Università di Genova

Project for Computer Engineering students

NoSQL data store to support a large scale application:

Hospital Billing & Information System

Martyna Majewska

Monika Mazella

1. Domain and Application

Hospital Billing & Information System focusing on managing patient records, tracking encounters, procedures, facilitate billing and insurance claims with payer data.

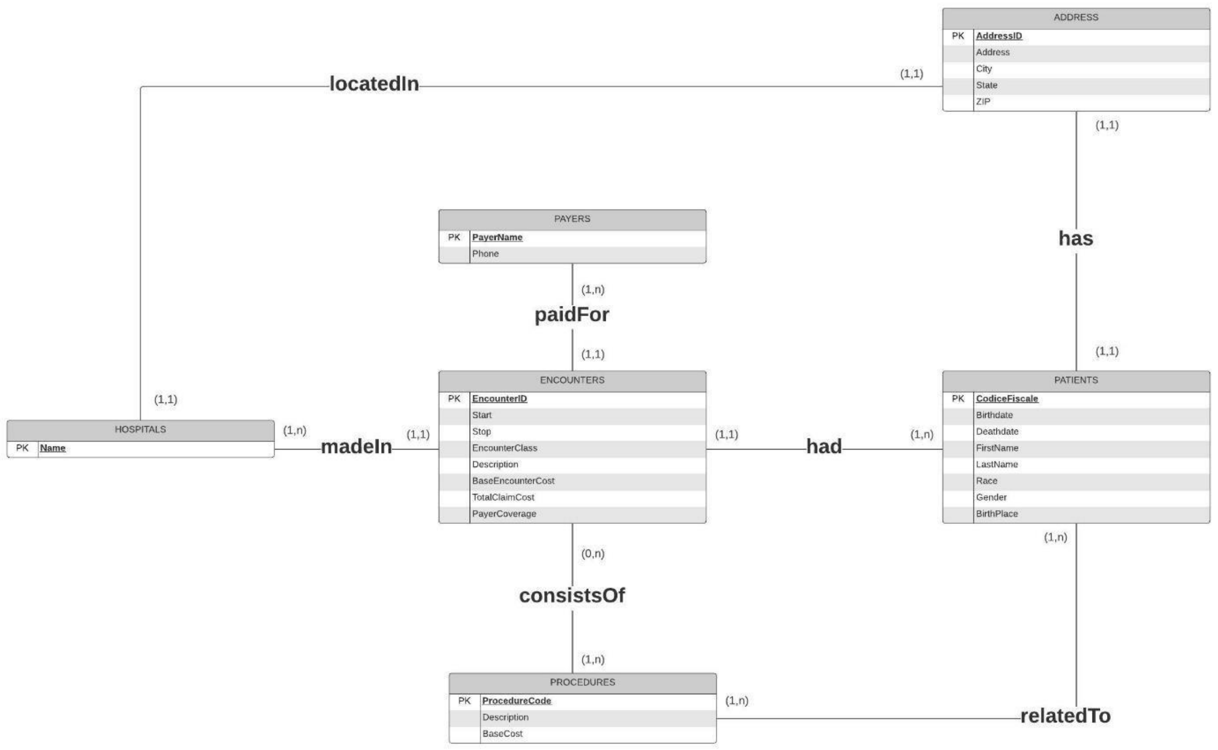
2. Nature of Application, System Requirements

Application is mostly used by the hospital and admission staff to control patient's billings and issued procedures. It uses both, **transactional** and **analytical** operations, so it is **read/write intensive** and it needs to support real-time updates to patient records, procedures they had and their costs. Frequent reads for medical history as well as visit details (history of procedures), billing records including total costs of encounters, coverage by insurance (or its lack).

Additionally, **batch processing** will be needed for tasks like generating hospital performance reports (yearly, monthly hospital income, average income per patient, average encounter time, types of encounters performed).

While **high consistency** is crucial and for us, it is placed in the first place, **availability** of the system should also be mediated. Admission staff, receptionists need real-time data on the patient's current treatment – need to know current costs of treatments, to have accurate calculation of total costs (soon after doctor adds some procedures, or insurance company covers some value). Doctors and nurses need to see up to date patient procedure history. While it is not necessary as the consistency, system should provide some availability to update and write new records to the system.

