVOKE INSERT ON your\_table\_name FROM user1;  
  
*-- Grant INSERT permission to user2*  
**GRANT INSERT ON your\_table\_name TO user2;**

**Transaction Control Language (TCL):**

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

**Ans:** To insert a few rows into the courses table and permanently save these changes using COMMIT, execute the following SQL statements:

INSERT INTO courses (course\_id, course\_name, credits)  
VALUES  
 ('CS101', 'Introduction to Computer Science', 3),  
 ('MA201', 'Calculus I', 4),  
 ('ENGL102', 'Academic Writing', 3);  
  
COMMIT;

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

**Ans:** To insert additional rows and then use ROLLBACK to undo the last insert operation, follow these steps within a database transaction:

* **Start a Transaction:** Begin a new transaction to group your database operations. This ensures that all changes within the transaction are treated as a single, atomic unit.

BEGIN TRANSACTION;

* **Insert Rows:** Execute INSERT statements to add the desired rows to your table.

INSERT INTO YourTableName (Column1, Column2) VALUES ('ValueA', 'ValueB');  
 INSERT INTO YourTableName (Column1, Column2) VALUES ('ValueC', 'ValueD');

* **Perform the Last Insert Operation:** Execute the specific INSERT statement that you intend to undo.

INSERT INTO YourTableName (Column1, Column2) VALUES ('ValueE', 'ValueF');

* **Undo the Last Insert using ROLLBACK:** Use the ROLLBACK statement to revert all changes made since the BEGIN TRANSACTION statement, effectively undoing the last insert operation (and any preceding uncommitted inserts within the same transaction).

ROLLBACK;

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

**Ans:** To create a SAVEPOINT before updating the courses table and use it to roll back specific changes, follow these steps within a transaction:

* **Start a Transaction:** Initiate a transaction to ensure that all operations are treated as a single, atomic unit. This is crucial for using SAVEPOINT effectively.

START TRANSACTION;

* **Create the SAVEPOINT:** Before executing the UPDATE statement that you might need to roll back, create a SAVEPOINT and assign it a name. This marks a point in the transaction to which you can later return.

SAVEPOINT before\_courses\_update;

* **Perform Updates:** Execute the UPDATE statements on the courses table.

UPDATE courses  
 SET course\_name = 'Advanced Database Management'  
 WHERE course\_id = 101;  
  
 UPDATE courses  
 SET credits = 4  
 WHERE course\_id = 102;

* **Roll Back to the SAVEPOINT (if needed):** If you determine that the updates made after the SAVEPOINT need to be undone while keeping prior changes within the transaction, use ROLLBACK TO SAVEPOINT.

ROLLBACK TO SAVEPOINT before\_courses\_update;

* **Commit or Rollback the Transaction:** After rolling back to the SAVEPOINT (or if no rollback was needed), you must either COMMIT the entire transaction to make the remaining changes permanent or ROLLBACK the entire transaction to undo all changes since START TRANSACTION.

COMMIT;

OR

ROLLBACK;

**SQL Joins:**

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

**Ans:** To create the departments and employees tables, populate them with data, and then perform an INNER JOIN to display employees with their respective departments, follow these SQL steps: Create the departments table.

CREATE TABLE departments (  
 department\_id INT PRIMARY KEY,  
 department\_name VARCHAR(50) NOT NULL  
 );

Create the employees table.

CREATE TABLE employees (  
 employee\_id INT PRIMARY KEY,  
 first\_name VARCHAR(50) NOT NULL,  
 last\_name VARCHAR(50) NOT NULL,  
 department\_id INT,  
 FOREIGN KEY (department\_id) REFERENCES departments(department\_id)  
 );

Insert data into the departments table.

INSERT INTO departments (department\_id, department\_name) VALUES  
 (1, 'Human Resources'),  
 (2, 'Engineering'),  
 (3, 'Sales'),  
 (4, 'Marketing');

Insert data into the employees table.

INSERT INTO employees (employee\_id, first\_name, last\_name, department\_id) VALUES  
 (101, 'Alice', 'Smith', 2),  
 (102, 'Bob', 'Johnson', 3),  
 (103, 'Charlie', 'Brown', 1),  
 (104, 'Diana', 'Miller', 2),  
 (105, 'Eve', 'Davis', NULL);

* Perform an INNER JOIN to display employees and their departments:

SELECT  
 e.first\_name,  
 e.last\_name,  
 d.department\_name  
 FROM  
 employees e  
 INNER JOIN  
 departments d ON e.department\_id = d.department\_id;

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

**Ans:** To display all departments, including those that do not have any employees, a LEFT JOIN (also known as LEFT OUTER JOIN) should be used. This join returns all rows from the "left" table (in this case, the Departments table) and the matching rows from the "right" table (the Employees table). If a department in the Departments table does not have a corresponding employee in the Employees table, the columns from the Employees table will show NULL values for that department.

