Database Schema Design for a Non-Relational Information System

Silvana Greca
Department of Informatics
Faculty of Natural Sciences
University of Tirana

Agenda

- NoSQL & MongoDB background
- Data modeling approach
- Data dictionary & visual schema
- Retrieval & maintenance requirements
- Implementation examples
- Conclusion

What is NoSQL

- Stands for **N**ot **O**nly **SQL??**
- Class of non-relational data storage systems
- Usually do not require a fixed table schema nor do they use the concept of joins
  - Distributed data storage systems
- **Many NoSQL systems relax one or more ACID properties**, particularly consistency, in order to achieve higher scalability and availability.

  However, modern systems such as MongoDB support ACID transactions with certain trade-offs.

- Non-relational database paradigm
- Designed for scalability and flexibility
- Often trades strict consistency for availability
- Supports semi-structured and unstructured data

NoSQL: Categories

- Key-value

- Graph database

- Document-oriented

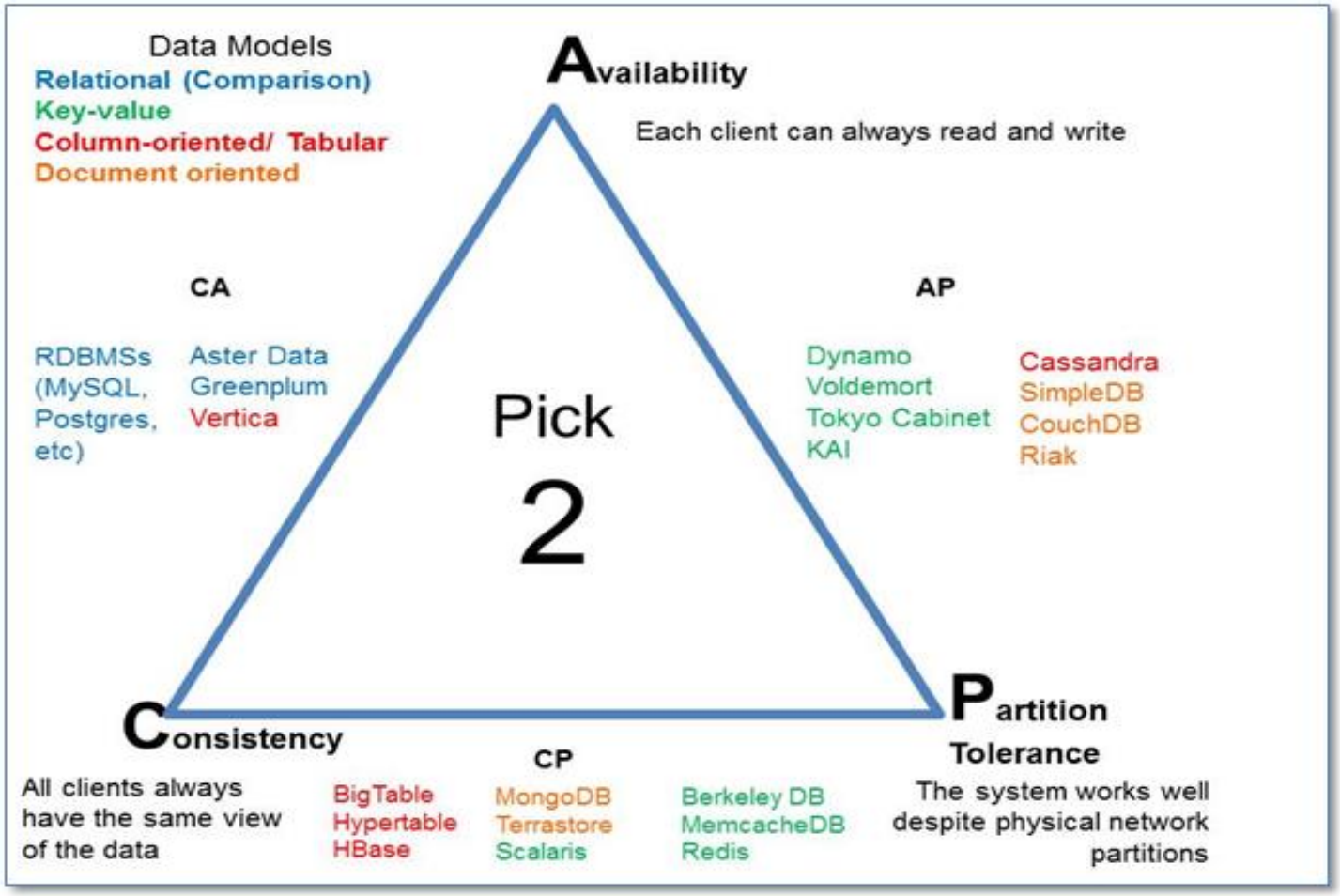
- Column family

CAP Theorem

- Three properties of a system
  - **Consistency** (all copies have same value)
  - **Availability** (system can run even if parts have failed)
    - All nodes can still accept reads and writes
  - **Partition Tolerance** (Even if part is down, others can take over)

- CAP "Theorem":
  - You can have at most two of these three properties for any system
  - Pick two !!!

CAP Theorem


Data Models
Relational (Comparison)
Key-value
Column-oriented/ Tabular
Document oriented
Availability
Each client can always read and write
CA
RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica
Pick 2
AP
Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak
Consistency
All clients always have the same view of the data
CP
BigTable
Hypertable
HBase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis
Partition Tolerance
The system works well despite physical network partitions

The BASE Properties

- The CAP theorem proves that it is impossible to guarantee strict Consistency and Availability while being able to tolerate network partitions
- This resulted in databases with relaxed ACID guarantees
- In particular, such databases apply the BASE properties:
  - **B**asically **A**vailable: the system guarantees Availability
  - **S**oft-State: the state of the system may change over time
  - **E**ventual Consistency: the system will *eventually* become consistent

What does NoSQL Not Provide

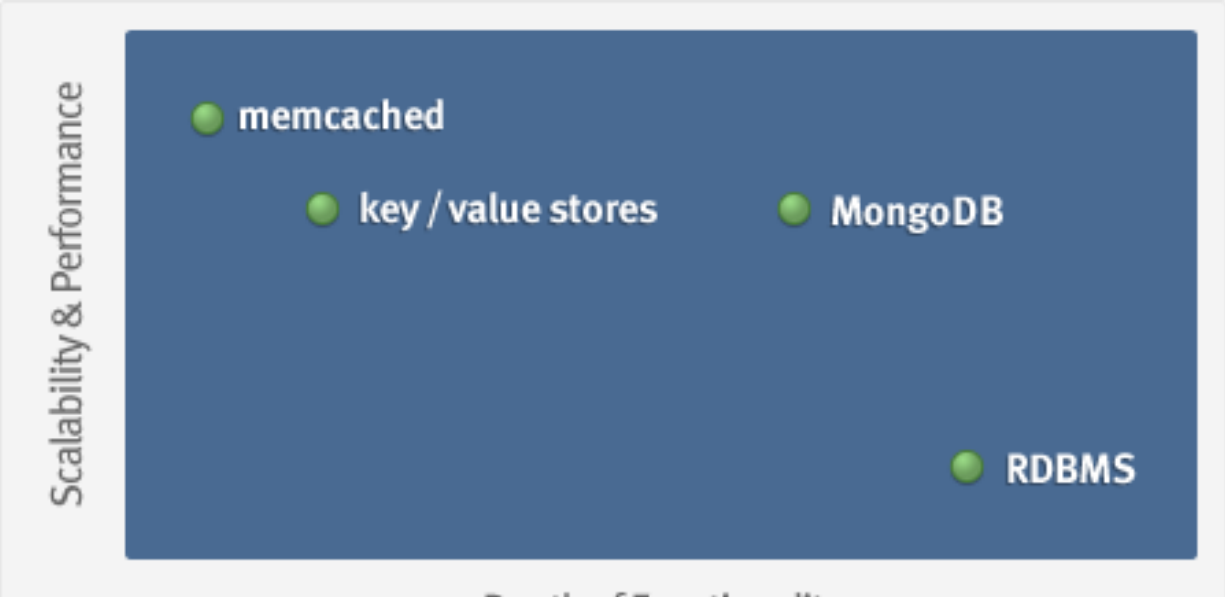
- No built-in join
- Historically, NoSQL systems did not support ACID transactions.
- Modern NoSQL databases, such as MongoDB, now provide ACID transaction support while prioritizing scalability and performance.
- No SQL

What is MongoDB?

