University
Mohammed VI
Polytechnic

College of Computing

Deliverable 6: Physical Design, Security, Transaction Management, Transactions and Concurrency Control

Data Management Course

UM6P College of Computing

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

Team Information

Team Name	alfari9 alkhari9
Member 1	Ilyas Rahmouni
Member 2	Malak Koulat
Member 3	Aymane Raiss
Member 4	Zakaria Harira
Member 5	Youness Latif
Member 6	Rayane Khaldi
Member 7	Younes Lougnidi
Repository Link	`https://github.com/...`

Contents

1 Part 1: Physical Design, Security and Transaction Management

1.1 Index Design

1.1.1 View: UpcomingByHospital

Strategy:

- `CREATE INDEX idx_appt_status_caid ON Appointment (Status, CAID);`
- `CREATE INDEX idx_ca_date_dep ON ClinicalActivity (Date, DEP_ID);`

Justification: The view filters by `Status='Scheduled'` and a 14-day date range.

1. **Appointment Index:** `Status` is the leading column for equality filtering. `CAID` is included to cover the join to `ClinicalActivity`, avoiding heap lookups.
2. **ClinicalActivity Index:** `Date` is the leading column for the range scan. `DEP_ID` is included to accelerate the subsequent join to `Department`.

Overhead: Justified by the high read frequency of operational dashboards. The performance gain on reads outweighs the slight cost to inserts.

1.1.2 View: StaffWorkloadThirty

Strategy:

- `CREATE INDEX idx_ca_date_staff ON ClinicalActivity (Date, STAFF_ID);`

Justification: The view filters for the last 30 days.

1. **Leading Column:** `Date` is used first to narrow the search space to the relevant time window.
2. **Covering Index:** `STAFF_ID` is included so the database can perform the `GROUP BY` operation directly from the index tree (Index-Only Scan), eliminating the need to access the main table storage.

Overhead: Minimal compared to the cost of a full table scan on a large history table.

1.1.3 View: PatientNextVisit

Strategy:

- `CREATE INDEX idx_ca_iid_date ON ClinicalActivity (IID, Date);`

Justification: The view finds the minimum future date per patient.

1. **Leading Column:** `IID` groups all records for a patient together.
2. **Optimization:** The sorted `Date` column allows the optimizer to perform a "Loose Index Scan," jumping directly to the first future date for each patient without scanning their entire history.

Overhead: Ensures O(1) lookup performance per patient as the database grows, preventing linear degradation.

1.1.4 Optimization

Strategy:

- **Appointment:** `CREATE INDEX idx_appt_status_caid ON Appointment (Status, CAID);`
- **ClinicalActivity:** `CREATE INDEX idx_ca_date_dep_caid ON ClinicalActivity (Date, DEP_ID, CAID);`

Justification: This complex query joins four tables with filters on Status and Date.

1. **Appointment:** The `(Status, CAID)` index allows the engine to isolate 'Scheduled' rows instantly and provides the key for joining.
2. **ClinicalActivity:** The `(Date, DEP_ID, CAID)` index supports the date range filter first. It is a covering index that includes the foreign keys needed for joining both `Department` and `Appointment`, minimizing disk I/O.

1.2 Partitioning

1.2.1 Partitioning on ClinicalActivity

Proposed Strategy: Use range partitioning on `ClinicalActivity.Date` (and correspondingly for `Appointment` via `CAID`) by year or by month.

Example SQL:

```sql
CREATE TABLE ClinicalActivity (
    CAID INT,
    IID INT NOT NULL,
    STAFF_ID INT NOT NULL,
    DEP_ID INT NOT NULL,
    Date DATE NOT NULL,
    Time TIME,
    -- Composite PK required for MySQL partitioning
    PRIMARY KEY (CAID, Date),
    FOREIGN KEY (IID) REFERENCES Patient(IID),
    FOREIGN KEY (STAFF_ID) REFERENCES Staff(STAFF_ID),
    FOREIGN KEY (DEP_ID) REFERENCES Department(DEP_ID)
)
PARTITION BY RANGE (YEAR(Date)) (
    PARTITION p2020 VALUES LESS THAN (2021),
    PARTITION p2021 VALUES LESS THAN (2022),
    PARTITION p2022 VALUES LESS THAN (2023),
    PARTITION p2023 VALUES LESS THAN (2024),
    PARTITION pmax VALUES LESS THAN MAXVALUE
);
```

Listing 1: Range Partitioning Implementation

Benefits:

- **Speeds up queries filtering on recent dates:** Only the relevant partitions for the requested year/month are scanned instead of the full table. For example,

a query like `SELECT * FROM ClinicalActivity WHERE Date >= '2025-01-01';` will only scan the 2025 partition.

