Deliverable 5: Views, Triggers and Application Development

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 Task 1: Views and Triggers

This section details the implementation of SQL Views and Triggers for the MNHS database.

1.1 Views

1.1.1 UpcomingByHospital

Description: Returns scheduled appointment counts per hospital for the next 14 days.

```sql
CREATE or replace VIEW UpcomingByHospital AS
SELECT
    h.Name AS HospitalName,
    ca.Date AS ApptDate,
    COUNT(*) AS ScheduledCount
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 CURDATE() AND DATE_ADD(CURDATE(), INTERVAL
    14 DAY)
GROUP BY h.HID, h.Name, ca.Date;
```

Listing 1: UpcomingByHospital View

1.1.2 Drug Pricing Summary

Description: Summarizes average, minimum, and maximum medication prices per hospital.

```sql
CREATE VIEW DrugPricingSummary AS
SELECT
    H.HID,
    H.Name AS HospitalName,
    M.DrugID,
    M.Name AS MedicationName,
    AVG(S.Unit_Price) AS AvgUnitPrice,
    MIN(S.Unit_Price) AS MinUnitPrice,
    MAX(S.Unit_Price) AS MaxUnitPrice,
    MAX(S.StockTimestamp) AS LastStockTimestamp
FROM Stock S
JOIN Hospital H ON S.HID = H.HID
JOIN Medication M ON S.DrugID = M.DrugID
GROUP BY H.HID, H.Name, M.DrugID, M.Name;
```

Listing 2: DrugPricingSummary View

1.1.3 StaffWorkloadThirty

Description: Shows staff appointment counts (total and by status) for the last 30 days.

```sql
CREATE VIEW StaffWorkloadThirty AS
SELECT
    Staff.STAFF_ID,
    Staff.FullName,
    COUNT(Appointment.CAID) AS TotalAppointments,
    SUM(CASE WHEN Appointment.Status = 'Scheduled' THEN 1 ELSE 0
       END) AS ScheduledCount,
    SUM(CASE WHEN Appointment.Status = 'Completed' THEN 1 ELSE 0
       END) AS CompletedCount,
    SUM(CASE WHEN Appointment.Status = 'Cancelled' THEN 1 ELSE 0
       END) AS CancelledCount
FROM Staff
LEFT JOIN ClinicalActivity ON Staff.STAFF_ID = ClinicalActivity.
   STAFF_ID
    AND DATEDIFF(NOW(), ClinicalActivity.Date) < 30
LEFT JOIN Appointment ON ClinicalActivity.CAID = Appointment.CAID
GROUP BY Staff.STAFF_ID, Staff.FullName;
```

Listing 3: StaffWorkloadThirty View

1.1.4 Patient Next Visit

Description: Shows the next scheduled appointment details for each patient.

```sql
CREATE VIEW PatientNextVisit AS
SELECT
    p.IID,
    p.FullName,
    ca.Date AS NextApptDate,
    d.Name AS DepartmentName,
    h.Name AS HospitalName,
    h.City
FROM Patient p
JOIN ClinicalActivity AS ca ON ca.IID = p.IID
JOIN Appointment a ON ca.CAID = a.CAID
JOIN Department d ON ca.DEP_ID = d.DEP_ID
JOIN Hospital h ON d.HID = h.HID
WHERE a.Status = 'Scheduled'
AND ca.Date > CURDATE()
AND ca.Date = (
    SELECT MIN(ca2.Date)
    FROM ClinicalActivity AS ca2
    JOIN Appointment a2 ON ca2.CAID = a2.CAID
    WHERE ca2.IID = p.IID
    AND a2.Status = 'Scheduled'
    AND ca2.Date > CURDATE()
);
```

Listing 4: PatientNextVisit View

1.2 Triggers

1.2.1 Reject Double Booking

Description: Blocks concurrent appointments for the same staff member.

```
DELIMITER //

CREATE TRIGGER prevent_double_booking_insert
BEFORE INSERT ON Appointment
FOR EACH ROW
BEGIN
    DECLARE v_staff INT;
    DECLARE v_date DATE;
    DECLARE v_time TIME;

    SELECT STAFF_ID, Date, Time INTO v_staff, v_date, v_time
    FROM ClinicalActivity
    WHERE CAID = NEW.CAID;

    IF EXISTS (
        SELECT 1
        FROM ClinicalActivity
        WHERE STAFF_ID = v_staff
        AND Date = v_date
        AND Time = v_time
        AND CAID <> NEW.CAID
    ) THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Double booking detected: Staff is
                already scheduled at this time.';
    END IF;
END//

CREATE TRIGGER prevent_double_booking_update
BEFORE UPDATE ON Appointment
FOR EACH ROW
BEGIN
    DECLARE v_staff INT;
    DECLARE v_date DATE;
    DECLARE v_time TIME;

    SELECT STAFF_ID, Date, Time INTO v_staff, v_date, v_time
    FROM ClinicalActivity
    WHERE CAID = NEW.CAID;

    IF EXISTS (
        SELECT 1
        FROM ClinicalActivity
        WHERE STAFF_ID = v_staff
        AND Date = v_date
        AND Time = v_time
        AND CAID <> NEW.CAID
```

```
    ) THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Double booking detected: Staff is
                already scheduled at this time.';
    END IF;
END//

DELIMITER ;
```

Listing 5: Double Booking Prevention Triggers

1.2.2 Recompute Expense Total

Description: Updates expense totals upon prescription changes; blocks if prices are missing.

```
DELIMITER //

CREATE PROCEDURE RecomputeExpenseTotal(IN p_prescription_id INT)
BEGIN
    DECLARE hosp_id INT;
    DECLARE new_total DECIMAL(10,2);
    DECLARE missing_price INT DEFAULT 0;

    SELECT h.HID INTO hosp_id
    FROM Prescription p
    JOIN ClinicalActivity ca ON p.CAID = ca.CAID
    JOIN Department d ON ca.DEP_ID = d.DEP_ID
    JOIN Hospital h ON d.HID = h.HID
    WHERE p.PID = p_prescription_id;

    SELECT COUNT(*) INTO missing_price
    FROM Include i
    LEFT JOIN Stock s ON i.DrugID = s.DrugID AND s.HID = hosp_id
    WHERE i.PID = p_prescription_id AND s.Unit_Price IS NULL;

    IF missing_price > 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cannot compute expense: missing unit
           price for one or more medications';
    ELSE
        SELECT COALESCE(SUM(s.Unit_Price), 0) INTO new_total
        FROM Include i
        JOIN Stock s ON i.DrugID = s.DrugID AND s.HID = hosp_id
        WHERE i.PID = p_prescription_id;

        UPDATE Expense e
        JOIN Prescription p ON e.CAID = p.CAID
        SET e.Total = new_total
        WHERE p.PID = p_prescription_id;
    END IF;
END//
```

```sql

CREATE TRIGGER RecomputeExpenseAfterInsert
AFTER INSERT ON Include
FOR EACH ROW
BEGIN
    CALL RecomputeExpenseTotal(NEW.PID);
END//

CREATE TRIGGER RecomputeExpenseAfterUpdate
AFTER UPDATE ON Include
FOR EACH ROW
BEGIN
    CALL RecomputeExpenseTotal(NEW.PID);
END//

CREATE TRIGGER RecomputeExpenseAfterDelete
AFTER DELETE ON Include
FOR EACH ROW
BEGIN
    CALL RecomputeExpenseTotal(OLD.PID);
END//

DELIMITER ;
```

Listing 6: Expense Recalculation Triggers

1.2.3 Prevent Negative or Inconsistent Stock

Description: Enforces positive prices and prevents negative stock quantities.

```sql
DELIMITER //

CREATE TRIGGER PreventInvalidStock
BEFORE INSERT ON Stock
FOR EACH ROW
BEGIN
    IF NEW.Qty < 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'ERROR: Quantity cannot be negative.';
    END IF;

    IF NEW.Unit_Price <= 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'ERROR: Unit price must be positive.';
    END IF;

    IF NEW.ReorderLevel < 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'ERROR: Reorder level cannot be
            negative.';
    END IF;
END//
```

```sql

CREATE TRIGGER PreventInvalidStockUpdate
BEFORE UPDATE ON Stock
FOR EACH ROW
BEGIN
    IF NEW.Qty < 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'ERROR: Quantity cannot be negative.';
    END IF;

    IF NEW.Unit_Price <= 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'ERROR: Unit price must be positive.';
    END IF;

    IF NEW.ReorderLevel < 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'ERROR: Reorder level cannot be
           negative.';
    END IF;

    IF NEW.Qty < OLD.Qty AND NEW.Qty < 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'ERROR: Cannot decrease quantity below
           zero.';
    END IF;
END//

DELIMITER ;
```

