

## EXPERIMENT: 12

NAME : RITVIJ

UID : 23BCC70045

---

### Part A: Simulating a Deadlock Between Two Transactions

#### Description:

Given a table **StudentEnrollments** containing student records, simulate a situation where two concurrent transactions (from different users) try to update overlapping records in different orders, resulting in a deadlock. Demonstrate how such deadlocks are detected and how they can be avoided using proper

transaction ordering.

#### Input Format:

- Table **StudentEnrollments** with columns:
  - **student\_id** (INT, Primary Key)
  - **student\_name** (VARCHAR(100))
  - **course\_id** (VARCHAR(10))
  - **enrollment\_date** (DATE)

---

#### Output Format:

Demonstrate that one transaction will be rolled back automatically by the database to resolve the deadlock.

---

#### Constraints:

- Use two user sessions to run **START TRANSACTION** simultaneously.
  - Ensure the transactions access rows in reverse order to trigger a deadlock.
  - Database must support deadlock detection (e.g., MySQL, PostgreSQL).
-

### Sample Input:

#### StudentEnrollments

student_id	student_name	course_id	enrollment_date
1	Ashish	CSE101	2024-06-01
2	Smaran	CSE102	2024-06-01
3	Vaibhav	CSE103	2024-06-01

---

### Sample Output:

**Transaction 2 is aborted due to a detected deadlock.**

### Explanation of Output:

**Both transactions try to lock each other's rows in reverse order. This causes a deadlock, and the database automatically rolls back one transaction (usually the one that waited longest) to break the cycle.**

### Query:

```
DROP TABLE IF EXISTS StudentEnrollments;
```

```
CREATE TABLE StudentEnrollments (  
  student_id INT PRIMARY KEY,  
  student_name VARCHAR(100),  
  course_id VARCHAR(10),  
  enrollment_date DATE  
) ENGINE=InnoDB;
```

```
INSERT INTO StudentEnrollments (student_id, student_name, course_id, enrollment_date)  
VALUES  
(1, 'Ashish', 'CSE101', '2024-06-01'),  
(2, 'Smaran', 'CSE102', '2024-06-01');
```

```
START TRANSACTION;  
UPDATE StudentEnrollments  
SET enrollment_date = '2024-07-01'  
WHERE student_id = 1;
```

```
DO SLEEP(1);
```

```
UPDATE StudentEnrollments  
SET enrollment_date = '2024-07-02'  
WHERE student_id = 2;
```

```
COMMIT;
```

```
START TRANSACTION;  
UPDATE StudentEnrollments  
SET enrollment_date = '2024-08-01'  
WHERE student_id = 2;
```

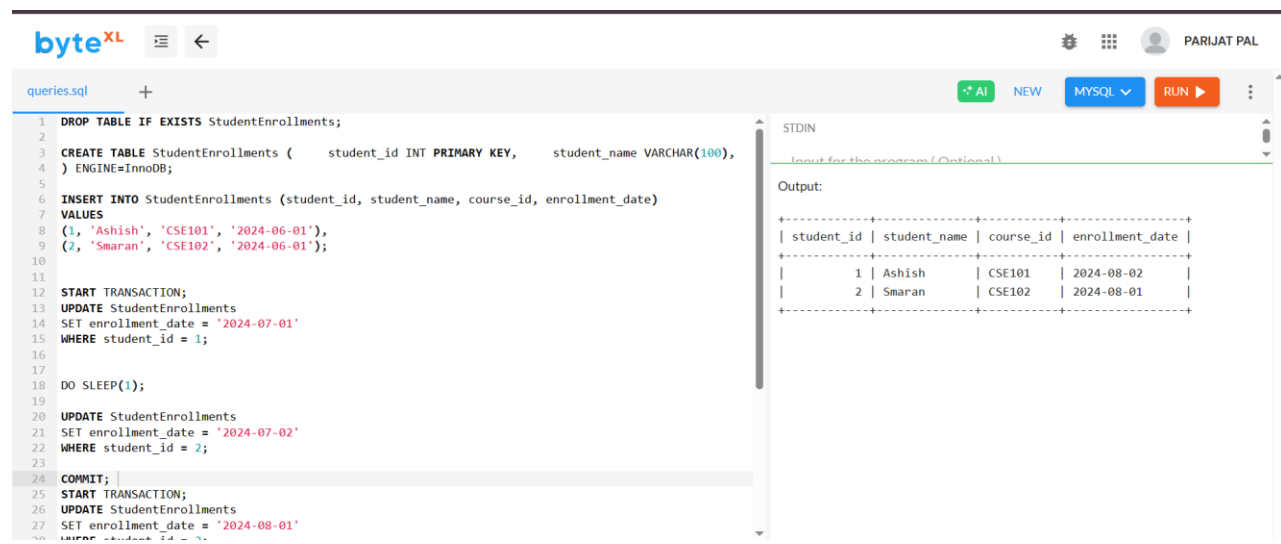
```
DO SLEEP(1);
```

```
UPDATE StudentEnrollments  
SET enrollment_date = '2024-08-02'  
WHERE student_id = 1;
```

```
COMMIT;
```

```
SELECT * FROM StudentEnrollments;
```