3. Conceptual schema



4. Workload and frequent operations

- <u>What procedures has a patient with a given Codice Fiscale undergone?</u>
  Query will show all procedures associated with the patient's encounters, relevant for both doctors and patient's billings.
  - Doctors: To review the patient's medical history and ongoing treatment.
  - Billing Staff: To confirm billed procedures for payment processing and insurance claims.
- <u>What is the total amount of money that a given payer in given month must cover for encounters?</u>
  Query summarizes costs covered by a specific insurance company in a given month. It is relevant for the insurance companies while at the end of the month they can see how much they spent in given hospital.
  - Insurance Companies: To see how much they spent at the hospital in that month and help plan their budgets.

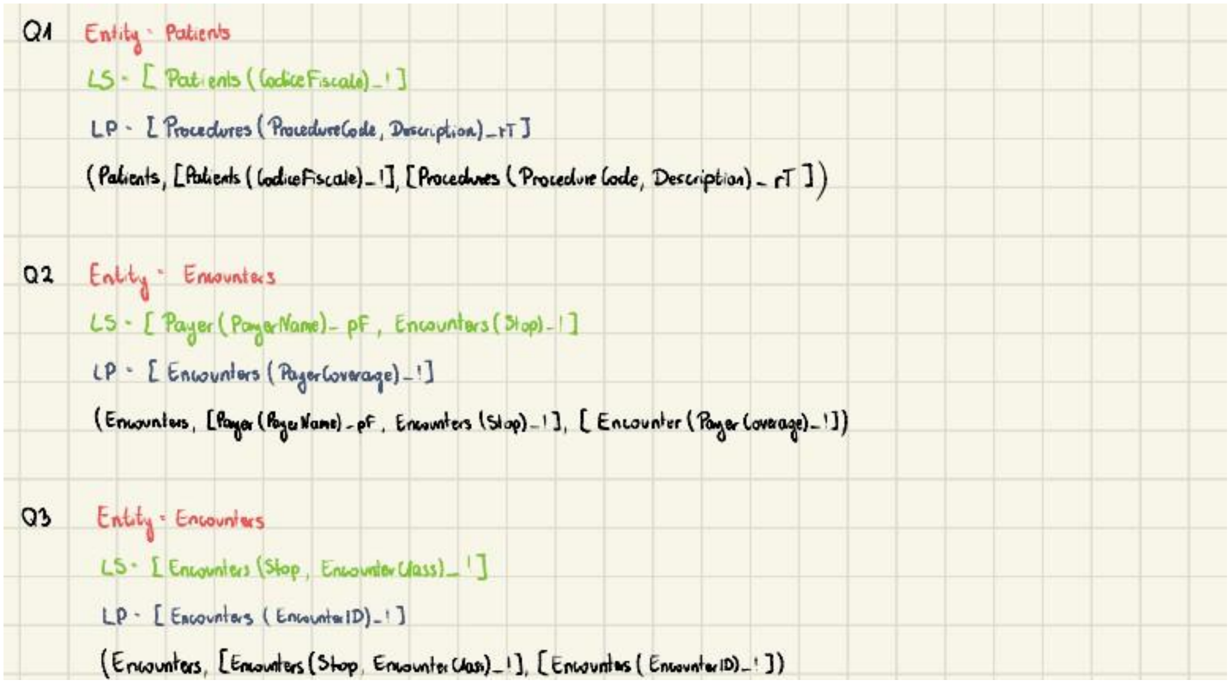
    - Hospital Finance Teams: To generate monthly billings and check completeness of payments from insurance companies.

- <u>How many times in a given month was it necessary to perform 'urgent care'?</u>
  Query counts the number of urgent care procedures for performance evaluation.
    - Performance Evaluation: Providing hospital administrators with metrics to assess the demand and efficiency of urgent care services.
    - Resource Management: Helping in planning staff allocation, equipment needs, and improving patient response times based on historical trends.

5. Aggregate-oriented design methodology

STEP 1

Q1 – Given patient Codice Fiscale, determine what procedures he/she had.
Q2 – Given payer name and given month, determine total amount of money that needs to be covered by payer/insurance company.
Q3 – Given month, determine number of encounters that was labeled ‘urgent care’.