Listing 7: Stock Validation Triggers

1.2.4 Protect Referential Integrity on Patient Delete

Description: Prevents deleting patients who have existing clinical activities.

```sql
DELIMITER //

CREATE TRIGGER PreventPatientDelete
BEFORE DELETE ON Patient
FOR EACH ROW
BEGIN
    IF EXISTS (
        SELECT IID FROM ClinicalActivity ca
        WHERE ca.IID = OLD.IID
    ) THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cannot delete patient. Please
           reassign or delete dependent clinical activities first
           .';
    END IF;
END//
```

```
DELIMITER ;
```

Listing 8: Patient Delete Protection Trigger

2 Task 2 : Testing (Views and Triggers in action)

2.1 SQL Views

2.1.1 Sample Query 1: Upcoming By Hospital

Objective: Shows scheduled appointments per hospital for the next 14 days **Query:**

```sql
SELECT * FROM UpcomingByHospital
WHERE HospitalName = 'CHU Ibn Rochd'
ORDER BY ApptDate;
```

Listing 9: UpcomingByHospital View

#	HospitalName	ApptDate	ScheduledCoun
1	CHU Ibn Rochd	2025-11-30	1
2	CHU Ibn Rochd	2025-12-03	1
3	CHU Ibn Rochd	2025-12-05	1
4	CHU Ibn Rochd	2025-12-14	1

Figure 1: Output Table for example Query

2.1.2 Sample Query 2: Drug Pricing Summary

Objective: Summarizes medication pricing statistics per hospital **Query:**

```sql
SELECT * FROM DrugPricingSummary
WHERE MedicationName IN ('Doliprane', 'Glucophage', 'Triatec')
ORDER BY MedicationName, AvgUnitPrice;
```

Listing 10: DrugPricingSummary View

#	HID	HospitalName	DrugID	MedicationName	AvgUnitPrice	MinUnitPrice	MaxUnitPrice	LastStockTimestam
1	1	CHU Ibn Rochd	100	Doliprane	15.000000	15.00	15.00	2025-11-27 00:38:05
2	3	CHU Mohammed VI	100	Doliprane	15.000000	15.00	15.00	2025-11-27 00:38:05
3	5	Hôpital Mohammed V	100	Doliprane	16.000000	16.00	16.00	2024-11-28 10:00:00
4	1	CHU Ibn Rochd	105	Glucophage	35.000000	35.00	35.00	2024-11-27 20:00:00
5	2	Hôpital Avicenne	105	Glucophage	41.000000	40.00	42.00	2025-11-27 00:38:05
6	1	CHU Ibn Rochd	106	Triatec	22.000000	22.00	22.00	2024-11-27 22:00:00
7	2	Hôpital Avicenne	106	Triatec	25.000000	25.00	25.00	2024-11-28 09:00:00

Figure 2: Output Table for example Query

2.1.3 Sample Query 3: Staff Workload Thirty

Objective: Shows staff appointment workload over the last 30 days **Query:**

```sql
SELECT * FROM StaffWorkloadThirty
WHERE TotalAppointments > 0
ORDER BY CompletedCount;
```

Listing 11: StaffWorkloadThirty View

#	STAFF_ID	FullName	TotalAppointment	ScheduledCoun	CompletedCoun	CancelledCoun
1	7	Dr. Karim Chraibi	2	2	0	0
2	12	Dr. Asmaa Chaoui	1	1	0	0
3	19	Dr. Samira Alami	2	2	0	0
4	20	Dr. Rachid Naciri	1	1	0	0
5	2	Dr. Leila Berrada	6	5	1	0
6	18	Dr. Redouane Aouad	1	0	1	0
7	26	Dr. Hicham Amrani	2	1	1	0
8	10	Dr. Houda Tazi	4	2	2	0
9	11	Dr. Nabil Daoudi	5	3	2	0

Figure 3: Output Table for example Query

2.1.4 Sample Query 4: Patient Next Visit

Objective: Shows the next scheduled visit for each patient **Query:**

```sql
SELECT * FROM PatientNextVisit
WHERE NextApptDate BETWEEN CURDATE() AND DATE_ADD(CURDATE(),
    INTERVAL 7 DAY)
ORDER BY NextApptDate;
```

Listing 12: PatientNextVisit View

#	IID	FullName	NextApptDat	DepartmentNam	HospitalName	City
1	102	Yassir Tazi	2025-12-02	Pédiatrie	Hôpital Avicenne	Rabat
2	103	Haja Khadija	2025-12-03	Cardiologie	CHU Ibn Rochd	Casablanca
3	104	Karim Bouanani	2025-12-05	Orthopédie	CHU Ibn Rochd	Casablanca
4	105	Salma Bennani	2025-12-07	Pédiatrie	Hôpital Avicenne	Rabat

Figure 4: Output Table for example Query

2.2 Triggers

2.2.1 Test Case 1: Double Booking Prevention

Scenario: We will try to create a conflicting appointment for Dr. Leila Berrada (STAFF_ID = 2). Note : She already has an appointment on 2025-11-28 at 11:00:00 (CAID 1003) **Query:**

```sql
INSERT INTO ClinicalActivity (CAID, IID, STAFF_ID, DEP_ID, Date,
    Time) VALUES
(6001, 101, 2, 11, '2025-11-28', '11:00:00');
INSERT INTO Appointment (CAID, Reason, Status) VALUES (6001, '
    Conflict test same time', 'Scheduled');
```

Listing 13: Insert Statments

```
Error Code: 1644. Double booking detected: Staff is already
    scheduled at this time.
```

Listing 14: Error Message

2.2.2 Test Case 2: Expense Recalculation

Scenario: We will add a new medication to prescription 1 - this will trigger expense recalculation The trigger is blocking this insert because:

1. Prescription 1 (PID=1) is linked to ClinicalActivity CAID=1000
2. CAID=1000 belongs to Department 11 (Cardiologie) in Hospital 1 (CHU Ibn Rochd)
3. The trigger checks if Mopral (DrugID=108) has a Unit_Price at Hospital 1
4. Currently, Mopral only has stock at Hospital 3 (CHU Mohammed VI) with price 80.00
5. There is NO stock record for Mopral at Hospital 1, so Unit_Price is NULL
6. The trigger detects missing price and blocks with error: "Cannot compute expense: missing unit price"

Query:

```sql
INSERT INTO Include (PID, DrugID, Dosage, Duration) VALUES
(1, 108, '1 capsule daily', '30 days');
```

Listing 15: Insert Statments

```
Error Code: 1644. Cannot compute expense: missing unit price for
    one or more medications
```

Listing 16: Error Message

2.2.3 Test Case 3: Stock Validation

Scenario: We will try to insert negative quantity, negative price and lastly negative reorder level.) **Query:**

```sql
INSERT INTO Stock (HID, DrugID, Unit_Price, Qty, ReorderLevel,
    StockTimestamp) VALUES
(1, 110, 25.00, -10, 5, NOW());
INSERT INTO Stock (HID, DrugID, Unit_Price, Qty, ReorderLevel,
    StockTimestamp) VALUES
```

```
(1, 110, -5.00, 50, 10, NOW());
INSERT INTO Stock (HID, DrugID, Unit_Price, Qty, ReorderLevel,
    StockTimestamp) VALUES
(1, 110, 25.00, 50, -5, NOW());
```

Listing 17: Insert Statments

```
Error Code: 1644. ERROR: Quantity cannot be negative.
```

Listing 18: Error Message 1

```
Error Code: 1644. ERROR: Unit price must be positive.
```

Listing 19: Error Message 2

```
Error Code: 1644. ERROR: Reorder level cannot be negative.
```

Listing 20: Error Message 3

2.2.4 Test Case 4: Patient Delete Protection

Scenario: We will try to delete a patient with an existing clinical activity **Query:**

```
DELETE FROM Patient WHERE IID = 100;
```

Listing 21: Delete Statment

```
Error Code: 1644. Cannot delete patient. Please reassign or
    delete dependent clinical activities first.
```

Listing 22: Error Message

3 Task 3: Application Layer for MNHS Database

3.1 Summary

The primary objective of this project is to develop a secure and user-friendly ”Application Layer” for the MNHS database.The project prioritizes architectural robustness through the implementation of security best practices, including parameterized queries and environment-based credential management.

3.2 Technical Stack

3.2.1 The Frontend: Streamlit

The application interface is built using **Streamlit**, a Python-based framework designed for the rapid development of data-driven web applications. Streamlit abstracts the frontend complexity (HTML/CSS/JavaScript), allowing for a direct integration of the Python application logic with the user interface.

