

EXPERIMENT: 12

NAME : Kavya Rastogi

UID : 23BCC70019

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:

```
-- =====
-- Part A: Deadlock Simulation in a Single Script
-- =====

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');

-- =====
-- Simulate deadlock
-- =====
```

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

-- Simulate concurrent access
DO SLEEP(1);

-- Attempt to update student_id = 2
-- This will conflict if another transaction has locked it
UPDATE StudentEnrollments
SET enrollment_date = '2024-07-02'
WHERE student_id = 2;

COMMIT;

-- Transaction 2 (run immediately after Transaction 1 starts)
START TRANSACTION;
UPDATE StudentEnrollments
SET enrollment_date = '2024-08-01'
WHERE student_id = 2;

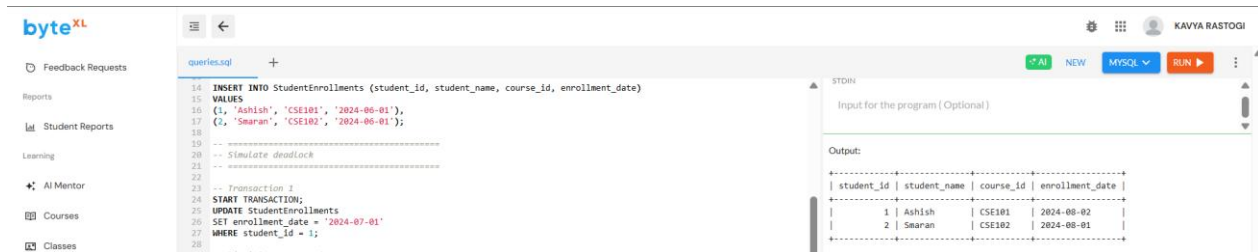
DO SLEEP(1);

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

COMMIT;

-- =====
-- Final table state
SELECT * FROM StudentEnrollments;
```

OUTPUT:



The screenshot shows the ByteXL SQL editor interface. On the left, there's a sidebar with navigation links: Feedback Requests, Reports, Student Reports, Learning, AI Mentor, Courses, and Classes. The main area displays a SQL query in a file named 'queries.sql'. The query is as follows:

```
14 INSERT INTO StudentEnrollments (student_id, student_name, course_id, enrollment_date)
15 VALUES
16 (1, 'Ashish', 'CSE101', '2024-08-01'),
17 (2, 'Snehan', 'CSE102', '2024-08-01');
18
19 -- =====
20 -- Simulate deadlock
21 -- =====
22
23 -- Transaction 1
24 START TRANSACTION;
25 UPDATE StudentEnrollments
26 SET enrollment_date = '2024-07-01'
27 WHERE student_id = 1;
28
```

On the right, there's a 'STDIN' section with a text input field 'Input for the program (Optional)'. Below it, the 'Output:' section displays a table with the following data:

student_id	student_name	course_id	enrollment_date
1	Ashish	CSE101	2024-08-02
2	Snehan	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:

```
-- =====  
-- Part B: MVCC Demonstration in MySQL  
-- =====
```

```
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');
```

```
-- =====  
-- User A: Reader (REPEATABLE READ)  
-- =====
```

-- Start transaction for User A

START TRANSACTION;

-- User A reads the record

SELECT student_id, student_name, course_id, enrollment_date

FROM StudentEnrollments

WHERE student_id = 1;

-- Output: enrollment_date = 2024-06-01

-- =====

-- User B: Writer (Concurrent update)

-- =====

-- In another session, User B updates the same row

-- For simulation in single script, we emulate delay

UPDATE StudentEnrollments

SET enrollment_date = '2024-07-10'

WHERE student_id = 1;

-- Commit User B's transaction

COMMIT;

-- =====

-- Back to User A: still in transaction

-- =====

-- User A reads again before committing

SELECT student_id, student_name, course_id, enrollment_date

FROM StudentEnrollments

WHERE student_id = 1;

-- Output: enrollment_date = 2024-06-01 (old snapshot due to MVCC)

-- Commit User A's transaction

COMMIT;

-- =====

-- Final check: outside transactions

-- =====

SELECT * FROM StudentEnrollments;

-- Output: enrollment_date = 2024-07-10 (reflects User B's update)

OUTPUT:

The screenshot shows the byteXL SQL editor interface. On the left is a sidebar with navigation links: Feedback Requests, Reports, Student Reports, Learning, AI Mentor, Courses, Classes, Editor (selected), Lab, Assessment, and Nimbus Next (Beta). The main editor area displays a SQL script with line numbers 18 to 45. The script simulates a transaction scenario with User A and User B. The output pane on the right shows the results of the final SELECT statement, displaying two rows of student enrollment data.