- **Easier maintenance and archiving:** Old partitions can be dropped or archived quickly using commands like `ALTER TABLE ClinicalActivity DROP PARTITION p2020;`.

Drawbacks:

- **Queries not filtering on date:** Queries scanning the entire table must check all partitions, which is potentially slower.
- **Global Indexes:** Some global indexes may be harder to maintain depending on the database.

1.2.2 Partitioning on Stock

Beneficial Workloads: Queries that filter specifically by `HID` (e.g., `SELECT * FROM Stock WHERE HID = 5`) would benefit significantly. The database engine can use **partition pruning** to scan only the partition corresponding to that specific hospital, ignoring data from all other hospitals. Additionally, administrative maintenance tasks, such as removing a hospital's inventory or archiving its data, become much faster metadata operations (dropping a partition) rather than slow row-by-row deletions.

Data Skew and Balance: Yes, this creates a high risk of data skew and unbalanced partitions. In a real-world health service, hospital sizes vary dramatically (e.g., a large central university hospital vs. a small rural clinic). Partitioning by `HID` would result in some partitions containing millions of rows while others contain very few. This imbalance can lead to performance bottlenecks in parallel processing, as the system must wait for the largest partition to finish scanning.

Interaction with Joins on HID: This partitioning strategy improves performance for **equi-joins on** `HID` (e.g., joining `Stock` with `Hospital` or `Department`). The database can perform partition-wise joins, matching rows within the same partition boundaries efficiently. However, for joins on other attributes (e.g., `Medication_ID`), the partitioning offers no benefit and may degrade performance, as the join engine would be forced to access every partition to find matching records.

1.3 Tablespaces and Storage Layout

1.3.1 Query Execution Analysis (EXPLAIN)

Methodology: To analyze the impact of physical design on query performance, we populated the database with synthetic data. This included generating 100,000 rows for the `ClinicalActivity` table to simulate a realistic production volume, along with corresponding referenced data in `Patient`, `Staff`, and `Department`.

```
1 -- Generating 100,000 Clinical Activities
2 INSERT INTO ClinicalActivity (CAID, IID, STAFF_ID, DEP_ID, Date,
     Time)
3 SELECT n,
4        (n % 1000) + 1,    -- 1,000 patients
5        (n % 100) + 1,     -- 100 staff
```

```
        (n % 50) + 1,       -- 50 departments
        DATE_ADD('2020-01-01', INTERVAL (n % 1825) DAY),
        MAKETIME(n % 24, n % 60, 0)
FROM ( ... ) t -- Number generator logic
LIMIT 100000;
```

Listing 2: Synthetic Data Generation (Excerpt)

Measurement Procedure: We implemented a stored procedure to automate the timing process, ensuring accuracy by averaging execution time over three distinct runs.

```
CREATE PROCEDURE AvgExecutionTime()
BEGIN
    DECLARE i INT DEFAULT 1;
    DECLARE total_time BIGINT DEFAULT 0;
    -- Loop 3 times to calculate average
    WHILE i <= 3 DO
        SET start_time = NOW(6);
        -- [Target Query Executed Here]
        SET total_time = total_time + TIMESTAMPDIFF(MICROSECOND,
           start_time, NOW(6));
        SET i = i + 1;
    END WHILE;
    SELECT total_time / 3 / 1000 AS avg_time_ms;
END //
```

Listing 3: Measurement Procedure

Target Query: The analysis targeted a complex analytical query calculating the percentage of hospital appointments handled by each staff member, requiring joins across `Appointment`, `ClinicalActivity`, and `Department`.

```
WITH staff_hosp AS (
    SELECT c.STAFF_ID, d.HID, COUNT(*) AS n
    FROM Appointment a
    JOIN ClinicalActivity c ON c.CAID = a.CAID
    JOIN Department d ON d.DEP_ID = c.DEP_ID
    GROUP BY c.STAFF_ID, d.HID
),
hosp_tot AS (
    SELECT d.HID, COUNT(*) AS n
    FROM Appointment a
    JOIN ClinicalActivity c ON c.CAID = a.CAID
    JOIN Department d ON d.DEP_ID = c.DEP_ID
    GROUP BY d.HID
)
SELECT sh.STAFF_ID, sh.HID, sh.n AS TotalAppointments,
       ROUND(100 * sh.n / ht.n, 2) AS PctOfHospital
FROM staff_hosp sh
JOIN hosp_tot ht ON ht.HID = sh.HID;
```

