EXPERIMENT-12
ADVANCED DATABASE MANAGEMENT SYSTEMS

Aim:
To demonstrate deadlock scenarios, how they occur, and how MVCC allows concurrent reads and writes without blocking, ensuring data consistency in high-concurrency environments.

Theory:

1. **Deadlock:** Occurs when two or more transactions hold locks on resources and wait for each other indefinitely.
2. **Deadlock Detection:** Modern databases detect deadlocks and abort one transaction automatically.
3. **MVCC (Multiversion Concurrency Control):** Each transaction sees a **snapshot of data** at the start, allowing readers and writers to operate concurrently without blocking.
4. **Isolation Levels:** Repeatable Read or Snapshot Isolation ensures consistent views for readers while writers update.
5. **Concurrency Behavior Comparison:** Traditional locking blocks readers until writers finish, while MVCC allows non-blocking reads.

CODE:

Part A: Simulating a Deadlock Between Two Transactions

```
-- Create table
CREATE TABLE StudentEnrollments (
   student_id INT PRIMARY KEY,
   student_name VARCHAR(100),
   course_id VARCHAR(10),
   enrollment_date DATE
);

-- Insert sample data
INSERT INTO StudentEnrollments VALUES
(1, 'Ashish', 'CSE101', '2024-06-01'),
(2, 'Smaran', 'CSE102', '2024-06-01'),
(3, 'Vaibhav', 'CSE103', '2024-06-01');

-- User A Transaction
-- Session 1
START TRANSACTION;
```

```
UPDATE StudentEnrollments SET enrollment_date = '2024-07-01' WHERE
student_id = 1;
-- Wait/hold

-- User B Transaction
-- Session 2
START TRANSACTION;
UPDATE StudentEnrollments SET enrollment_date = '2024-07-02' WHERE
student_id = 2;
-- Now both try to update the other user's row in reverse order
UPDATE StudentEnrollments SET enrollment_date = '2024-07-03' WHERE
student_id = 1; -- This triggers deadlock

-- Database detects deadlock and aborts one transaction automatically
COMMIT; -- Only the surviving transaction will commit
```

Part B: Applying MVCC to Prevent Conflicts During Concurrent Reads/ Writes

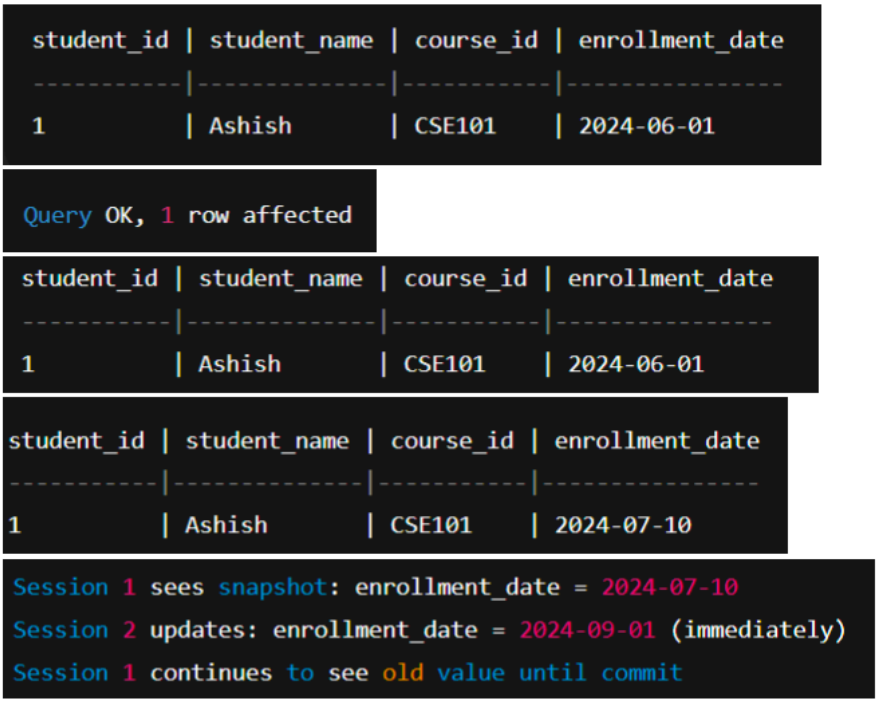
```
-- Assuming InnoDB or PostgreSQL with MVCC enabled
-- User A reads the row in a transaction
-- Session 1
START TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM StudentEnrollments WHERE student_id = 1;

-- User B updates the same row concurrently
-- Session 2
START TRANSACTION;
UPDATE StudentEnrollments SET enrollment_date = '2024-07-10' WHERE
student_id = 1;
COMMIT;

-- User A still sees old value in transaction
SELECT * FROM StudentEnrollments WHERE student_id = 1;
-- Output: enrollment_date = '2024-06-01'

COMMIT; -- User A ends transaction
```

Part C: Comparing Behavior With and Without MVCC in High-Concurrency

```
-- Scenario 1: Traditional Locking (SELECT FOR UPDATE)
-- Session 1 (Writer)
START TRANSACTION;
SELECT * FROM StudentEnrollments WHERE student_id = 1 FOR UPDATE;
```

```
UPDATE StudentEnrollments SET enrollment_date = '2024-07-15' WHERE
student_id = 1;

-- Session 2 (Reader)
START TRANSACTION;
SELECT * FROM StudentEnrollments WHERE student_id = 1;
-- This will be blocked until Session 1 commits
COMMIT;

-- Scenario 2: MVCC-enabled reads (no locking)
-- Session 3 (Reader)
START TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM StudentEnrollments WHERE student_id = 1;
-- Sees enrollment_date = '2024-06-01' even while writer updates

-- Session 4 (Writer)
START TRANSACTION;
UPDATE StudentEnrollments SET enrollment_date = '2024-07-20' WHERE
student_id = 1;
COMMIT;

-- Session 3 continues to see old value until commit
COMMIT;
```

OUTPUT:



Learning Outcomes:

1. Understand **deadlock scenarios** and how databases handle them.
2. Learn how **transaction ordering** can prevent deadlocks.
3. Learn **MVCC** and how it provides **consistent snapshots** for readers.
4. Compare **traditional locking vs MVCC** behavior in concurrent environments.
5. Understand how to **design transactions** to maximize concurrency without data inconsistency.