Name : Akshansh Kaund  al
UID – 23BCS13369
SEC – 622(B)

# Practice 1 – Transactions & Concurrency Control

## Part A: Insert Multiple Fee Payments in a Transaction

Objective: Insert multiple payments as a single transaction to demonstrate Atomicity. Either all rows are inserted or none are.

### SQL Code:

```
CREATE TABLE FeePayments (
   payment_id INT PRIMARY KEY,
   student_name VARCHAR(100) NOT NULL,
   amount DECIMAL(10,2) CHECK (amount > 0),
   payment_date DATE
);

-- Insert multiple rows in one transaction
START TRANSACTION;
INSERT INTO FeePayments(payment_id, student_name, amount, payment_date)
VALUES (1, 'Ashish', 5000.00, '2024-06-01'),
     (2, 'Smaran', 4500.00, '2024-06-02'),
     (3, 'Vaibhav', 5500.00, '2024-06-03');
COMMIT;
```

### Simulated Output (Terminal):

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO FeePayments(payment_id, student_name, amount, payment_date)
    -> VALUES (1, 'Ashish', 5000.00, '2024-06-01'),
    ->        (2, 'Smaran', 4500.00, '2024-06-02'),
    ->        (3, 'Vaibhav', 5500.00, '2024-06-03');
Query OK, 3 rows affected (0.02 sec)
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

### *Simulated Output (Clean Table):*

| payment_id | student_name | amount | payment_date |
|---|---|---|---|
| 1 | Ashish | 5000.00 | 2024-06-01 |
| 2 | Smaran | 4500.00 | 2024-06-02 |
| 3 | Vaibhav | 5500.00 | 2024-06-03 |

## Part B: Demonstrate ROLLBACK for Failed Payment Insertion

Objective: Show that when a transaction contains an invalid operation, ROLLBACK undoes all changes.

### SQL Code:

```
-- Assume FeePayments table exists as above and already has entries 1,2,3
-- Start a transaction that will fail due to constraint violation
START TRANSACTION;
INSERT INTO FeePayments(payment_id, student_name, amount, payment_date)
VALUES (4, 'Kiran', 4800.00, '2024-06-04'),
    (1, 'Ashish', -100.00, '2024-06-05'); -- duplicate ID and negative amount
-- Error occurs, so rollback
ROLLBACK;
```

### Simulated Output (Terminal - Rollback):

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO FeePayments(payment_id, student_name, amount, payment_date)
    -> VALUES (4, 'Kiran', 4800.00, '2024-06-04'),
    ->        (1, 'Ashish', -100.00, '2024-06-05');
ERROR 1406 (22003): Data out of range or CHECK constraint violated
mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)
```

*Table after Rollback (Clean):*

| payment_id | student_name | amount | payment_date |
|---|---|---|---|
| 1 | Ashish | 5000.00 | 2024-06-01 |
| 2 | Smaran | 4500.00 | 2024-06-02 |
| 3 | Vaibhav | 5500.00 | 2024-06-03 |

## Part C: Simulate Partial Failure and Ensure Consistent State

Objective: Demonstrate that a transaction with one valid and one invalid insert is fully rolled back.

### SQL Code:

```
START TRANSACTION;
INSERT INTO FeePayments(payment_id, student_name, amount, payment_date)
VALUES (4, 'Kiran', 4800.00, '2024-06-04'),
    (5, NULL, 5000.00, '2024-06-05'); -- NULL student_name causes NOT NULL violation
ROLLBACK;
```

### Simulated Output (Terminal - NULL error):

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO FeePayments(payment_id, student_name, amount, payment_date)
    -> VALUES (4, 'Kiran', 4800.00, '2024-06-04'),
    ->        (5, NULL, 5000.00, '2024-06-05');
ERROR 1048 (23000): Column 'student_name' cannot be null
mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)
```

*Table after Rollback (Clean):*

| payment_id | student_name | amount | payment_date |
|---|---|---|---|
| 1 | Ashish | 5000.00 | 2024-06-01 |
| 2 | Smaran | 4500.00 | 2024-06-02 |
| 3 | Vaibhav | 5500.00 | 2024-06-03 |

## Part D: Verify ACID Compliance with Transaction Flow

Objective: Combine the techniques and explain how ACID properties are preserved.

