UM6P
University
Mohammed VI
Polytechnic

College of Computing

Deliverable #6: Physical Design & Transaction Management

Data Management Course

UM6P College of Computing

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

Session: Fall 2025

Team Information

Team Name	Groupe2
Member 1	Abir Fakhreddine
Member 2	Malak El Assali
Member 3	Nada El Farissi
Member 4	Amine Chrif
Member 5	Anass Fertat
Member 6	Yasser Hallou
Repository Link	`https://github.com/beaNoBeebea`

Part I

Part 1: Physical Design, Security and Transaction Management

1 Introduction

This deliverable offers an in-depth exploration of database performance optimization and transaction management within the Moroccan National Health Services (MNHS) system. Building on earlier stages of the project, it applies more advanced physical design techniques and concurrency control mechanisms to respond to the real challenges of large-scale healthcare data. These challenges include declining query performance as data grows, the need to efficiently partition and manage historical records, and ensuring consistent data access when many users interact with the system at the same time. By laveraging advanced MySQL features such as secondary indexing, range and hash partitioning, different transaction isolation levels, and locking strategies, we were able to improve system responsiveness while preserving data integrity. Developed as part of the Data Management course, this work reflects a solid understanding of key concepts like query execution plans, optimal index selection., ACID guarantees, and conflict serializability. The project combines thoughtful design choices with hands-on performance testing, resulting in clear performance gains while upholding the reliability expectations of a healthcare information system.

2 Index Design

2.1 Index Design for UpcomingByHospital View

2.1.1 Index #1: Composite Index on Appointment (Status, CAID)

- **Which predicates/JOINS it accelerates?**
  - This index allows us to filter by appointments, this means we can keep those who are relevant to the view: only the appointments where (status = 'Scheduled').
  - It also helps join using `CAID` Consequence: searching is faster as we don't loop through every single appointment each time the view is executed, the join is also more efficient that way.
- **Leading column choice justification:**
  - Here, we filter the table on `status` before joining, so we narrow down the table and only join the necessary rows using `CAID`.

- **Overhead on INSERT/UPDATE operations:**
  - There is a small overhead when we create new appointment or when we update the status of an appointment. This is acceptable because often times reads are significantly more frequent than writes, so it is better to keep the indexing.

2.1.2 Index #2: Composite Index on ClinicalActivity (Date, DEP_ID, CAID)

- **Which predicates/JOINS it accelerates?**
  - The index allows us to keep only the appointments scheduled in the following two weeks, this doesn't take account of many unnecessary rows so it speeds up the process. After filtering, this query joins on `DEP-ID` and then on `CAID`.
- **Leading column choice justification:**
  - Choosing date as a leading column allows us to only focus on the small, relevant part of the data for our view.
- **Overhead on INSERT/UPDATE operations:**
  - There is some overhead when inserting rows into `ClinicalActivity` due to additional sorting and insertion work.
  - This is acceptable if the reads >> writes.

2.1.3 Index #3: Composite Index on Department (HID, DEP_ID)

- **Which predicates/JOINS it accelerates?**
  - In the view, MySQL retrieves a hospital and must find all departments in that hospital, therefore, indexing on HID and DEP-ID makes this process much more efficient.
- **Leading column choice justification:**
  - We start the join with `Hospital` → `Department`, so we join using `HID` first..
- **Overhead on INSERT/UPDATE operations:**
  - The overhead is very small since departments rarely change within hospitals, therefore, writing is minimal.

2.2 Index Design for StaffWorkloadThirty View

The `StaffWorkloadThirty` view calculates 30-day performance metrics for staff members by joining `Staff`, `ClinicalActivity`, and `Appointment` tables. The tables used in the view are:

- Staff
- ClinicalActivity
- Appointment

2.2.1 Index on ClinicalActivity Table

We created a composite index on `ClinicalActivity(Staff_ID, Date)`.

