University
Mohammed VI
Polytechnic

College of Computing

Deliverable # 7: Transactions and Concurrency Control

Data Management Course

UM6P College of Computing

Professor: Karima Echihabi **Program:** Computer Engineering

Session: Fall 2025

Team Information

Team Name	Leo fl Bernabéu
Member 1	Yassine SQUALLI-HOUSSAINI
Member 2	Omar TSOULI
Member 3	Yahya TALIB
Member 4	Adam RHYA
Member 5	Hamza TAALOUCHT
Member 6	Diaa Eddine ZAINI
Repository Link	https://github.com/therealzaini/DMG_LAB2_LEO_FL_BERNABEU

1 Revisiting ACID Transactions:

1. Atomicity & Durability:

Property: Atomicity and Durability.

Justification: The system ensures the transaction is treated as a single unit ("all or nothing"). By detecting the incomplete transaction upon recovery and retrying until success, the system enforces Atomicity (ensuring the logical unit of work completes) and Durability (ensuring the committed state persists despite the crash).

2. Isolation (and Consistency):

Property: Violation of Isolation (and consequently Consistency).

Justification: Isolation is violated because two transactions running concurrently affected each other (race condition) leading to an invalid state. Consistency is violated because the rule "only one physical slot exists" was broken when both users received confirmation.

3. Isolation:

Property: Isolation.

Justification: This behavior demonstrates proper Isolation (specifically preventing Dirty Reads). Staff B is protected from seeing the intermediate, uncommitted writes of Staff A. They only see the data after Staff A commits.

4. Durability:

Property: Violation of Durability.

Justification: Durability guarantees that once a transaction is committed (saved), it should not be lost even after a power failure. Since the data was "saved" but missing after the restart, this property failed.

5. Consistency:

Property: Consistency.

Justification: The system maintains data validity according to defined rules (stock totals must be correct, no negative numbers). The transition from one valid state to another is preserved regardless of concurrency.

2 Implementing Atomic Transactions in MySQL:

Atomic scheduling of an appointment:

a. SQL Code Fragment:

```
START TRANSACTION;
INSERT INTO ClinicalActivity (CAID, IID, STAFF_ID, DEP_ID, Date, Time)
```

```
VALUES (101, 5, 20, 2, '2025-12-15', '09:00:00');
INSERT INTO Appointment (CAID, Reason, Status)
VALUES (101, 'General Checkup', 'Scheduled');
-- If both inserts succeed:
COMMIT;
-- If an error occurs (handled via application logic or stored procedure handler):
-- ROLLBACK;
```

b. Explanation:

Atomicity: Grouping these statements ensures Atomicity ("all or nothing"). The `START TRANSACTION` and `COMMIT`/`ROLLBACK` commands ensure that the database never ends up in a partial state where a ClinicalActivity exists without a corresponding Appointment. Without Transaction (Autocommit): If executed separately in autocommit mode, the first `INSERT` could succeed and the second could fail (e.g., due to a constraint violation or crash). This would leave "orphan" data in the database, violating data integrity.

Atomic update of stock and expense:

a. Pseudocode:

```
START TRANSACTION
    TRY:
        UPDATE Stock SET Qty = Qty - dispensed_amount
        WHERE medication_id = ...;

        -- Trigger logic typically runs automatically, but if manual:
        UPDATE Expense SET Total = calculated_new_total
        WHERE activity_id = ...;

        COMMIT
    CATCH Error:
        ROLLBACK
```

b.Important Properties:

Atomicity: Crucial to ensure that stock is not deducted without the expense being recorded (or vice versa).

Consistency: Essential to ensure the invariants (Stock count matches reality, Expense matches dispensed items) remain valid.

3 Identifying Types of Schedules:

1. Are the schedules S1 and S2 equivalent? Justify your answer.

→ Yes, they are equivalent.

a. Same Transactions: Both S1 and S2 involve the exact same transactions (T1 and T2).

b. Same Internal Order: The order of actions for each individual transaction is preserved in both schedules (e.g., R(A) always before W(A)).

c. Same Final State: The final state of the database is the same because T1 operates only on A and T2 operates only on B. Since they access completely different (disjoint) data items, they do not interfere with each other, so the result is identical regardless of the order.