### SQL Code (Demonstrative):

```
-- Atomicity: group multiple operations
START TRANSACTION;
INSERT INTO FeePayments(payment_id, student_name, amount, payment_date) VALUES (6, 'Riya', 5200.00, '2024-06-06');
UPDATE FeePayments SET amount = amount + 100 WHERE payment_id = 2;
COMMIT;

-- Isolation: use SELECT ... FOR UPDATE to lock rows during complex operations
START TRANSACTION;
SELECT * FROM FeePayments WHERE payment_id = 6 FOR UPDATE;
-- perform checks and updates
COMMIT;

-- Durability: once committed, changes persist even after DB restart (demonstrated by SELECT after COMMIT)
SELECT * FROM FeePayments;
```

### Simulated ACID Timeline (Terminal):

```
Session 1 (Atomicity & Isolation):
START TRANSACTION;
SELECT * FROM FeePayments WHERE payment_id=6 FOR UPDATE;
-- holds lock, performs multiple inserts and updates
Session 2 (Blocked due to Isolation):
Attempting to INSERT/UPDATE the same payment_id=6 will wait until session1 commits.
COMMIT;
-- Changes durable (Durability)
```

## Conclusion

This document demonstrated transactions for FeePayments table covering Atomicity, Consistency, Isolation, and Durability. Simulated terminal outputs and clean tables show that commits make changes persistent while errors trigger rollbacks to keep the database in a consistent state.

# Practice 2 – Transactions & Concurrency Control

Assignment Solution

This document contains solutions to the given assignment with SQL code, explanations, dry runs, and sample outputs.

## Part A: Prevent Duplicate Enrollments Using Locking

Objective: Prevent two users from enrolling the same student into the same course simultaneously using unique constraints and transactions.

### SQL Code:

```
CREATE TABLE StudentEnrollments (
   enrollment_id INT PRIMARY KEY AUTO_INCREMENT,
   student_name VARCHAR(100),
   course_id VARCHAR(10),
   enrollment_date DATE,
   UNIQUE(student_name, course_id)
);

-- User 1 transaction
START TRANSACTION;
INSERT INTO StudentEnrollments(student_name, course_id, enrollment_date)
VALUES('Ashish', 'CSE101', '2024-07-01');
COMMIT;

-- User 2 transaction (will fail due to duplicate pair)
START TRANSACTION;
INSERT INTO StudentEnrollments(student_name, course_id, enrollment_date)
VALUES('Ashish', 'CSE101', '2024-07-01');
COMMIT;
```

### Explanation:

• The UNIQUE constraint ensures each (student_name, course_id) pair is unique.
• User 1 inserts successfully.
• User 2's transaction fails with a constraint violation, preventing duplicate enrollment.

### Sample Output:

User 1: Record inserted successfully.
User 2: ERROR 1062 (23000): Duplicate entry 'Ashish-CSE101' for key 'student_course_unique'

## Part B: Use SELECT FOR UPDATE to Lock Student Record

Objective: Use row-level locking via SELECT FOR UPDATE to prevent conflicts while verifying a student before enrollment.

### SQL Code:

```
-- User A
START TRANSACTION;
SELECT * FROM StudentEnrollments
WHERE student_name='Ashish' AND course_id='CSE101'
FOR UPDATE;

-- (Row is now locked until commit/rollback)

-- User B (while User A has not committed yet)
UPDATE StudentEnrollments
SET enrollment_date='2024-07-02'
WHERE student_name='Ashish' AND course_id='CSE101';
-- This query will be blocked until User A commits or rollbacks.
```

### Explanation:

• User A locks the row with SELECT FOR UPDATE.
• User B's update is blocked until User A finishes.
• Ensures consistency and avoids simultaneous conflicting updates.

### Sample Output:

User A: Row selected and locked.
User B: Query waiting... (blocked until User A commits).

## Part C: Demonstrate Locking Preserving Consistency

Objective: Show how locking ensures consistent data when multiple users attempt concurrent updates.

