University
Mohammed VI
Polytechnic

College of
Computing

Deliverable 7: Physical Design, Security and Transaction Management

Data Management Course

UM6P College of Computing

Professor: Karima Echihabi **Program:** Computer Engineering
Session: Fall 2025

Team Information

Team Name	Sekel
Member 1	Anas Mahdad
Member 2	Rania Moujahed
Member 3	Haitam Laghmam
Member 4	Abdelmoughit Labrighi
Member 5	Youssef Ouazzani
Member 6	Mohammed Idriss Nana
Repository Link	`https://github.com/Data-Project-Hierarchy`

1 Introduction

This lab covers two main topics in database management using the MNHS healthcare database. First, we look at physical design—how Io use indexes and partitioning to speed up queries. Second, we explore transactions and concurrency control—how to keep data consistent when multiple users access it at the same time. Both parts aim to make the database faster, reliable, and ready for real-world healthcare use.

2 Requirements and Methodology

Part I: Physical Design, Security and Transaction Management

1. Index Design

1.1 Indexes for the Given Views

UpcomingByHospital

- **Appointment**: `B+Tree(Status, CAID)`
- **ClinicalActivity**: `B+Tree(Date, CAID, DEP_ID)`

StaffWorkloadThirty

- **ClinicalActivity**: `B+Tree(Date, CAID, STAFF_ID)`
- **Appointment**: `B+Tree(Status, CAID)`

PatientNextVisit

- **Appointment**: `B+Tree(Status, CAID)`
- **ClinicalActivity**: `B+Tree(Date, CAID, DEP_ID)`

1.2 Indexes for a Frequent Query

Consider the following query executed very frequently by the application:

```sql
SELECT H.Name, C.Date, COUNT(*) AS NumAppt
FROM Hospital H
JOIN Department D ON D.HID = H.HID
JOIN ClinicalActivity C ON C.DEP_ID = D.DEP_ID
JOIN Appointment A ON A.CAID = C.CAID
WHERE A.Status = 'Scheduled'
  AND C.Date BETWEEN ? AND ?
GROUP BY H.Name, C.Date;
```

- **Appointment**: `B+Tree(Status, CAID)` The leading column `Status` matches the equality predicate in the `WHERE` clause and allows early filtering of scheduled appointments. Including `CAID` optimizes the join with `ClinicalActivity` and enables index-only access on the `Appointment` table.

- **ClinicalActivity**: `B+Tree(Date, DEP_ID)` The leading column `Date` supports the range predicate on appointment dates and enables efficient B+Tree range scans. Including `DEP_ID` accelerates joins with `Department` and often avoids accessing the base `ClinicalActivity` table.
- **Department**: `B+Tree(HID, DEP_ID)` Using `HID` as the leading column supports the join with `Hospital`. The subsequent join with `ClinicalActivity` relies on the primary key `DEP_ID`, making this index effective for a query pattern that is executed frequently.

Overhead. These secondary indexes introduce additional cost during inserts and updates, since index structures must be maintained. However, this overhead is justified by the substantial performance gains for a highly frequent query.

2. Partitioning

2.1 Partitioning ClinicalActivity by Date

We chose range partitioning on `ClinicalActivity.Date` by year.

Rationale. Queries targeting recent dates benefit from partition pruning. Yearly partitioning simplifies maintenance tasks such as archiving old data. Partitioning by month was avoided to prevent an excessive number of partitions.

Drawback. Queries that do not filter on date must scan all partitions.

2.2 Partitioning Stock by Region

For the `Stock` table, partitioning by region was chosen instead of per hospital.

Discussion. Partitioning per hospital would degrade performance for queries operating on many hospitals or without hospital filters. Regional partitioning provides a balance between performance and complexity, although data skew may occur due to uneven population distribution.