- **Which predicates/JOINS it accelerates?**
  - The index on `CA(Staff_ID)` accelerates the JOIN condition on `CA.Staff_ID = S.Staff_ID`.
  - The index on `CA(Date)` accelerates the filtering: `"CA.Date >= CURDATE() - INTERVAL 30 DAY"`, so that when searching, the Clinical Activities are already filtered by their date and therefore are faster to find.
- **Why is Staff_ID the leading column?**
  - `STAFF_ID` should be the leading column because the query first filters by staff (`CA.STAFF_ID = S.STAFF_ID`) and only then by date range (`CA.Date >= ...`).
  - With the index on `(STAFF_ID, Date)`, the database can go straight to one staff member's rows and then only scan their last 30 days, instead of scanning all recent rows for all staff and then filtering by staff afterward.
- **Overhead on INSERT/UPDATE?**
  - Every time a new row is inserted into `ClinicalActivity`, the database updates the index `(staff_id, date)` which slows write operations, and adds extra disk writes for each insert.

Note: We also might consider indexing `CA.CAID` or `A.CAID` to accelerate the other joins, but both are primary keys of `ClinicalActivity` and `Appointment`, so they are automatically indexed.

2.3 Index Design for PatientNextVisit View

- In this part, we'll design secondary indexes for base tables used by the PatientNextVisit view.
The tables used are:

- Patient
- ClinicalActivity
- Appointment
- Department
- Hospital

2.3.1 Index #1 On ClinicalActivity

ClinicalActivity appears more than once in that view; first in the main query (ca.IID), in the subquery (ca2.IID), and we also compute MIN(ca2.Date). This is a clear clue as to the need of an index on (IID, Date).

- **Which predicates/JOINS it accelerates?**
  - It helps the join condition: `ca.IID = p.IID`.
  - It helps the filters `ca.Date > CURDATE()` in the main query, and `ca2.Date > CURDATE()` in the subquery.
  - It also helps computing `MIN(ca2.Date)` for each IID, because the dates are already sorted inside each IID in the index.
- **Why is IID the leading column?**
  - The view we're analyzing is PatientNextVisit, so we care about per-patient next dates.
  - Putting IID first means the index groups rows by patient, and inside each patient group, rows are ordered by Date.
- **Overhead on INSERT/UPDATE?**
  - Whenever we insert a new row into ClinicalActivity, MySQL inserts it into the table but also inserts it into the index in the correct sorted position, which results in a bit more work.
  - The same goes when we update an IID or a Date for a row, that means we have to move the index entry to rearrange the rows in the right order.
  - So we get slower writes to ClinicalActivity, plus the obvious extra disk space to store the index.

2.3.2 Index #2 On Appointment

Appointment is used two times, in the main query (CAID), and in the subquery (CAID, Status). CAID is the primary key of Appointment, so it already has an index on CAID, but we also filter a lot on Status.

- **Which predicates/JOINS it accelerates?**
  - It helps the filter `apt.Status = 'Scheduled'` and `apt2.Status = 'Scheduled'`.
- **Why is Status the leading column?**
  - Almost every query in this view filters on `Status = 'Scheduled'`. Moreover, putting Status first means the index is grouped by Status and all 'Scheduled' appointments are together.

  - This makes it easy for the optimizer to select only the 'Scheduled' rows, instead of scanning all statuses.

- **Overhead on INSERT/UPDATE?**
  - There is some extra work done during insertion, because, with this index, besides inserting into the table and the primary key index, it now also has to be inserted into idx_app.
  - The same goes for UPDATEs. Every time we update, the index entry must be changed too; so this is equivalent to a delete then a reinsert.
  - So to reiterate, writes to Appointment get a bit slower, and extra disk space is used.

2.4 Index Design for Frequent Query Pattern

The following query pattern is executed frequently by the MNHS application for hospital reporting:

```
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;
```

We propose the following secondary indexes to optimize this query:

2.4.1 Index #1: Composite Index on Appointment (Status, CAID)

- **Creation:**

```
CREATE INDEX idx1 ON Appointment (Status, CAID);

```

- **Optimizer Usage:** Instead of searching the whole appointment table row by row, this index helps us to jump immediately to rows where status = "Scheduled" is located. By including CAID, it extracts that value for the join operations without the need to retrieve the whole row.