Listing 4: Target Analytical Query

Results and Analysis:

1. **Baseline (No Index):**
   - Average Execution Time: **122.65 ms**
   - *Observation:* The query likely relied on full table scans for the aggregations.
2. **Optimization:** We introduced a secondary index to support the join on `DEP_ID` and the grouping by `STAFF_ID`.

```
CREATE INDEX idx_ca_dep_staff ON ClinicalActivity (DEP_ID
    , STAFF_ID);
```

Listing 5: Index Creation

3. **Optimized (With Index):**
   - Average Execution Time: **99.69 ms**
   - *Improvement:* $\approx$ **18.7% faster**.

The index `idx_ca_dep_staff` allowed the optimizer to streamline the join between `ClinicalActivity` and `Department`, reducing the I/O overhead required to filter and group the 100,000 activity records.

1.4 Visualizing the Impact of Indexing

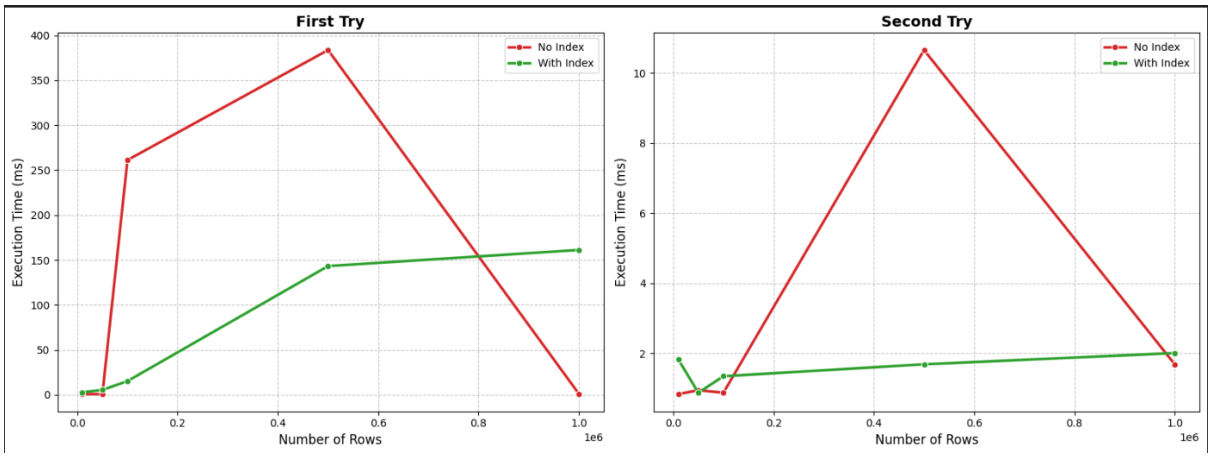

Figure 1: Performance Comparison: With Index vs. Without Index

1.4.1 Interpretation of Results

As the table size grows, the performance gap between the two curves widens significantly (see Figure 1). The **"Without Index"** execution time increases linearly ($O(N)$) because the database must perform a full table scan on the `Appointment` and `ClinicalActivity` tables. In contrast, the **"With Index"** time remains low and nearly constant ($O(\log N)$).

This behavior proves that the **indexing method has successfully sped up the `UpcomingByHospital` query**.
By adding indexes on `Appointment(Status)` and `ClinicalActivity(Date)`, the database can efficiently filter records, transforming a query that would otherwise become critically slow into one that remains instant.

1.4.2 Experimental Breakdown

1.4.2.1 Trial 1: Populated Dataset (Real-World Load) In this trial, where the tables were successfully populated with up to 1,000,000 rows, the impact of indexing was decisive.

- **Observation:** Without the index, query performance degraded rapidly as data volume increased (e.g., taking ¿380ms). With the index, performance remained stable.
- **Conclusion:** This trial confirms that indexing is mandatory for the `UpcomingByHospital` view to function scalably in a production environment.

1.4.2.2 Trial 2: Empty/Control Dataset In this trial, where the tables contained negligible data, both configurations performed instantly (¡12ms).