3. Tablespaces and Storage Layout

3.1 Data Population

Synthetic data was generated using the following stored procedure:

```
DELIMITER //

CREATE PROCEDURE PopulateClinicalActivityWithStaff(IN n INT)
BEGIN
    DECLARE i INT DEFAULT 1;
    DECLARE maxStaff INT;
    DECLARE minStaff INT;
    DECLARE maxCaid INT;

    SELECT MAX(STAFF_ID) INTO maxStaff FROM Staff;
    SELECT MIN(STAFF_ID) INTO minStaff FROM Staff;
    SELECT MAX(CAID) INTO maxCaid FROM ClinicalActivity;
    WHILE i <= n DO
        INSERT INTO ClinicalActivity (CAID, Date, IID, DEP_ID,
           STAFF_ID, Time)
        VALUES (
            i + maxCaid,
            DATE_ADD('2020-01-01', INTERVAL FLOOR(RAND() * 365 *
               5) DAY),
            FLOOR(1 + RAND() * 5),
            FLOOR(1 + RAND() * 5) * 10,
            FLOOR(minStaff + RAND() * (maxStaff - minStaff)),
            '10:00:00'
        );
        SET i = i + 1;
    END WHILE;
END//

DELIMITER ;

CALL PopulateClinicalActivityWithStaff(500000);
```

Appointments were generated as follows:

```
INSERT INTO Appointment (CAID, Status)
SELECT CAID,
       CASE WHEN RAND() < 0.5
            THEN 'Scheduled'
            ELSE 'Completed'
       END
FROM ClinicalActivity;
```

3.2 Analyzed Query

```
EXPLAIN ANALYZE
SELECT H.Name AS HospitalName,
       C.Date AS ApptDate,
       COUNT(*) AS ScheduledCount
FROM Appointment A
JOIN ClinicalActivity C ON A.CAID = C.CAID
JOIN Department D ON C.DEP_ID = D.DEP_ID
JOIN Hospital H ON D.HID = H.HID
WHERE A.Status = 'Scheduled'
  AND C.Date >= CURDATE()
  AND C.Date < DATE_ADD(CURDATE(), INTERVAL 14 DAY)
GROUP BY H.Name, C.Date;
```

3.3 Index Creation

```
CREATE INDEX idx_date_caid_depid
ON ClinicalActivity (Date, CAID, DEP_ID);
```

3.4 Execution Plan Analysis

Before Optimization

Execution Plan Before Indexing

```
-> Table scan on <temporary>  (actual time=24.1..24.1 rows=1 loops=1)
    -> Aggregate using temporary table  (actual time=24.1..24.1 rows=1 loops=1)
        -> Nested loop inner join  (cost=1495 rows=369) (actual time=0.659..24
        ↪  rows=1 loops=1)
            -> Nested loop inner join  (cost=1108 rows=1107) (actual
            ↪  time=0.633..24 rows=1 loops=1)
                -> Nested loop inner join  (cost=2.5 rows=5) (actual
                ↪  time=0.046..0.14 rows=5 loops=1)
                    -> Filter: (d.HID is not null)  (cost=0.75 rows=5) (actual
                    ↪  time=0.0282..0.0425 rows=5 loops=1)
                        -> Covering index scan on D using HID  (cost=0.75
                        ↪  rows=5) (actual time=0.0261..0.0356 rows=5 loops=1)
                    -> Single-row index lookup on H using PRIMARY (HID=d.HID)
                    ↪  (cost=0.27 rows=1) (actual time=0.0182..0.0183 rows=1
                    ↪  loops=5)
                -> Filter: ((c.`Date` >= <cache>(curdate())) and (c.`Date` <
                ↪  <cache>((curdate() + interval 14 day))))
                    (cost=26.2 rows=221) (actual time=3.66..4.77 rows=0.2
                    ↪  loops=5)
                    -> Index lookup on C using DEP_ID (DEP_ID=d.DEP_ID)
                        (cost=26.2 rows=1993) (actual time=0.453..4.47 rows=2000
                        ↪  loops=5)
            -> Filter: (a.`Status` = 'Scheduled')  (cost=0.25 rows=0.333)
                (actual time=0.0253..0.0257 rows=1 loops=1)
                -> Single-row index lookup on A using PRIMARY (CAID=c.CAID)
                    (cost=0.25 rows=1) (actual time=0.0201..0.0203 rows=1
                    ↪  loops=1)
```

