NFO-B 512

SCIENTIFIC AND CLINICAL DATA MANAGEMENT

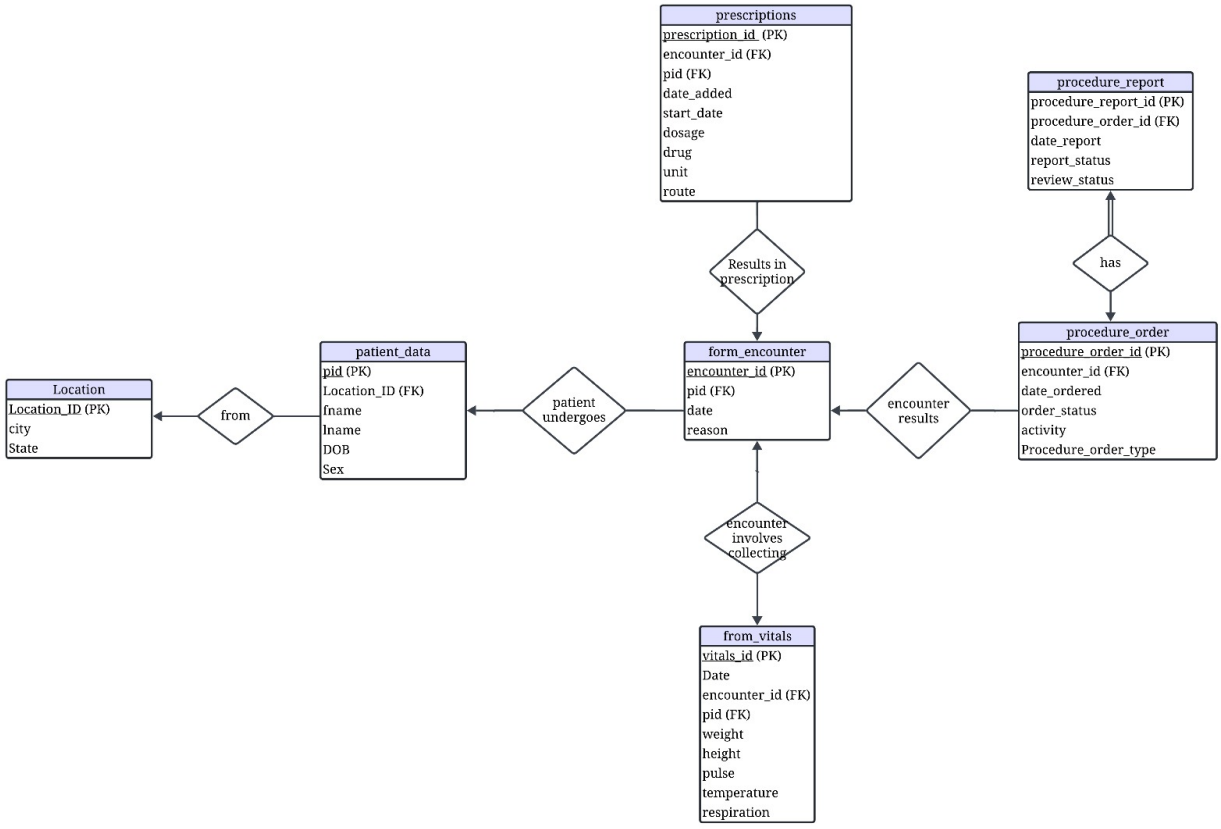
FINAL REPORT

GROUP 3: EMERGENCY ROOM VISIT ANALYSIS – A DATABASE MANAGEMENT PERSPECTIVE

SRI RAMYA PANJA

Database Design

Entity relationship diagram



Selected Entities and schema

Patient Data Table

Schema: Patient_Data (pid, fname, lname, DOB, Sex, Location_ID)

Location Table

Schema: Location (Location_ID, City, State)

Form_Encounter Table

Schema: Form_Encounter (encounter_id, pid, date, reason.)

Prescription Table

Schema: Prescription (prescription_id, encounter_id, date_added, date_modified, start_date, dosage, drug, unit, route)

Procedure Order Table

Schema: Procedure_Order (procedure_order_id, encounter_id, procedure_order_type date_ordered, date_collected, order_status, activity)

Procedure Report Table

Schema: Procedure_Report (procedure_report_id, procedure_order_id, date_collected, date_report, report_status, review_status)

Form_vitals

Schema: form_vitals (vitals_id, Date, encounter_id, pid, weight, height, pulse, temperature, respiration)

Key Attributes & Foreign Keys:

Primary keys: pid, Location_ID, encounter_id, vital_id, prescription_id, procedure_order_id, procedure_report_id

Foreign keys:

Patient_data.Location_ID → Location

Form_encounter.pid → Patient_data

Form_vitals.encounter_id → Form_encounter

Prescriptions.encounter_id → Form_encounter

Procedure_order.encounter_id → Form_encounter

Procedure_report.procedure_order_id → Procedure_order

Functional Dependencies (All in 3NF):

Location: Location_ID → city, state

Patient_data: pid → fname, lname, DOB, sex, Location_ID

Form_encounter: encounter_id → pid, date, reason.

