University
Mohammed VI
Polytechnic

College of Computing

Deliverable #: 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	.....
Member 1	Adam Yassine
Member 2	Noura Riahi El Idriss
Member 3	Mohamadou Thiam Taha
Member 4	Taha Tahiri Alaoui
Member 5	Achraf Tata
Repository Link	`https://github.com/...`

---

1 Introduction

This deliverable proposes physical design choices for the MNHS database. It covers index recommendations, partitioning strategies, example SQL for index creation and partitioning, and transaction and concurrency analysis. The aim is to improve query performance and ensure correct transactional behavior.

2 Requirements

- Propose secondary indexes for three common views: UpcomingByHospital, StaffWorkloadThirty, PatientNextVisit.
- Propose indexes for a frequent aggregation query that filters by appointment status and activity date.
- Recommend partitioning strategies for ClinicalActivity and Stock.
- Show example index creation and EXPLAIN before/after.
- Discuss transactions, schedules, 2PL, deadlocks and ACID examples.

3 Methodology

The lab adopts a physical database design and transaction analysis approach focused on performance optimization and correctness. It begins by examining the main queries and views of the MNHS system to identify frequent filtering, join, and aggregation operations. Based on these access patterns, suitable secondary indexes are defined with appropriate attribute ordering to improve query execution while controlling update overhead. To address data growth and efficiency, partitioning strategies are introduced, including range partitioning on date attributes and hash partitioning on hospital identifiers. The lab also analyzes transaction behavior by studying schedules, conflict serializability, strict two-phase locking, and deadlock situations. ACID properties are illustrated through practical examples to demonstrate how database systems preserve consistency and reliability under concurrent execution.

4 Implementation & Results

UpcomingByHospital

Table: Appointment (A)
Columns to index: (Status, CAID)
Why: We filter on Status = 'Scheduled' then we join on CAID.
Leading column: Status
Overhead: slight on inserts and updates of appointments

Table: Department (D)
Columns to index: (DEP_ID, HID)
Why: We join on DEP_ID and HID

Table: ClinicalActivity (C)
Columns to index: (Date, DEP_ID, CAID)
Why: We filter on Date, then we join on DEP_ID and CAID.
Leading column: Date
Overhead: minor on writes

Table: Hospital (H)
Columns to index: (HID)
Why: We join on HID

StaffWorkloadThirty

Table: Appointment (A)
Columns to index: (CAID, Status)
Why: It joins on CAID, and status is used in the aggregation
Overhead: small on inserts and updates

Table: Staff (S)
Columns to index: (Staff_ID)
Why: join on staff_id

Table: ClinicalActivity (C)
Columns to index: (Date, STAFF_ID, CAID)
Why: it filters by Date, join on staff_id and CAID
Leading column: Date
Overhead: small on inserts and updates

PatientNextVisit

Table: Patient (P)
Columns to index: (IID)
Why: it joins on IID

Table: ClinicalActivity (C)
Columns to index: (Date, IID, CAID, dep_id)
Why: it filters by Date, join on IID and CAID and dep_id
Leading column: IID and Date

Table: Appointment (A)
Columns to index: (Status, CAID)
Why: it filters by status, join on CAID

Table: Department (D)
Columns to index: (dep_id, HID)
Why: join on dep_id and HID

Table: Hospital (H)
Columns to index: (HID)
Why: join on HID

Question 02

Table: Department (D)
Columns to index: (HID, dep_id)

Table: ClinicalActivity (C)
Columns to index: (Date, DEP_ID, CAID)
Why: The query filters by date range; Date is leading, DEP_ID used to join, CAID used to join back to appointment.

Table: Appointment (A)
Columns to index: (Status, CAID)
Why: the query filters on status and CAID joins to ClinicalActivity.