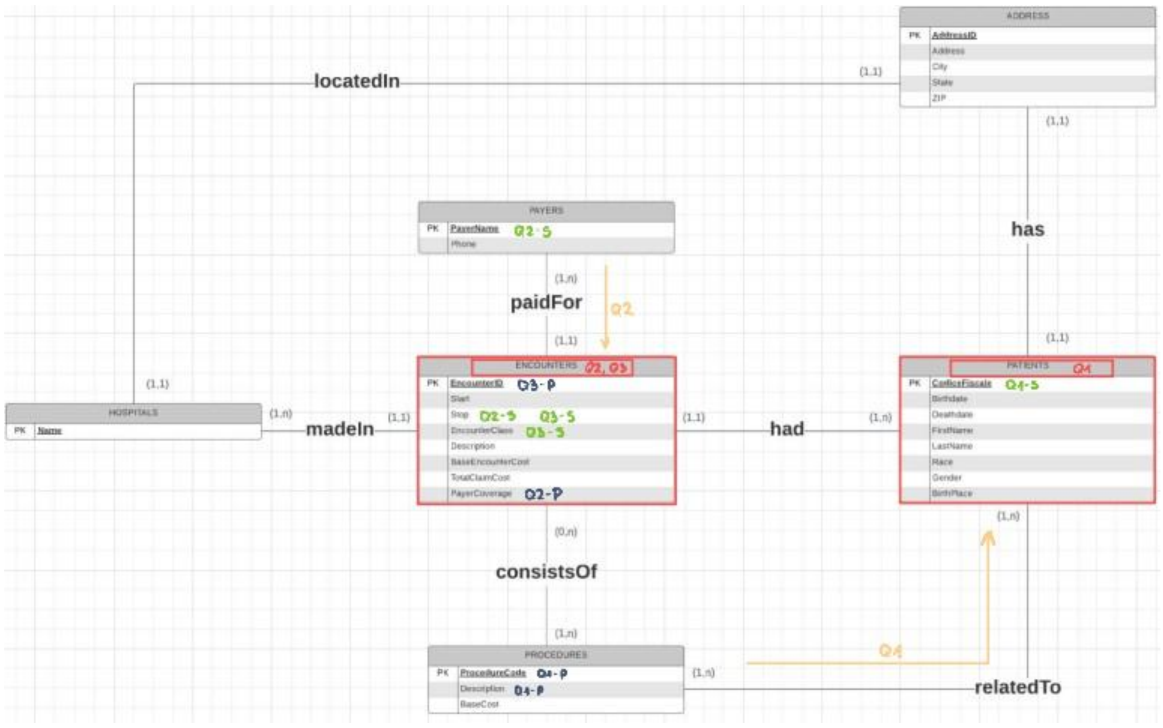


Q1 Entity - Patients
LS - [Patients (CodiceFiscale)_!]
LP - [Procedures (ProcedureCode, Description)_rT]
(Patients, [Patients (CodiceFiscale)_!], [Procedures (Procedure Code, Description)_rT])

Q2 Entity - Encounters
LS - [Payer (PayerName)_pF , Encounters (Stop)_!]
LP - [Encounters (PayerCoverage)_!]
(Encounters, [Payer (PayerName)_pF , Encounters (Stop)_!], [Encounter (Payer Coverage)_!])

Q3 Entity - Encounters
LS - [Encounters (Stop, EncounterClass)_!]
LP - [Encounters (EncounterID)_!]
(Encounters, [Encounters (Stop, EncounterClass)_!], [Encounters (EncounterID)_!])

STEP 2



STEP 3

<u>Q1:</u>

patients: { <u>CodiceFiscale</u>, hadProcedures:[{ProcedureCode, Description}] }

<u>Q2, Q3:</u>

encounters: { <u>EncounterID</u>, Stop, EncounterClass, PayerCoverage, PayerName}

6. Chosen system: MongoDB

a. <u>Schema for selected system:</u>

Q1:

patients: { _id, <u>CodiceFiscale</u>, hadProcedures:[{ProcedureCode, Description}] }

Selection Attributes Q1: {CodiceFiscale}

Shard Key (+unique index): {CodiceFiscale}, hashed

Q2, Q3:

encounters: { _id, <u>EncounterID</u>, Stop, EncounterClass, PayerCoverage, PayerName }

Selection Attributes Q2: { Stop, PayerName}

Selection Attributes Q3: { Stop, EncounterClass }

Shard Key (+ non unique index): { Stop }, hashed

Unique index: { Stop, EncounterID }

b. Operations from workload in MongoDB:

Q1:

```
db.patients.find({"CodiceFiscale":"CF002J6UIMM"},{"hadProcedures.ProcedureCode":1,
"hadProcedures.Description" : 1, "_id":0}).pretty()
```