2.4.2 Index #2: Composite Index on ClinicalActivity (Date, DEP_ID)

- **Creation:**

```
CREATE INDEX idx2 ON ClinicalActivity (Date, DEP_ID);

```

- **Optimizer Usage:** Same as the first index—instead of reading every record on ClinicalActivity table, it helps by jumping immediately to the start date until it reaches end date. By including DEP_ID, it provides the values for the join operations.

2.4.3 Additional Supporting Indexes

To further optimize the query, we added two supporting indexes:

```sql
CREATE INDEX idx3 ON Department(DEP_ID, HID);
CREATE INDEX idx4 ON Hospital(HID, Name);
```

3 Partitioning Strategies

3.1 Range Partitioning by Date (ClinicalActivity and Appointment)

- ClinicalActivity and Appointement are perpetually used throughout the previous labs - ie. PatientNextVisit view - joined with a condition on ClinicalActivity.Date. So it appears it would be beneficial to partition based on it as the amount of data scla es - Only ClinicalActivity has a Date column. - Appointment does not have a Date column; it just points to ClinicalActivity via CAID. - So our strategy will be to partition the ClinicalActivity table by Date - We partition the ClinicalActivity table by year of the Date column using RANGE partitioning on YEAR(Date).

```sql
PARTITION BY RANGE (YEAR(Date)) (
    PARTITION p2018 VALUES LESS THAN (2019),
    PARTITION p2019 VALUES LESS THAN (2020),
    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 p2024 VALUES LESS THAN (2025),
    PARTITION pMax  VALUES LESS THAN MAXVALUE
);
```

- Some of our most interesting queries (like PatientNextVisit) involve:

```sql
ClinicalActivity.Date > CURDATE()
apt.Status = 'Scheduled'
```

- Joining Appointment to ClinicalActivity via CAID. - Because ClinicalActivity is now split by YEAR(Date): - So, if today is in 2024, and we search for future visits:

```sql
WHERE ca.Date > CURDATE()
```

- With the partitioning, the database will only need to look into: p2024, pMax. It can completely skip over p2018, p2019, p2020, etc. - This reduces the number of rows read and improves performance, especially when most queries focus on recent data, and makes older data management faster and safer. - However, queries that do not filter on Date cannot take advantage of partition pruning. On the contrary, the database may need to scan multiple or all partitions, which adds some overhead. - For those queries, partitioning does not help and can be slightly slower than a single non-partitioned table.

3.2 Hash Partitioning by HID (Stock)

- **Rationale:** Stock is used multiple times during the management of the database, like for compute expense total, staffworkloadthirty, and others... so it would be beneficial to partition Stock on HID. More specifically we choose to partition by Hospital ID because most of our queries are hospital based ones (examples: pricing per hospital in DrugPricingSummary, the expense computation trigger joins Stock using the HID of the CA's hospital, and other smaller tasks that include the "low stock per hospital" and "staff share within their hospital"). All that said, partitioning by hospital will enable us to avoid scanning rows for other hospitals and therefore improve performance.
- **Implementation:**

```sql
CREATE TABLE Stock (
    HID INT,
    MID INT,
    StockTimestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
    UnitPrice DECIMAL(10,2) CHECK (UnitPrice >= 0),
    Qty INT DEFAULT 0 CHECK (Qty >= 0),
    ReorderLevel INT DEFAULT 10 CHECK (ReorderLevel >= 0),
    PRIMARY KEY (HID, MID, StockTimestamp),
    FOREIGN KEY (HID) REFERENCES Hospital(HID),
    FOREIGN KEY (MID) REFERENCES Medication(MID))
PARTITION BY HASH (HID)
PARTITIONS 16;

```

- **Workloads that benefit:** Workloads that benefit the most from this partitioning are the ones that filter or group by a specific hospital. For example, DrugPricingSummary computes pricing per Hospital, so the queries access Stock by HID. The expense computation trigger also joins Stock using HID of the clinical activity's hospital, so every trigger execution goes through the stock rows for one hospital. Moreover, tasks like checking low stock per hospital and calculating staff share within their hospital repeatedly access stock data within one single HID, so partitioning Stock on HID reduces significantly the unnecessary scans of unneeded Stock rows.
- **Potential data skew:** Yes, in the case where some hospitals have much more data than other because they are more active/bigger then their partition become of big size whereas small hospitals' partition stays small.
- **Interaction with joins on HID:** Partitioning Stock by HID necessarily helps joins that use HID since the Stock rows are already filtered and will read only the partitions that contain that specific HID. But of course if a join involves all or many hospitals, then the

partition doesn't reduces the work that much because many if not all partitions must be accessed anyway.