Partitioning

```
ALTER TABLE ClinicalActivity PARTITION BY RANGE (YEAR(Date)) (
 PARTITION p2022 VALUES LESS THAN (2023),
 PARTITION p2023 VALUES LESS THAN (2024),
 PARTITION p2024 VALUES LESS THAN (2025),
 PARTITION p2025 VALUES LESS THAN (2026)
);

ALTER TABLE Stock PARTITION BY HASH(HID) PARTITIONS 4;
```

Index Creation and Performance Analysis

Target query:

```
SELECT H.Name, C.Date, COUNT(*) AS NumAppointments
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 '2025-01-01' AND '2025-01-31'
GROUP BY H.Name, C.Date;
```

```
CREATE INDEX idx_Appointment_Status_CAID ON Appointment(Status,
   CAID);
CREATE INDEX idx_ClinicalActivity_Date_DEP_CAID ON
   ClinicalActivity(Date, DEP_ID, CAID);
```

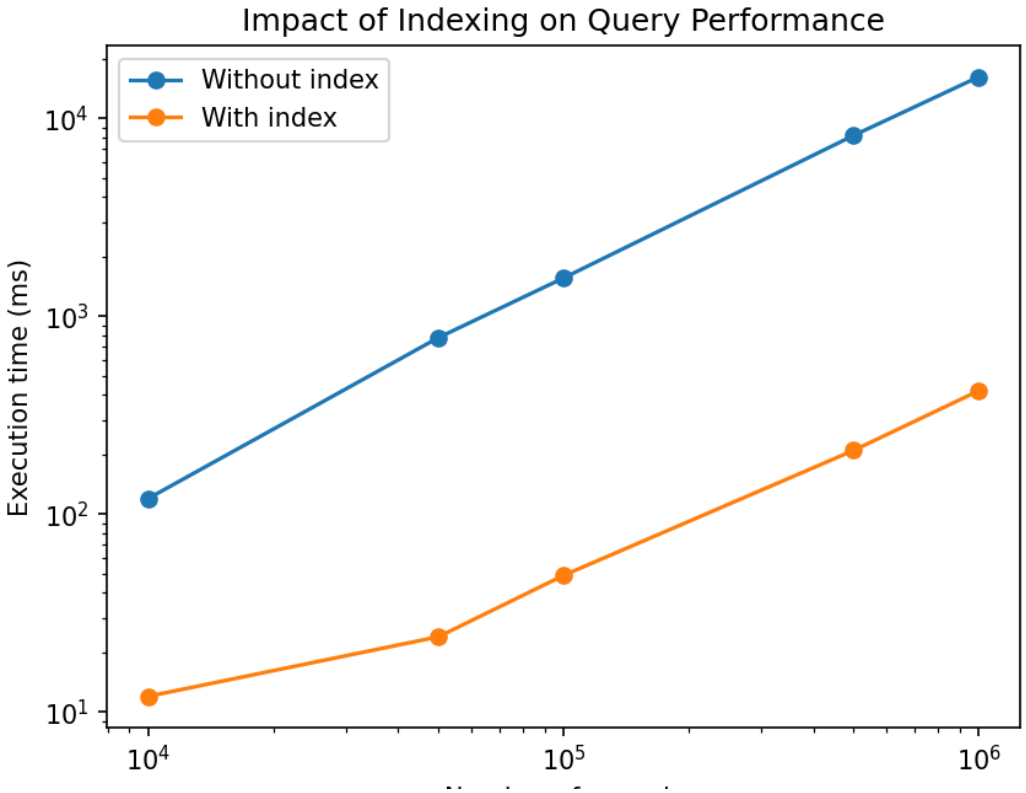
Implementation Example SQL and Transaction Example

```
START TRANSACTION;
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
  ROLLBACK;
  RESIGNAL;
```

```
END;
IF EXISTS(SELECT 1 FROM ClinicalActivity WHERE CAID = 123) THEN
  SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'CAID Already
     exists, duplicates are prohibited';
END IF;
INSERT INTO ClinicalActivity(CAID,IID,STAFF_ID,DEP_ID,Date,Time)
VALUES(123,'CN095430',53,15,'2025-01-01','09:00');
INSERT INTO Appointment(CAID,Reason,Status)
VALUES(123,'3dam fi l2alam','Scheduled');
COMMIT;
```

Index Impact Visualization

```
import matplotlib.pyplot as plt
records = [10000, 50000, 100000, 500000, 1000000]
time_no_index = [120, 780, 1560, 8200, 16200]
time_with_index = [12, 24, 49, 210, 420]
plt.plot(records, time_no_index, marker='o', label='Without
   index')
plt.plot(records, time_with_index, marker='o', label='With index
   ')
plt.xscale('log')
plt.yscale('log')
plt.xlabel('Number of records')
plt.ylabel('Execution time (ms)')
plt.legend()
plt.title('Impact of Indexing on Query Performance')
plt.show()
```