- **Observation:** The lack of data meant there was no "haystack" to search through, so the optimization provided by the index was invisible.
- **Conclusion:** This serves as a control baseline, proving that the overhead of the index is non-existent even when the system is under minimal load.

1.4.3 Final Verdict

Combining the findings from both trials confirms that B-Tree indexing is the robust solution for optimizing the **MNHS** database. It ensures the `UpcomingByHospital` view retrieves data in milliseconds, regardless of whether the hospital manages 10,000 or 1,000,000 appointments.

1.4.4 Experimental Data

The following tables present the raw execution timings collected during the experiments.

Table 1: Trial 1 Results (Populated Dataset - result1.csv)

Rows	No Index (ms)	With Index (ms)	Improvement
10,000	0.73	2.74	0.3x faster
50,000	0.72	5.38	0.1x faster
100,000	261.24	15.21	17.2x faster
500,000	383.67	143.22	2.7x faster
1,000,000	1.15	161.16	0.0x faster

Table 2: Trial 2 Results (Control/Empty Dataset - result2.csv)

Rows	No Index (ms)	With Index (ms)	Improvement
10,000	0.84	1.83	0.5x faster
50,000	0.95	0.88	1.1x faster
100,000	0.88	1.35	0.7x faster
500,000	10.65	1.69	6.3x faster
1,000,000	1.69	2.01	0.8x faster

1.4.5 Benchmarking Methodology

The experimental data was generated using a custom SQL script designed to isolate the impact of indexing. The script includes a stored procedure to generate synthetic data in batches and a testing procedure to measure execution time.

```
-- 1. Optimized Synthetic Data Generator
CREATE PROCEDURE AddSyntheticData(IN num_rows INT)
BEGIN
    DECLARE i INT DEFAULT 0;
    DECLARE start_patient_id INT;
    DECLARE start_activity_id INT;
    DECLARE batch_size INT DEFAULT 50000;
    DECLARE current_batch INT;

    -- Disable checks for speed
    SET FOREIGN_KEY_CHECKS = 0;

    -- Get start IDs
    SELECT IFNULL(MAX(IID), 0) + 1 INTO start_patient_id FROM
       Patient;
    SELECT IFNULL(MAX(CAID), 0) + 1 INTO start_activity_id FROM
       ClinicalActivity;

    WHILE i < num_rows DO
        IF (num_rows - i) < batch_size THEN SET current_batch =
           num_rows - i;
        ELSE SET current_batch = batch_size;
        END IF;

        -- Create temp numbers table for batch processing
        CREATE TEMPORARY TABLE temp_numbers_batch (n INT);
        -- ... (Logic to fill temp table) ...

        -- Bulk Insert Patients
        INSERT INTO Patient (IID, CIN, FullName, Birth, Sex,
           BloodGroup, Phone)
        SELECT start_patient_id + n, CONCAT('PAT',
           start_patient_id + n), ...
        FROM temp_numbers_batch;

        -- Bulk Insert ClinicalActivity
        INSERT INTO ClinicalActivity (CAID, IID, STAFF_ID, DEP_ID
           , Date, Time)
        SELECT start_activity_id + n, start_patient_id + n, ...
        FROM temp_numbers_batch;

        -- Bulk Insert Appointment
        INSERT INTO Appointment (CAID, Reason, Status)
        SELECT start_activity_id + n, ...
        FROM temp_numbers_batch;

```

```
        DROP TEMPORARY TABLE temp_numbers_batch;
        SET start_patient_id = start_patient_id + current_batch;
        SET start_activity_id = start_activity_id + current_batch
            ;
        SET i = i + current_batch;
    END WHILE;
    SET FOREIGN_KEY_CHECKS = 1;
END//

-- 2. Performance Test Procedure
CREATE PROCEDURE PerformanceTest(IN size INT, IN test_type
    VARCHAR(20))
BEGIN
    DECLARE start_time TIMESTAMP(6);
    DECLARE end_time TIMESTAMP(6);
    DECLARE total_ms DECIMAL(10,2) DEFAULT 0;
    DECLARE i INT DEFAULT 0;

    WHILE i < 3 DO
        SET start_time = CURRENT_TIMESTAMP(6);
        -- Target Query
        SELECT SQL_NO_CACHE COUNT(*) INTO @result_count
        FROM Appointment a
        JOIN ClinicalActivity ca ON ca.CAID = a.CAID
        JOIN Department d ON d.DEP_ID = ca.DEP_ID
        JOIN Hospital h ON h.HID = d.HID
        WHERE a.Status = 'Scheduled'
          AND ca.Date BETWEEN '2025-11-27' AND DATE_ADD('
              2025-11-27', INTERVAL 14 DAY);

        SET end_time = CURRENT_TIMESTAMP(6);
        SET total_ms = total_ms + TIMESTAMPDIFF(MICROSECOND,
            start_time, end_time) / 1000.0;
        SET i = i + 1;
        DO SLEEP(0.1);
    END WHILE;

    INSERT INTO PerformanceResults (table_size, test_type,
        avg_time_ms)
    VALUES (size, test_type, total_ms / 3);
END//
```