Q2:

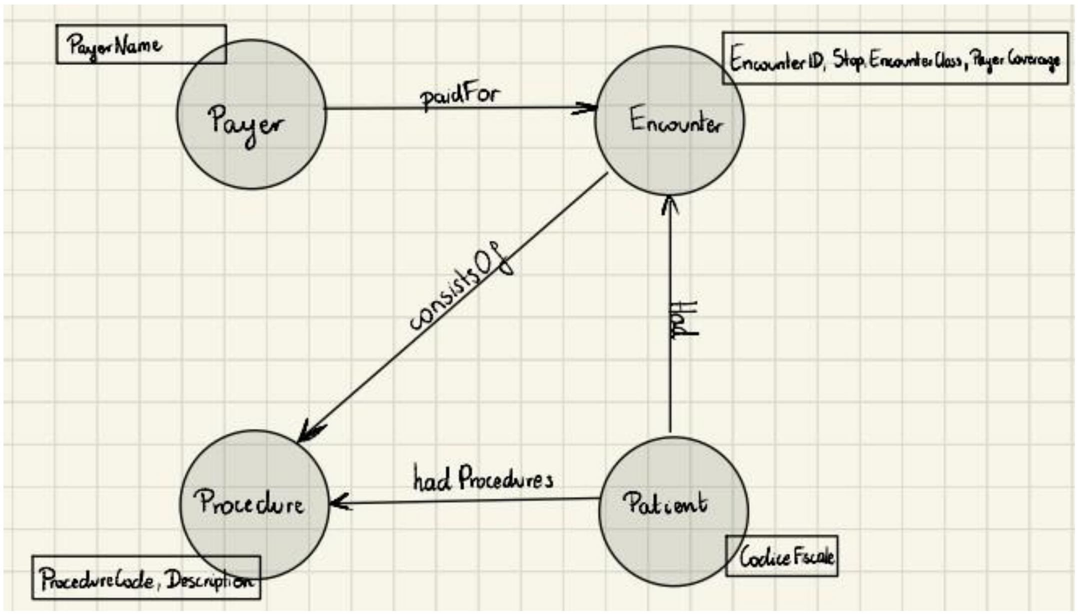
```
db.encounters.aggregate([ {$addFields:{ "Month":{$dateToString:{format:"%Y-%m",date:{
$toDate:"$Stop"}}}}}, {$match:{"PayerName":"Medicare", "Month":"2011-01"}},
{$group:{_id:"$PayerName",toPay:{$sum:"$PayerCoverage"}}}]);
```

Q3:

```
db.encounters.aggregate([{$addFields:{"Month":{$dateToString:{format:"%Y-%m",
date:{$toDate:"$Stop"}}}}}, {$match: {"Month":"2011-02", "EncounterClass":"urgentcare"}},
{$count: "urgentCareCounts"}]);
```

7. Design in Neo4J

a. Schema for Neo4j:



b. Operations of the workload in Neo4j:

Q1:

```
MATCH (p:Patient)-[:hadProcedures]->(proc:Procedure)
WHERE p.CodiceFiscale = "CF002J6UIMM"
RETURN proc.ProcedureCode, proc.Description;
```

```
Q2:
MATCH (payer:Payer)-[:paidFor]->(enc:Encounter)
WHERE payer.PayerName= "Medicare" AND datetime(enc.Stop).month = 1 AND
datetime(enc.Stop).year = 2011
RETURN SUM(enc.PayerCoverage) AS TotalCoverage;

Q3:
MATCH (enc:Encounter)
WHERE datetime(enc.Stop).month = 1 AND datetime(enc.Stop).year = 2011 AND
enc.EncounterClass = "urgentcare"
RETURN COUNT(enc) AS UrgentCareCount;
```

8. Most suitable back-end system for given application: MongoDB

As we place high consistency at the first place in our application as the back-end system we would use MongoDB. Considering the features of application and identified workload it would be the most suitable system for keeping critical data (patient records, procedures, and billing) that considers the importance of data accuracy.

MongoDB is optimized for frequent reads and writes, essential for updates and querying historical patient data.

However, the main advantage of MongoDB over other NoSQL systems is that **all nested document components can be accessed during those read and write operations.** The structure of an aggregate is visible at the logical and application level (useful for operations like adding a new procedure to a patient).