In this implementation, Streamlit serves three critical functions:

- **Navigation Control:** `st.sidebar` is utilized to create a modular navigation menu, separating distinct business functions (e.g., Patient Directory, Scheduling, Analytics) into isolated views.

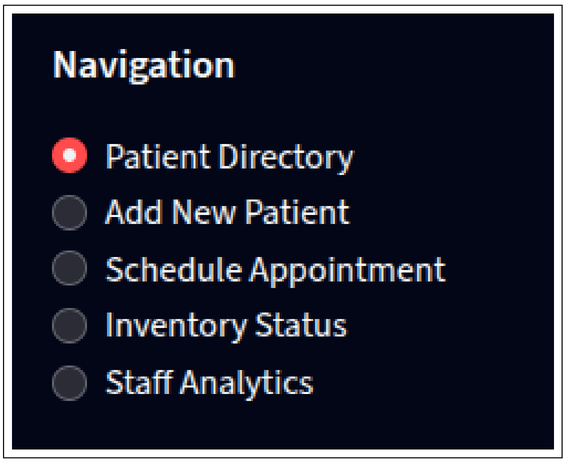


Figure 5: Navigation Menu

```python
        st.sidebar.markdown("### Navigation")
        menu_options = [
        "Patient Directory",
        "Add New Patient",
        "Schedule Appointment",
        "Inventory Status",
        "Staff Analytics"
        ]
        choice = st.sidebar.radio("Go to", menu_options, label_visibility="collapsed")
```

Listing 23: Python code for sidebar/navigation

- **Input Handling:** `st.form` and `st.form_submit_button` are employed to batch user inputs. This prevents premature database queries by ensuring that transactional data (such as appointment details) is only sent to the backend upon explicit user confirmation.

```python
    with st.form("appt_form"):
            st.markdown("#### Appointment Details")
            c1, c2, c3 = st.columns(3)
            with c1:
```

```python
                st.text_input("New Activity ID (Auto)", value=next_caid, disabled=True)
            with c2:
                date = st.date_input("Date")
            with c3:
                time = st.time_input("Time")

            reason = st.text_area("Reason for visit", height=100)
            submitted = st.form_submit_button("Confirm Schedule", type="primary", use_container_width=True)
```

Listing 24: Python code for the form in the schedule appointment section

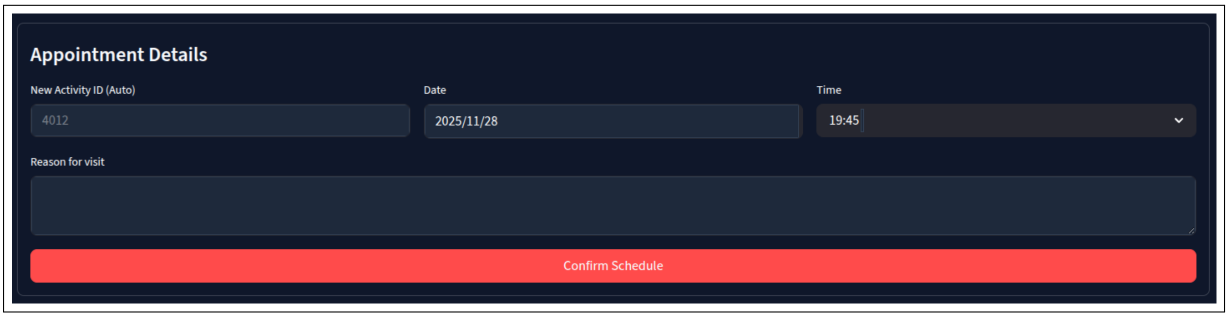
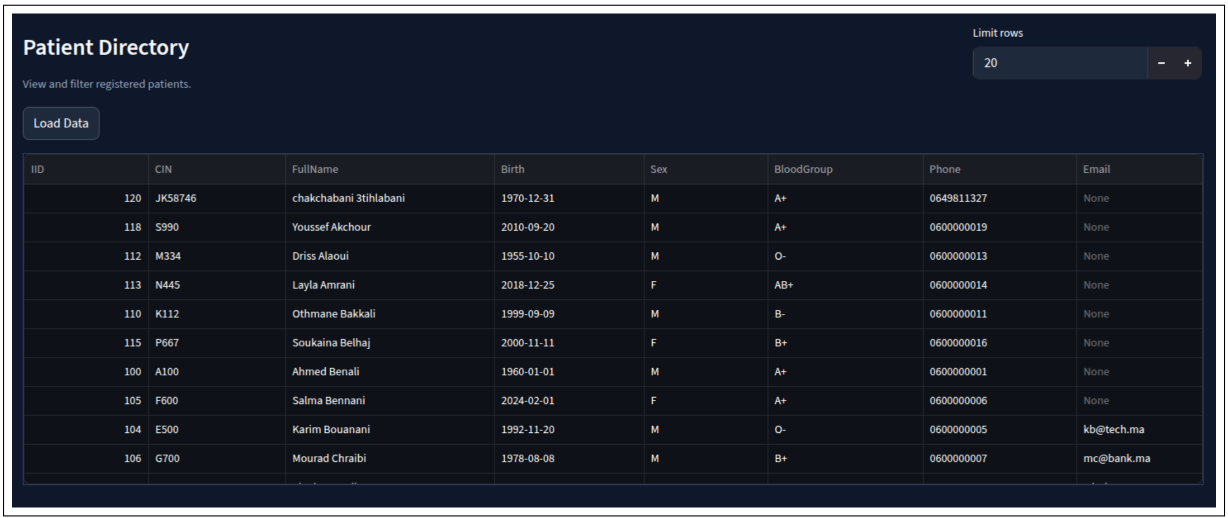


Figure 6: Appointment Form

- **Data Visualization:** The framework native integration with **Pandas** allows for the direct rendering of SQL query results into interactive tables (`st.dataframe`) and charts.

```python
    if st.button("Load Data"):
            with st.spinner("Fetching..."):
                results = list_patients_ordered_by_last_name(limit)
                if results:
                    df = pd.DataFrame(results)
                    st.dataframe(df, use_container_width=True, hide_index=True)
                else:
                    st.info("No records found.")
```

Listing 25: Python code for the rendering of the patients table using st.dataframe



Patient Directory

View and filter registered patients.

Load Data

Limit rows

20

IID	CIN	FullName	Birth	Sex	BloodGroup	Phone	Email
120	JK58746	chakchabani 3tihlabani	1970-12-31	M	A+	0649811327	None
118	S990	Youssef Akchour	2010-09-20	M	A+	0600000019	None
112	M334	Driss Alaoui	1955-10-10	M	O-	0600000013	None
113	N445	Layla Amrani	2018-12-25	F	AB+	0600000014	None
110	K112	Othmane Bakkali	1999-09-09	M	B-	0600000011	None
115	P667	Soukaina Belhaj	2000-11-11	F	B+	0600000016	None
100	A100	Ahmed Benali	1960-01-01	M	A+	0600000001	None
105	F600	Salma Bennani	2024-02-01	F	A+	0600000006	None
104	E500	Karim Bouanani	1992-11-20	M	O-	0600000005	kb@tech.ma
106	G700	Mourad Chraibi	1978-08-08	M	B+	0600000007	mc@bank.ma

Figure 7: Patients Table

3.2.2 The Backend: TiDB

The application backend relies on **TiDB**, an open-source distributed SQL database.