- **Defination:** MongoDB is an **open source**, **document-oriented** database designed with both scalability and developer agility in mind.
- Instead of storing your data in tables and rows as you would with a relational database, in MongoDB you store **JSON-like documents** with **dynamic schemas**.
- **MongoDB is a schema-flexible document-oriented database.**
- It does not enforce a fixed schema at the storage level, but schemas can be defined and enforced at the application or database validation level.
- **Document-Oriented DB**
  - Unit object is a document instead of a row (tuple) in relational DBs

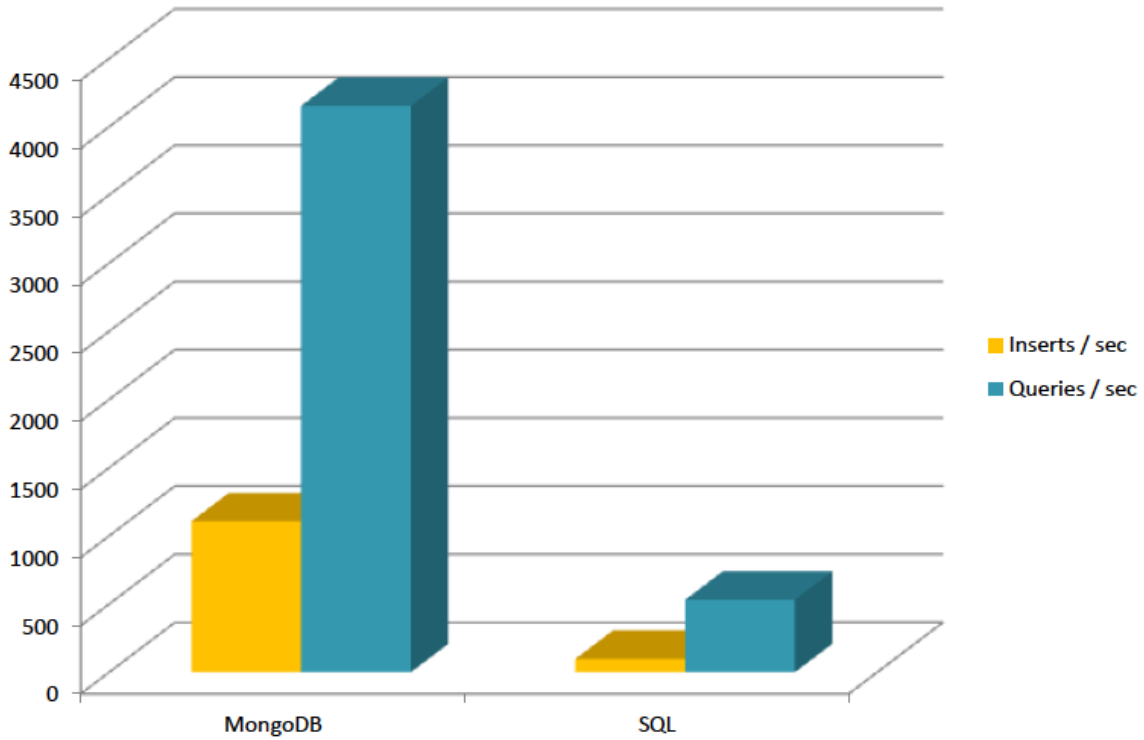
MongoDB: Goal

- **Goal:** bridge the gap between key-value stores (which are fast and scalable) and relational databases (which have rich functionality).
- Support complex, semi-structured data
- Optimize data retrieval



Is It Fast?

- **MongoDB provides high performance for semi-structured and hierarchical data** by storing related data together in documents, reducing the need for joins and enabling efficient data retrieval.



Integration with Others

- C
- C++
- Erlang
- Haskell
- Java
- Javascript
- .NET (C# F#, PowerShell, etc)
- Node.js
- Perl
- PHP
- Python
- Ruby
- Scala

Data Modeling

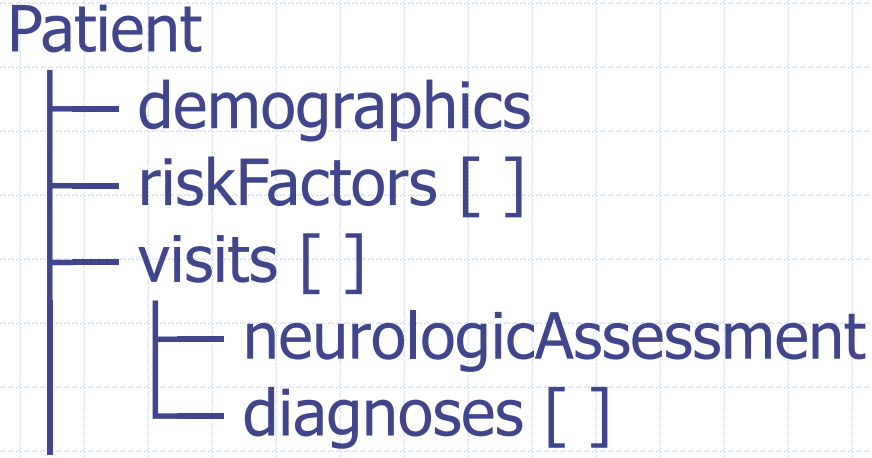
- BSON format (binary JSON)
- Developers can easily map to modern object-oriented languages without a complicated ORM layer.
- lightweight, traversable, efficient
- Collections & documents
- Embedded documents
- Arrays for one-to-many relationships
- Aggregate-oriented design