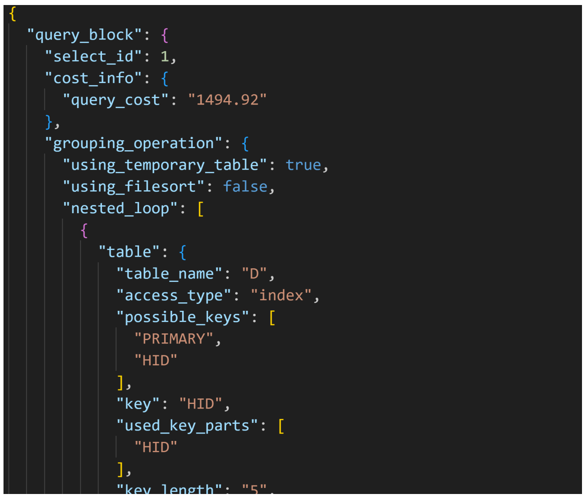
Before indexing. Before introducing the composite index, the optimizer drove the plan from `Department` into `ClinicalActivity` using `DEP_ID`. The date predicate was applied late, leading to large intermediate results, high estimated costs (around 1495), and execution times in the tens of seconds (approximately 22.8 seconds on average for 10k rows).

After Optimization

Execution Plan After Indexing

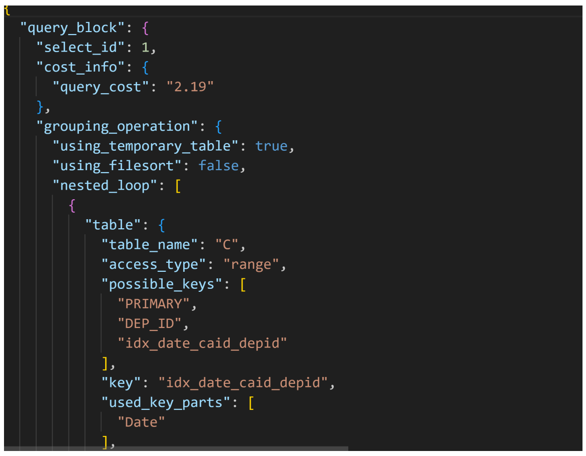
```
-> Table scan on <temporary>  (actual time=0.0627..0.063 rows=1 loops=1)
    -> Aggregate using temporary table  (actual time=0.0617..0.0617 rows=1
    ↪  loops=1)
        -> Nested loop inner join  (cost=2.19 rows=0.333) (actual
        ↪  time=0.0411..0.0448 rows=1 loops=1)
            -> Nested loop inner join  (cost=1.84 rows=1) (actual
            ↪  time=0.0331..0.0364 rows=1 loops=1)
                -> Nested loop inner join  (cost=1.49 rows=1) (actual
                ↪  time=0.0283..0.0314 rows=1 loops=1)
                    -> Filter: ((c.`Date` >= <cache>(curdate())) and (c.`Date` <
                    ↪  <cache>((curdate() + interval 14 day))))
                        (cost=1.14 rows=1) (actual time=0.0197..0.0226 rows=1
                        ↪  loops=1)
                        -> Covering index range scan on C using
                        ↪  idx_date_caid_depid
                            over ('2025-12-13' <= Date < '2025-12-27')
                            (cost=1.14 rows=1) (actual time=0.0158..0.0185
                            ↪  rows=1 loops=1)
                    -> Filter: (d.HID is not null)  (cost=0.35 rows=1)
                        (actual time=0.0077..0.0078 rows=1 loops=1)
                        -> Single-row index lookup on D using PRIMARY
                        ↪  (DEP_ID=c.DEP_ID)
                            (cost=0.35 rows=1) (actual time=0.0072..0.0072
                            ↪  rows=1 loops=1)
                -> Single-row index lookup on H using PRIMARY (HID=d.HID)
                    (cost=0.35 rows=1) (actual time=0.0043..0.0044 rows=1
                    ↪  loops=1)
            -> Filter: (a.`Status` = 'Scheduled')  (cost=0.283 rows=0.333)
                (actual time=0.0077..0.0079 rows=1 loops=1)
                -> Single-row index lookup on A using PRIMARY (CAID=c.CAID)
                    (cost=0.283 rows=1) (actual time=0.0058..0.0058 rows=1
                    ↪  loops=1)
```

After indexing. After creating the index `(Date, CAID, DEP_ID)`, the optimizer started with a covering range scan on `ClinicalActivity.Date`. Subsequent joins were performed using efficient primary-key lookups. The estimated cost dropped to approximately 2.19, and execution time decreased to around 0.06 seconds, representing a speedup of roughly 360–380×.



```
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "1494.92"
    },
    "grouping_operation": {
      "using_temporary_table": true,
      "using_filesort": false,
      "nested_loop": [
        {
          "table": {
            "table_name": "D",
            "access_type": "index",
            "possible_keys": [
              "PRIMARY",
              "HID"
            ],
            "key": "HID",
            "used_key_parts": [
              "HID"
            ],
            "key_length": "5",
```