Implementation Logic:

- **Protocol:** The application interacts with TiDB exactly like a standard single-node MySQL instance. The `mysql-connector-python` library is used to manage the connection, requiring no specialized TiDB-specific client code.
- **Secure Connection:** To maintain security best practices, the connection logic is encapsulated in a helper function. This function reads sensitive credentials (Host, Port, User, Password) dynamically from environment variables (`.env`) rather than hard-coding them, ensuring the distributed cluster is accessed securely.

```python
# Local .env configuration
def get_connection():
    """
    Establishes a database connection with a hybrid approach:
    1. Tries to load from Streamlit Secrets (st.secrets) for
       Cloud deployment.
    2. Falls back to local .env file using python-dotenv for
       local dev.
    """
    try:
        # Attempt to access Streamlit Secrets
        # This will work if .streamlit/secrets.toml exists or on
           Streamlit Cloud
        return mysql.connector.connect(
            host=st.secrets["mysql"]["host"],
            port=st.secrets["mysql"]["port"],
            database=st.secrets["mysql"]["database"],
            user=st.secrets["mysql"]["user"],
            password=st.secrets["mysql"]["password"]
        )
```

```python
    except (FileNotFoundError, KeyError):
        # Fallback: Load environment variables from .env file
        load_dotenv()

        # Ensure port is an integer
        port_val = os.getenv("MYSQL_PORT", 3306)

        return mysql.connector.connect(
            host=os.getenv("MYSQL_HOST"),
            port=int(port_val),
            database=os.getenv("MYSQL_DB"),
            user=os.getenv("MYSQL_USER"),
            password=os.getenv("MYSQL_PASSWORD")
        )
```

Listing 26: TiDB Connection Implementation

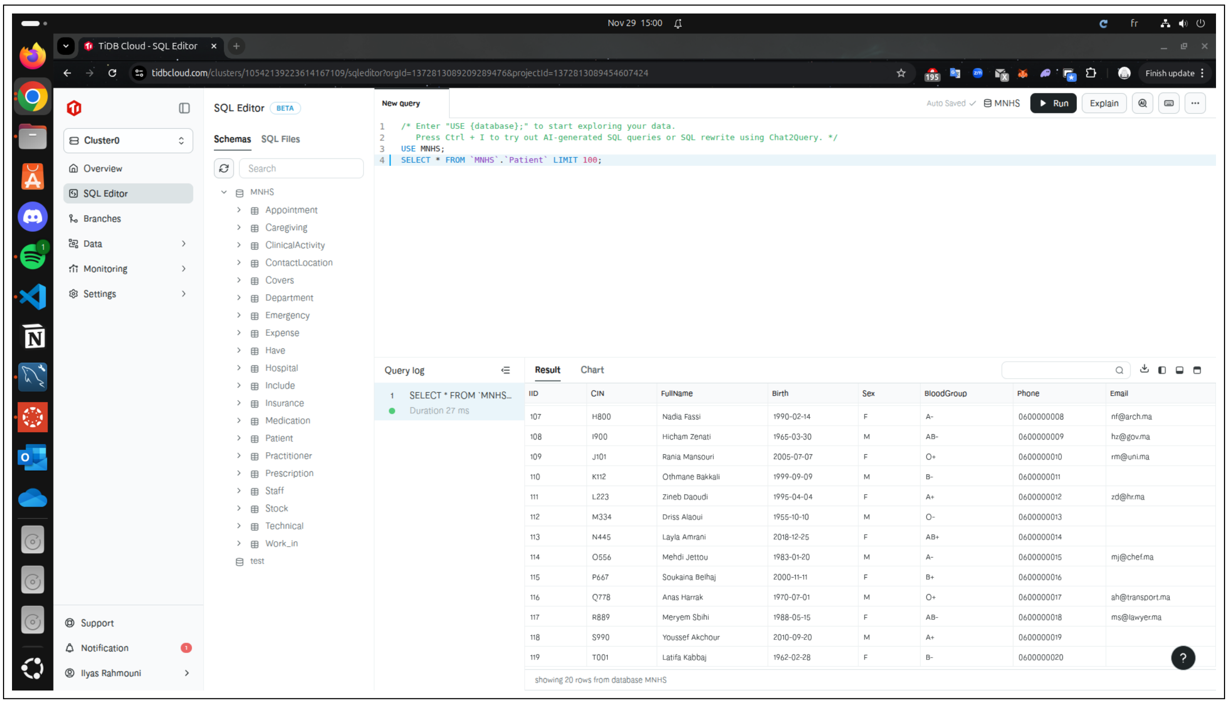


Figure 8: TiDB SQL Editor

3.3 Implementation Code Logic

3.3.1 Module 1: Patient Directory (list_patients)

The Patient Directory module provides the user with a searchable, ordered view of registered patients. The lab specifically asks for the first 20 patients, this is set as the default value for the limit rows option. But we took the liberty to add the possibility of choosing the number of rows to show as an option.

3.3.1.1 UI Implementation: Server-Side Pagination

To ensure optimal performance, the application avoids fetching the entire dataset at once. Instead, it implements a "Server-Side Pagination" strategy controlled by the user.

- **Dynamic Limit Control:** A Streamlit number input (`st.number_input`) allows the user to specify exactly how many records to retrieve (e.g., 10, 20, 50).

```python
    limit = st.number_input("Limit rows", 5, 100, 20)
```

Listing 27: Python code for the limit choice

Figure 9: Dynamic Limit Control

- **Performance Impact:** This input is passed directly to the SQL `LIMIT` clause. This ensures that the heavy lifting is done by the database engine (TiDB), and only the requested subset of data is transmitted over the network to the Python frontend.

```python
    if st.button("Load Data"):
            with st.spinner("Fetching..."):
                results = list_patients_ordered_by_last_name(
                    limit)
```

Listing 28: Python code calling the function list patients ordered by last name at the press of Load Data button with the limit as an argument to the function

3.3.1.2 SQL Logic: String Manipulation for Sorting

A task to implement for this application layer was to order patients by their **Last Name**. However, the database schema provided stores the name as a single string in the `FullName` column (e.g., "Ahmed Benali").

Standard SQL sorting (`ORDER BY FullName`) would incorrectly sort by the first name. To resolve this without altering the schema, the application utilizes the `SUBSTRING_INDEX` function.

The Sorting Algorithm:

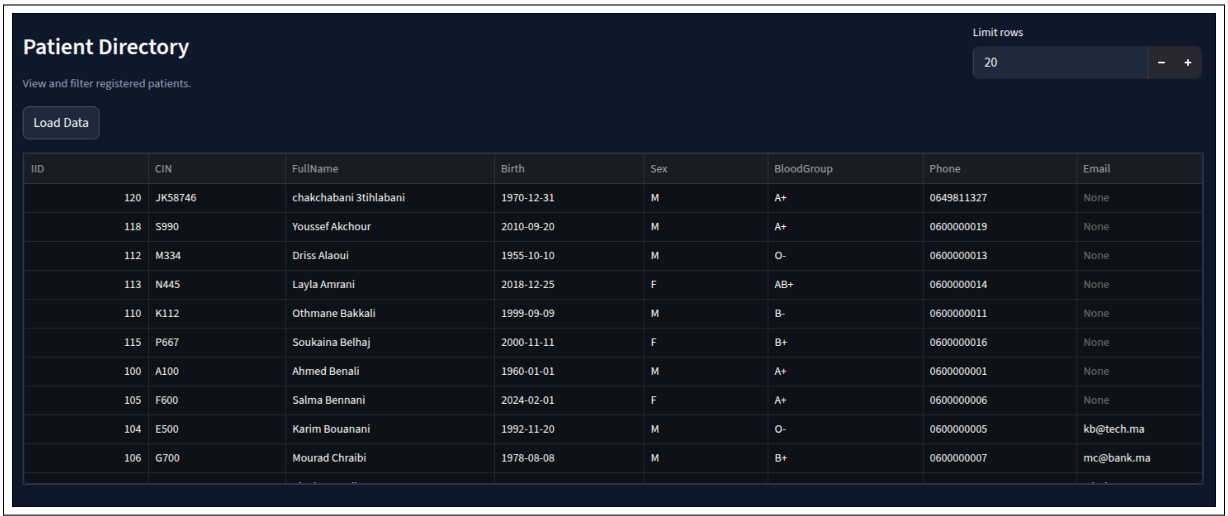
1. The query isolates the string segment occurring after the last space character using `SUBSTRING_INDEX(FullName, ' ', -1)`.

2. This extracted token (the surname) is used as the primary sort key.

3. The full name is used as a secondary sort key to resolve ties.

```python
def list_patients_ordered_by_last_name(limit=20):
    sql = """
    SELECT IID, CIN, FullName, Birth, Sex, BloodGroup, Phone,
       Email
    FROM Patient
    ORDER BY SUBSTRING_INDEX(FullName, ' ', -1)
    LIMIT %s
    """
    try:
        with get_connection() as cnx:
            with cnx.cursor(dictionary=True) as cur:
                cur.execute(sql, (limit,))
                return cur.fetchall()
    except Exception as e:
        st.error(f"Error fetching patients: {e}")
        return []
```

Listing 29: Python code including SQL Query for Surname Sorting



IID	CIN	FullName	Birth	Sex	BloodGroup	Phone	Email
120	JK58746	chakchabani 3tihlabani	1970-12-31	M	A+	0649811327	None
118	S990	Youssef Akchour	2010-09-20	M	A+	0600000019	None
112	M334	Driss Alaoui	1955-10-10	M	O-	0600000013	None
113	N445	Layla Amrani	2018-12-25	F	AB+	0600000014	None
110	K112	Othmane Bakkali	1999-09-09	M	B-	0600000011	None
115	P667	Soukaina Belhaj	2000-11-11	F	B+	0600000016	None
100	A100	Ahmed Benali	1960-01-01	M	A+	0600000001	None
105	F600	Salma Bennani	2024-02-01	F	A+	0600000006	None
104	E500	Karim Bouanani	1992-11-20	M	O-	0600000005	kb@tech.ma
106	G700	Mourad Chraibi	1978-08-08	M	B+	0600000007	mc@bank.ma

Figure 10: Patients Table Output

3.3.2 Module 2: Intelligent Scheduling System (schedule_appt)

The implementation of this module/functionality prioritizes user experience by abstraction the actual complex data base identifiers and automating the CAID choice.