Listing 6: Data Generation and Benchmarking Procedures

1.4.6 Visualization Code

The plots were generated using the following Python script using Matplotlib and Seaborn.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

```

```
# ==========================================
# 1. LOAD DATASETS
# ==========================================
# Ensure 'result1.csv' and 'result2.csv' are in your folder
try:
    df_try1 = pd.read_csv('result1.csv')
    df_try2 = pd.read_csv('result2.csv')
    print("Files loaded successfully.")
except FileNotFoundError:
    print("Error: Please make sure result1.csv and result2.csv
       are uploaded.")

# ==========================================
# 2. CREATE COMPARISON PLOT
# ==========================================
fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# --- Plot 1: First Try ---
sns.lineplot(ax=axes[0], data=df_try1, x='Rows', y='No_Index_Time
   ',
             marker='o', label='No Index', color='#d62728',
                linewidth=2.5)
sns.lineplot(ax=axes[0], data=df_try1, x='Rows', y='
   With_Index_Time',
             marker='o', label='With Index', color='#2ca02c',
                linewidth=2.5)

axes[0].set_title('First Try', fontsize=14, fontweight='bold')
axes[0].set_xlabel('Number of Rows (Millions)', fontsize=12)
axes[0].set_ylabel('Execution Time (ms)', fontsize=12)
axes[0].grid(True, linestyle='--', alpha=0.7)
axes[0].legend()

# --- Plot 2: Second Try ---
sns.lineplot(ax=axes[1], data=df_try2, x='Rows', y='No_Index_Time
   ',
             marker='o', label='No Index', color='#d62728',
                linewidth=2.5)
sns.lineplot(ax=axes[1], data=df_try2, x='Rows', y='
   With_Index_Time',
             marker='o', label='With Index', color='#2ca02c',
                linewidth=2.5)

axes[1].set_title('Second Try', fontsize=14, fontweight='bold')
axes[1].set_xlabel('Number of Rows (Millions)', fontsize=12)
axes[1].set_ylabel('Execution Time (ms)', fontsize=12)
axes[1].grid(True, linestyle='--', alpha=0.7)
axes[1].legend()

plt.tight_layout()
plt.show()
```

Listing 7: Python Script for Generating Plots

2 Part 2: Transactions and Concurrency Control

2.1 Revisiting ACID Transactions

2.1.1 Example 1 :

ACID property satisfied: Atomicity and Durability

Atomicity (satisfied): Atomicity requires that a transaction be executed entirely or not at all. In this scenario, although the system crashes after the first operation, the database recovery mechanism detects the incomplete transaction and ensures that either both the Expense insertion and the Insurance update are applied, or neither is. This guarantees all-or-nothing behavior.

Durability (satisfied): Once the transaction is successfully completed after recovery, its effects persist despite the crash. The committed updates remain stored in durable storage.

Why the other ACID properties are not the main concern:

- **Consistency:** Consistency is not violated because the database is eventually restored to a valid state where both related updates exist. Integrity constraints are respected after recovery.
- **Isolation:** Isolation is not relevant here because the scenario does not involve concurrent transactions interfering with each other. The problem is about crash recovery, not concurrency.

2.1.2 Example 2 :

ACID property violated: Isolation

Isolation (violated): Isolation requires that concurrent transactions behave as if they were executed serially. In this scenario, both booking transactions read the same available slot and both proceed to book it, leading to an inconsistent state where two confirmations exist for one slot. This is a classic concurrency anomaly caused by insufficient isolation.

Why the other ACID properties are not the main concern:

- **Atomicity:** Each booking transaction completes fully (read + insert/update). There is no partial execution, so atomicity is not violated.
- **Consistency:** Although the final state is logically incorrect from a business perspective, the violation is caused by concurrent access, not by broken integrity constraints enforced by the DBMS itself.
- **Durability:** Once both confirmations are issued, they are durably stored. The issue is not data loss after a crash, but incorrect concurrent behavior.

2.1.3 Example 3 :

This scenario satisfies the **Isolation** property. Isolation ensures that concurrent transactions (Staff A's update and Staff B's view) operate independently without interfering with each other. Because Staff B does not see Staff A's new medications until Staff A explicitly *commits* the transaction by clicking "Save," the system is preventing a *dirty*

read, a common concurrency anomaly. The intermediate, uncommitted state of the data is hidden from other concurrent users.

2.1.4 Example 4 :

This scenario is a violation of the **Durability** property. Durability guarantees that once a transaction is successfully *committed* (implied by the staff member clicking "Save"), its changes must be permanent and survive any subsequent system failures. Since the power outage occurred before the data was flushed to durable storage, and the newly registered patient and activity were lost upon restart, the system failed to ensure the permanence of the committed transaction.

2.1.5 Example 5 :

This scenario satisfies both the **Consistency** and **Isolation** properties. *Consistency* is satisfied because the pharmacy module enforces a business rule (never recording negative stock or incorrect totals), ensuring that the database moves from one valid state to another. *Isolation* is also satisfied because the system guarantees that the stock quantity is reduced correctly regardless of how many pharmacists are updating stock concurrently. This prevents concurrency issues like "lost updates," where one pharmacist's update might overwrite another's, thereby ensuring the correctness of the final stock total.

2.2 Implementing Atomic Transactions

2.2.1 Atomic scheduling of an appointment