(a) Query execution before indexing



```
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "2.19"
    },
    "grouping_operation": {
      "using_temporary_table": true,
      "using_filesort": false,
      "nested_loop": [
        {
          "table": {
            "table_name": "C",
            "access_type": "range",
            "possible_keys": [
              "PRIMARY",
              "DEP_ID",
              "idx_date_caid_depid"
            ],
            "key": "idx_date_caid_depid",
            "used_key_parts": [
              "Date"
            ],
```

(b) Query execution after indexing

Figure 1: Comparison of query execution results before and after indexing

4. Visualization of Index Impact

4.1 Python Plotting Code

```
import matplotlib.pyplot as plt
from statistics import mean

sizes = [10000, 50000, 100000, 500000, 1000000]
labels = ['10k', '50k', '100k', '500k', '1M']

before = {
    10000: [24.1, 16.7, 27.6],
    50000: [68.5, 62.1, 86.2],
    100000: [107, 112, 135],
    500000: [1202, 1156, 1118],
    1000000: [3145, 4892, 3762]
}

after = {
    10000: [0.063, 0.061, 0.064],
    50000: [0.068, 0.072, 0.064],
    100000: [0.069, 0.049, 0.039],
    500000: [0.055, 0.053, 0.065],
    1000000: [0.32, 1.44, 0.045]
}

before_avg = [mean(before[s]) for s in sizes]
after_avg = [mean(after[s]) for s in sizes]

plt.figure(figsize=(10,6))
plt.plot(labels, before_avg, marker='o', label='Without Index')
plt.plot(labels, after_avg, marker='s', label='With Index')
plt.yscale('log')
plt.xlabel('Dataset Size')
plt.ylabel('Execution Time (seconds)')
plt.legend()
plt.grid(True)
plt.savefig('idx_perf.png', dpi=300)
plt.show()
```

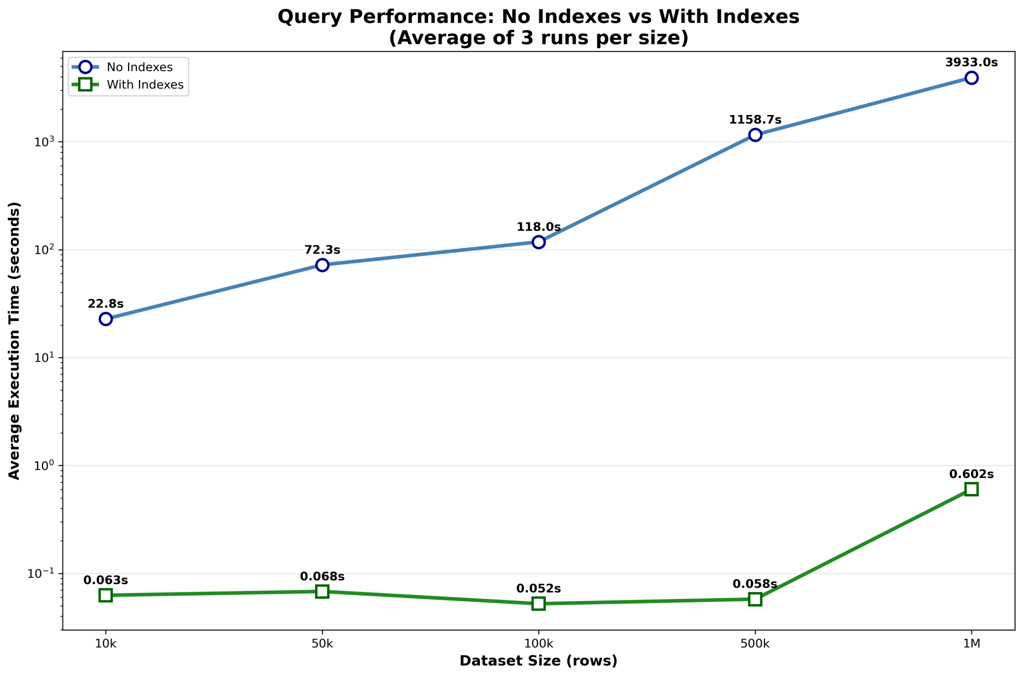


Figure 2: Query execution time with and without secondary indexes (log scale).