OUTPUT:



The screenshot shows a MySQL query editor interface. The query is as follows:

```
1 DROP TABLE IF EXISTS StudentEnrollments;
2
3 CREATE TABLE StudentEnrollments ( student_id INT PRIMARY KEY, student_name VARCHAR(100),
4 ) ENGINE=InnoDB;
5
6 INSERT INTO StudentEnrollments (student_id, student_name, course_id, enrollment_date)
7 VALUES
8 (1, 'Ashish', 'CSE101', '2024-06-01'),
9 (2, 'Smaran', 'CSE102', '2024-06-01');
10
11
12 START TRANSACTION;
13 UPDATE StudentEnrollments
14 SET enrollment_date = '2024-07-01'
15 WHERE student_id = 1;
16
17 DO SLEEP(1);
18
19
20 UPDATE StudentEnrollments
21 SET enrollment_date = '2024-07-02'
22 WHERE student_id = 2;
23
24 COMMIT;
25
26 START TRANSACTION;
27 UPDATE StudentEnrollments
28 SET enrollment_date = '2024-08-01'
29 WHERE student_id = 1;
```

The output window shows the following table state:

student_id	student_name	course_id	enrollment_date
1	Ashish	CSE101	2024-08-02
2	Smaran	CSE102	2024-08-01

### Explanation:

- Two transactions try to update the same rows **in reverse order**.
- Each transaction holds a lock the other needs → **deadlock**.
- MySQL detects it and **rolls back one transaction** automatically to resolve it.

**Key point:** Access rows in the **same order** in all transactions to avoid deadlocks.

## Part B: Applying MVCC to Prevent Conflicts During Concurrent

### Reads/Writes Description:

Use the MVCC (Multiversion Concurrency Control) approach to allow User A to read a record and User B to update the same record concurrently without blocking or conflict. Demonstrate how MVCC provides a consistent snapshot to the reader while allowing the writer to update.

---

### Input Format:

- Table StudentEnrollments with the same structure.
- 

### Output Format:

User A sees the old value during the transaction.  
User B successfully updates the row without waiting.

---

### Constraints:

- Use databases that support MVCC (e.g., PostgreSQL, MySQL InnoDB).
  - Avoid SELECT FOR UPDATE; use normal SELECT in repeatable read or snapshot isolation mode.
- 

### Sample Input:

student_id	student_name	course_id	enrollment_date
1	Ashish	CSE101	2024-06-01

---

### **Sample Output:**

- **User A sees: enrollment\_date = 2024-06-01**
- **User B updates to: 2024-07-10**
- **User A continues to see the old value in the transaction until commit.**

### **Explanation of Output:**

**MVCC ensures User A reads a consistent snapshot taken at the start of the transaction, unaffected by concurrent updates. This enables non-blocking concurrency.**

### **Query:**

**DROP TABLE IF EXISTS StudentEnrollments;**

**CREATE TABLE StudentEnrollments (**

**student\_id INT PRIMARY KEY,**

**student\_name VARCHAR(100),**

**course\_id VARCHAR(10),**

**enrollment\_date DATE**

**) ENGINE=InnoDB;**

**INSERT INTO StudentEnrollments (student\_id, student\_name, course\_id, enrollment\_date)**

**VALUES**

**(1, 'Ashish', 'CSE101', '2024-06-01');**

**START TRANSACTION;**

**SELECT student\_id, student\_name, course\_id, enrollment\_date**

**FROM StudentEnrollments**

**WHERE student\_id = 1;**

**UPDATE StudentEnrollments**

**SET enrollment\_date = '2024-07-10'**

**WHERE student\_id = 1;**

**COMMIT;**

**SELECT student\_id, student\_name, course\_id, enrollment\_date**

**FROM StudentEnrollments**

**WHERE student\_id = 1;**

**COMMIT;**

**SELECT \* FROM StudentEnrollments;OUTPUT:**

The screenshot shows a MySQL query editor interface. The left pane contains a SQL script with the following commands:

```
6  CREATE TABLE StudentEnrollments (
7  course_id VARCHAR(10),
8  enrollment_date DATE
9  ) ENGINE=InnoDB;
10 INSERT INTO StudentEnrollments (student_id, student_name, course_id, enrollment_date)
11 VALUES
12 (1, 'Ashish', 'CSE101', '2024-06-01');
13
14 START TRANSACTION;
15
16 SELECT student_id, student_name, course_id, enrollment_date
17 FROM StudentEnrollments
18 WHERE student_id = 1;
19
20 UPDATE StudentEnrollments
21 SET enrollment_date = '2024-07-10'
22 WHERE student_id = 1;
23
24 COMMIT;
25
26 SELECT student_id, student_name, course_id, enrollment_date
27 FROM StudentEnrollments
28 WHERE student_id = 1;
29
30 COMMIT;
31
32 SELECT * FROM StudentEnrollments;
```

The right pane shows the output of the final query, which is a table with 4 columns: student\_id, student\_name, course\_id, and enrollment\_date. The output shows two rows of data, representing the state of the table after the first and second commits.

