|  |  |
| --- | --- |
| **EXP NO:7** | **Implementation of SQL Triggers for Data Manipulation Events** |
| **DATE:** |

## AIM:

To implement complex transactions and understand how to apply Data Control Language (DCL) and Transaction Control Language (TCL) commands in SQL to manage database integrity, security, and consistency during multiple operations**.**

## ALGORITHM:

1. Create a sample database and relevant tables (e.g., Accounts, Transactions).
2. Insert initial data into the tables for transaction testing.
3. Begin a transaction using START TRANSACTION.
4. Perform multiple operations (e.g., update balances, insert transaction records).
5. Use conditions to simulate success/failure scenarios.
6. Use SAVEPOINT to mark rollback points.
7. Use ROLLBACK TO SAVEPOINT if a certain condition fails.
8. Commit the transaction with COMMIT if all steps succeed.
9. Grant and revoke user privileges using DCL commands (GRANT, REVOKE).
10. Ensure proper isolation and consistency using TCL commands (COMMIT, ROLLBACK, SAVEPOINT).

## PROGRAM:

## USE BankDB;

## CREATE TABLE IF NOT EXISTS Accounts (

## acc\_id INT PRIMARY KEY,

## acc\_holder VARCHAR(50),

## balance DECIMAL(10, 2)

## );

## DELETE FROM Accounts WHERE acc\_id IN (1, 2);

## INSERT INTO Accounts VALUES

## (1, 'Vaishnavy', 1000.00),

## (2, 'Shreya', 1500.00);

## SELECT \* FROM Accounts;

## START TRANSACTION;

## UPDATE Accounts SET balance = balance - 200 WHERE acc\_id = 1;

## SELECT \* FROM Accounts;

## SAVEPOINT after\_debit;

## UPDATE Accounts SET balance = balance + 200 WHERE acc\_id = 2;

## SELECT \* FROM Accounts;

## DELIMITER //

## DROP PROCEDURE IF EXISTS HandleTransaction;

## CREATE PROCEDURE HandleTransaction()

## BEGIN

## DECLARE vaishnavy\_balance DECIMAL(10, 2);

## SELECT balance INTO vaishnavy\_balance FROM Accounts WHERE acc\_id = 1;

## IF vaishnavy\_balance < 0 THEN

## ROLLBACK TO SAVEPOINT after\_debit;

## END IF;

## END //

## DELIMITER ;

## CALL HandleTransaction();

## SELECT \* FROM Accounts;

## COMMIT;

## SELECT \* FROM Accounts;

## CREATE USER IF NOT EXISTS 'student'@'localhost' IDENTIFIED BY 'password123';

## GRANT SELECT, UPDATE ON BankDB.Accounts TO 'student'@'localhost';

## REVOKE UPDATE ON BankDB.Accounts FROM 'student'@'localhost';

## OUTPUT:

## 

**RESULT:**

Successfully executed a transaction involving fund transfer using START TRANSACTION, SAVEPOINT, ROLLBACK, and COMMIT. Also demonstrated GRANT and REVOKE privileges using DCL on the Accounts table.

|  |  |
| --- | --- |
| **EXP NO:8** | **Implementation of SQL Triggers for Data Manipulation Events** |
| **DATE:** |

## AIM:

To implement SQL triggers that automatically execute before or after INSERT, DELETE, and UPDATE operations on a table, helping enforce business rules and maintain data integrity without manual intervention.

## ALGORITHM:

1. Create a database and select it using USE.
2. Create a main table called Employees.
3. Create a Logs table to record changes made to Employees.
4. Insert initial records into the Employees table.
5. Write an AFTER INSERT trigger to log any new record added.
6. Write an AFTER UPDATE trigger to log updated details.
7. Write a BEFORE DELETE trigger to save the deleted record.
8. Test the insert operation and verify logs.
9. Test the update operation and verify logs.
10. Test the delete operation and verify logs.