With this system, application will provide eventual consistency at replicas at the same time addressing availability with replica sets, making it fault-tolerant and operational during node failures (taking care of high-importance and sensitive data).

9. System configuration needed in MongoDB for storing and processing

Partitioning (sharding):
- **Hashed-based** (hash function on shard keys)

Replica Set:
- **Master-slave approach** (one master node - receiving requests, two slave nodes)

- **Fault-tolerace:** use of primary replicas (accepting read, write operations), secondary replicas (only read operations)

Consistency, Availability:

- **CP** Theorem, but trade-off consistency for some availability (loosening some of consistency requirements and making writes on a **QUORUM** of replica nodes, or even making system asynchronous but we don't think it would be that necessary in this case)

Indexing:

- Ensure **indexes** (CodiceFiscale) and **compound indexes** (Stop, EncounteID) on shard keys.

System transactions:

- **ACID** - single write operation will modify multiple documents, and modification at each document is atomic (on a level of a single document).

Each node Resources:

- **Storage about 1T**
- **CPU:** Multi-core processor (6-8 cores), handling concurrent operations.
- **RAM: 16GB (for real time updates)**
  (Our database is much smaller than (we assume) it would be for a real data-store.)

10. Logical schema in system MongoDB.

Q1:
patients: { _id, <u>CodiceFiscale</u>, hadProcedures:[{ProcedureCode, Description}] }
Selection Attributes Q1: {CodiceFiscale}
Shard Key (+unique index): {CodiceFiscale}, hashed

```
db.patients.createIndex({"CodiceFiscale": 1}, {unique: true})
db.adminCommand({shardCollection: "db.patients", key: {"CodiceFiscale": 1}, field: "hashed"})
```

Q2, Q3:
encounters: { _id, <u>EncounterID</u>, Stop, EncounterClass, PayerCoverage, PayerName }
Selection Attributes Q2: { Stop, PayerName }
Selection Attributes Q3: { Stop, EncounterClass }
Shard Key (+ non unique index): { Stop }, hashed
Unique index { Stop, EncounterID }

```
db.encounters.createIndex({"Stop": 1, "EncounterID": 1}, {unique: true})
db.encounters.createIndex({"Stop": 1}, {unique: false})
db.adminCommand({shardCollection: "db.encounters", key: {"Stop": 1}, field: "hashed"})
```

11. Creating instance of schema in the MongoDB

Dataset source: https://mavenanalytics.io/data-playground

Importing files to local machine:
scp "encounters.json" user16@130.251.61.97:
scp "patients.json" user16@130.251.61.97:

In mongo db we created indexes:
db.patients.createIndex({“CodiceFiscale”: 1}, {unique: true})
db.adminCommand({shardCollection: “db.patients”, key: {“CodiceFiscale”: 1}, field: “hashed”})*

db.encounters.createIndex({“Stop”: 1, “EncounterID”: 1}, {unique: true})
db.encounters.createIndex({“Stop”: 1}, {unique: false})
db.adminCommand({shardCollection: “db.encounters”, key: {“Stop”: 1}, field: “hashed”})*

**since we want to import data from the existing collections, and we are not allowed to partition/shard a collection (admin privileges are needed to enable partitioning)*

Importing data from the json files to populate the collections:
mongoimport --username=user16 --password=8u5oz9 --collection=encounters --db=user16_db --file=encounters.json --jsonArray

mongoimport --username=user16 --password=8u5oz9 --collection=patients --db=user16_db --file=patients.json --jsonArray

To verify data:
- db.patients.find().pretty();
- db.encounters.find().pretty();

12. Implement the workload in MongoDB (already done previously):

Q1:
db.patients.find({"CodiceFiscale":"CF002J6UIMM"},{"hadProcedures.ProcedureCode":1, "hadProcedures.Description" : 1, "_id":0}).pretty()

Q2:

```
db.encounters.aggregate([ {$addFields:{ "Month":{$dateToString:{format:"%Y-%m",date:{$toDate
:"$Stop"}}}}}, {$match:{"PayerName":"Medicare", "Month":"2011-01"}},
{$group:{_id:"$PayerName",toPay:{$sum:"$PayerCoverage"}}}]);
```

Q3:

```
db.encounters.aggregate([{$addFields:{"Month":{$dateToString:{format:"%Y-%m",
date:{$toDate:"$Stop"}}}}}, {$match: {"Month":"2011-02", "EncounterClass":"urgentcare"}},
{$count: "urgentCareCounts"}]);
```