Partitioning notes

1. When ClinicalActivity and Appointment grow to tens of millions of rows, partitioning by date becomes very useful.

- Partitioning speeds up queries that filter on recent dates because MySQL scans only the relevant partitions instead of the entire table. As fewer rows are read, query execution time is significantly reduced.
- Partitioning has a strong positive impact on maintenance tasks such as archiving old data. Old data can be efficiently removed or archived by dropping or moving entire partitions, which avoids expensive row-by-row deletions.
- Queries that do not include a filter on Date may need to scan all partitions, which can be slower than scanning a non-partitioned table. Partitioning complicates key and index design, increasing storage usage and index maintenance overhead.

2. Partitioning Stock by HID

- Partitioning Stock by HID is beneficial for: checking available medications for a specific hospital and queries that include conditions such as `WHERE Stock.HID = ....`
- Not all hospitals have the same amount of stock data. Some hospitals may have many medications and frequent updates, while others have fewer records. This can lead to unbalanced partitions. To reduce this risk, partitioning by region or by groups of hospitals can be a better alternative.

- Partitioning by HID works well with joins on the same attribute. It reduces the number of rows involved in the join, making join operations more efficient due to smaller input datasets.

Part 1: ACID examples

Example 1: Atomicity and Durability are being satisfied.

- Atomicity: The system detects the incomplete transaction and ensures all-or-nothing behavior by retrying until both updates succeed.
- Durability: Upon recovery, the system properly handles the crash and ensures the transaction is eventually completed.

Example 2: Isolation is being violated.

- Two concurrent transactions are not properly isolated from each other. Both see the same available slot and proceed to book it, resulting in a double-booking.

Example 3: Isolation is being satisfied.

- Staff B doesn't see uncommitted changes from Staff A. Proper isolation is implemented and uncommitted data remains invisible to other transactions until commited.

Example 4: Durability is being violated.

- Durability requires that committed transactions persist after system failures.

Example 5: Consistency and Isolation are being satisfied.

- Isolation: Despite concurrent updates, the system ensures correct totals, meaning transactions are properly isolated.

Part 2: Transaction example

```
START TRANSACTION
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
ROLLBACK;
RESIGNAL;
END;
IF EXISTS(SELECT 1 FROM ClinicalActivity WHERE CAID = 123) THEN
SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'CAID Already exists,
    duplicates are prohibited';
END IF;
INSERT INTO ClinicalActivity(CAID,IID,STAFF_ID,DEP_ID,Date,Time)
VALUES(123,'CN095430',53,15,"certain date","certain time")
Insert INTO Appointment(CAID,Reason,Status) VALUES(123,"3dam fi
   l2alam",'Scheduled')
COMMIT;
```

2. Atomic update of stock and expense

(a) Pseudocode:

```
BEGIN TRANSACTION
SET error_handler TO:
ON ANY SQL ERROR:
ROLLBACK TRANSACTION
EXIT (log or notify application)
FOR each medication M in Includes WHERE M.PRES_ID =
   prescription_id:
  FIND corresponding stock S in Stock WHERE S.MED_ID = M.MED_ID
     AND S.HOSP_ID = M.HOSP_ID
  S.Qty = S.Qty - M.Quantity
END FOR
'''The necessary triggers would start here'''
IF EXISTS any Stock S such that:
  S.Qty < 0 AND S.MED_ID is in (medications of the prescription)
THEN
  RAISE ERROR "Insufficient stock for prescription"
END IF
COMMIT TRANSACTION
```

(b) ACID properties:

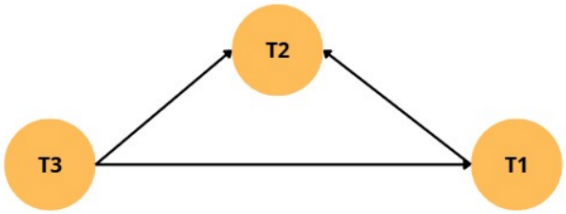
- Atomicity : All stock updates and expense recalculations must succeed together or fail together. Partial updates would cause inventory/billing inconsistencies.
- Consistency : Business rules are maintained.
- Isolation : Other transactions do not see intermediate states where stock is updated but expenses aren't yet.
- Durability : Once committed, the changes made persist.