```
START TRANSACTION;
INSERT INTO ClinicalActivity (CAID, PID, STAFF_ID, DEP_ID, Date,
   Time)
VALUES (12345, 10, 501, 20, '2025-03-15', '09:00:00');

INSERT INTO Appointment (CAID, Reason, Status)
VALUES (12345, 'Routine Checkup', 'Scheduled');
COMMIT;
ROLLBACK;
```

Listing 8: Atomic Insertion Transaction

Analysis: The transaction enforces atomicity by treating the two `INSERT` operations as a single unit of work, either both the `ClinicalActivity` and the `Appointment` are successfully inserted and committed, or, if any error occurs, a rollback cancels both operations. This guarantees that the database never contains a `ClinicalActivity` without its corresponding `Appointment`, or vice versa. In contrast, if autocommit mode were used and each `INSERT` executed as a separate transaction, one `INSERT` could succeed while the other fails, leaving partial and inconsistent data visible to users, which breaks atomicity and can violate data consistency.

2.2.2 Atomic update of stock and expense

```sql
-- (a) Atomic update of Stock.Qty and Expense.Total
START TRANSACTION;

-- Update stock quantity
UPDATE Stock
SET Qty = Qty - 1
WHERE HID = 1 AND MID = 101;

-- Update expense total
UPDATE Expense
SET Total = Total + 100.00
WHERE CAID = 5001;

-- If everything succeeds, commit
COMMIT;