SELECT  
 d.department\_id,  
 d.department\_name,  
 e.employee\_id,  
 e.employee\_name  
FROM  
 Departments d  
LEFT JOIN  
 Employees e ON d.department\_id = e.department\_id;

SQL Group By:

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

**Ans:** To group employees by department and count the number of employees in each department using SQL's GROUP BY clause, a query structured as follows is used:

SELECT Department, COUNT(\*) AS NumberOfEmployees  
FROM Employees  
GROUP BY Department;

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

**Ans:** To find the average salary of employees in each department using the AVG aggregate function, a SQL query involving the GROUP BY clause is required.

SELECT  
 department\_name,  
 AVG(salary) AS average\_salary  
FROM  
 employees  
GROUP BY  
 department\_name;

**SQL Stored Procedure:**

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

**Ans:** A stored procedure can be created to retrieve employees from the Employees table based on a specified department. This procedure will accept a department identifier as an input parameter and return all employee records associated with that department.

CREATE PROCEDURE GetEmployeesByDepartment  
 @DepartmentID INT  
AS  
BEGIN  
 SELECT   
 EmployeeID,  
 FirstName,  
 LastName,  
 DepartmentID,  
 Salary *-- Include other relevant columns as needed*  
 FROM   
 Employees  
 WHERE   
 DepartmentID = @DepartmentID;  
END;

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

**Ans:** A stored procedure that accepts course\_id as input and returns the course details can be implemented as follows, assuming a database table named Courses with columns such as course\_id, course\_name, description, and credits.

CREATE PROCEDURE GetCourseDetails  
 @course\_id INT  
AS  
BEGIN  
 SELECT   
 course\_id,  
 course\_name,  
 description,  
 credits  
 FROM   
 Courses  
 WHERE   
 course\_id = @course\_id;  
END;

**SQL View:**

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

**Ans:** To create a view that displays all employees along with their department names, assuming you have an employees table and a departments table, you would typically join these two tables on a common column, such as department\_id.

CREATE VIEW EmployeeDepartmentView AS  
SELECT  
 e.employee\_id,  
 e.first\_name,  
 e.last\_name,  
 d.department\_name  
FROM  
 employees e  
JOIN  
 departments d ON e.department\_id = d.department\_id;

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

**Ans:** To modify a database view to exclude employees earning less than $50,000, you would need to use a SELECT statement with a WHERE clause in your SQL query for the view. Assuming your table is named employees and has a salary column, the command to create or update the view would be: CREATE OR REPLACE VIEW HighSalaryView AS SELECT \* FROM employees WHERE salary >= 50000;.

CREATE VIEW HighSalaryEmployees AS

SELECT \*

FROM Employees

WHERE Salary >= 50000;

**SQL Triggers:**

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

**Ans:** To automatically log changes to the employees table when a new employee is added, a trigger can be created. This trigger will fire after an INSERT operation on the employees table and record the details of the newly added employee into a separate audit table.

1. Create the Audit Table:

First, an audit table is required to store the log entries. For example, employee\_audit:

CREATE TABLE employee\_audit (  
 audit\_id INT PRIMARY KEY AUTO\_INCREMENT,   
 employee\_id INT NOT NULL,  
 action\_type VARCHAR(50) NOT NULL,  
 action\_timestamp DATETIME NOT NULL,  
 details TEXT  
);

2. Create the Trigger:

The trigger is defined to execute after an INSERT operation on the employees table. It will capture the employee\_id of the new employee and insert a record into the employee\_audit table.

For MySQL:

DELIMITER *//*  
  
CREATE TRIGGER log\_employee\_insert  
AFTER INSERT ON employees  
FOR EACH ROW  
BEGIN  
 INSERT INTO employee\_audit (employee\_id, action\_type, action\_timestamp, details)  
 VALUES (NEW.employee\_id, 'INSERT', NOW(), CONCAT('New employee added: ', NEW.name));  
END;  
*//*  
  
DELIMITER ;

For PostgreSQL/Oracle:

CREATE OR REPLACE FUNCTION log\_employee\_insert\_func()  
RETURNS TRIGGER AS $$  
BEGIN  
 INSERT INTO employee\_audit (employee\_id, action\_type, action\_timestamp, details)  
 VALUES (NEW.employee\_id, 'INSERT', NOW(), 'New employee added: ' || NEW.name);  
 RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER log\_employee\_insert  
AFTER INSERT ON employees  
FOR EACH ROW  
EXECUTE FUNCTION log\_employee\_insert\_func();

**2: Create a trigger to update the last\_modified timestamp whenever an employee record is updated.**

**Ans:** To create a trigger that automatically updates a last\_modified timestamp whenever an employee record is updated, the following SQL trigger can be used. This example assumes a table named employees with a column last\_modified of a suitable timestamp or datetime type.