## PROGRAM:

## -- Create and select the database

## CREATE DATABASE IF NOT EXISTS Experiment8;

## USE Experiment8;

## -- Drop existing triggers and tables to avoid conflicts

## DROP TRIGGER IF EXISTS after\_employee\_insert;

## DROP TRIGGER IF EXISTS after\_employee\_update;

## DROP TRIGGER IF EXISTS before\_employee\_delete;

## DROP TABLE IF EXISTS Logs;

## DROP TABLE IF EXISTS Employees;

## -- Recreate the Employees and Logs tables

## CREATE TABLE Employees (

## EmpID INT PRIMARY KEY,

## Name VARCHAR(50),

## Salary DECIMAL(10,2)

## );

## CREATE TABLE Logs (

## LogID INT AUTO\_INCREMENT PRIMARY KEY,

## ActionType VARCHAR(20),

## EmpID INT,

## Name VARCHAR(50),

## Salary DECIMAL(10,2),

## ActionTime TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

## );

## -- Sample data

## INSERT INTO Employees VALUES

## (1, 'Rajarajan', 50000),

## (2, 'Sivakanesh', 60000);

## -- Triggers

## DELIMITER //

## CREATE TRIGGER after\_employee\_insert

## AFTER INSERT ON Employees

## FOR EACH ROW

## BEGIN

## INSERT INTO Logs (ActionType, EmpID, Name, Salary)

## VALUES ('INSERT', NEW.EmpID, NEW.Name, NEW.Salary);

## END;

## //

## CREATE TRIGGER after\_employee\_update

## AFTER UPDATE ON Employees

## FOR EACH ROW

## BEGIN

## INSERT INTO Logs (ActionType, EmpID, Name, Salary)

## VALUES ('UPDATE', NEW.EmpID, NEW.Name, NEW.Salary);

## END;

## //

## CREATE TRIGGER before\_employee\_delete

## BEFORE DELETE ON Employees

## FOR EACH ROW

## BEGIN

## INSERT INTO Logs (ActionType, EmpID, Name, Salary)

## VALUES ('DELETE', OLD.EmpID, OLD.Name, OLD.Salary);

## END;

## //

## DELIMITER ;

## -- Test triggers

## INSERT INTO Employees VALUES (3, 'Yogesh', 70000);

## UPDATE Employees SET Salary = 65000 WHERE EmpID = 2;

## DELETE FROM Employees WHERE EmpID = 1;

## -- View Logs

## SELECT \* FROM Logs;

## OUTPUT:

## 

**RESULT:**

Successfully created and executed SQL triggers for INSERT, UPDATE, and DELETE operations. All changes were automatically captured in the Logs table using trigger.

|  |  |
| --- | --- |
| **EXP NO:9** | **Implementation of View and index.** |
| **DATE:** |

## AIM:

To create views and indexes on large database tables to optimize data access and improve performance in relational database systems.

## ALGORITHM:

1. Start your MySQL server and open the SQL environment.
2. Create a database and use it.
3. Create a table with a large number of records (manually or with a script).
4. Insert a large dataset to simulate real-world conditions.
5. Create a view to display a specific subset of data for ease of access.
6. Create an index on one or more frequently searched columns.
7. Query the view to retrieve meaningful information.
8. Execute a query using the indexed column and observe performance.
9. Modify the view or index if necessary to suit performance goals.
10. Drop the view and index once testing is complete, if needed.

## PROGRAM:

## -- Drop the database if it exists

## DROP DATABASE IF EXISTS Experiment9;

## -- Create the database

## CREATE DATABASE Experiment9;

## -- Use the database

## USE Experiment9;

## -- Drop the table if it exists

## DROP TABLE IF EXISTS StudentMarks;

## -- Create the table

## CREATE TABLE StudentMarks (

## RollNo INT PRIMARY KEY,

## Name VARCHAR(50),

## Marks INT

## );

## -- Insert values into the table

## INSERT INTO StudentMarks VALUES

## (301, 'Yogesh', 78),

## (302, 'Vaishnavy', 91),

## (303, 'Sivakanesh', 45);

## -- Drop the view if it exists

## DROP VIEW IF EXISTS HighScorers;

## -- Create the view for high scorers

## CREATE VIEW HighScorers AS

## SELECT Name, Marks FROM StudentMarks WHERE Marks > 50;

## -- Query the view

## SELECT \* FROM HighScorers;

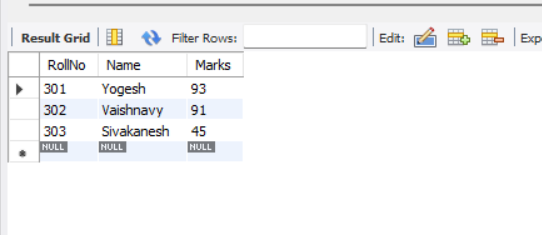
## -- Update Yogesh's marks using RollNo

## UPDATE StudentMarks SET Marks = 93 WHERE RollNo = 301;

## -- Query the StudentMarks table to see the updated marks

## SELECT \* FROM StudentMarks;

## OUTPUT:

****

**RESULT:**

The experiment was successfully executed. A view was created to display specific student records, and indexes were applied to enhance query performance. This approach is essential for managing large datasets in real-world applications.

|  |  |
| --- | --- |
| **EXP NO:10** | **Implementation of XML Database Using XML Schema.** |
| **DATE:** |

## AIM:

To design a simple XML database and validate its structure using an XML Schema Definition (XSD), ensuring that the data adheres to specified constraints and format rules, which is essential for data integrity in XML-based systems.