Data Dictionary – Neurological Patient

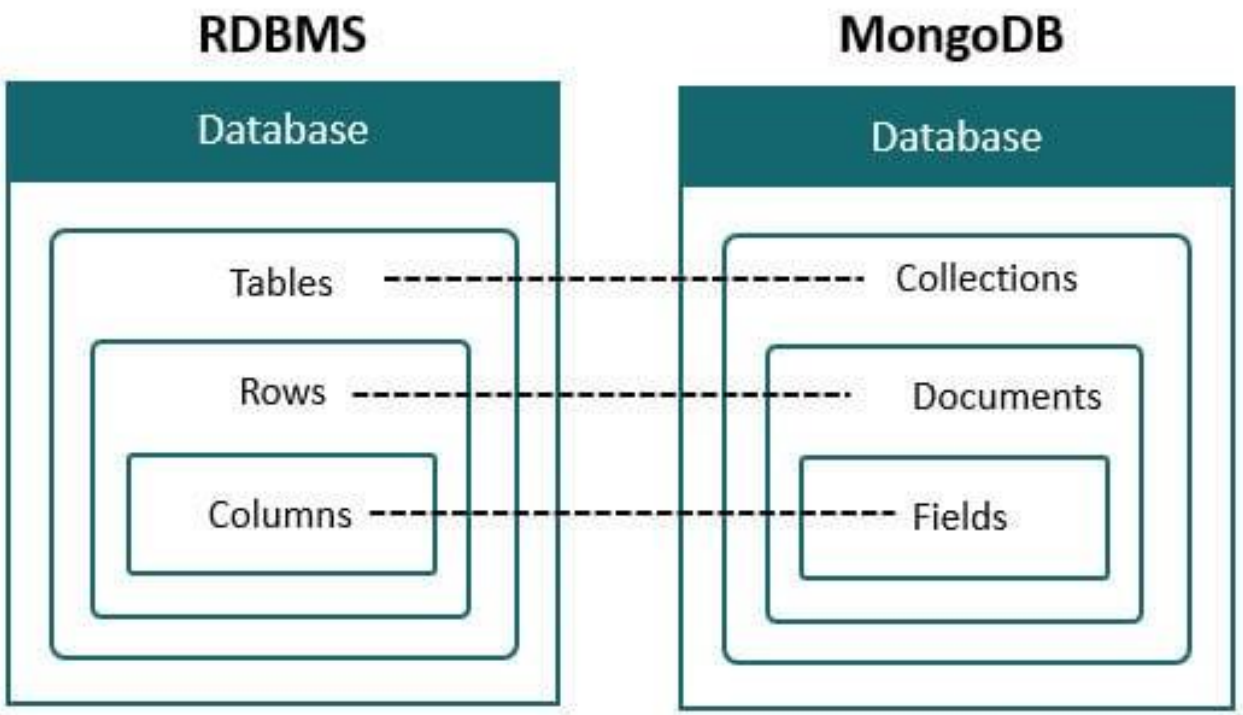
- **Patient Document**
  - patientId (String): Unique identifier of the patient
  - demographics (Object): Personal and administrative patient data
  - riskFactors (Array): Known medical risk factors
  - visits (Array): Clinical encounters over time
  - createdAt, updatedAt (Date): Metadata for document lifecycle
- **Visit Subdocument**
  - visitId (String): Unique visit identifier
  - date (Date): Visit date
  - complaints (Array): Reported symptoms
  - neurologicAssessment (Object): Neurological examination results
  - diagnoses (Array): Diagnosed neurological conditions

Database Visual Schema

- Single **Patient** document as the main aggregate
- Nested structures represent real-world hierarchy
- One-to-many relationships modeled through arrays
- No foreign keys or join tables
- Data stored together based on access patterns



Terms Mapping: DB vs. MongoDB



JSON

Field Name

Field Value

One document

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "height_cm": 167.6,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

- ***Field Value***
  - Scalar (Int, Boolean, String, Date, …)
  - Document (Embedding or Nesting)
  - Array of JSON objects

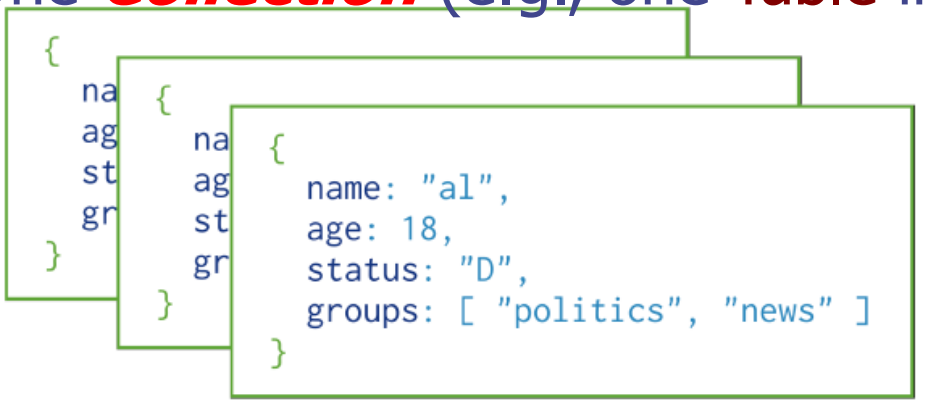
MongoDB Model

One ***document*** (e.g., one tuple in RDBMS)

```
{
  name: "sue",          <-- field: value
  age: 26,              <-- field: value
  status: "A",          <-- field: value
  groups: [ "news", "sports" ]   <-- field: value
}
```

- The field names **cannot** start with the **$** character
- The field names **cannot** contain the **.** character

One ***Collection*** (e.g., one Table in RDBMS)



Collection

- Max size of single document 16MB

```
{
  _id: ObjectId(7df78ad8902c)
  title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
  by: 'tutorials point',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100,
  comments: [
    {
      user:'user1',
      message: 'My first comment',
      dateCreated: new Date(2011,1,20,2,15),
      like: 0
    },
    {
      user:'user2',
      message: 'My second comments',
      dateCreated: new Date(2011,1,25,7,45),
      like: 5
    }
  ]
}
```

Example Document in MongoDB

- _id is a special column in each document
- Unique within each collection
- _id ←→ Primary Key in RDBMS
- The default ObjectId value is 12 bytes, but _id can be of any data type.
- Or:
  - 1st 4 bytes → timestamp
  - Next 3 bytes → machine id
  - Next 2 bytes → Process id
  - Last 3 bytes → incremental values

Design Rationale – Key Modeling Decisions

- Patient-centric document design
- Use of embedded documents for visits and diagnoses
- Avoidance of joins to optimize read performance
- Acceptance of controlled data redundancy
- Schema flexibility to support evolving clinical data
- Query-driven schema design

Data Modeling Strategies in MongoDB

- Embedded model
- Referenced model
- Hybrid model
- Choice depends on:
  - data structure
  - access patterns
  - scalability needs

Comparison of MongoDB Data Modeling Approaches

Aspect	Embedded	Referenced	Hybrid
Data locality	High	Low	Medium
Read performance	Excellent	Moderate	Good
Write scalability	Limited	High	High
Query complexity	Low	High	Medium
Document growth risk	Yes	No	Controlled

Chosen Modeling Strategy

- Embedded model selected
- Patient-centric access pattern
- Complete medical history retrieval
- Atomic updates at document level
- Hybrid model considered for large-scale scenarios