Form_vitals: vitals_id → date, encounter_id, pid, weight, height, pulse, temperature, respiration

Prescriptions: prescription_id → encounter_id, date_added, date_modified, start_date, dosage, drug, unit, route Procedure_order: procedure_order_id → encounter_id, status, activity

Procedure_report: procedure_report_id → procedure_order_id, date_collected, date_report, report_status, review_status

Cardinality Constraints:

Location → Patient_data:
One-to-Many – Many patients can belong to one location.

Patient_data → Form_Encounter:
One-to-Many – A single patient can have multiple emergency room encounters.

Form_Encounter → Prescriptions:
One-to-Many – Each encounter can result in multiple prescriptions for that visit.

Form_Encounter → Form_Vitals:
One-to-One – Each encounter is linked to one set of vitals recorded during that visit.

Form_Encounter → Procedure_Order:
One-to-Many – One encounter can lead to several procedure orders.

Procedure_Order → Procedure_Report:
One-to-One – Each procedure order is associated with a single procedure report.

Describe the database design used in your project and explain why it is effective for your specific role in the project. Provide examples to support your explanation.

We started by identifying keywords from the scenario questions to select relevant entities from the OpenEMR database. Our primary goal was to build a clean, normalized schema in 3rd Normal Form, with strong referential integrity and no redundancy.

We chose binary relationships because they are easier to maintain and normalize effectively. Using the context from the CSV files provided (downloaded from OpenEMR), we analyzed the structure of tables like Form_Encounter, Patient_Data, and Prescriptions. For instance, the Prescriptions table contains encounter_id, which directly references the encounter_id in the Form_Encounter table, allowing us to link both. Similarly, pid was clearly a unique identifier for patients in the Patient_Data table and also appeared in the Form_Encounter table, helping us establish a foreign key relationship there.

We also examined the attributes within each table to determine which could act as primary keys and what values they determine. For example, in the Patient_Data table, we identified pid → fname, lname, DOB, sex, Location_ID as the functional dependency, confirming that all attributes depend entirely on the primary key. This logic was applied across all entities, and we derived functional dependencies accordingly, ensuring each table satisfies 3NF.

ERD Design Explanation

The key objective of our ERD was to keep all relationships binary.

Entities, Attributes, and Relationships

Location

Attributes: Location_ID, City, State

Location_ID is the primary key.

Patient_Data

Attributes: pid, fname, lname, DOB, sex, Location_ID

pid is the primary key.

Location_ID is a foreign key referencing Location.

Form_Encounter

Attributes: encounter_id, pid, date, reason, onset_date, sensitivity

encounter_id is the primary key.

pid is a foreign key referencing Patient_Data.

Form_Vitals

Attributes: vitals_id, date, encounter_id, pid, weight, height, pulse, temperature, respiration

vitals_id is the primary key.

encounter_id is a foreign key referencing Form_Encounter.

Prescriptions

Attributes: prescription_id, encounter_id, date_added, dosage, drug, unit, route

prescription_id is the primary key.

encounter_id is a foreign key referencing Form_Encounter.

Procedure_Order

Attributes: procedure_order_id, encounter_id, procedure_order_type date_ordered, date_collected, order_status, activity

procedure_order_id is the primary key.

encounter_id is a foreign key referencing Form_Encounter.

Procedure_Report

Attributes: procedure_report_id, procedure_order_id, date_collected, date_report, report_status, review_status

procedure_report_id is the primary key.

Cardinality Constraints:

Location → Patient_data:
One-to-Many – Many patients can belong to one location.

Patient_data → Form_Encounter:
One-to-Many – A single patient can have multiple emergency room encounters.

Form_Encounter → Prescriptions:
One-to-Many – Each encounter can result in multiple prescriptions for that visit.

Form_Encounter → Form_Vitals:
One-to-One – Each encounter is linked to one set of vitals recorded during that visit.

Form_Encounter → Procedure_Order:
One-to-Many – One encounter can lead to several procedure orders.