Part 3: Identifying Types of Schedules

S1 : R1(A),R2(B),W1(A),W2(B)
S2 : R1(A),W1(A),R2(B),W2(B)
Result equivalence: Starting from the same database state, these 2 schedules produce the same final state after execution and there are no conflict between them and as S1 is serializable it is equivalent to S2 = R1(A),W1(A),R2(B),W2(B)

Part 4: Conflict Serializability

1) Dependency graph for S3: Here S3 is conflict Serializable because it's dependency graph is acyclical



Part 5: 2PL

Schedule 1:
T1 : R(A),W(A),R(B),W(B)
T2 : R(A),W(A)
S : R1(A),W1(A),R1(B),W1(B),R2(A),W2(A)
The schedule is compatible with S2PL because: R1(A) T1 gets an S lock on A which is then promoted to an X lock on A with W1(A) then T1 gets an S-lock on B with R1(B) then promoted to S-lock on B with W1(B) Then T2 asks for an S lock on A with R2(A) which is granted because of the X lock of T1 on A has been released as T1 was commited so all the locks are granted and there is no conflict between T1 and T2.

Schedule 2:
T1 : R(A),W(A),R(B)
T2 : R(B),W(B)
S : R1(A),W1(A),R2(B),W2(B),R1(B)
Here the schedule is not compatible with S2PL because: T1 gets an S lock on A then it becomes an X lock on A, after that T2 gets an S lock on B which becomes an X lock on B then T1 requests for an S lock on B which can't be granted because of T2's X lock on B so T1 waits for the release of the lock which can't happen because all the locks are released after the commit of T1 which can't happen as T1 is supposed to end after T2.

Schedule 3:
T1 : R(A),W(A),R(B),W(B)
T2 : R(C),W(C)
S : R1(A),W1(A),R1(B),W1(B),R2(C),W2(C)
The schedule here is compatible with S2PL because: T1 requests for an S lock then X lock on A and S lock then X lock on B then T2 requests for an S lock then X lock an C, there is no conflict between these Lock requests so all of them are granted, the transactions are then commited and all the locks are released at the end of each transaction

Schedule 4:
T1 : R(A),W(A),R(B)
T2 : R(A),W(A),R(B),W(B)
S : R1(A),W1(A),R2(A),W2(A),R1(B),R2(B),W2(B)
The schedule is not compatible with S2PL because: T1 gets an S-lock on A with R1(A) then it is promoted to an X-lock with W1(A) then T2 requests for an S-lock on A with R2(A) it is not granted so T2 waits on the X lock to be released which doesn't happen because for the X-lock to be released which doesn't happen because T1 can't be commited before W2(A) happens.

Part 6: Deadlocks in MNHS

Given schedule: S: R_1(A), R_2(B), W_1(B), W_2(A) Timeline analysis:

- R_1(A): T_1 acquires S-lock on A
- R_2(B): T_2 acquires S-lock on B
- W_1(B): T_1 needs X-lock on B, but T_2 holds S-lock on B → T_1 waits for T_2
- W_2(A): T_2 needs X-lock on A, but T_1 holds S-lock on A → T_2 waits for T_1

Wait-for Graph:
Deadlock analysis: Yes, there is a deadlock. Explanation:

- T_1 is waiting on T_2: T_1 needs to acquire X-lock on B, but T_2 holds S-lock on B
- T_2 is waiting on T_1: T_2 needs to acquire X-lock on A, but T_1 holds S-lock on A
- There is a cycle in wait-for graph: T_1 → T_2 → T_1 therefore a deadlock

DBMS deadlock resolution: The DBMS should :

1. Detect the deadlock using the wait-for graph or timeout mechanism
2. Select a transaction to abort ; depending on the work done or priority...
3. Rollback the victim transaction completely, releasing all its locks
4. Allow the other transaction to proceed

5 Discussion

Challenges: balancing read performance and write overhead for indexes; partitioning requires Date to be part of keys and can complicate key design; hash partitioning can create skew. Observations: indexes on Status and Date give large speedups for the frequent query. Partitioning helps queries that filter by date and helps archival.

6 Conclusion

We proposed indexes for the three views and for the frequent aggregation query. We suggested range partitioning by year for ClinicalActivity and hash partitioning on HID for Stock. We demonstrated simple transaction code to ensure atomicity and discussed concurrency schedules and deadlock detection.