4 Performance Analysis

4.1 Data Generation Methodology

- **Objective:** To create realistic synthetic datasets for performance testing of indexes and queries across different data volumes.
- **Implementation:** We developed a MySQL stored procedure to populate ClinicalActivity and Appointment tables with synthetic data spread over multiple years.

```sql
DELIMITER //
CREATE PROCEDURE PopulateClinicalData(IN n INT)
BEGIN
    DECLARE i INT DEFAULT 1;
    WHILE i <= n DO
        INSERT INTO ClinicalActivity (CAID, IID, STAFF_ID, DEP_ID, Date, Time)
        VALUES (
            i,
            FLOOR(1 + RAND() * 1000),
            FLOOR(1 + RAND() * 100),
            FLOOR(1 + RAND() * 50),
            DATE_ADD('2020-01-01', INTERVAL FLOOR(RAND() * 365 * 5) DAY),
            SEC_TO_TIME(FLOOR(RAND() * 86400))
        );
        INSERT INTO Appointment (CAID, Reason, Status)
        VALUES (
            i,
            CONCAT('Consultation Reason ', i),
            ELT(1 + FLOOR(RAND() * 3), 'Scheduled', 'Completed', 'Cancelled')
        );
        SET i = i + 1;
    END WHILE;
END //
DELIMITER ;

```

- **Data Characteristics:**
  - Random patient IDs (range: 1-1000)
  - Random staff IDs (range: 1-100)
  - Random department IDs (range: 1-50)
  - Dates distributed between 2020-2025
  - Random times throughout the day
  - Appointment statuses randomly assigned: ’Scheduled’, ’Completed’, or ’Cancelled’
- **Usage:** This procedure was executed multiple times with different parameters (n = 10,000, 50,000, 100,000, 500,000, 1,000,000) to generate datasets of varying sizes for performance benchmarking.

4.2 EXPLAIN Experiment: Before and After Indexing

4.2.1 Performance Comparison Analysis

- **Query:**

```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 ;

```

- **Access Paths:**
  - **Without Index:** While some tables were accessed via fast lookups (type: eq_ref, type: ref), the EXPLAIN output for the Hospital table showed type: ALL, indicating a highly inefficient Full Table Scan. This means the database had to read every row in the table.
  - **With Index:** The EXPLAIN output for the Hospital table switched to type: index, with the chosen key being idx4. This confirms the optimizer used an efficient Index Seek to jump directly to the relevant records.
- **Estimated Rows:**

  - **Without Index:** The estimated rows for the ClinicalActivity table was approximately 2080 rows. This high number confirms the optimizer planned to inspect nearly every record.
  - **With Index:** The estimated rows for the Appointment table dropped to approximately 1960 rows. This reduction proves the index successfully filtered the data before the join process began.

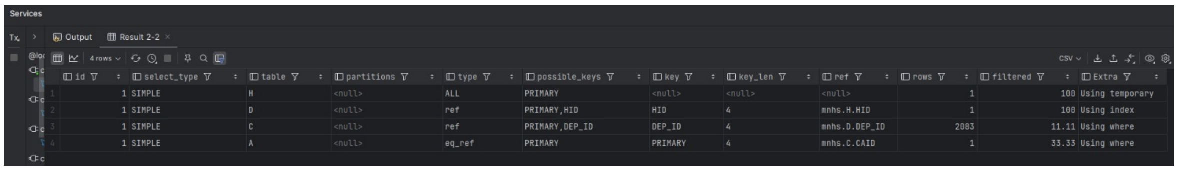
- **Join Order:**
  - **Without Index:** Without an index, the database knows the first step that needs to be checked is A.Status = 'Scheduled'. To avoid this slow start, it shifts and starts with faster operations like joins, meaning it will join on the full table before checking A.Status = 'Scheduled'.
  - **With Index:** The optimizer's plan changes because starting with checking A.Status = 'Scheduled' is fast. It begins with this operation and then passes to join operations, so it joins the minimum possible and gains a lot of time.