Defined Schema

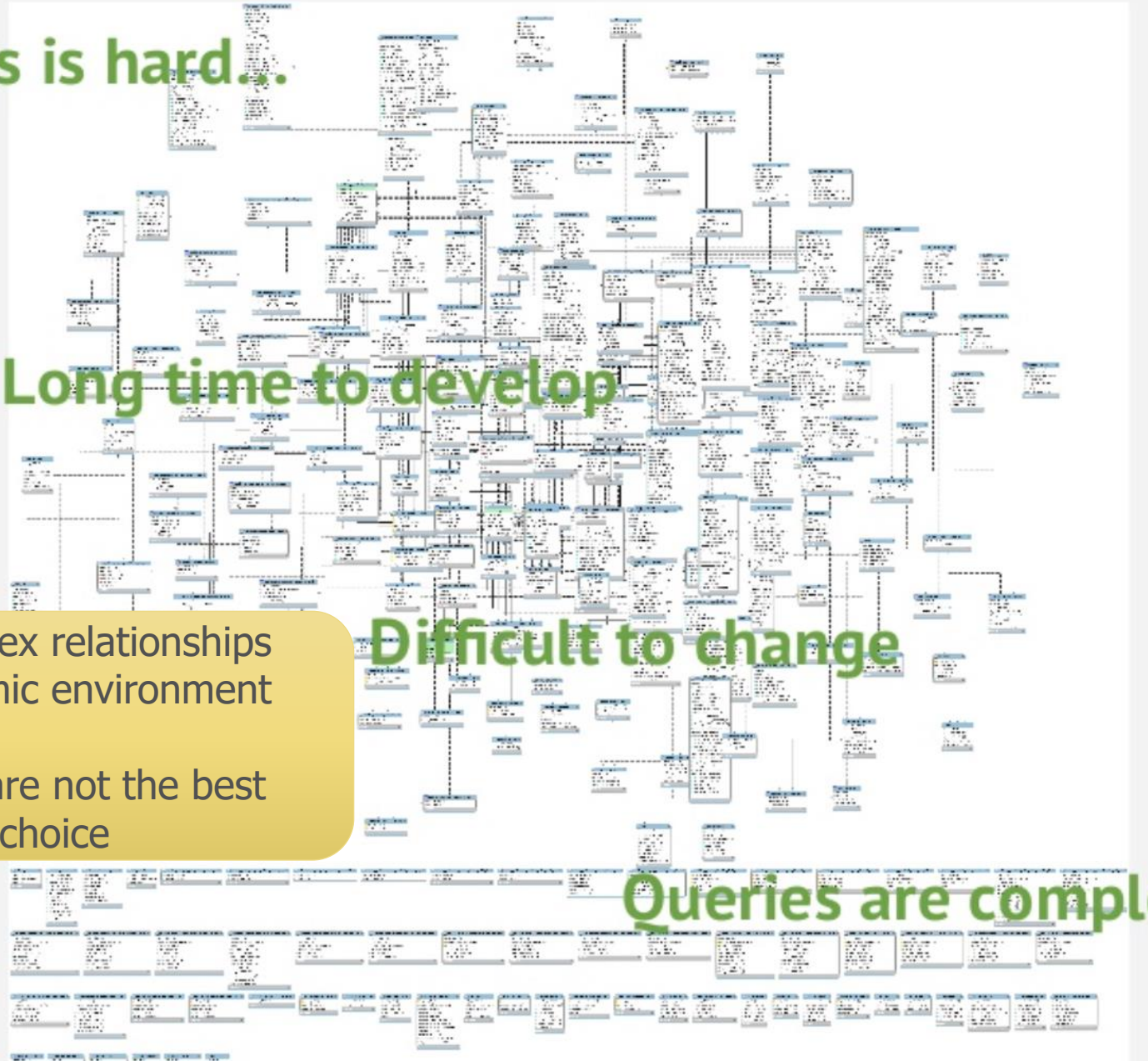
MongoDB does not require a predefined schema at the storage level.

Schema constraints can be enforced using schema validation or at the application layer.

Data Model Comparison

Relational DB vs. NoSQL

Data Model Comparison Relational DB vs. NoSQL



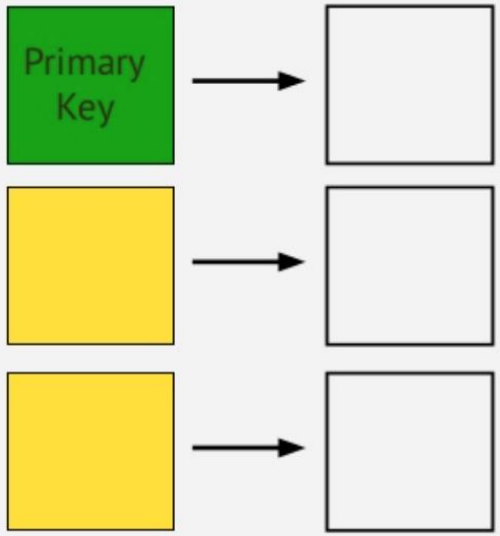
- Complex relationships
- Dynamic environment

RDBMS are not the best choice

Relational Data Model

Relational Record

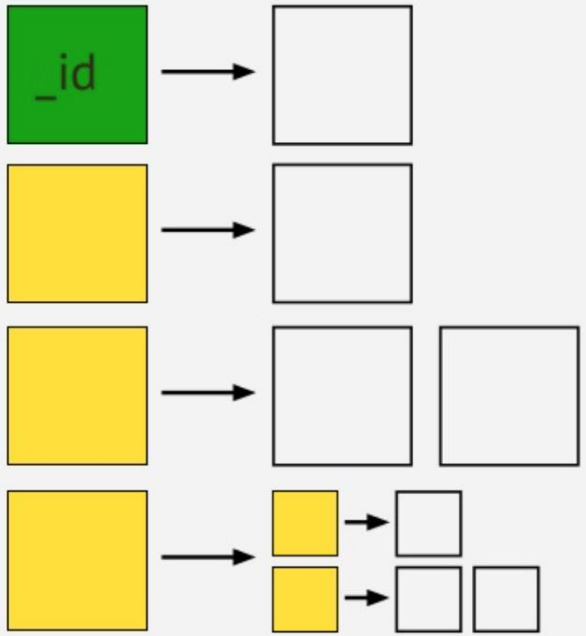
- Two-dimensional storage
- Field contains a single value
- Query on any field
- Very structured schema
- Poor data locality requires many tables, joins, and indexes.



Document Data Model

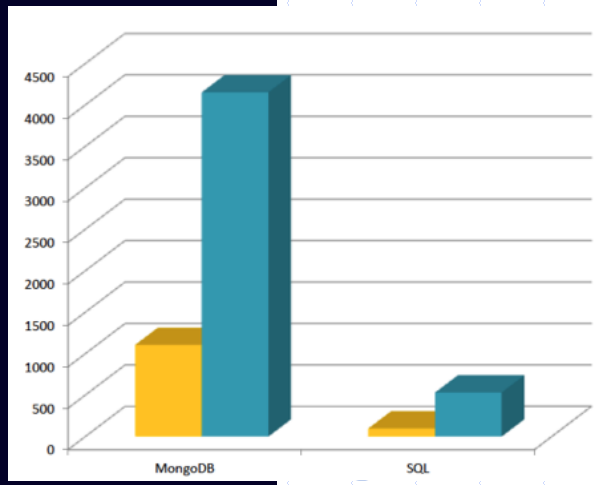
MongoDB Document

- **N-dimensional** storage
- Field can contain **many** values and **embedded** values
- Query on **any field & level**
- **Flexible** schema
- Optimal data locality requires fewer **indexes** and provides better **performance**



MongoDB does not support joins in the traditional relational sense.

Data relationships are modeled using embedded documents, references, or application-level logic.