CREATE OR REPLACE TRIGGER update\_employee\_last\_modified  
BEFORE UPDATE ON employees  
FOR EACH ROW  
BEGIN  
 :NEW.last\_modified := SYSTIMESTAMP;   
END;

**Introduction to PL/SQL:**

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

**Ans:** A PL/SQL block to print the total number of employees from the employees table involves declaring a variable to store the count, executing a SELECT COUNT(\*) statement into that variable, and then using DBMS\_OUTPUT.PUT\_LINE to display the result.

SET SERVEROUTPUT ON;   
  
DECLARE  
 v\_total\_employees NUMBER;

BEGIN  
 SELECT COUNT(\*)  
 INTO v\_total\_employees  
 FROM employees;  
 DBMS\_OUTPUT.PUT\_LINE('Total number of employees: ' || v\_total\_employees);  
END;

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

**Ans:** A PL/SQL block to calculate the total sales from an orders table and display the result is provided below. This assumes the orders table has a total\_amount column representing the total value of each order. If the orders table stores quantity and price separately, the SUM function would need to calculate quantity \* price for each order.

DECLARE  
 v\_total\_sales NUMBER;   
BEGIN  
 SELECT SUM(total\_amount)  
 INTO v\_total\_sales  
 FROM orders;

DBMS\_OUTPUT.PUT\_LINE('Total Sales: ' || v\_total\_sales);  
  
EXCEPTION  
 WHEN NO\_DATA\_FOUND THEN  
 DBMS\_OUTPUT.PUT\_LINE('No orders found in the table.');  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);  
END;

**PL/SQL Control Structures:**

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

**Ans:** A PL/SQL block utilizing an IF-THEN condition to determine an employee's department and display a corresponding message is presented below. This example assumes the existence of an employees table with employee\_id and department\_id columns, and a departments table with department\_id and department\_name columns.

SET SERVEROUTPUT ON;  
  
DECLARE  
 v\_employee\_id NUMBER := 101; *-- Replace with the desired employee ID*  
 v\_department\_name VARCHAR2(50);  
BEGIN  
 *-- Retrieve the department name for the given employee ID*  
 SELECT d.department\_name  
 INTO v\_department\_name  
 FROM employees e  
 JOIN departments d ON e.department\_id = d.department\_id  
 WHERE e.employee\_id = v\_employee\_id;  
  
 *-- Check the department and display a message*  
 IF v\_department\_name = 'IT' THEN  
 DBMS\_OUTPUT.PUT\_LINE('Employee ' || v\_employee\_id || ' belongs to the IT department.');  
 ELSIF v\_department\_name = 'Sales' THEN  
 DBMS\_OUTPUT.PUT\_LINE('Employee ' || v\_employee\_id || ' belongs to the Sales department.');  
 ELSIF v\_department\_name = 'HR' THEN  
 DBMS\_OUTPUT.PUT\_LINE('Employee ' || v\_employee\_id || ' belongs to the HR department.');  
 ELSE  
 DBMS\_OUTPUT.PUT\_LINE('Employee ' || v\_employee\_id || ' belongs to an unknown or unhandled department: ' || v\_department\_name || '.');  
 END IF;  
  
EXCEPTION  
 WHEN NO\_DATA\_FOUND THEN  
 DBMS\_OUTPUT.PUT\_LINE('Employee with ID ' || v\_employee\_id || ' not found.');  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);  
END;

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

**Ans:** Not Answer.

**SQL Cursors:**

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

**Ans:** A PL/SQL block using an explicit cursor to retrieve and display employee details can be constructed as follows:

DECLARE  
 *-- Declare record type variable to hold fetched data*  
 l\_employee\_rec employees%ROWTYPE;   
  
 *-- Declare explicit cursor*  
 CURSOR c\_employees IS  
 SELECT employee\_id, first\_name, last\_name, email, phone\_number, hire\_date, job\_id, salary, department\_id  
 FROM employees  
 ORDER BY last\_name, first\_name;   
  
BEGIN  
 *-- Open the cursor*  
 OPEN c\_employees;  
  
 LOOP  
 *-- Fetch a row into the record variable*  
 FETCH c\_employees INTO l\_employee\_rec;  
  
 *-- Exit the loop when no more rows are found*  
 EXIT WHEN c\_employees%NOTFOUND;  
  