Procedure_Order → Procedure_Report:
One-to-One – Each procedure order is associated with a single procedure report.

procedure_order_id is a foreign key referencing Procedure_Order.

Elective questions

1.Describe how your database design allows for the addition of new attributes. How would you implement these changes using SQL?

Our database design is flexible enough to accommodate new attributes, especially since all our entities are in 3rd Normal Form (3NF). If a new attribute needs to be added to an existing entity and it functionally depends on the primary key, it can be integrated without affecting normalization. Using SQL, this would typically involve an ALTER TABLE command to add a new column. However, since our data was sourced from the OpenEMR system and imported via CSV files, any newly added attributes would not have corresponding data available in those original datasets. As a result, the values for the new attribute would remain NULL unless manually updated or populated from an additional data source.

2. Evaluate your database's normalization. To which normal form is it normalized? If not normalized, propose improvements.

Our database is in Third Normal Form (3NF). First, there are no repeating groups, so it satisfies 1NF. There are no partial dependencies, so it's in 2NF. And finally, all the attributes depend only on the primary key, not on other non-key attributes, so no transitive dependencies, which makes it 3NF. For example, In the Patient_data table, pid is the primary key, and all other attributes like fname, lname, DOB, sex, and Location_ID depend directly on it.

Same with the Form_Encounter table, encounter_id is the primary key and attributes like pid, reason, onset_date, etc., are directly dependent on it. We figured this out by checking the actual CSVs we imported from OpenEMR and understanding how things are linked. For instance, pid was unique in the Patient_data table and was used in Form_Encounter, so we knew it had to be a foreign key there. In form_vitals,

Area for imporovements: we included both pid and encounter_id , but since encounter_id already links to the encounter and that can give us pid, it's a bit redundant as we can get the details using a join when needed as there is referential integrity.

3. Discuss the selection of primary keys in your design, including the use of composite keys (if any).

We selected primary keys based on what uniquely identifies each record in a table. For example, pid uniquely identifies a patient in the Patient_data table, and encounter_id does the same for Form_Encounter. We made sure all other attributes in each table functionally depend on these keys. This helped us keep the design in Third Normal Form (3NF), no partial or transitive dependencies.

We also made sure the database supports lossless joins. For instance, if we join Patient_data and Form_Encounter on pid, the patient and their visits match correctly, and no data is lost in the process. This shows that our foreign key relationships and design choices preserve data integrity when querying across tables.

4. How have you implemented referential integrity? Provide a specific example from your project.

We used foreign keys to maintain referential integrity between related tables. For example, Form_Encounter.pid connects to Patient_data.pid, Form_vitals.encounter_id links back to Form_Encounter Same with Prescriptions.encounter_id and Procedure_Report.procedure_order_id

This helped us keep the relationships clean and meaningful. It also made querying easier, we could just join on the foreign key and pull connected info without losing anything. So, the joins were lossless, and everything stayed consistent.

Data Analytics:

Which attributes were used for analytics? Provide SQL queries used to retrieve this data from your database.

For our analysis, we selected following attributes from different tables:

From Form_Encounter:

reason (to identify common visit reasons)

pid (to link with patient demographics)

From Vitals:

temperature, pulse, respiration (to flag abnormal vitals)

From Patient:

DOB (used to calculate patient age)

sex (for demographic breakdowns)

From Procedure_Order:

procedure_order_type (to see the type and frequency of procedures)

From Prescription:

encounter_id and pid (to track how many patients received medication after an ER visit)

Query 1. What are the most frequent reasons for emergency room visits?

```
# Q1: What are the most frequent reasons for emergency room visits?
cursor.execute('''
    SELECT reason, COUNT(*) AS visit_count
    FROM Form_Encounter
    GROUP BY reason
    ORDER BY visit_count DESC
    LIMIT 5;
''')
```

Rationale for SQL query: To find the most common reasons for ER visits, I used the Form_Encounter table and grouped records by the "reason" column. Then I counted each group using COUNT (*), sorted them with ORDER BY in descending order

Query 2. Are there patterns in patient vital signs that indicate a higher risk of hospitalization after an ER visit?

```
query = '''
SELECT
    summary.Hospitalized,
    SUM(summary.High_Temperature) AS High_Temperature,
    SUM(summary.High_Pulse) AS High_Pulse,
    SUM(summary.High_Respiration) AS High_Respiration
FROM (
    SELECT
        CASE
            WHEN LOWER(fe.reason) LIKE '%admission%' THEN 'Hospitalized'
            ELSE 'Not Hospitalized'
        END AS Hospitalized,
        CASE WHEN v.temperature > 38 THEN 1 ELSE 0 END AS High_Temperature,
        CASE WHEN v.pulse > 100 THEN 1 ELSE 0 END AS High_Pulse,
        CASE WHEN v.respiration > 24 THEN 1 ELSE 0 END AS High_Respiration
    FROM
        Vitals v
    INNER JOIN
        Form_Encounter fe ON v.encounter_id = fe.encounter_id
    WHERE
        v.temperature IS NOT NULL
        AND v.pulse IS NOT NULL
        AND v.respiration IS NOT NULL
) AS summary
GROUP BY
    summary.Hospitalized;

'''
```