Data Retrieval Requirements

- Retrieve complete patient history
- Filter patients by diagnosis
- Analyze neurological assessments
- Support aggregation queries

Data Maintenance Requirements

- Insert new patient documents
- Append new visits to existing patients
- Update nested clinical data atomically
- Preserve historical medical records
- Support schema evolution without migrations
- Maintain consistency at document level

MongoDB CRUD Operations

- **C**reate
  - db.collection.insertOne(<document>)
  - db.collection. insertMany(<document>)
  - db.createcollection()
- **R**ead
  - db.collection.find(<query>, <projection>)
  - db.collection.findOne(<query>, <projection>)
- **U**pdate
  - db.collection.updateOne(<query>, <update>, <options>)
  - db.collection.updateMany()
  - db.collection.replaceOne()
- **D**elete
  - db.collection.deleteOne()
  - db.collection.deleteMany()

CRUD Examples

```
db.users.insertOne(          <-- collection
   {
      name: "sue",           <-- field: value
      age: 26,               <-- field: value
      status: "pending"      <-- field: value
   }                                              } document
)
```

```
> db.student.updateOne({name: "Sumit"},{$set:{age: 24 }})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }
> db.student.find().pretty()
{
        "_id" : ObjectId("5e540cdc92e6dfa3fc48ddae"),
        "name" : "Sumit",
        "age" : 24,
        "branch" : "CSE",
        "course" : "C++ STL",
        "mode" : "online",
        "paid" : true,
        "amount" : 1499
}
```

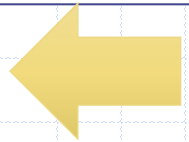
```
> db.student.find().pretty()
{
        "_id" : ObjectId("5e540cdc92e6dfa3fc48ddae"),
        "name" : "Sumit",
        "age" : 20,
        "branch" : "CSE",
        "course" : "C++ STL",
        "mode" : "online",
        "paid" : true,
        "amount" : 1499
}
```

```
> db.student.deleteOne({name: "Sumit"})
{ "acknowledged" : true, "deletedCount" : 1 }
> db.student.find().pretty()
{
        "_id" : ObjectId("5e540d3192e6dfa3fc48ddaf"),
        "name" : "Sumit",
        "age" : 20,
        "branch" : "CSE",
        "course" : "C++ STL",
        "mode" : "online",
        "paid" : true,
        "amount" : 1499,
        "year" : 2020
}
```

Multi-Document Insertion (Use of Arrays)

```
var mydocuments =
    [
      {
        item: "ABC2",
        details: { model: "14Q3", manufacturer: "M1 Corporation" },
        stock: [ { size: "M", qty: 50 } ],
        category: "clothing"
      },
      {
        item: "MNO2",
        details: { model: "14Q3", manufacturer: "ABC Company" },
        stock: [ { size: "S", qty: 5 }, { size: "M", qty: 5 }, { size: "L", qty: 1 } ],
        category: "clothing"
      },
      {
        item: "IJK2",
        details: { model: "14Q2", manufacturer: "M5 Corporation" },
        stock: [ { size: "S", qty: 5 }, { size: "L", qty: 1 } ],
        category: "houseware"
      }
    ];
```

```
db.inventory.insert( mydocuments );
```



All the documents are inserted at once

Replace a document

Query Condition

```
db.employee.replaceOne(
  { name: "Sonu" },
  { name: "Sonu", age: 25, branch: "CSE", department: "Designing" }
);
```

New doc

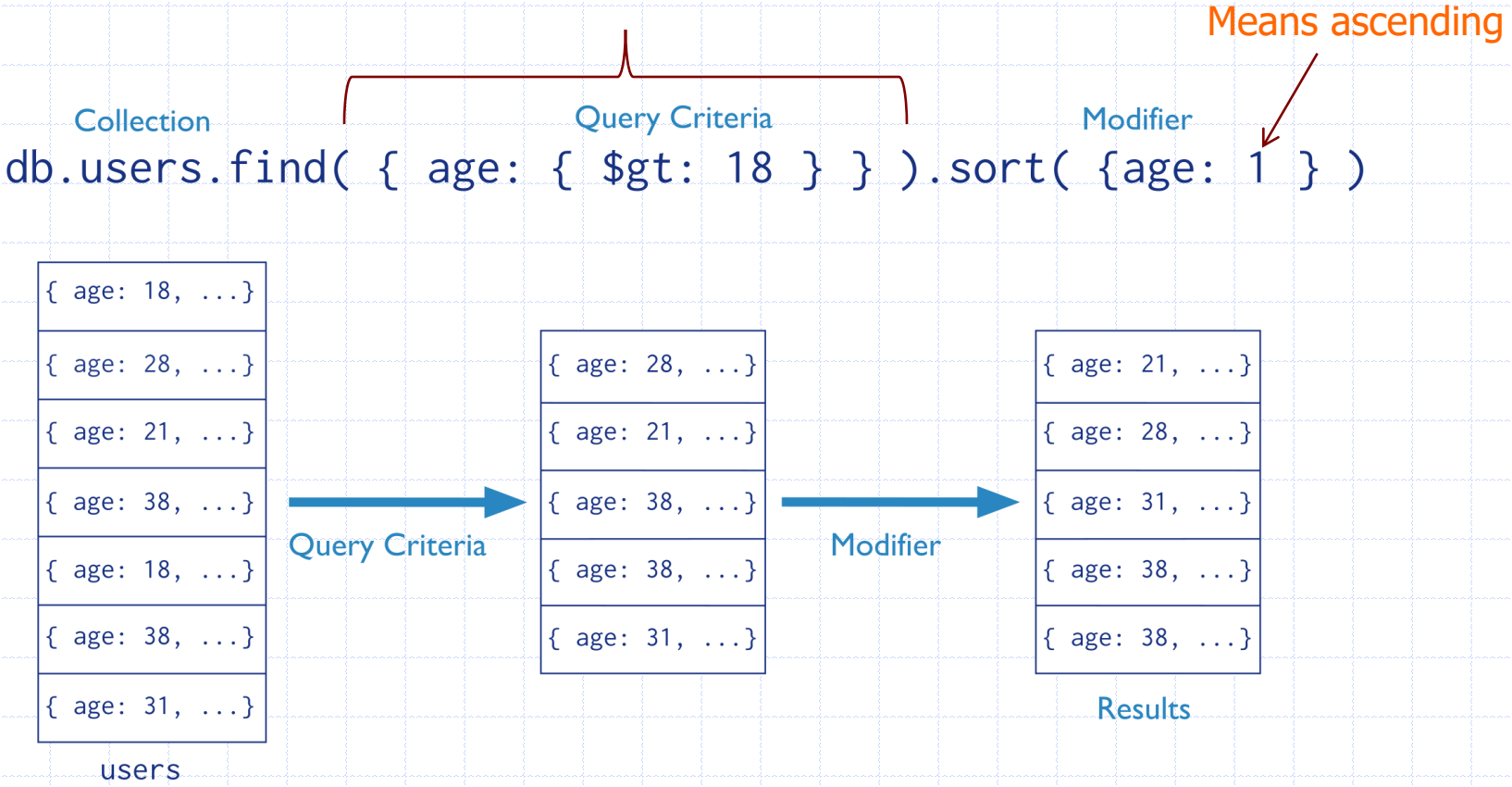
we are replacing a document of an employee whose name is Sonu.

Update

```
> db.employee.updateMany({branch: "CSE"}, {$set: {salary: 35000}})
{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }
> db.employee.find().pretty()
{
        "_id" : ObjectId("5e49177592e6dfa3fc48dd73"),
        "name" : "Sonu",
        "age" : 26,
        "branch" : "CSE",
        "department" : "HR",
        "salary" : 35000
}
{
        "_id" : ObjectId("5e49813692e6dfa3fc48dd74"),
        "name" : "Rohit",
```

we are updating the salary of those employees whose branch is CSE

Query Language in MongoDB: Find() Operator



Find() + Projection

```
db.users.find(
    { age: { $gt: 18 } },
    { name: 1, address: 1 }
).limit(5)
```