Interpretation. Without indexes, execution time grows rapidly with dataset size. With the appropriate index, execution time remains almost constant, demonstrating the scalability benefits of indexing.

Part II: Transactions and Concurrency Control

Part 1: Revisiting ACID Transactions

For each example or scenario below, identify which ACID property or properties is being satisfied or violated. Briefly justify your answer.

Example 1: A MNHS billing service records an **Expense** row linked to a ClinicalActivity and then updates the corresponding **Insurance** claim. After inserting the **Expense**, the system crashes before updating the claim. Upon recovery, the system detects the incomplete transaction and retries until both updates succeed.

Answer: Here the Atomicity property is being enforced, since even if there is a crash mid-transaction, once the system is up again it will make sure the transaction gets finished before any further action. Thus the transaction is enforced as being one singular action.

Example 2: Two MNHS receptionists attempt to book the *last* available appointment slot for the same doctor and time. Both use the web application concurrently and both receive a confirmation, but only one physical slot exists.

Answer: Here both Consistency and Isolation properties are violated, since at most **one** appointment can be booked for the same doctor at the same time (violating consistency), and isolation since transaction should act like they happened one after the other (even if they get interleaved in the background) thus one should not receive a confirmation of the appointment if we were to satisfy the Isolation property.

Example 3: A staff member (Staff A) enters new medications into a shared **Prescription/Includes** list for a patient. Another staff member (Staff B) is viewing the same patient's medication list through the application at the same time, but does not see Staff A's changes until Staff A clicks "Save" and the transaction is committed.

Answer: In the example, we see the Isolation property being enforced, since Staff B can't see half done work from Staff A, pressing "Save" is a signal for the transaction being committed. Preventing a Dirty Read!

Example 4: An administrative staff member registers a new patient (Patient and ContactLocation rows) and records an initial ClinicalActivity. After saving the activity, a power outage occurs before the data is flushed to durable storage. When the database restarts, the newly registered patient and activity are missing.

Answer: Here the Durability property has been violated, since the staff member saved the activity, but a crash voided the saved data, thus a committed activity was not saved.

Example 5: The pharmacy module ensures that every time a medication is dispensed, the corresponding **Stock.Qty** is reduced by exactly the dispensed amount, regardless of how many pharmacists are updating stock concurrently. The system never records negative stock or incorrect totals.

Answer: Here both Isolation and Consistency are being satisfied since the Stock.Qty is reduced (by the exact amount) regardless of how many updates there are while still enforcing business rules (No negative stock/ incorrect totals), underlying the fact that each transaction behaves like it is done one after the other all while respecting business constraints.

Part 2: Implementing Atomic Transactions in MySQL

In this section, you will move from reasoning about ACID to implementing atomic transactions in MySQL. The goal is to ensure ”all-or-nothing” behavior: either all statements of a logical operation succeed, or none of them are visible.

1. Atomic scheduling of an appointment.

Consider the following logical operation in MNHS:

- Create a new ClinicalActivity row,
- Create the corresponding Appointment row with Status = ’Scheduled’.

(a) Using MySQL transaction control, write a code fragment that groups these two INSERT statements into a single transaction.

```
START TRANSACTION;

INSERT INTO ClinicalActivity (CAID, IID, STAFF_ID, DEP_ID, Date,
   Time)
VALUES (10023, 501, 12, 3, '2025-01-20', '09:30:00');

INSERT INTO Appointment (CAID, Reason, Status)
VALUES (10023, 'Initial consultation', 'Scheduled');

-- if both INSERTs succeed:
COMMIT;

-- otherwise:
ROLLBACK;
```

(b) Explain how this transaction enforces the atomicity ACID property for this operation? What could go wrong if each INSERT were executed in autocommit mode as two separate transactions?

Answer: The transaction only commits after both insertions succeed, if one of these inserts return an error then there is no committing and we rollback; and if there is no error, the code commits and the insertions are saved.

The atomicity of the transaction is preserved since the transaction can only be committed after the whole transaction is over, otherwise it won’t be committed. Thus it is equivalent to the transaction being one singular block.