3.3.2.1 User-Friendly Input Abstraction A key usability challenge in database applications is that end-users (medical staff) do not know internal Primary Keys (e.g., that "Dr. Leila" is `STAFF_ID=2` or "Cardiology" is `DEP_ID=11`).

To resolve this, the application implements a **Translation Layer** using Streamlit and Python dictionaries:

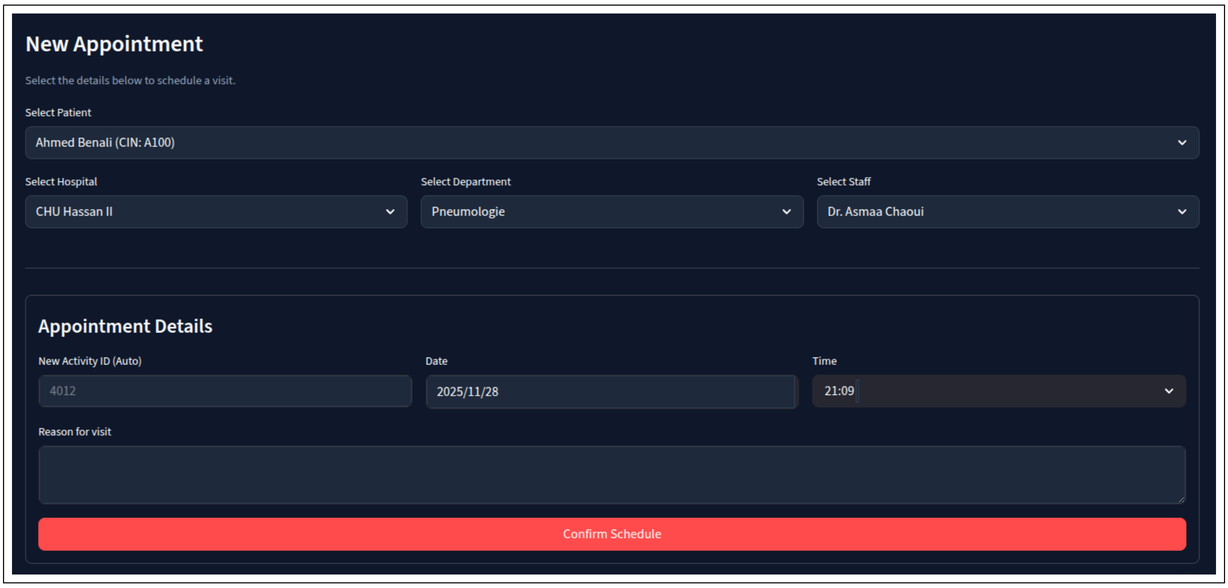


Figure 11: User-Friendly interface

1. **Fetching:** On page load, the application executes `SELECT` queries to retrieve human-readable names (Staff Full Name, Hospital Name, Department Name).

```
    all_patients = get_all_patients()
    all_hospitals = get_all_hospitals()
    all_staff = get_all_staff()
```

Listing 30: Python code for fetching process on page load

2. **Mapping:** These results are stored in a Python dictionary mapping the Name (Key) to the ID (Value) as seen above.

```
    patient_options = {f"{p['FullName']} (CIN: {p['CIN']})":
       p['IID'] for p in all_patients}
    # patient_options is the dictionary in question
    # it contains patient names and cin as keys and iids as
       values
```

Listing 31: Python code for patients dictionnary as an example

3. **Selection:** The user interacts with a `st.selectbox` displaying only the names.

```
selected_patient_label = st.selectbox("Select Patient",
  options=list(patient_options.keys()))
```

Listing 32: Python code for selection

4. **Translation:** Upon submission, the application looks up the corresponding ID from the dictionary to use in the SQL `INSERT` statement.

```
selected_iid = patient_options[selected_patient_label]
```

Listing 33: Python code for iid lookup

3.3.2.2 Automated ID Generation To prevent "Duplicate Key" errors and reduce manual entry, the application automatically calculates the next available Clinical Activity ID (`CAID`).

Logic: Before the transaction begins, the application queries the maximum existing ID in the `ClinicalActivity` table. The new ID is calculated as $MAX(CAID) + 1$.

```
def get_next_caid():
    sql = "SELECT MAX(CAID) as max_id FROM ClinicalActivity"
    try:
        with get_connection() as cnx:
            with cnx.cursor(dictionary=True) as cur:
                cur.execute(sql)
                res = cur.fetchone()
                if res and res['max_id']:
                    return res['max_id'] + 1
                return 1000
    except: return 1000
```

Listing 34: Next CAID Calculation helper function

```
next_caid = get_next_caid()
```

Listing 35: Function call at page load

3.3.2.3 Transactional Integrity The scheduling process involves two distinct SQL write operations:

1. Creating the parent event in the `ClinicalActivity` table.
2. Creating the specific status in the `Appointment` table.

To ensure data consistency, if the second insert fails (e.g., due to a constraint violation), the first insert is automatically reversed (`ROLLBACK`)

```
if submitted:
    if selected_dep_id is None:
        st.error("Invalid Department.")
    else:
        try:
```

```
            schedule_appointment(next_caid, selected_iid,
               selected_staff_id, selected_dep_id, str(date), str
               (time), reason)
                        st.success(f"Appointment scheduled (ID: {
                           next_caid})")
        except Exception as e:
            st.error(f"Failed to schedule: {e}")
```

Listing 36: schedule_appointment function call at the press of Confirme Schedule button

```
def schedule_appointment(caid, iid, staff_id, dep_id, date_str,
   time_str, reason):
    """
    Schedules an appointment with 'Double Booking' protection.
    """
    # 1. Validation SQL: Check if staff is already booked
    check_sql = """
    SELECT CAID
    FROM ClinicalActivity
    WHERE STAFF_ID = %s AND Date = %s AND Time = %s
    LIMIT 1
    """
    ins_ca = """
    INSERT INTO ClinicalActivity (CAID, IID, STAFF_ID, DEP_ID,
       Date, Time)
    VALUES (%s, %s, %s, %s, %s, %s)
    """
    ins_appt = """
    INSERT INTO Appointment (CAID, Reason, Status)
    VALUES (%s, %s, 'Scheduled')
    """
    with get_connection() as cnx:
        try:
            with cnx.cursor() as cur:
                # --- [TRIGGER LOGIC START] ---
                # Check for double booking before doing anything
                cur.execute(check_sql, (staff_id, date_str,
                   time_str))
                conflict = cur.fetchone()
                if conflict:
                    # STOP! Raise an error to prevent the insert
                    raise ValueError(f"Double Booking Error:
                       Staff {staff_id} is already busy at {
                       time_str} on {date_str}.")
                # --- [TRIGGER LOGIC END] ---
                # If no conflict, proceed with the transaction
                cur.execute(ins_ca, (caid, iid, staff_id, dep_id,
                    date_str, time_str))
                cur.execute(ins_appt, (caid, reason))
                cnx.commit()
                return True
        except Exception as e:
```

```
                cnx.rollback()
                raise e
```

Listing 37: schedule_appointment function

3.3.2.4 Note on Architectural Adaptation The final implementation of this scheduling module includes specific validation logic to enforce the "Double Booking" constraint. However, due to the architectural differences between standard MySQL and the distributed TiDB backend, specifically the lack of support for `CREATE TRIGGER`, this validation was strategically shifted from the database layer to the application layer. A detailed technical justification for this design choice is presented in 2.4 Adaptation Strategy

3.3.3 Module 3: Inventory & Low Stock (low_stock)

This module detects only low quantity medication stocks including also medications with no stock at all. Unlike a standard data retrieval, this functionality utilizes various techniques to filter and load only the wanted rows in the specified format.

3.3.3.1 SQL Query Explanation: A standard SQL query on the `Stock` table can easily find items where `Qty < ReorderLevel`. However, if a hospital has completely run out of a medication, or never logged it, there may be **no row at all** in the `Stock` table for that specific Hospital-Drug combination.