- **Timing Difference:**

Table 1: EXPLAIN Experiment Performance Comparison

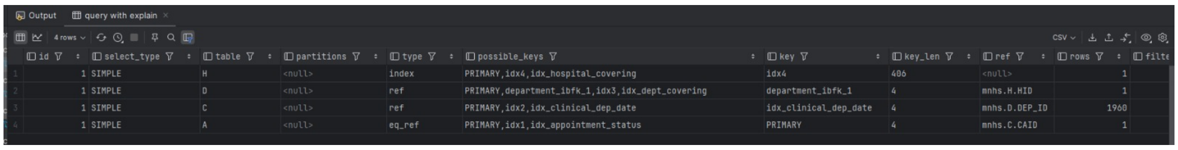
Metric	**Without Index**	**With Index**
Execution Time (Run 1)	10 ms	0 ms
Execution Time (Run 2)	11 ms	4 ms
Execution Time (Run 3)	11 ms	6 ms
Average Execution Time	**10.667 ms**	**3.333 ms**
Performance Improvement	**Baseline**	**68.8% faster**

4.2.2 Visual Evidence



	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	H	<null>	ALL	PRIMARY	<null>	<null>	<null>	1	100	Using temporary
2	1	SIMPLE	D	<null>	ref	PRIMARY,HID	HID	4	mnhs.H.HID	1	100	Using index
3	1	SIMPLE	C	<null>	ref	PRIMARY,DEP_ID	DEP_ID	4	mnhs.D.DEP_ID	2083	11.11	Using where
4	1	SIMPLE	A	<null>	eq_ref	PRIMARY	PRIMARY	4	mnhs.C.CAID	1	33.33	Using where

Figure 1: EXPLAIN Output Without Index



	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filte
1	1	SIMPLE	H	<null>	index	PRIMARY,idx4,idx_hospital_covering	idx4	406	<null>	1	
2	1	SIMPLE	D	<null>	ref	PRIMARY,department_ibfk_1,idx3,idx_dept_covering	department_ibfk_1	4	mnhs.H.HID	1	
3	1	SIMPLE	C	<null>	ref	PRIMARY,idx2,idx_clinical_dep_date	idx_clinical_dep_date	4	mnhs.D.DEP_ID	1960	
4	1	SIMPLE	A	<null>	eq_ref	PRIMARY,idx1,idx_appointment_status	PRIMARY	4	mnhs.C.CAID	1	

Figure 2: EXPLAIN Output With Index

4.3 Visualizing the Impact of Indexing

Table 2: Query Execution Time Scaling

Rows	Without Index (s)	With Index (s)	Speedup
10,000	0.110	0.031	3.5x
50,000	0.125	0.125	1.0x
100,000	0.500	0.172	2.9x
500,000	2.735	0.687	4.0x
1,000,000	18.219	3.078	5.9x

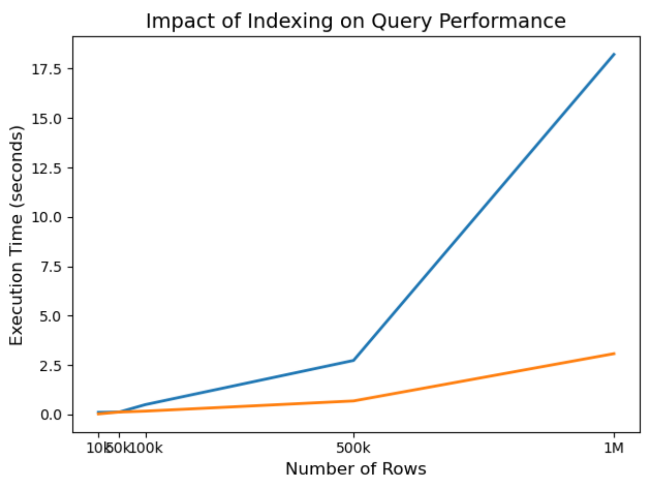


Figure 3: Query Performance: With vs. Without Index

Indexing is very efficient as dataset size grows, reducing time execution of queries (18.2s for 1M rows without index to 3.1s with index).