If each INSERT were executed in autocommit mode as two separate transactions there’s room for multiple errors,

One of them being a clinical activity being inserted without its associated appointment, resulting in a logic error

And another is the appointment being inserted with a CAID that is not defined in the Clinical Activities table resulting in a MySQL error.

2. Atomic update of stock and expense.

Suppose you implement a logical operation that:

- Updates Stock.Qty for all medications dispensed in a given Prescription (using Includes),
- Relies on your trigger logic to recompute the corresponding Expense.Total for the underlying ClinicalActivity.

(a) Describe (in pseudocode) how you would wrap these updates inside a single transaction so that a failure in any UPDATE or trigger execution causes a ROLLBACK.

```
START TRANSACTION;
try
    -- for each medication listed in this prescription
    for each row inc in Includes where inc.PrescriptionID =
        prescription_id do
         -- decrease stock for that medication
         UPDATE Stock
         SET Qty = Qty - inc.DispensedQty
         WHERE Stock.MedID = inc.MedID
         -- on each UPDATE, the Expense trigger fires
         -- and recomputes Expense.Total for the ClinicalActivity
    end for
    -- if we got here, all UPDATEs and triggers have not resulted
         in an error
    COMMIT;
    -- Thus committing is the signal that we're all good
catch any error (from UPDATE or trigger)
    ROLLBACK;
end try;
```

(b) Briefly indicate which ACID properties are particularly important for this scenario (atomicity, consistency, isolation, durability) and why?

Answer: Atomicity is the most important property here:

We're updating Stock.Qty and (via triggers) recomputing Expense.Total as one logical operation.

Atomicity ensures that either all stock updates and all related expense updates happen, or none of them do.

So we never end up with:

- stock reduced but expense not updated, or
- expense updated but stock not changed.

Part 3: Identifying Types of Schedules

Consider two simple transactions on the MNHS database:

$$T_1 : R(A), W(A)$$

$$T_2 : R(B), W(B)$$

You can think of A and B as attributes such as Stock.Qty for two different medications.
Schedules:

$$S_1 : R_1(A), R_2(B), W_1(A), W_2(B)$$
$$S_2 : R_1(A), W_1(A), R_2(B), W_2(B)$$

1. Are the schedules S_1 and S_2 equivalent? Justify your answer.

Answer: Two schedules S and S' are equivalent if:

- They contain the same operations of the same transactions
- For any database:
  - we have the same final state
  - and reads the same values, returns the same output

S1 and S2 contain the same operations of the same transactions
We have the same final state (W1(A) and W2(B) operate on different tables)
Read the same values (The write operates on the opposite tables thus the table is unchanged before the schedule)
Thus S1 and S2 are equivalent.

2. Is S_1 serializable? If yes, give an equivalent serial schedule.

Answer: S1 is equivalent to S2 which is a serial schedule.
Thus S1 is serializable.

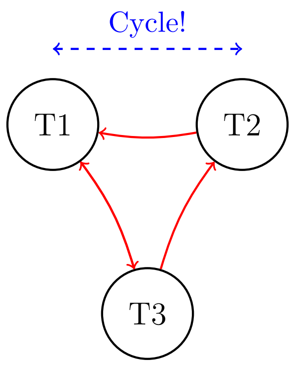
Part 4: Conflict Serializability

Consider three transactions on the MNHS database:

$$T_1 : R(A), W(A)$$
$$T_2 : W(A), R(B)$$
$$T_3 : R(A), W(B)$$

Schedule:
$$S_3 : R_1(A), W_2(A), R_3(A), W_1(A), W_3(B), R_2(B)$$

1. Construct the precedence (dependency) graph for S_3.



Precedence Graph Explanation: The graph shows a cycle between T1, T2, and T3, indicating that the schedule is not conflict serializable.

2. Is S_3 conflict serializable? Justify your answer using the graph.

Answer: The graph has a cycle thus, S3 is not conflict serializable. The cycle T1 $\rightarrow$ T3 $\rightarrow$ T2 $\rightarrow$ T1 (and also T3 $\rightarrow$ T1) means there is no serial schedule that is equivalent to S3.

Part 5: 2PL

Assume a standard lock-based concurrency control where transactions acquire shared/exclusive locks on data items (e.g., rows representing clinical activities, expenses, or stock).

For each schedule below, state whether it can result from a legal strict 2PL execution. Briefly justify your answer.