### SQL Code:

```
-- Without Locking (Problematic)
User A: UPDATE StudentEnrollments
    SET enrollment_date='2024-07-03'
    WHERE student_name='Ashish' AND course_id='CSE101';

User B (simultaneous): UPDATE StudentEnrollments
    SET enrollment_date='2024-07-04'
    WHERE student_name='Ashish' AND course_id='CSE101';

-- Result: Last update overwrites previous one (race condition).

-- With Locking
User A:
START TRANSACTION;
SELECT * FROM StudentEnrollments
WHERE student_name='Ashish' AND course_id='CSE101' FOR UPDATE;
UPDATE StudentEnrollments SET enrollment_date='2024-07-03'
WHERE student_name='Ashish' AND course_id='CSE101';
COMMIT;

User B (waits until User A finishes):
START TRANSACTION;
UPDATE StudentEnrollments SET enrollment_date='2024-07-04'
WHERE student_name='Ashish' AND course_id='CSE101';
COMMIT;
```

### Explanation:

• Without locking, both users' updates conflict, and only the last one persists.
• With locking, transactions are serialized: User B waits until User A finishes, preserving

consistency.

### Sample Output:

Final Table Row:
student_name = Ashish, course_id = CSE101, enrollment_date = 2024-07-04

## Conclusion

This assignment demonstrates how transactions and locking mechanisms prevent race conditions, duplicate entries, and inconsistent data. Using UNIQUE constraints, SELECT FOR UPDATE, and proper transaction management ensures that database operations follow ACID properties, preserving integrity and consistency in concurrent environments.

# Practice 3 – Transactions & Concurrency Control

## Part A: Simulating a Deadlock Between Two Transactions

Setup SQL:

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

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

Steps to Reproduce Deadlock:

```
-- Session 1
START TRANSACTION;
UPDATE StudentEnrollments SET course_id = 'CSE201' WHERE student_id = 1;
UPDATE StudentEnrollments SET course_id = 'CSE202' WHERE student_id = 2;

-- Session 2
START TRANSACTION;
UPDATE StudentEnrollments SET course_id = 'CSE301' WHERE student_id = 2;
UPDATE StudentEnrollments SET course_id = 'CSE302' WHERE student_id = 1;
```

Expected Output:

One transaction will fail with error:
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction.

Explanation:

Both transactions try to lock each other's rows in reverse order, creating a deadlock. The database detects the deadlock and rolls back one transaction automatically.

## Part B: Applying MVCC to Prevent Conflicts

Steps:

```
-- Session 1 (User A)
START TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT enrollment_date FROM StudentEnrollments WHERE student_id = 1;
-- Sees: 2024-06-01

-- Session 2 (User B)
START TRANSACTION;
UPDATE StudentEnrollments SET enrollment_date = '2024-07-10' WHERE student_id = 1;
COMMIT;

-- Session 1 (User A continues)
SELECT enrollment_date FROM StudentEnrollments WHERE student_id = 1;
-- Still sees: 2024-06-01

COMMIT;

-- After commit, new transaction sees updated value 2024-07-10
```

Explanation:

MVCC ensures that User A reads a consistent snapshot of data from the start of the transaction. User B can update concurrently without blocking User A.

## Part C: Comparing Behavior With and Without MVCC

Without MVCC (using SELECT FOR UPDATE):

```
-- Session 1
START TRANSACTION;
SELECT * FROM StudentEnrollments WHERE student_id=1 FOR UPDATE;

-- Session 2
SELECT * FROM StudentEnrollments WHERE student_id=1;
-- Blocks until Session 1 commits
```

With MVCC (normal SELECT):

```
-- Session 1
START TRANSACTION;
UPDATE StudentEnrollments SET enrollment_date='2024-08-01' WHERE student_id=1;

-- Session 2
SELECT * FROM StudentEnrollments WHERE student_id=1;
-- Sees old value, no blocking
```

Explanation:

In locking systems, readers block until writers commit. With MVCC, readers see a consistent snapshot while writers can still update, avoiding blocking.