Rationale for SQL query: To figure out if abnormal vitals were linked to hospitalization. The dataset didn't have a clear "hospitalized" attribute, so we looked at the reason column in the Form_Encounter table. After checking the CSV manually, we noticed that cases involving admission had the word "admission" in the reason string. So, we used LIKE '%admission%' in the SQL query to classify visits as either 'Hospitalized' or 'Not Hospitalized'. The inner query flagged abnormal vitals using CASE WHEN conditions based on thresholds for temperature, pulse, and respiration. The outer query grouped the results by hospitalization status and summed up the abnormal flags using SUM() to get the total count for each group.

Query 3. Which procedures are most performed in the emergency room, and how do they vary by patient age group?

```
cursor.execute('''
    SELECT
        pa.sex,
        CASE
            WHEN TIMESTAMPDIFF(YEAR, pa.DOB, CURDATE()) <= 18 THEN '0-18'
            WHEN TIMESTAMPDIFF(YEAR, pa.DOB, CURDATE()) <= 35 THEN '19-35'
            WHEN TIMESTAMPDIFF(YEAR, pa.DOB, CURDATE()) <= 50 THEN '36-50'
            ELSE '51+'
        END AS Age_Group,
        COUNT(*) AS Visit_Count
    FROM
        Form_Encounter fe
    JOIN
        Patient pa ON fe.pid = pa.pid
    GROUP BY
        pa.sex, Age_Group
    ORDER BY
        Visit_Count DESC;
''')
```

Rationale for SQL query: We wanted to see which procedures are done the most in the ER and how they differ across age groups. So, we got the procedure types and used the patient's date of birth to calculate their age. Then, using a CASE statement, we grouped them into clear age ranges like 0-18, 19-35, 36-50, and 51+. We used COUNT to see how many times each procedure happened in each age group. To get this info, we joined the Procedure_Order table with Form_Encounter using encounter_id, and then joined that with the Patient table using pid. We grouped the data by procedure and age group, and then ordered it by visit count.

Query 4. Are there differences in emergency room visit frequency based on demographics?

```
cursor.execute('''
    SELECT
        pa.sex,
        CASE
            WHEN TIMESTAMPDIFF(YEAR, pa.DOB, CURDATE()) <= 18 THEN '0-18'
            WHEN TIMESTAMPDIFF(YEAR, pa.DOB, CURDATE()) <= 35 THEN '19-35'
            WHEN TIMESTAMPDIFF(YEAR, pa.DOB, CURDATE()) <= 50 THEN '36-50'
            ELSE '51+'
        END AS Age_Group,
        COUNT(*) AS Visit_Count
    FROM
        Form_Encounter fe
    JOIN
        Patient pa ON fe.pid = pa.pid
    GROUP BY
        pa.sex, Age_Group
    ORDER BY
        Visit_Count DESC;
''')
```

Rationale for SQL query: I used this query to find out if certain age groups and genders visit the emergency room more often. It joins the Form_Encounter table with the Patient table to access each patient's date of birth and sex. Then it calculates their age and groups them into four age ranges: 0–18, 19–35, 36 -50, and 51+. For each combination of age group and sex, it counts the number of ER visits using COUNT (*)

Query 5. How often do emergency room patients require follow-up prescriptions, and what are the most prescribed medications?

```
# Q5: How often do emergency room patients require follow-up prescriptions, and what are the most pr
#Getting total distinct patients from Form_Encounter

cursor.execute('''
    SELECT COUNT(DISTINCT pid) FROM Form_Encounter
''')
total_patients = cursor.fetchone()[0]

#Getting distinct patients who received at least one prescription
cursor.execute('''
    SELECT COUNT(DISTINCT fe.pid)
    FROM Prescription p
    JOIN Form_Encounter fe ON p.encounter_id = fe.encounter_id
''')
patients_with_prescription = cursor.fetchone()[0]

#Calculate percentages
patients_without_prescription = total_patients - patients_with_prescription
percentages = [
    (patients_with_prescription / total_patients) * 100,
    (patients_without_prescription / total_patients) * 100
]
```