Schedule 1

$$T_1 : R(A), W(A), R(B), W(B)$$

$$T_2 : R(A), W(A)$$

$$S : R_1(A), W_1(A), R_1(B), W_1(B), R_2(A), W_2(A)$$

Answer: It is compatible with strict 2PL
We could do this: S: X1(A),R1(A),W1(A), X1(B), R1(B), W1(B), R-X1, X2(A),R2(A),W2(A)
Thus T2 acquires locks only after T1 acquires then releases them.

Schedule 2

$$T_1 : R(A), W(A), R(B)$$

$$T_2 : R(B), W(B)$$

$$S : R_1(A), W_1(A), R_2(B), W_2(B), R_1(B)$$

Answer: It is compatible with Strict 2-PL:
S : X1(A), R1(A), W1(A), X2(B), R2(B), W2(B), R-X2, X1(B) R1(B) R-X1
T1 never releases a lock before it has all it needs (A and B).
T2 holds X(B) until R-X2
T1 reads B only after T2 commits (R-X2).

Schedule 3

$$T_1 : R(A), W(A), R(B), W(B)$$

$$T_2 : R(C), W(C)$$

$$S : R_1(A), W_1(A), R_1(B), W_1(B), R_2(C), W_2(C)$$

Answer: It is compatible with strict 2-PL:
S: X1(A), R1(A), W1(A), X1(B), R1(B), W1(B), R-X1, X2(C), R2(C), W2(C).

Schedule 4

$$T_1 : R(A), W(A), R(B)$$

$$T_2 : R(A), W(A), R(B), W(B)$$

$$S : R_1(A), W_1(A), R_2(A), W_2(A), R_1(B), R_2(B), W_2(B)$$

Answer: It is **NOT** compatible with strict 2-PL:
T1 writes A before T2 accesses A, but later T1 still needs to read B. To let T2 access A, T1 would have to release its lock on A before it has acquired a lock on B, entering the shrinking phase and then acquiring a new lock. This violates the 2PL growing/shrinking rule (and strict 2PL's "hold write locks until commit").

Part 6: Deadlocks in MNHS

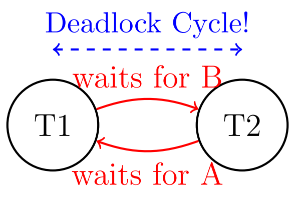
Consider the following situation on the MNHS database. Transaction T_1 is updating stock for a medication, while transaction T_2 is updating the corresponding expense for a clinical activity that uses that medication.

Schedule:

$$S : R_1(A), R_2(B), W_1(B), W_2(A)$$

Assume each read/write requires and holds a lock on the corresponding item.

1. Draw the wait-for graph for this schedule.



Wait-for Graph: T1 holds lock on A and waits for B, while T2 holds lock on B and waits for A.

2. Determine whether there is a deadlock. If yes, explain:

- **Which transaction is waiting on which?**
  Answer: T1 is waiting on T2 to release B (for W1(B)).
  T2 is waiting on T1 to release A (for W2(A)).

- **Where the cycle appears in the wait-for graph?**
  Answer: Exactly the 2-edge cycle: T1 → T2 → T1.

- **What the DBMS should do to resolve the deadlock?**
  Answer: Detect the cycle in the wait-for graph.
  Choose one transaction to abort/rollback (e.g. T2), and release its locks.
  Then the other transaction (T1) can proceed, after which the victim could be restarted if needed.

3 Discussion

For indexing, the trick was picking the right column order in composite indexes. Wrong indexes caused slow joins and late filtering; good ones made queries run hundreds of times faster. For transactions, the hard part was finding subtle bugs like double bookings or lost updates. We learned that strict locking prevents problems but can slow things down if overused. In healthcare, you can't afford mistakes, so balancing safety with speed is key.

4 Conclusion

This lab shows how good database design—both in how data is stored and how it's handled—keeps healthcare systems fast, consistent, and ready for real-world use.

Smart indexes and partitioning make queries run faster, while careful transaction design prevents errors, double bookings, and crashes when many users work at once. But both need upkeep: indexes can slow down over time as data changes, and locking must be balanced so it protects data without hurting performance.

Part 2 gave us insight into why ACID properties are crucial for functional, reliable databases and how to implement them in practice. We learned how to verify whether schedules are serializable, check strict 2PL properties, and manage locks to prevent issues like deadlocks.