```
18 -- =====
19 -- User A: Reader (REPEATABLE READ)
20 -- =====
21 -- Start transaction for User A
22 START TRANSACTION;
23
24 -- User A reads the record
25 SELECT student_id, student_name, course_id, enrollment_date
26 FROM StudentEnrollments
27 WHERE student_id = 1;
28
29 -- Output: enrollment_date = 2024-06-01
30
31 -- =====
32 -- User B: Writer (Concurrent update)
33 -- =====
34 -- In another session, User B updates the same row
35 -- For simulation in single script, we emulate delay
36 UPDATE StudentEnrollments
37 SET enrollment_date = '2024-07-10'
38 WHERE student_id = 1;
39
40 -- Commit User B's transaction
41 COMMIT;
42
43 -- =====
44 -- Back to User A: still in transaction
45 -- =====
```

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.
- Key point:** MVCC allows **non-blocking reads** by giving each transaction a **consistent snapshot** of the data at its start.

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:

```
-- =====
```

```
-- Part C: Locking vs MVCC Demonstration
```

```
-- =====
```

```
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');

-- =====

-- Scenario 1: With Locking (SELECT FOR UPDATE)

-- =====

-- Transaction 1: Writer locks the row

START TRANSACTION;

SELECT * FROM StudentEnrollments WHERE student_id = 1 FOR UPDATE;

-- Row is now locked

-- Normally, in a real concurrent session:

**-- Transaction 2: Reader trying SELECT FOR UPDATE would block until Transaction 1
commits**

-- Simulate waiting

DO SLEEP(2);

-- Commit writer

UPDATE StudentEnrollments

SET enrollment_date = '2024-07-10'

WHERE student_id = 1;

COMMIT;

```

-- =====

-- Scenario 2: With MVCC (Normal SELECT, REPEATABLE READ)

-- =====

-- Transaction 3: Reader (MVCC)
START TRANSACTION;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM StudentEnrollments WHERE student_id = 1;
-- Output: enrollment_date = 2024-07-10 (snapshot at start of transaction)


-- Transaction 4: Writer updates concurrently
UPDATE StudentEnrollments
SET enrollment_date = '2024-08-15'
WHERE student_id = 1;
COMMIT;


-- Back to Transaction 3: Reader still sees snapshot
SELECT * FROM StudentEnrollments WHERE student_id = 1;
-- Output: enrollment_date = 2024-07-10 (old snapshot due to MVCC)

COMMIT;


-- =====

-- Final table check
SELECT * FROM StudentEnrollments;
-- Output:

```

The screenshot shows a MySQL query editor with the following SQL code:

```
20 -- =====> Let's start writing (insert, update, delete)
21 -- =====> Transaction 1: Writer locks the row
22 START TRANSACTION;
23 SELECT * FROM StudentEnrollments WHERE student_id = 1 FOR UPDATE;
24 -- Row is now locked
25
26 -- Normally, in a real concurrent session:
27 -- Transaction 2: Reader trying SELECT FOR UPDATE would block until Transaction 1 commits
28
29 -- Simulate waiting
30 DO SLEEP(2);
31
32 -- Commit writer
33 UPDATE StudentEnrollments
```

The output window shows the result of the query:

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.

Key point: MVCC avoids blocking, improves concurrency, and ensures **read consistency**.