I used this query to figure out how many unique patients got at least one prescription after an emergency room visit. Since prescriptions are linked to specific encounters using encounter_id, I joined the Prescription table with the Form_Encounter table. Then I used COUNT (DISTINCT fe.pid) to make sure each patient is only counted once, even if they got multiple prescriptions. This helped me understand how common follow-up prescriptions are among ER patients.

```
#Step 5: SQL Query - Top 5 Prescribed Medications
cursor.execute('''
    SELECT
        p.drug,
        COUNT(*) AS prescription_count
    FROM
        Prescription p
    JOIN
        Form_Encounter fe ON p.encounter_id = fe.encounter_id
    WHERE
        p.drug IS NOT NULL
    GROUP BY
        p.drug
    ORDER BY
        prescription_count DESC
    LIMIT 5;
''')
```

This query helped me find out which drugs were prescribed the most. I joined the same two tables again using encounter_id, filtered out any rows where the drug field is null, and then grouped by drug name. I used COUNT(*) to get how many times each drug was prescribed and sorted the

result in descending order to get the top 5 most prescribed drugs. This gave me insight into common treatments given after ER visits.

Electives Questions

2. Explain a scenario where you optimized a query for better performance. Describe the modifications made

For Query 2, we needed to find out which patients were hospitalized, but there was no direct ‘hospitalized’ attribute in the database. Instead of adding new data or changing the schema, we optimized the query by understanding the dataset contextually.

We went through the CSV files and noticed that in the Form_Encounter table, the reason column sometimes included the word ‘Admission,’ which implied that the patient was likely hospitalized.

So, we used a LIKE '%admission%' condition to filter those records. We also converted the reason to lowercase using LOWER() to make sure the match was case-insensitive. This allowed us to indirectly group patients into ‘Hospitalized’ and ‘Not Hospitalized’ based on actual encounter reasons.

That small contextual understanding helped us avoid adding extra logic or columns, and made the query both effective and efficient.

```
# Step 2: SQL Query
query = '''
SELECT
    summary.Hospitalized,
    SUM(summary.High_Temperature) AS High_Temperature,
    SUM(summary.High_Pulse) AS High_Pulse,
    SUM(summary.High_Respiration) AS High_Respiration
FROM (
    SELECT
        CASE
            WHEN LOWER(fe.reason) LIKE '%admission%' THEN 'Hospitalized'
            ELSE 'Not Hospitalized'
        END AS Hospitalized,
        CASE WHEN v.temperature > 38 THEN 1 ELSE 0 END AS High_Temperature,
        CASE WHEN v.pulse > 100 THEN 1 ELSE 0 END AS High_Pulse,
        CASE WHEN v.respiration > 24 THEN 1 ELSE 0 END AS High_Respiration
    FROM
        Vitals v
    INNER JOIN
        Form_Encounter fe ON v.encounter_id = fe.encounter_id
    WHERE
        v.temperature IS NOT NULL
        AND v.pulse IS NOT NULL
        AND v.respiration IS NOT NULL
) AS summary
GROUP BY
    summary.Hospitalized;

'''
```

3. If your project involved integrating data from multiple sources, how did you ensure data consistency and integrity?

Yes, we imported multiple CSV files from OpenEMR and loaded them into our own phpMyAdmin database. Before that, we designed our schema in a way that made sense for the data, only kept the attributes we needed, mapped the relationships, and made sure it followed 3NF to avoid any redundancy.

For example, we saw that both Patient_data and Form_Encounter shared the pid, so we made pid a foreign key in Form_Encounter. Same with encounter_id, which was used in Vitals, Prescription, and Procedure_Order. These links helped us join tables later without any data loss.

We also made sure that the data types in our tables matched the CSV files, for instance, date_added in Prescription and DOB in Patient_data were both set as DATE so the import wouldn't break.

After uploading, we verified the joins by running queries, like counting how many Vitals entries actually matched with Form_Encounter, to make sure the foreign keys were working as intended. All of this helped us keep the data consistent and accurate across sources.

```
# Create Vitals table
cursor.execute('''
CREATE TABLE IF NOT EXISTS Vitals (
    vital_id INT PRIMARY KEY,
    pid INT,
    date DATE,
    weight FLOAT,
    height FLOAT,
    temperature FLOAT,
    pulse FLOAT,
    respiration FLOAT,
    encounter_id INT,
    FOREIGN KEY (pid) REFERENCES Patient(pid),
    FOREIGN KEY (encounter_id) REFERENCES Form_Encounter(encounter_id)
);
```