To solve this, the application implements a **Cross-Join Strategy**:

1. **Cartesian Product:** A `CROSS JOIN` is performed between the `Hospital` table and the `Medication` table. This generates a theoretical matrix of every possible medication in every hospital.

```sql
    FROM Medication M
    CROSS JOIN Hospital H
```

Listing 38: Cross-Join Logic for Missing Stock

2. **Left Join:** The application then `LEFT JOIN`s the actual `Stock` table against this matrix.

```sql
    LEFT JOIN Stock S ON M.DrugID = S.DrugID AND H.HID = S.
       HID
```

Listing 39: Left Join the actual Stock Table

3. **Null Handling:** If no stock record matches, the database returns `NULL`. The query uses `COALESCE(Qty, 0)` to interpret this `NULL` as a quantity of zero, triggering the alert.

```sql
    COALESCE(S.Qty, 0) AS CurrentQuantity,
    COALESCE(S.ReorderLevel, 0) AS ReorderLevel,
```

Listing 40: COALESCE Function

4. **COALESCE Function:** The `COALESCE` function safeguards the query logic by converting any `NULL` values, resulting from missing inventory records, into a computational zero, ensuring that complete stockouts are accurately identified and flagged.

```sql
    SELECT
    M.Name AS MedicationName,
    H.Name AS HospitalName,
    COALESCE(S.Qty, 0) AS CurrentQuantity,
    COALESCE(S.ReorderLevel, 0) AS ReorderLevel,
    CASE
        WHEN S.Qty IS NULL THEN 'No Stock'
        WHEN S.Qty < S.ReorderLevel THEN 'Low Stock'
        ELSE 'Adequate'
    END AS StockStatus
```

```sql
    FROM Medication M
    CROSS JOIN Hospital H
    LEFT JOIN Stock S ON M.DrugID = S.DrugID AND H.HID = S.HID
    WHERE S.Qty IS NULL OR S.Qty < S.ReorderLevel
    ORDER BY M.Name, H.Name;
```

Listing 41: Full Sql Query

3.3.3.2 UI Implementation — Visual Alerting : To make this data actionable for administrators, the application avoids displaying a raw wall of text. Instead, it utilizes **Pandas Styling** to apply conditional formatting to the data before rendering it in Streamlit.

Logic: A Python function iterates through the result set:

- **Red Highlight:** Applied if the status is "No Stock" (Qty = 0).
- **Orange Highlight:** Applied if the status is "Low Stock" (0 ¡ Qty ¡ ReorderLevel).

```python
    def style_status(val):
        color = '#ef4444' if val == 'No Stock' else '#f59e0b'
            if val == 'Low Stock' else '#94A3B8'
        return f'color: {color}; font-weight: 500'
```

Listing 42: Python Function used for the styling

3.3.3.3 UI Implementation — Donut Chart : To provide an immediate assessment of the hospital's overall inventory health, the interface includes a summary Donut Chart built using **Plotly Express**.

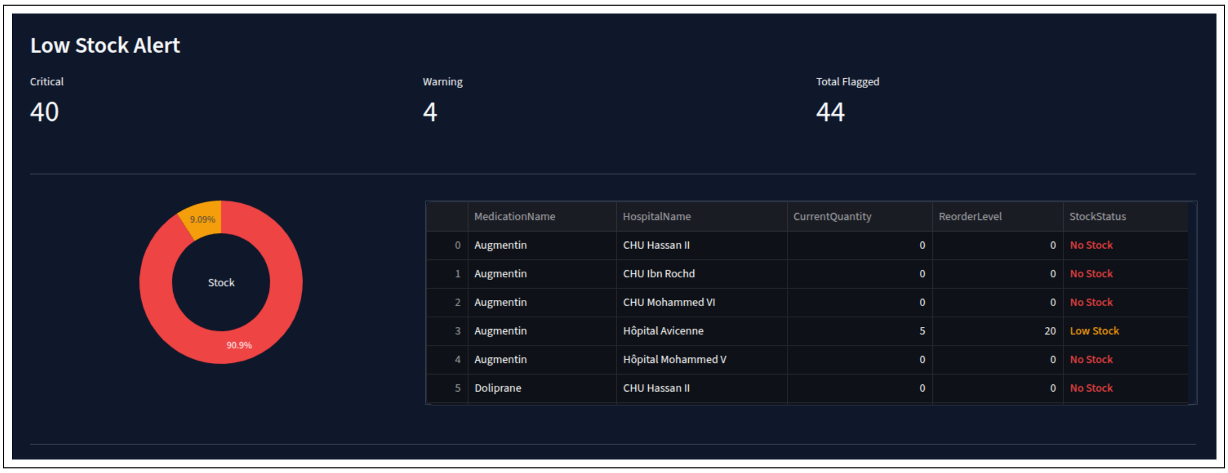
Code Logic:

1. **Aggregation:** The application aggregates the SQL results using `value_counts()` on the 'StockStatus' column to determine the distribution of critical vs. adequate stock.

```python
status_counts = df['StockStatus'].value_counts()
```

Listing 43: value_counts() function on the StockStatus

2. **Visualization:** A pie chart with a center hole (`hole=0.6`) is generated to create a modern "Donut" aesthetic.
3. **Semantic Coloring:** To ensure consistency with the table highlights, the chart utilizes a specific color sequence: Red (`#ef4444`) for shortages, Orange (`#f59e0b`) for low stock, and Grey for adequate items.



	MedicationName	HospitalName	CurrentQuantity	ReorderLevel	StockStatus
0	Augmentin	CHU Hassan II	0	0	No Stock
1	Augmentin	CHU Ibn Rochd	0	0	No Stock
2	Augmentin	CHU Mohammed VI	0	0	No Stock
3	Augmentin	Hôpital Avicenne	5	20	Low Stock
4	Augmentin	Hôpital Mohammed V	0	0	No Stock
5	Doliprane	CHU Hassan II	0	0	No Stock

Figure 12: Low Stock Interface

```python
# Create a Donut chart with semantic colors
fig = px.pie(
    values=status_counts.values,
    names=status_counts.index,
    hole=0.6, # Creates the donut shape
    # Red for Critical, Orange for Warning
    color_discrete_sequence=['#ef4444', '#f59e0b', '#475569'],
    template="plotly_dark"
)
st.plotly_chart(fig, use_container_width=True)
```

Listing 44: Plotly Donut Chart Configuration

3.3.4 Module 4: Staff Workload Analytics (staff_share)

This module provides the user with performance metrics, calculating the share of appointments handled by each staff member relative to their specific hospital's total workload. This requires an SQL aggregation and a visualization strategy capable of handling multi-hospital employment.

3.3.4.1 SQL Logic: Calculating a percentage share requires: the count for the individual staff member and the total count for the hospital. We will use Subqueries to achieve the desired result (Nested Subqueries).

The Query Structure:

1. **Subquery S (Staff):** Counts appointments grouped by `STAFF_ID` and `HID`. This yields the numerator (Individual Workload).

```sql
    SELECT ca.STAFF_ID, st.FullName, dep.HID, COUNT(a.CAID)
       AS StaffApp
    FROM Appointment a
    JOIN ClinicalActivity ca ON a.CAID = ca.CAID
    JOIN Staff st ON ca.STAFF_ID = st.STAFF_ID
    JOIN Department dep ON ca.DEP_ID = dep.DEP_ID
    GROUP BY ca.STAFF_ID, st.FullName, dep.HID
```

Listing 45: Staff Subquery

2. **Subquery H (Hospital):** Counts appointments grouped by `HID` only. This yields the denominator (Hospital Total).

```sql
    SELECT dep.HID, Hosp.Name, COUNT(a.CAID) AS HospitalApp
    FROM Appointment a
    JOIN ClinicalActivity ca ON ca.CAID = a.CAID
    JOIN Department dep ON ca.DEP_ID = dep.DEP_ID
    JOIN Hospital Hosp ON dep.HID = Hosp.HID
    GROUP BY dep.HID, Hosp.Name
```

Listing 46: Hospital Subquery

3. **Main Query:** Joins these two virtual tables on `HID` to calculate the percentage: $\frac{\text{Staff Appts}}{\text{Hospital Appts}} \times 100$.

```sql
    SELECT
    s.STAFF_ID,
    s.FullName AS StaffName,
    h.Name AS HospitalName,
    s.StaffApp AS TotalAppointments,
    ROUND((s.StaffApp / h.HospitalApp) * 100, 2) AS
       PercentageShare
    FROM (
        SELECT ca.STAFF_ID, st.FullName, dep.HID, COUNT(a.CAID)
           AS StaffApp
        FROM Appointment a
        JOIN ClinicalActivity ca ON a.CAID = ca.CAID
```

```sql
        JOIN Staff st ON ca.STAFF_ID = st.STAFF_ID
        JOIN Department dep ON ca.DEP_ID = dep.DEP_ID
        GROUP BY ca.STAFF_ID, st.FullName, dep.HID
    ) s
    JOIN (
        SELECT dep.HID, Hosp.Name, COUNT(a.CAID) AS HospitalApp
        FROM Appointment a
        JOIN ClinicalActivity ca ON ca.CAID = a.CAID
        JOIN Department dep ON ca.DEP_ID = dep.DEP_ID
        JOIN Hospital Hosp ON dep.HID = Hosp.HID
        GROUP BY dep.HID, Hosp.Name
    ) h ON s.HID = h.HID
    ORDER BY h.HID, PercentageShare DESC;
```

Listing 47: Full Query for appointment share

3.3.4.2 UI Implementation — Bar Chart A significant challenge in visualizing this data is that many doctors work across multiple hospital sites. Using a standard bar chart often results in data points for the same doctor overlapping illegibly.