collection
query criteria
projection
cursor modifier

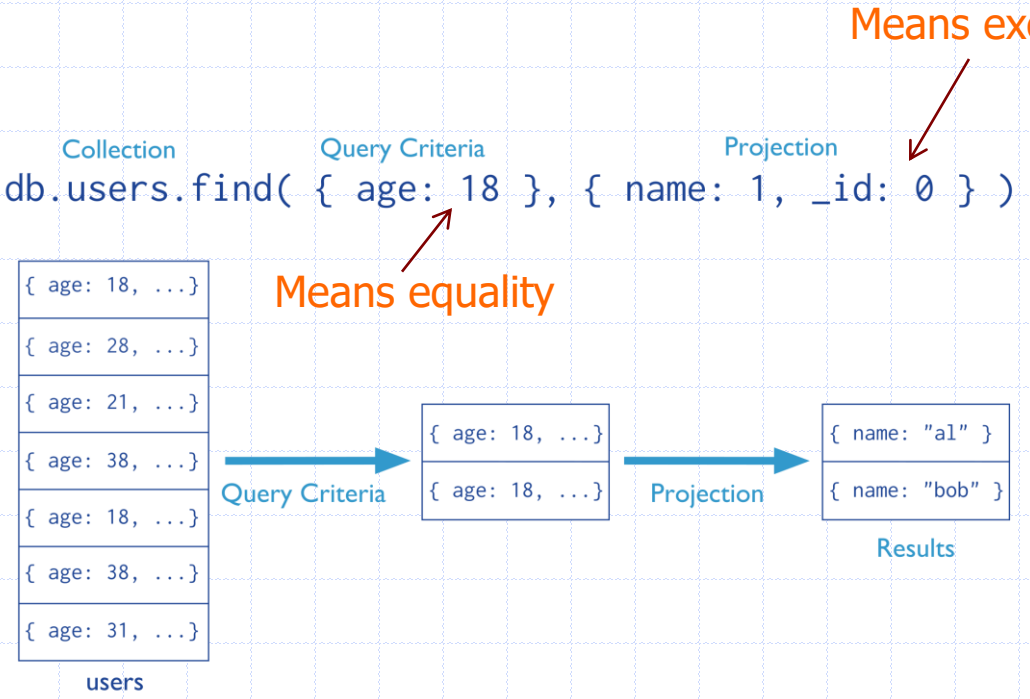
Means inclusion **+**
_id is always automatically included

Equivalent to in SQL:

```
SELECT _id, name, address
FROM   users
WHERE  age > 18
LIMIT  5
```

projection
table
select criteria
cursor modifier

Find(): Exclude Fields



Cannot mix "inclusion & exclusion" in the same operator except for ***_id***

Find() More Examples

Report all documents in the "inventory" collection

```
db.inventory.find( )
```

```
db.inventory.find( {} )
```

Equivalent to in SQL:

```
Select *
From inventory;
```

Report all documents in the "inventory" collection Where type = 'food' or 'snacks'

```
db.inventory.find(
        { type: { $in: [ 'food',
'snacks' ] } }
 )
```

Equivalent to in SQL:

```
Select *
From inventory
Where type in
        ('food', 'snacks')
```

Find(): AND & OR

AND Semantics

```
db.inventory.find( { type: 'food', price: { $lt: 9.95 } } )
```

OR Semantics

```
db.inventory.find(
    {
      $or: [ { qty: { $gt: 100 } }, { price: { $lt: 9.95 } } ]
    }
)
```

AND + OR Semantics

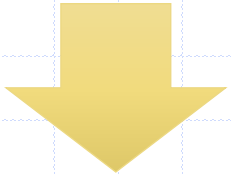
```
db.inventory.find(
    {
      type: 'food',
      $or: [ { qty: { $gt: 100 } }, { price: { $lt: 9.95 } } ]
    }
)
```

Type = 'food' and (qty > 100 or price < 9.95)

Querying Complex Types

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "height_cm": 167.6,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Documents can be complex, E.g., (Arrays, embedded documents, any nesting of these, many levels)



Queries get complex too !!!

Array Manipulation- Exact Match

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

```
db.inventory.find( { ratings: [ 5, 8, 9 ] } )
```

The operation returns the following document:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
```

Array Manipulation - Search By Element

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

```
db.inventory.find( { ratings: 5 } )
```

The operation returns the following documents:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
```

Notice: if a document has "ratings" as an Integer field = 5, it will be returned

Array Manipulation - Search By Position

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

```
db.inventory.find( { 'ratings.0': 5 } )
```

The operation returns the following documents:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
```

Notice: if a document has "ratings" as an Integer field = 5, it *will not be* returned

Embedded Object Matching (Exact doc Matching)

```
{
        name: "Joe",
        address: {
                city: "San Francisco",
                state: "CA" } ,
        likes: [ 'scuba', 'math', 'literature' ]
}
```

```
db.persons.find( { "address" : { state: "CA" }} )  //don't match

db.persons.find( { "address" : {city: "San Francisco", state: "CA" }} ) // match

db.persons.find( { "address" : {state: "CA" , city: "San Francisco"}} ) //don't match
```

Exact-match (entire object)

Embedded Object Matching (Field Matching)

```
{
        name: "Joe",
        address: {
                city: "San Francisco",
                state: "CA" } ,
        likes: [ 'scuba', 'math', 'literature' ]
}
```

Find the user documents where the address's state = 'CA'

db.persons.find({"address.state" : "CA"}) Using dot notation

Collection Modeling

- Modeling multiple collections that reference each other
- In Relational DBs ➔ FK-PK Relationships
- In MongoDB, two options
  - Referencing
  - Embedding

FK-PK in Relational DBs

- Create “Students” relation

```
CREATE TABLE Students
        (sid CHAR(20),
         name CHAR(20),
         login CHAR(10),
         age INTEGER,
         gpa REAL);
```

- Create “Courses” relation

```
CREATE TABLE Courses
        (cid Varchar2(20),
         name varchar2(50),
         maxCredits integer,
         graduateFlag char(1));
```

Foreign key

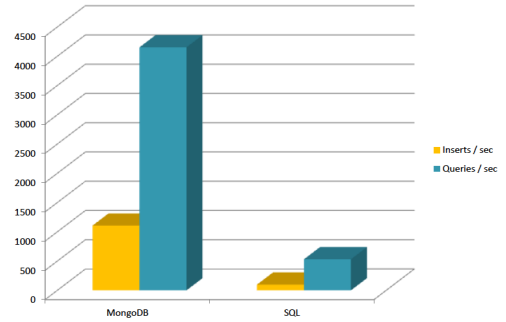
Foreign key

- Create “Enrolled” relation