Part II

Part 2: Transactions and Concurrency Control

5 ACID Properties Analysis

5.0.1 Example 1: Billing and Insurance Claim Recovery

- **Scenario:** 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.

- **ACID Analysis:**
  - **Atomicity:** Satisfied. Although the system crashed in the middle, the system detects this anomaly upon recovery and retries the command until both Expense insertion and Insurance updated succeeded. This transaction verifies the all or nothing property.
  - **Consistency:** Satisfied. If the transaction doesn't commit, the system is already in a valid state. If it does commit, the system insures that the Expense row and the insurance claim stay coherent with one another.
  - **Isolation:** Not particularly addressed by this example, so we assume it is satisfied.
  - **Durability:** Satisfied. Once the transaction commits, the changes are performed and stored on the database.

5.0.2 Example 2: Double-Booking Appointment Slot

- **Scenario:** 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.
- **ACID Analysis:**
  - **Atomicity:** If we're looking at each individual booking, then atomicity is verified. However, the fact that two valid atomic transactions are taking place at the same time poses a problem.
  - **Consistency:** Violated. The final state of the database is incoherent, it has two confirmed appointments for the exact same doctor and time. This violates the integrity constraints.
  - **Isolation:** Violated. Both users think the slot is available before commit of the transactions. Since the transactions are not properly isolated, both of them commited and created inconsistency.
  - **Durability:** Satisfied. If we're looking at the separate transactions, after each receptionist commits, the changes persist.

5.0.3 Example 3: Concurrent Medication List Access

- **Scenario:** 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.
- **ACID Analysis:**
  - **Atomicity:** The changes appear once all of them are saved so it respects the fact that a transaction is indivisible.

- **Isolation:** When transaction of staff A runs at the same time as staff B is reading, staff B is not aware of the changes staff A is making untill the transaction is finished, so they can only the new medication after the commit without seeing the progress of the transaction beforehand.
- **Consistency and Durability:** Example3 does not describe the violation of the other two ACID properties, since there was no issue mentioned when committing (so Consistency is also satisfied), and once the changes are saved, they remain in the system (so durability is also satisfied)
- **Overall:** All ACID properties are satisfied and NONE were violated.

5.0.4 Example 4: Power Outage During Patient Registration

- **Scenario:** An administrative staff member registers a new patient (Patient and Contact-Location 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.
- **ACID Analysis:**
  - **Durability:** Violated. When a transaction is committed its changes should be permanent so that if a system failure happens the changes made should still be present. However the situation in example4 does not satisfy that: even if the changes were committed, after the system failure the changes were not saved.
  - **Atomicity:** Satisfied. The changes are saved all at once, and when the crash happened both patient and activity disappeared.
  - **Consistency and Isolation:** Example4 does not describe the violation of the other two ACID properties (isolation and consistency), since there was no issue mentionned when committing (so Consistency is also satisfied), and isolation because a simultaneous operation was not mentionned.
  - **Overall:** Atomicity, Consistency and isolation are satisfied but durability is not.

5.1 Example 5: Pharmacy Stock Management

- **Scenario:** 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.
- **ACID Analysis:**
  - **Consistency:** This example highlights the Consistency property, because every successful transaction keeps Stock.Qty within valid bounds, never negative or incorrect totals so that it follows already set business rules.

- **Isolation:** In this example, transaction isolation is used in such a way that concurrent updates to one same row are serialized, meaning it's always one and then the other in a way that doesn't break the correctness of the system. So regardless of how many pharmacists are updating stock concurrently, each transaction remains consistent and deals with updates in such a way that other updates aren't compromised.
- **Atomicity:** In this example, we have two tracks of actions. Either we check the stock, subtract amount from Stock.Qty and save everything, or something fails along the way and the whole transaction is aborted, so Stock.Qty remains unchanged. This showcases how everything is done in one transaction, and if any part of the transaction fails, no partial or incorrect update remains.