 *-- Display the employee details*  
 DBMS\_OUTPUT.PUT\_LINE('Employee ID: ' || l\_employee\_rec.employee\_id ||   
 ', Name: ' || l\_employee\_rec.first\_name || ' ' || l\_employee\_rec.last\_name ||   
 ', Salary: ' || l\_employee\_rec.salary ||   
 ', Department ID: ' || l\_employee\_rec.department\_id);  
 END LOOP;  
  
 *-- Close the cursor*  
 CLOSE c\_employees;  
  
END;

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

**Ans:** To create and use a cursor to retrieve and display all courses one by one in SQL (e.g., PL/SQL or Transact-SQL), follow these steps:

* **Declare the Cursor**: Define the cursor, associating it with a SELECT statement that retrieves the desired course information from your Courses table.

DECLARE course\_cursor CURSOR FOR  
 SELECT CourseID, CourseName, Credits  
 FROM Courses;

* **Open the Cursor**: Initialize the cursor, which executes the SELECT statement and populates the cursor with the result set.

OPEN course\_cursor;

* **Fetch and Display Data**: Loop through the cursor, fetching one row at a time and displaying the course details. This typically involves using a FETCH statement within a loop.

*-- For PL/SQL (Oracle)*  
 LOOP  
 FETCH course\_cursor INTO v\_course\_id, v\_course\_name, v\_credits;  
 EXIT WHEN course\_cursor%NOTFOUND;  
 DBMS\_OUTPUT.PUT\_LINE('Course ID: ' || v\_course\_id || ', Name: ' || v\_course\_name || ', Credits: ' || v\_credits);  
 END LOOP;  
  
 *-- For Transact-SQL (SQL Server)*  
 FETCH NEXT FROM course\_cursor INTO @CourseID, @CourseName, @Credits;  
 WHILE @@FETCH\_STATUS = 0  
 BEGIN  
 PRINT 'Course ID: ' + CAST(@CourseID AS VARCHAR) + ', Name: ' + @CourseName + ', Credits: ' + CAST(@Credits AS VARCHAR);  
 FETCH NEXT FROM course\_cursor INTO @CourseID, @CourseName, @Credits;  
 END;

* **Close the Cursor**: Release the resources held by the cursor after processing all rows.

CLOSE course\_cursor;

**Rollback and Commit Savepoint:**

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

**Ans:** To perform a transaction involving a savepoint, record insertions, and a rollback to the savepoint, the following SQL commands can be used:

*-- Start a new transaction*  
START TRANSACTION;  
  
*-- Insert the first set of records*  
INSERT INTO your\_table (column1, column2) VALUES ('value1', 'valueA');  
INSERT INTO your\_table (column1, column2) VALUES ('value2', 'valueB');  
  
*-- Create a savepoint*  
SAVEPOINT my\_savepoint;  
  
*-- Insert additional records after the savepoint*  
INSERT INTO your\_table (column1, column2) VALUES ('value3', 'valueC');  
INSERT INTO your\_table (column1, column2) VALUES ('value4', 'valueD');  
  
*-- Rollback to the savepoint, undoing the insertions made after it*  
ROLLBACK TO my\_savepoint;  
  
*-- (Optional) Commit the transaction to save the changes made before the savepoint*  
-- COMMIT;

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

**Ans:** Committing part of a transaction and then rolling back the remaining changes using a savepoint involves the following steps:

* **Start a Transaction:** Initiate a new transaction to group a series of SQL statements.

START TRANSACTION;

* **Perform Initial Operations:** Execute the SQL statements that represent the part of the transaction you intend to commit.

INSERT INTO Customers (name, email) VALUES ('John Doe', 'john.doe@example.com');  
 UPDATE Products SET stock = stock - 1 WHERE product\_id = 123;

* **Create a Savepoint:** Define a savepoint at this stage. This marks a point in the transaction to which you can later roll back without affecting the changes made before the savepoint.

SAVEPOINT first\_part\_saved;

* **Perform Subsequent Operations:** Execute additional SQL statements that represent the changes you might want to roll back.

INSERT INTO Orders (customer\_id, product\_id, quantity) VALUES (1, 123, 1);  
 UPDATE Inventory SET last\_updated = NOW() WHERE item\_id = 456;

* **Rollback to the Savepoint:** If you decide to discard the changes made after the savepoint, use the ROLLBACK TO SAVEPOINT command. This will undo all operations performed after first\_part\_saved.

ROLLBACK TO SAVEPOINT first\_part\_saved;

* **Commit the Remaining Transaction:** Finally, commit the transaction. This will permanently save the changes made before the savepoint (and any changes made after rolling back to the savepoint if you performed new operations).

COMMIT;