```
CREATE TABLE Enrolled
        (sid CHAR(20),
         cid Varchar2(20),
         enrollDate date,
         grade CHAR(2));
```

- **Each tuple in “Enrolled” reference a specific student and a specific course**

In MongoDB

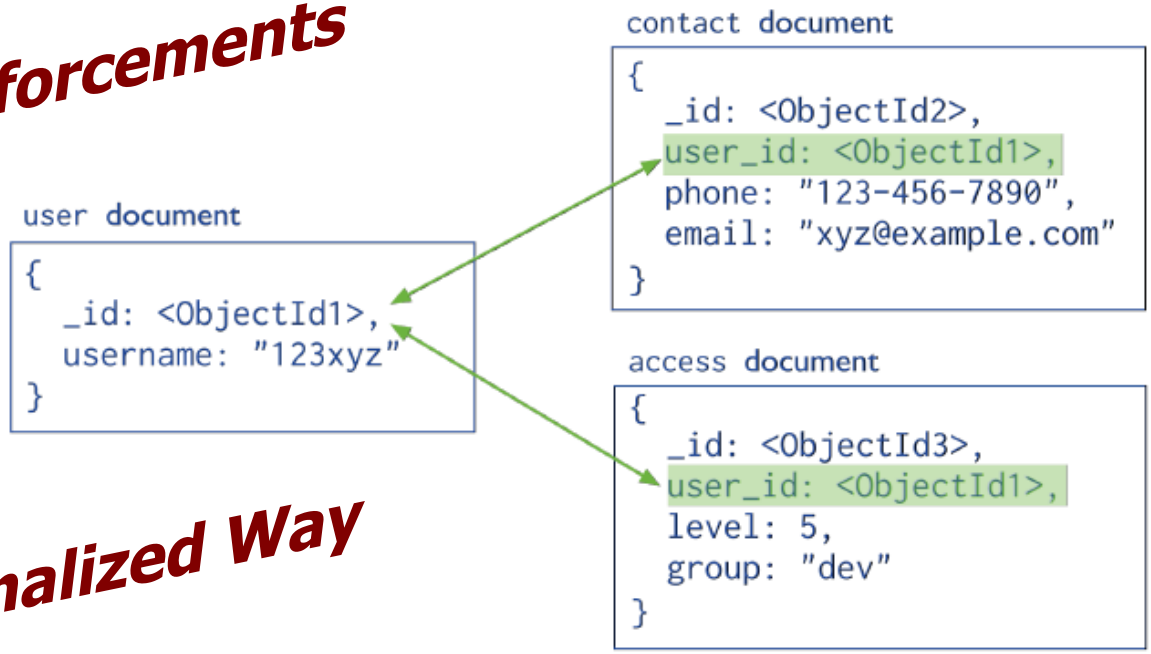


- ***Referencing*** between two collections
  - Use Id of one and put in the other
  - Very similar to FK-PK in Relational DBs
  - **Does not come with enforcement mechanism**
- ***Embedding*** between two collections
  - Put the document from one collection inside the other one

Hybrid Model

Referencing

No Enforcements



Normalized Way

- Have three collections in the DB: “User”, “Contact”, “Access”
- Link them by _id (or any other field(s))

Embedding

De-Normalized Way

- Have one collection in DB: "User"
- The others are embedded inside each user's document

Examples (1)

- "Patron" & "Addresses"

```
{
    _id: "joe",
    name: "Joe Bookreader"
}
```

```
{
    patron_id: "joe",
    street: "123 Fake Street",
    city: "Faketon",
    state: "MA",
    zip: "12345"
}
```

Referencing

- If it is 1-1 relationship
- If usually read the address with the name
- If address document usually does not expand

If most of these hold
➔ better use Embedding

Examples (2)

- "Patron" & "Addresses"

```
{
   _id: "joe",
   name: "Joe Bookreader",
   address: {
              street: "123 Fake Street",
              city: "Faketon",
              state: "MA",
              zip: "12345"
            }
}
```

Embedding

- When you read, you get the entire document at once
- In Referencing ➔ Need to issue multiple queries

Examples (3)

- What if a "Patron" can have many "Addresses"

```
{
    _id: "joe",
    name: "Joe Bookreader"
}
```

```
{
    patron_id: "joe",
    street: "123 Fake Street",
    city: "Faketon",
    state: "MA".
    zip: "12345"
}
```

Referencing

- Do you read them together ➔ Go for Embedding
- Are addresses dynamic (e.g., add new ones frequently)

➔ Go for Referencing

Examples (4)

- What if a "Patron" can have many "Addresses"