- **Overall:** No particular ACID properties were found violated in this example. All properties (Atomicity, Consistency, Isolation, Durability) are satisfied.

6 Atomic Transaction Implementation

6.0.1 Atomic Scheduling of an Appointment

- **Implementation:**

```sql
START TRANSACTION;

INSERT INTO ClinicalActivity (CAID, IID, STAFF_ID, DEP_ID, Date, Time)
VALUES (90001, 1004, 2001, 51, '2025-03-10', '09:00:00');

INSERT INTO Appointment (CAID, Reason, Status)
VALUES (90001, 'Consultation Cardiaque', 'Scheduled');

-- if everything is fine:
COMMIT;
-- otherwise:
ROLLBACK;

```

- **Atomicity Enforcement:** This transaction groups the two inserts (on ClinicalActivity and Appointment): they either both succeed or both fail, this verifies all or nothing atomicity property. This means we never end up with a Clinical activity that has no matching appointments.
- **Risk without Atomicity:** If we were to execute each INSERT in autocommit mode as two separate transactions, the database's integrity constraints could be violated. For example, in the case where we insert a clinical activity first and it commits, but the system crashes unexpectedly and the insert into appointments fails. In this case we end up with a clinical activity that has no corresponding appointment, this violates integrity constraints.

6.0.2 Atomic Update of Stock and Expense

- **Pseudocode Implementation:**

```
BEGIN TRANSACTION
Update Stock.Qty for all medication in the prescription
For each Clinical Activity the trigger
    Expense_total_update updates its expense
if any step fails(whether it is the Update or the trigger), ROLLBACK
if everything succeeds, COMMIT
```

- **Critical ACID Properties:**
  - **Atomicity:** Extremely important since the updating of the stock must either all succeed or not at all so that we don't end up with modification in the stock medication but no recomputation of the expense therefore corrupting the inventory and financial information about the data.
  - **Consistency:** Verified by the trigger itself and database constraints so that we don't have invalid data.
  - **Isolation:** Prevents other transactions from reading from a half-updated table and getting the wrong expense.

7 Schedule Analysis and Serializability

7.1 Identifying Types of Schedules (S1 & S2)

Consider two simple transactions on the MNHS database:

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

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

Where A and B could be attributes such as Stock.Qty for two different medications.
Schedules:

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

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

- **Are schedules S_1 and S_2 equivalent?**
  - We know that A and B are different items; they're the same attribute, but for different objects/rows.

  - In checking the conflicts between T_1 and T_2:
    - There's no conflict between $R_1(A)$ and $R_2(B)$ as A and B are different items.
    - No conflict between $R_1(A)$ and $W_2(B)$ for the same reason.
    - No conflict between $W_1(A)$ and $R_2(B)$.
    - No conflict between $W_1(A)$ and $W_2(B)$.
  - So there is no conflict at all between T_1 and T_2.
  - This means that within S_1 and S_2 we have the same internal order for T_1 and T_2, and since there is no conflict between T_1 and T_2, we can conclude that S_1 and S_2 are in fact equivalent.

- **Is S_1 serializable? If yes, give an equivalent serial schedule.**
  - Yes it is, because S_2 is a serial schedule (T_1 followed by T_2), and as shown by the previous question, S_1 and S_2 are equivalent.
  - So, S_1 is also serializable.
  - Equivalent serial schedule: T_1 followed by T_2 (which is S_2).

8 Conflict Serializability

Consider three transactions on the MNHS database: - T1 : R(A), W (A) - T2 : W (A), R(B) - T3 : R(A), W (B)

Schedule: - S3 : R1(A), W2(A), R3(A), W1(A), W3(B), R2(B)