2. Is S1 serializable? If yes, give an equivalent serial schedule.

→ Yes, S1 is serializable.

A schedule is defined as serializable if it is equivalent to some serial schedule5. S2 is a serial schedule because T1 runs from beginning to end without any interleaving with T2. Since we proved in Question 1 that S1 is equivalent to S2, S1 is therefore serializable and the equivalent serial schedule is S2.

4 Conflict Serializability:

1. Precedence (Dependency) Graph for S3:

Schedule: $R1(A), W2(A), R3(A), W1(A), W3(B), R2(B)$.

Conflicts (Edges):

$$R1(A) \text{ before } W2(A) \rightarrow \text{ Edge } T1 \rightarrow T2$$
$$W2(A) \text{ before } R3(A) \rightarrow \text{ Edge } T2 \rightarrow T3$$
$$W2(A) \text{ before } W1(A) \rightarrow \text{ Edge } T2 \rightarrow T1$$
$$R3(A) \text{ before } W1(A) \rightarrow \text{ Edge } T3 \rightarrow T1$$
$$W3(B) \text{ before } R2(B) \rightarrow \text{ Edge } T3 \rightarrow T2$$

2. Is S3 conflict serializable?

No, the dependency graph contains cycles12121212.

Cycle 1: T1 → T2 → T1 (Conflict on A between T1 and T2).

Cycle 2: T2 → T3 → T2 (Conflict on B between T3 and T2).

Since the graph is cyclic, the schedule is not conflict serializable.

5 2PL:

Schedule 1 :

Schedule 1 is compatible with strict 2PL, because T1 hold all its locks until it finishes (S1 is a serial schedule).

The growing phase: T1 acquires locks on A and B.

The shrinking phase: Upon completion, T1 releases all locks.

Schedule 2 :

Schedule 2 is compatible with strict 2PL, because T2 finishes before T1 needs lock on B (which could be the conflicting resource).

The growing phase: T1 acquire locks on A, T2 acquire locks on B.

The shrinking phase: T2 completes, releases locks on B so T1 can successfully acquire lock on B before it completes and releases all locks as well.

Schedule 3 :

Schedule 3 is compatible with strict 2PL, in fact it is very similar to schedule 1 (it is also a serial schedule).

T1 acquires locks on A and B, finishes, and releases them.

T2 acquires locks on C.

The transactions don't share any data, they never block each other, satisfying strict 2PL requirements.

Schedule 4 :

Schedule 4 is not compatible with strict 2PL.

T1 acquires X-Lock on A (since T1 writes to A) and T2 is executing R(A) before T1 has finished (T1 still has to do R(B) later).

Under strict 2PL, T1 must hold the X-lock on A until it commits (after R1(B)). As long as T1 holds X(A), T2 should be blocked from reading A.

6 Deadlocks in MNHS:

Schedule: $R1(A), R2(B), W1(B), W2(A)$.

1. Wait-for Graph :

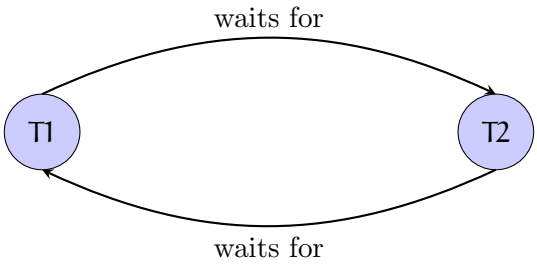
Node T1: Holds S-lock on A (from R1(A)). Needs X-lock on B (for W1(B)).

Node T2: Holds S-lock on B (from R2(B)). Needs X-lock on A (for W2(A)).

Edges:

T1 $\rightarrow$ T2: T1 is waiting for T2 to release the lock on B.

T2 $\rightarrow$ T1: T2 is waiting for T1 to release the lock on A.



2. Deadlock Analysis:

• Is there a deadlock?

→ Yes.

Explanation:

Waiting: T1 is waiting for T2 (to write B), and T2 is waiting for T1 (to write A).

Cycle: The graph has a cycle: T1 → T2 → T1.

Resolution: The DBMS must perform Deadlock Detection and Resolution16. It will typically select a "victim" transaction to abort and rollback (e.g., the one with the lower priority or most recent timestamp), allowing the other transaction to proceed and acquire the necessary locks.