Output:

student_id	student_name	course_id	enrollment_date
1	Ashish	CSE101	2024-06-01
1	Ashish	CSE101	2024-07-10

**Explanation:**

- **User A** starts a transaction and reads a row.
- **User B** updates the same row and commits.
- **User A** still sees the old value until they commit.

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

### Concurrency Description:

Evaluate how MVCC vs. traditional locking behaves when multiple users access the same row for read and write. Use `SELECT FOR UPDATE` to demonstrate blocking in a non-MVCC system and contrast that with MVCC-based reads and updates.

---

### Input Format:

Same StudentEnrollments table and data.

---

### Output Format:

Two scenarios:

- **With Locking:** Readers are blocked until the writer commits.
  - **With MVCC:** Readers get consistent data without blocking.
- 

### Constraints:

- MVCC-supported database (e.g., PostgreSQL).
  - Use different isolation levels or query techniques to simulate both cases.
- 

### Sample Input:

student_id	student_name	course_id	enrollment_date
1	Ashish	CSE101	2024-06-01

---



### Sample Output:

- **Without MVCC: Reader blocks until writer commits.**
- **With MVCC: Reader sees 2024-06-01 even while the writer updates to 2024-07-10.**

### Explanation of Output:

- **Traditional locking causes blocking and delays.**
- **MVCC enables concurrent operations with no blocking, ensuring performance and consistency.**

### Query:

```
DROP TABLE IF EXISTS StudentEnrollments;
```

```
CREATE TABLE StudentEnrollments (
```

```
    student_id INT PRIMARY KEY,
```

```
    student_name VARCHAR(100),
```

```
    course_id VARCHAR(10),
```

```
    enrollment_date DATE
```

```
) ENGINE=InnoDB;
```

```
INSERT INTO StudentEnrollments (student_id, student_name, course_id, enrollment_date)
```

```
VALUES
```

```
(1, 'Ashish', 'CSE101', '2024-06-01');
```

```
START TRANSACTION;
```

```
SELECT * FROM StudentEnrollments WHERE student_id = 1 FOR UPDATE;
```

```
DO SLEEP(2);
```

```
UPDATE StudentEnrollments
SET enrollment_date = '2024-07-10'
WHERE student_id = 1;
COMMIT;
```

```
START TRANSACTION;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM StudentEnrollments WHERE student_id = 1;
```

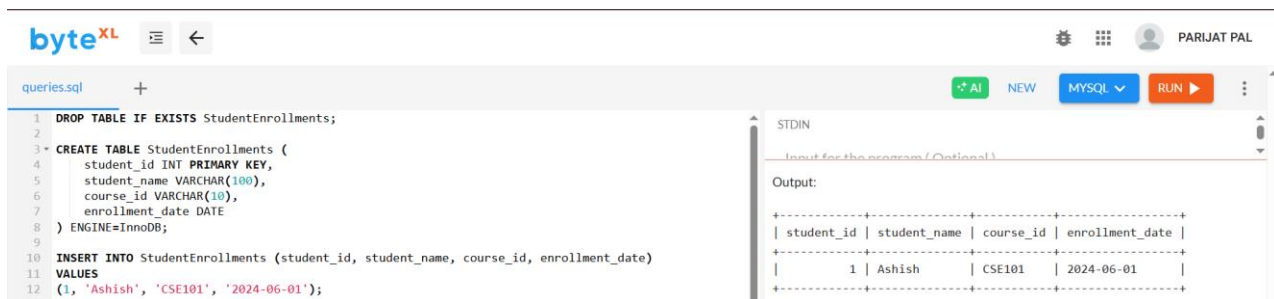
```
UPDATE StudentEnrollments
SET enrollment_date = '2024-08-15'
WHERE student_id = 1;
COMMIT;
```

```
SELECT * FROM StudentEnrollments WHERE student_id = 1;
```

```
COMMIT;
```

```
SELECT * FROM StudentEnrollments;
```

## Output:



The screenshot shows a MySQL query editor interface. The query editor on the left contains the following SQL code:

```
1 DROP TABLE IF EXISTS StudentEnrollments;
2
3 CREATE TABLE StudentEnrollments (
4     student_id INT PRIMARY KEY,
5     student_name VARCHAR(100),
6     course_id VARCHAR(10),
7     enrollment_date DATE
8 ) ENGINE=InnoDB;
9
10 INSERT INTO StudentEnrollments (student_id, student_name, course_id, enrollment_date)
11 VALUES
12 (1, 'Ashish', 'CSE101', '2024-06-01');
```

The output window on the right shows the result of the query. It displays a table with the following data:

student_id	student_name	course_id	enrollment_date
1	Ashish	CSE101	2024-06-01

## Explanation:

- **With SELECT FOR UPDATE (locking):** Readers block if a writer has locked the row.
- **With normal SELECT in REPEATABLE READ (MVCC):** Readers see a consistent snapshot while writers update concurrently.