8.1 Precedence Graph Construction

- Conflict on A (R_1 vs W_2):$R_1(A)$ happens before $W_2(A)$. Edge: $T_1 \rightarrow T_2$
- Conflict on A (W_2 vs R_3):$W_2(A)$ happens before $R_3(A)$. Edge: $T_2 \rightarrow T_3$
- Conflict on A (W_2 vs W_1):$W_2(A)$ happens before $W_1(A)$. Edge: $T_2 \rightarrow T_1$
- Conflict on A (R_3 vs W_1):$R_3(A)$ happens before $W_1(A)$. Edge: $T_3 \rightarrow T_1$
- Conflict on B (W_3 vs R_2):$W_3(B)$ happens before $R_2(B)$. Edge: $T_3 \rightarrow T_2$

8.2 Precedence Graph Visualization

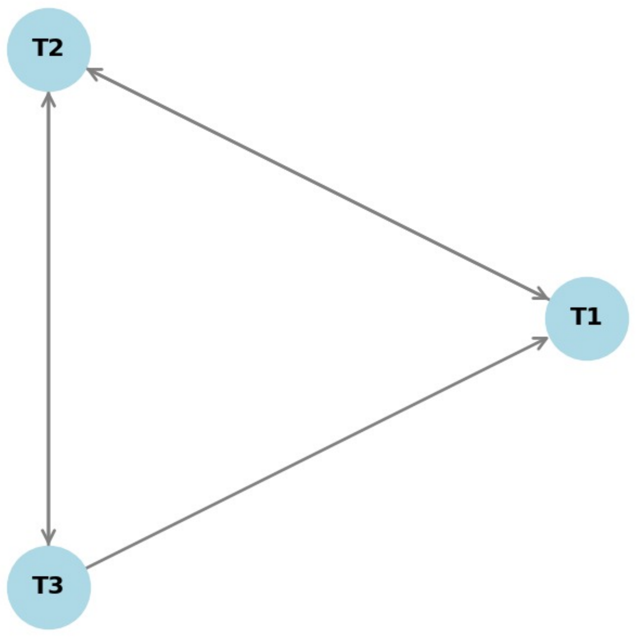


Figure 4: Precedence Graph for Schedule S3 Showing Conflict Cycles

The graph contains multiple cycles ($T_1 \leftrightarrow T_2$ and $T_2 \leftrightarrow T_3$). Therefore, schedule S_3 is not conflict serializable.

9 Two-Phase Locking (2PL) Analysis

9.1 Schedule 1 Analysis

No, it is not **2PL** Strict because T_1 acquires an **Exclusive Lock** ($\mathbf{X}(\mathbf{A})$) and it must hold it until the transaction is complete. The attempt by T_2 to enter its **Growing Phase** by acquiring $\mathbf{X}(\mathbf{A})$ is blocked by the conflicting lock held by T_1.

9.2 Schedule 2 Analysis

No, it is not **2PL** Strict because T_2 acquires an $\mathbf{X}(\mathbf{B})$ and without releasing this lock, T_1 tries to acquire an **Shared Lock** ($\mathbf{S}(\mathbf{B})$). This attempt to continue the T_1 **Growing Phase** is blocked by the conflicting lock held by T_2.

9.3 Schedule 3 Analysis

The schedule **3** follows the **2PL** rule because T_1 and T_2 operate on disjoint data items, ensuring no conflict in the lock acquisitions. Both transactions complete their **Growing Phase** without entering the **Shrinking Phase** (No shrinking phase). It satisfies the strictness because all locks were held until the end of the transaction.

9.4 Schedule 4 Analysis

No, it is not **2PL** Strict because T_1 is in its **Growing Phase** by acquiring **Shared Lock** $\mathbf{X}(\mathbf{A})$; however, T_2 holding a shared $\mathbf{S}(\mathbf{A})$ lock causes a conflict.cae

10 Deadlock Analysis and Resolution

- We have two transactions `T1` and `T2` in MNHS system:

- `T1` has `Stock A`, wants `Expense B`
- `T2` has `Expense B`, wants `Stock A`

Each transaction is waiting for the other → this is deadlock.
- **Wait-for graph:**

- T1 → T2 (waits for B)
- T2 → T1 (waits for A)

Cycle in wait-for graph means deadlock exists.
- **Resolution:** There are three ways to resolve the deadlock:

- Deadlock prevention
- Deadlock avoidance: assign priorities to transactions.
- Deadlock detection and resolution: create a waits-for-graph