The Solution: To resolve this, the application utilizes the **Plotly Express** library to render a **Horizontal Grouped Bar Chart**. This approach offers superior clarity over standard charts through three key mechanisms:

- **Grouped Mode (barmode='group'):** Instead of stacking values, this setting forces the chart to place bars side-by-side for the same staff member. If a doctor works in "Casablanca" and "Rabat," they appear as two distinct, adjacent bars, preventing any visual overlap.
- **Semantic Coloring:** The `HospitalName` is mapped to a discrete color sequence. This allows users to visually distinguish which hospital contributes to the specific workload share at a glance.
- **Horizontal Orientation:** Given that staff names can be long, setting `orientation='h'` ensures all labels are readable without rotation or truncation.

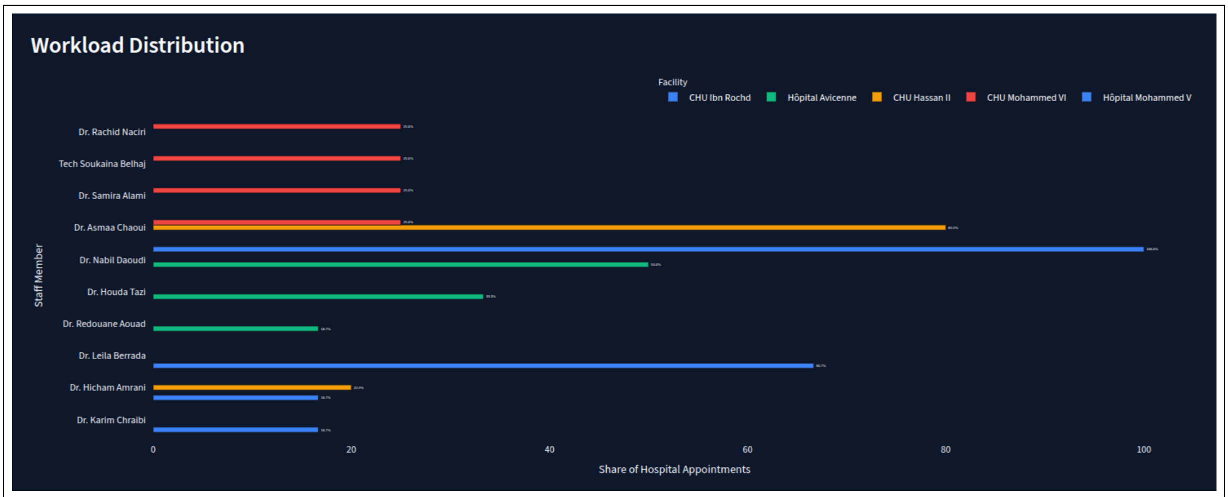


Figure 13: BarChart Final Result

```
# Using Plotly Express to handle multi-site overlaps
# Using px.bar with barmode='group' prevents overlapping.
fig = px.bar(
    df,
    x="PercentageShare",
    y="StaffName",
    color="HospitalName",
    orientation='h',
    barmode='group',
    text="PercentageShare",
    color_discrete_sequence=['#3b82f6', '#10b981', '#f59e0b', '#ef4444', "#b3ff00"],
    labels={"PercentageShare": "Share (%)", "StaffName": "Staff Member", "HospitalName": "Facility"}
)

fig.update_traces(texttemplate='%{text:.1f}%', textposition='outside')
fig.update_layout(
    title="",
    xaxis_title="Share of Hospital Appointments",
    template="plotly_dark",
    height=500,
    margin=dict(l=0, r=0, t=0, b=0),
    paper_bgcolor='rgba(0,0,0,0)',
    plot_bgcolor='rgba(0,0,0,0)',
    font=dict(color='#F8FAFC'),
    xaxis=dict(showgrid=False),
    yaxis=dict(showgrid=False),
    legend=dict(orientation="h", yanchor="bottom", y=1.02, xanchor="right", x=1)
)

st.plotly_chart(fig, use_container_width=True)
```

Listing 48: Plotly Grouped Bar Chart Logic

View Raw Data Table

	STAFF_ID	StaffName	HospitalName	TotalAppointments	PercentageShare
1	7	Dr. Karim Chraibi	CHU Ibn Rochd	1	16.67
2	26	Dr. Hicham Amrani	CHU Ibn Rochd	1	16.67
5	18	Dr. Redouane Aouad	Hôpital Avicenne	1	16.67
11	26	Dr. Hicham Amrani	CHU Hassan II	1	20
7	19	Dr. Samira Alami	CHU Mohammed VI	1	25
9	24	Tech Soukaina Belhaj	CHU Mohammed VI	1	25
8	20	Dr. Rachid Naciri	CHU Mohammed VI	1	25
6	12	Dr. Asmaa Chaoui	CHU Mohammed VI	1	25
4	10	Dr. Houda Tazi	Hôpital Avicenne	2	33.33
3	11	Dr. Nabil Daoudi	Hôpital Avicenne	3	50

Figure 14: Raw Data Table

3.3.5 Module 5: Patient Registration Utility (add_new_patient)

3.3.5.1 Context This module was not part of the original lab specifications. It was integrated into the final application following a recommendation from our Teaching Assistant, Miss Guerbouzi. We determined that a dedicated interface for registering new patients would serve as an invaluable tool for population and debugging, allowing us to rapidly generate test cases for the other modules.

3.3.5.2 Implementation Overview From a technical perspective, the architecture of this module is similar to the **Intelligent Scheduling System** (Module 2). It employs a similar strategy of form-based input handling and parameterized SQL insertion. To avoid redundancy, the functional logic of this module is not detailed in this document.

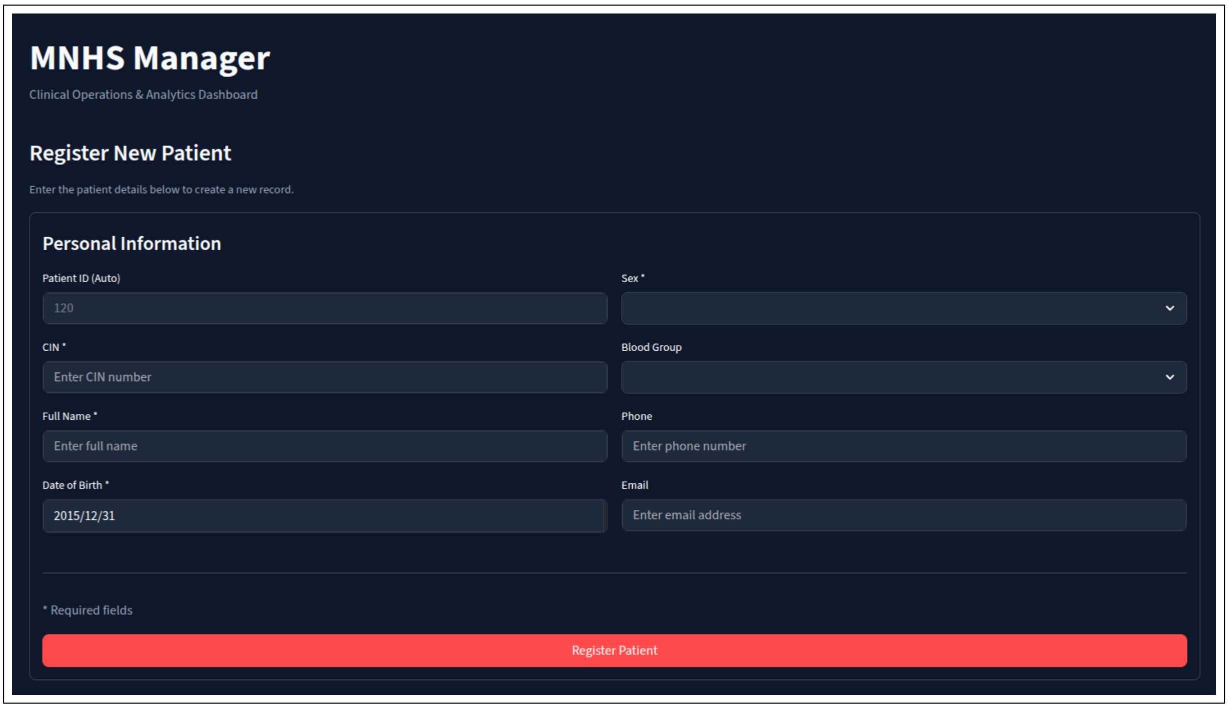


Figure 15: Add Patient Form

3.4 Adaptation Strategy: Handling TiDB Limitations

A core requirement of the lab specification was the implementation of four specific SQL Triggers to enforce business rules. However, the backend selected for this project, **TiDB**, does not currently support the `CREATE TRIGGER` statement found in standard MySQL 5.7.