## ALGORITHM:

1. Create a new XML file to store structured data.
2. Define a root element to represent the database (e.g., <library>, <students>, etc.).
3. Add nested elements representing records with various fields (e.g., name, ID, subject).
4. Save the XML file with a .xml extension.
5. Create a separate XSD file that defines the structure and data types of each element in the XML.
6. Define elements, attributes, types, restrictions, and sequences in the XSD.
7. Link the XSD to the XML file using xmlns:xsi and xsi:noNamespaceSchemaLocation.
8. Use an XML validator (online or IDE-based) to validate the XML file against the XSD.
9. Fix any validation errors if present.
10. Display the validated XML content with confirmation.

## PROGRAM:

## -- Step 1: Create the database

## CREATE DATABASE IF NOT EXISTS xml\_store;

## USE xml\_store;

## -- Step 2: Create table to store XML content

## CREATE TABLE IF NOT EXISTS xmll\_data (

## id INT AUTO\_INCREMENT PRIMARY KEY,

## xml\_name VARCHAR(100),

## xml\_content LONGTEXT

## );

## -- Step 3: Insert sample XML records

## INSERT INTO xmll\_data (xml\_name, xml\_content) VALUES

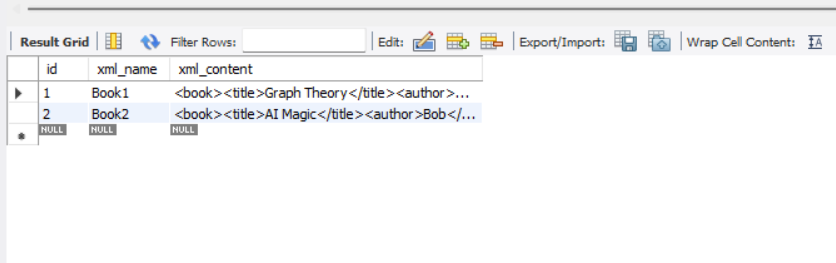
## ('Book1', '<book><title>Graph Theory</title><author>Alice</author><year>2024</year></book>'),

## ('Book2', '<book><title>AI Magic</title><author>Bob</author><year>2025</year></book>');

## -- Step 4: Retrieve XML data

## SELECT \* FROM xmll\_data;

## OUTPUT:`

****

**RESULT:**

The XML database was successfully created and validated using an XML Schema. The structure, data types, and constraints were enforced correctly, ensuring reliable and standardized XML data storage.

|  |  |
| --- | --- |
| **EXP NO:6** | **Implementation of UDF and Stored Procedure** |
| **DATE:** |

**AIM:**

To implement a **User-Defined Function (UDF)** and a **Stored Procedure** in MySQL for basic mathematical and student data operations.

**ALGORITHM:**

1. **Create a User-Defined Function (UDF)**
   * Define SquareNum to return the square of a number.
2. **Create the Students Table**
   * Define Students with ID, Name, and Age.
   * Insert sample records.
3. **Create a Stored Procedure**
   * Define GetStudent to fetch student details by ID.
4. **Execute & Verify Output**

**PROGRAM:**

DROP FUNCTION IF EXISTS SquareNum;

DELIMITER //

CREATE FUNCTION SquareNum(n INT) RETURNS INT

DETERMINISTIC

BEGIN

RETURN n \* n;

END //

DELIMITER ;

DROP TABLE IF EXISTS Students;

CREATE TABLE Students (

ID INT AUTO\_INCREMENT PRIMARY KEY,

Name VARCHAR(50),

Age INT

);

INSERT INTO Students (Name, Age)

VALUES

('Rajarajan', 22),

('Sivakaesh', 23),

('Vaishnavy', 21);

DELIMITER //

DROP PROCEDURE IF EXISTS GetStudent;

CREATE PROCEDURE GetStudent(IN stud\_id INT)

BEGIN

SELECT \* FROM Students WHERE ID = stud\_id;

END //

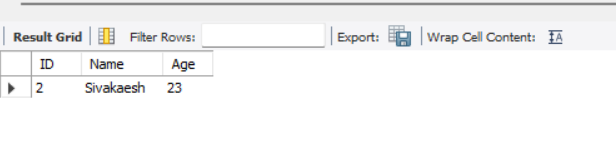
DELIMITER ;

CALL GetStudent(2); -- Should return Sivakaesh's record

**OUTPUT :**



**STORED PROCEDURE :**

****

**RESULT:**

Thus, the function and stored procedure were successfully executed, ensuring efficient data retrieval and calculations.