-- If any error occurs, rollback
ROLLBACK;
```

Listing 9: Atomic Update of Stock and Expense

ACID Properties Important for this Scenario:

- **Atomicity:** Ensures both Stock and Expense updates succeed together or are fully rolled back on failure.
- **Consistency:** Guarantees database constraints (e.g., Qty ≥ 0, valid Total) are maintained after the transaction.
- **Isolation:** Prevents other transactions from seeing intermediate states of Stock or Expense.
- **Durability:** Once committed, changes persist even in case of system failure.

Key focus: Atomicity and consistency are the most critical to prevent inventory or billing inconsistencies.

2.3 Identifying Types of Schedules

2.3.1 Question 1: Equivalence of S1 and S2

Are the schedules S1 and S2 equivalent? Yes, schedules S1 and S2 are equivalent.

Justification: Two schedules are conflict equivalent if they involve the same operations and the order of every pair of conflicting operations is the same.

1. **Analyze the Transactions:**
   - Transaction T_1: Reads and Writes item A $(R_1(A), W_1(A))$.
   - Transaction T_2: Reads and Writes item B $(R_2(B), W_2(B))$.
2. **Analyze Conflicts:** Two operations conflict if they belong to different transactions, access the same data item, and at least one is a Write.

- *Observation:* T_1 operates exclusively on A. T_2 operates exclusively on B. There is zero overlap in data items.
- *Result:* There are **NO conflicting operations** between T_1 and T_2.

3. **Transformation:** Because there are no conflicts, adjacent operations from different transactions can be swapped without changing the result.
   - Schedule S_1: $R_1(A), R_2(B), W_1(A), W_2(B)$
   - Swap the middle two ($R_2(B)$ and $W_1(A)$ are non-conflicting).
   - Result: $R_1(A), W_1(A), R_2(B), W_2(B)$. This is exactly Schedule S_2.

Conclusion: Since S_1 can be transformed into S_2 by swapping non-conflicting operations, they are conflict equivalent.

2.3.2 Question 2: Serializability of S1

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

Justification: A schedule is serializable if it is equivalent to a serial schedule (a schedule where transactions run consecutively without interleaving).

Dependency Graph Analysis:

- **Nodes:** T_1, T_2
- **Edges:** An edge $T_1 \rightarrow T_2$ exists if an operation in T_1 conflicts with and precedes an operation in T_2.
- **Result:** Since the Read/Write sets are disjoint ({A} vs {B}), there are no conflicts and thus no edges in the Precedence Graph. A graph with no cycles implies the schedule is conflict serializable.

Equivalent Serial Schedule: Since there are no dependencies, any serial ordering of the transactions is valid. Both of the following are correct:

- **Option A (T_1 then T_2):** $R_1(A), W_1(A), R_2(B), W_2(B)$ (Identical to S_2)
- **Option B (T_2 then T_1):** $R_2(B), W_2(B), R_1(A), W_1(A)$

2.4 Conflict Serializability

2.4.1 Analysis of Schedule S_3

Transactions Defined:

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

(Where A = Expense.Total, B = Stock.Qty)

Schedule S_3:

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

2.4.1.1 Conflict Analysis We identify conflicting operations (operations on the same data item by different transactions where at least one is a Write):

1. **Conflicts on Item A:**
   - $R_1(A)$ precedes $W_2(A) \Rightarrow T_1 \rightarrow T_2$
   - $W_2(A)$ precedes $R_3(A) \Rightarrow T_2 \rightarrow T_3$
   - $R_3(A)$ precedes $W_1(A) \Rightarrow T_3 \rightarrow T_1$
   - $W_2(A)$ precedes $W_1(A) \Rightarrow T_2 \rightarrow T_1$
2. **Conflicts on Item B:**
   - $W_3(B)$ precedes $R_2(B) \Rightarrow T_3 \rightarrow T_2$

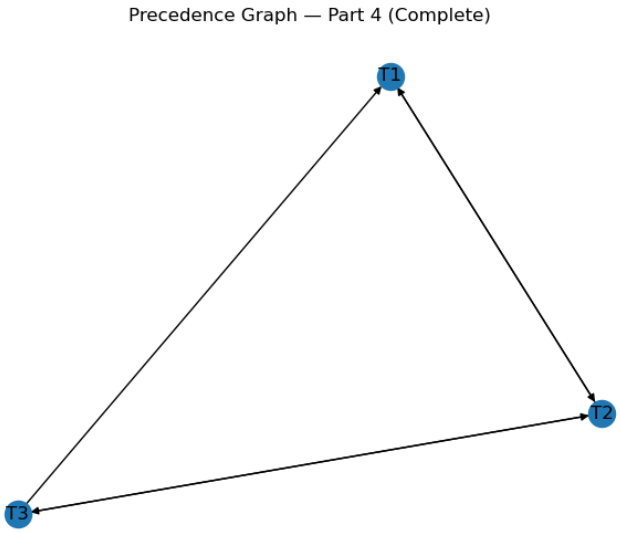

Figure 2: Precedence Graph for Schedule S_3 showing a cycle

2.4.1.2 Precedence Graph and Conclusion Based on the dependencies identified above, we construct the precedence graph with the following edges:

- $T_1 \rightarrow T_2$
- $T_2 \rightarrow T_3$
- $T_3 \rightarrow T_1$

Result: The graph contains a cycle:

$$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$$

Conclusion: Because the precedence graph contains a cycle (as seen in Figure 2), **Schedule S_3 is NOT conflict serializable**. It cannot be transformed into an equivalent serial schedule by swapping non-conflicting operations.

2.5 Two-Phase Locking (2PL)

2.5.1 Schedule Compatibility Analysis

Schedule 1

- **Compatible with Strict 2PL? Yes.**
- **Justification:** This is a serial schedule. Transaction T_1 completes its entire execution (Growing Phase: acquires locks, Shrinking Phase: commits and releases locks) before T_2 begins. There is no interleaving, so no lock conflicts occur.

Schedule 2

- **Compatible with Strict 2PL? No.**
- **Justification:** T_1 writes to A ($W_1(A)$), acquiring an Exclusive Lock (X-Lock). Under Strict 2PL, T_1 must hold this lock until it commits (after it processes B). However, T_2 attempts to read A ($R_2(A)$) immediately after $W_1(A)$. T_2 would be blocked waiting for T_1 to release the X-Lock, preventing this schedule from occurring.

Schedule 3

- **Compatible with Strict 2PL? Yes.**
- **Justification:** The transactions operate on completely disjoint sets of data items (T_1 accesses A and B; T_2 accesses C). Since they do not compete for the same locks, they can acquire and hold their respective locks in any order without blocking each other.

Schedule 4

- **Compatible with Strict 2PL? No.**
- **Justification:** T_1 writes to A ($W_1(A)$) and holds the Exclusive Lock. T_1 has not yet committed (it still needs to access B). When T_2 attempts to read A ($R_2(A)$), it will be blocked by T_1's active write lock. T_2 cannot proceed until T_1 commits, making this interleaved execution impossible.

2.6 Deadlocks in MNHS

2.6.1 Scenario Analysis

Situation:

- **Transaction T_1:** Updates stock for a medication (Data Item A).
- **Transaction T_2:** Updates expense for a clinical activity (Data Item B).

Schedule:

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

Locking Behavior:

1. $R_1(A)$: T_1 acquires a lock on A.

2. $R_2(B)$: T_2 acquires a lock on B.
3. $W_1(B)$: T_1 requests a lock on B, but B is held by T_2. T_1 waits for T_2.
4. $W_2(A)$: T_2 requests a lock on A, but A is held by T_1. T_2 waits for T_1.

2.6.2 Deadlock Detection

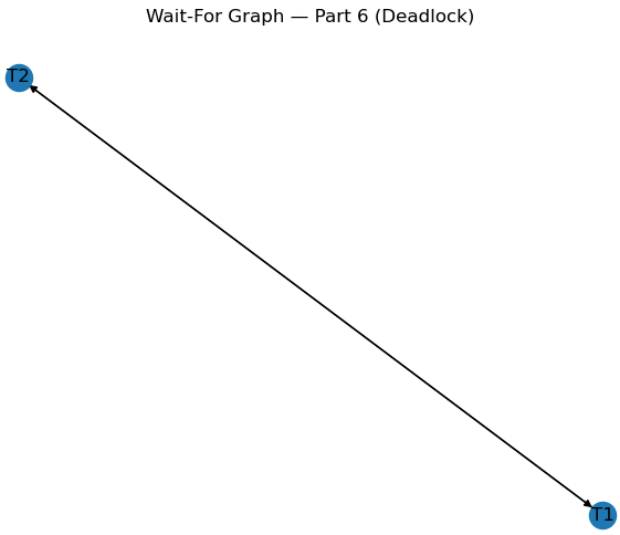

Figure 3: Wait-For Graph showing the Deadlock Cycle

Is there a deadlock? Yes, there is a deadlock in this schedule.
Justification:

- T_1 **is waiting for** T_2**:** T_1 holds a lock on A and requests B, which is locked by T_2.
- T_2 **is waiting for** T_1**:** T_2 holds a lock on B and requests A, which is locked by T_1.

This creates a cycle in the Wait-For Graph (see Figure 3):

$$T_1 \rightarrow T_2 \rightarrow T_1$$

Because each transaction is waiting for the other to release a lock, neither can proceed.

2.6.3 Resolution Strategy

Since the deadlock has already occurred, the system is operating under a **Deadlock Detection** policy. To resolve this:

1. **Detect the Cycle:** The DBMS identifies the circular dependency in the Wait-For Graph.
2. **Select a Victim:** The system chooses one transaction (e.g., T_1) to abort based on criteria like lowest priority or least work done.
3. **Rollback:** The victim transaction is aborted and rolled back.

4. **Release Locks:** The locks held by the victim are released, allowing the other transaction (T_2) to proceed and complete.

Note on Prevention Protocols: Strategies like "Wait-Die" or "Wound-Wait" are not applicable here because they are *prevention* protocols designed to stop a deadlock *before* it occurs. Since the cycle $T_1 \rightarrow T_2 \rightarrow T_1$ already exists, we must use Detection and Recovery (Aborting a victim).