To maintain functional parity with the requirements while leveraging the scalability of TiDB, the application implements a **Python Based Validation Architecture**. All constraints normally enforced by the database engine were migrated to the Python application layer.

The Requested Triggers were written as requested in 1.2 Triggers this choice was only made to allow deployement of the app while enforcing buisness rules.

The only trigger that was implemented is the double-booking prevention trigger as it is the only one needed in the tasks that were specified in the Lab.

3.4.1 The "No-Trigger" Architecture

In a traditional architecture, the database acts as the final gatekeeper of data integrity using procedural SQL. In this adapted architecture, the Python plays this role.

3.4.2 Double-Booking Prevention

Requirement: The system must reject an appointment request if the staff member is already scheduled for the exact same Date and Time.

1. **Verification Query:** Before any write operation, the application executes a `SELECT` query on the `ClinicalActivity` table, filtering by the requested `STAFF_ID`, `Date`, and `Time`.
2. **Conflict Detection:** The results are analyzed using `cursor.fetchone()`. If a record is returned, the time slot is identified as occupied.
3. **Transaction Abort:** Instead of letting the database throw a constraint error, the Python application explicitly raises a `ValueError`. This halts the execution flow immediately, preventing the `INSERT` statements from running and triggering a `ROLLBACK` of the transaction.

```python
# 1. Validation SQL: Check if staff is already booked
check_sql = """
    SELECT CAID FROM ClinicalActivity
    WHERE STAFF_ID = %s AND Date = %s AND Time = %s LIMIT 1
"""

# ... Inside the transaction ...
cur.execute(check_sql, (staff_id, date_str, time_str))
conflict = cur.fetchone()

if conflict:
    # STOP! Raise an error to prevent the insert
    raise ValueError(f"Double Booking Error: Staff {staff_id} is already busy.")
```

```

# If no conflict, proceed with the transaction
cur.execute(ins_ca, ...)
cur.execute(ins_appt, ...)
```

Listing 49: Double Booking Check Implementation

The full Python function can be found in 2.3.2 Module 2 : Intelligent Scheduling System

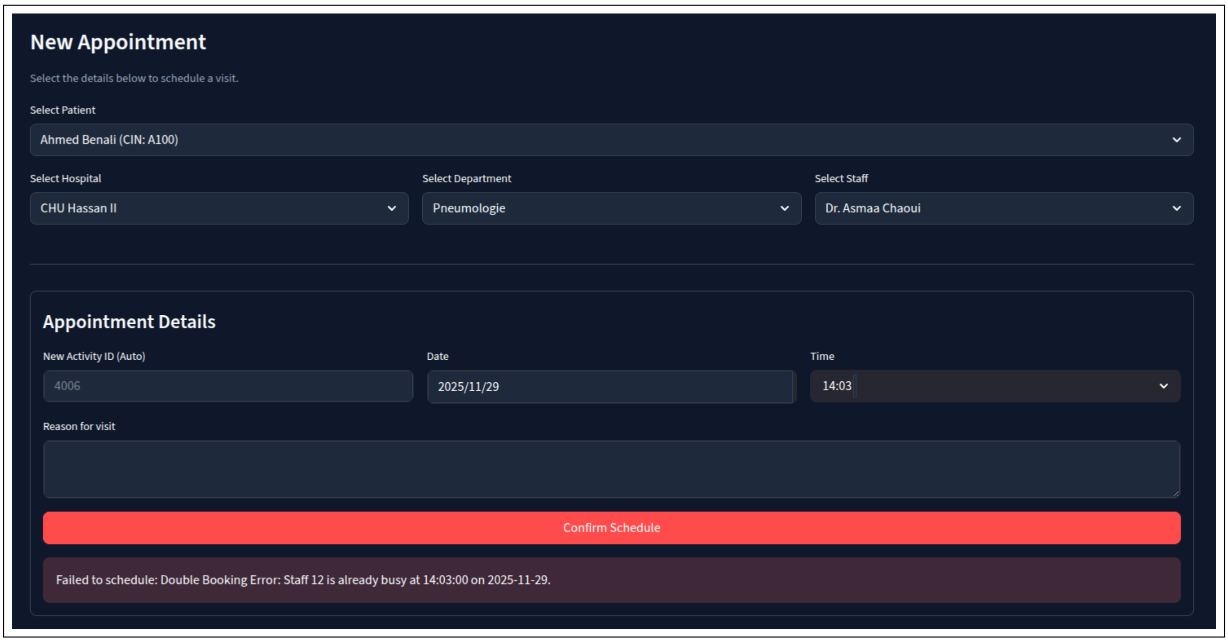


Figure 16: Double Booking Error Message

3.5 Conclusion

3.5.1 Summary of Achievements

This application layer successfully delivered a robust, full-stack "Application Layer" for the Moroccan National Health Services (MNHS) database. By integrating a Python-based Streamlit frontend with a distributed TiDB backend, the solution effectively bridges the gap between complex relational data and end-user accessibility.

Key technical achivements include:

- **Transactional Integrity:** The scheduling module ensures that the database maintains consistent states, even during complex multi-table write operations.
- **Architectural Adaptability:** The successful migration of business logic (constraints and validations) from the database layer to the application layer demonstrated the flexibility required when working with modern distributed SQL engines like TiDB, which may lack legacy features like Triggers.

3.5.2 Performance and Scalability

The chosen technology stack offers significant advantages for future growth. The use of **TiDB** ensures that the backend can scale horizontally to handle millions of patient records without complexity, while maintaining MySQL protocol compatibility. Furthermore, the **Streamlit** framework showed that complex, interactive dashboards can be deployed rapidly with a minimal code, making the system highly maintainable and easy to extend with new modules in the future.

3.6 Disclosure of AI Utilization

To maximize development efficiency and ensure a modern user interface, the frontend component of this application was developed with the assistance of Generative AI tools.

While the core database logic and SQL architecture were designed based on the lab specifications, We selected the **Streamlit** framework for the frontend interface. As we had no prior experience with this specific library, AI assistance was utilized to generate the initial code for the UI components (e.g., sidebars, forms, and layout containers).

It is important to note that we made a concerted effort to analyze, debug, and fully understand the generated code. The in-depth technical explanations provided in 2.3 Implementation and Code Logic, along with the specific code snippets details, serve as evidence of our understanding of the codebase.

3.7 Appendices

3.7.1 Project Repository and Source Code

To maintain the readability of this report, the full source code for the application is hosted in a public GitHub repository. This repository serves as the definitive source for the project's codebase, version history, and configuration files.

Repository Link:
`https://github.com/I-l-y-a-Z-z/DBMSApplicationLayerDeliverable5`

3.7.2 Local Deployment and Database Replication

The repository includes all necessary files to replicate the project environment locally:

- **Comprehensive Documentation (`README.md`):** A detailed guide providing step-by-step instructions for setting up the Python virtual environment, installing dependencies via `pip`, and launching the Streamlit server.
- **Database Generation Script (`query.sql`):** A monolithic SQL script is provided to fully reconstruct the MNHS database schema and populate it with the hyper-realistic dataset described in this report. This allows the application to be tested in a local MySQL or TiDB environment without requiring access to the production cluster.

3.7.3 Live Demonstration

For immediate testing and evaluation purposes, the application has been deployed to the Streamlit Cloud platform. This live instance connects directly to the production TiDB cluster, allowing users to interact with the full system functionality without the need for local installation or configuration.

Live Application URL:
`https://mnhslab6.streamlit.app`