```
{
   _id: "joe",
   name: "Joe Bookreader",
   addresses: [
                {
                  street: "123 Fake Street",
                  city: "Faketon",
                  state: "MA",
                  zip: "12345"
                },
                {
                  street: "1 Some Other Street",
                  city: "Boston",
                  state: "MA",
                  zip: "12345"
                }
              ]
}
```

Embedding

Use array of addresses

Examples (5)

- If addresses are added frequently …

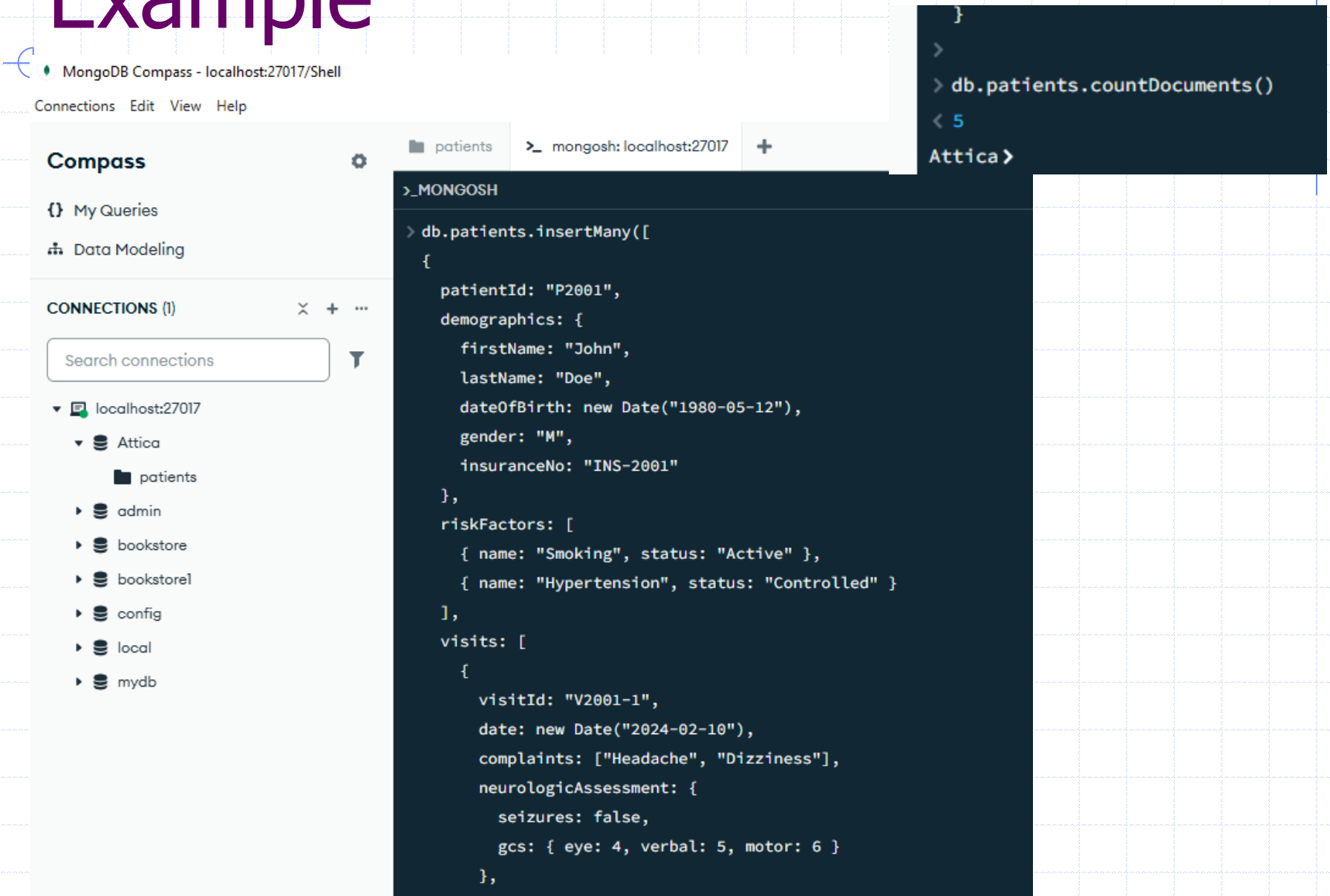
```
{
    _id: "joe",
    name: "Joe Bookreader",
    addresses: [
                  {
                    street: "123 Fake Street",
                    city: "Faketon",
                    state: "MA",
                    zip: "12345"
                  },
                  {
                    street: "1 Some Other Street",
                    city: "Boston",
                    state: "MA",
                    zip: "12345"
                  }
                ]
}
```

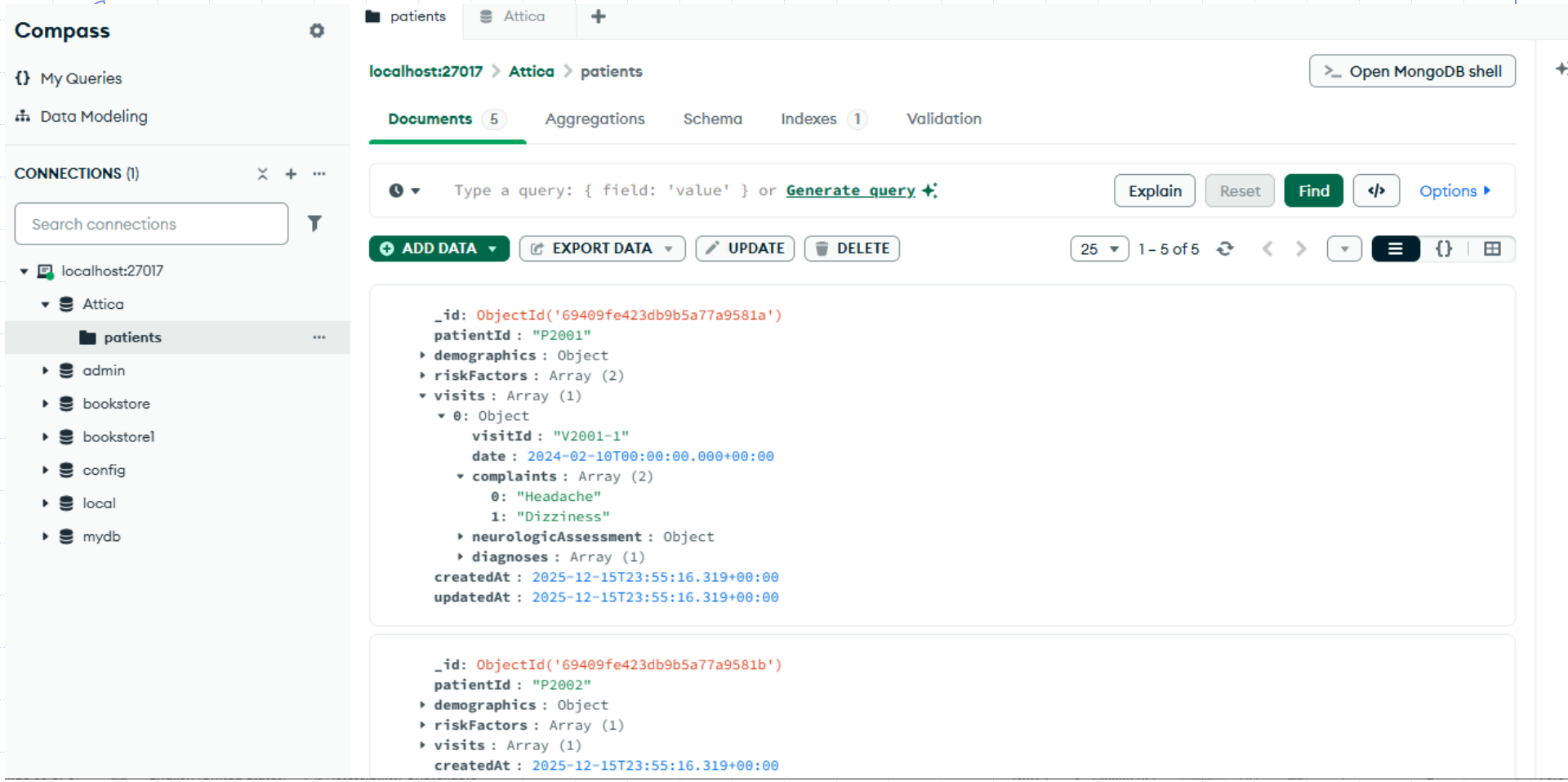
This array will expand frequently

Size of "Patron" document increases frequently

May trigger re-locating the document each time *(Bad)*

Example


MongoDB Compass - localhost:27017/Shell
Connections Edit View Help
Compass
My Queries
Data Modeling
CONNECTIONS (1)
Search connections
localhost:27017
Attica
patients
admin
bookstore
bookstore1
config
local
mydb
patients
mongosh: localhost:27017
>_MONGOSH
> db.patients.insertMany([
{
patientId: "P2001",
demographics: {
firstName: "John",
lastName: "Doe",
dateOfBirth: new Date("1980-05-12"),
gender: "M",
insuranceNo: "INS-2001"
},
riskFactors: [
{ name: "Smoking", status: "Active" },
{ name: "Hypertension", status: "Controlled" }
],
visits: [
{
visitId: "V2001-1",
date: new Date("2024-02-10"),
complaints: ["Headache", "Dizziness"],
neurologicAssessment: {
seizures: false,
gcs: { eye: 4, verbal: 5, motor: 6 }
},
}
> db.patients.countDocuments()
< 5
Attica>


Compass
My Queries
Data Modeling
CONNECTIONS (1)
Search connections
localhost:27017
Attica
patients
admin
bookstore
bookstore1
config
local
mydb
patients
Attica
localhost:27017 > Attica > patients
Open MongoDB shell
Documents 5
Aggregations
Schema
Indexes 1
Validation
Type a query: { field: 'value' } or Generate query
Explain
Reset
Find
Options
ADD DATA
EXPORT DATA
UPDATE
DELETE
25
1 – 5 of 5
_id: ObjectId('69409fe423db9b5a77a9581a')
patientId : "P2001"
demographics : Object
riskFactors : Array (2)
visits : Array (1)
0: Object
visitId : "V2001-1"
date : 2024-02-10T00:00:00.000+00:00
complaints : Array (2)
0: "Headache"
1: "Dizziness"
neurologicAssessment : Object
diagnoses : Array (1)
createdAt : 2025-12-15T23:55:16.319+00:00
updatedAt : 2025-12-15T23:55:16.319+00:00
_id: ObjectId('69409fe423db9b5a77a9581b')
patientId : "P2002"
demographics : Object
riskFactors : Array (1)
visits : Array (1)
createdAt : 2025-12-15T23:55:16.319+00:00

Queries

```
// All stroke-related patients
db.patients.find({ "visits.diagnoses.code": { $in: ["I63", "I64"] } })

// Patients with seizures
db.patients.find({ "visits.neurologicAssessment.seizures": true })

// Patient timeline
db.patients.find(
  { patientId: "P2005" },
  { demographics: 1, visits: 1 }
)

// Patients with Smoking risk factor
db.patients.find({ "riskFactors.name": "Smoking" })
```

Schema Validation

- JSON Schema validation
- Required fields enforcement
- Improves data quality
- Prevents malformed documents

```
>_MONGOSH

< 10
> db.runCommand({
    collMod: "patients",
    validator: {
      $jsonSchema: {
        bsonType: "object",
        required: ["patientId", "demographics", "visits", "createdAt", "updatedAt"],
        additionalProperties: true,
        properties: {
          patientId: { bsonType: "string", description: "Unique patient identifier" },

          demographics: {
            bsonType: "object",
            required: ["firstName", "lastName", "dateOfBirth", "gender", "insuranceNo"],
            properties: {
              firstName: { bsonType: "string" },
              lastName: { bsonType: "string" },
              dateOfBirth: { bsonType: "date" },
              gender: { enum: ["M", "F", "O"] },
              insuranceNo: { bsonType: "string" }
```

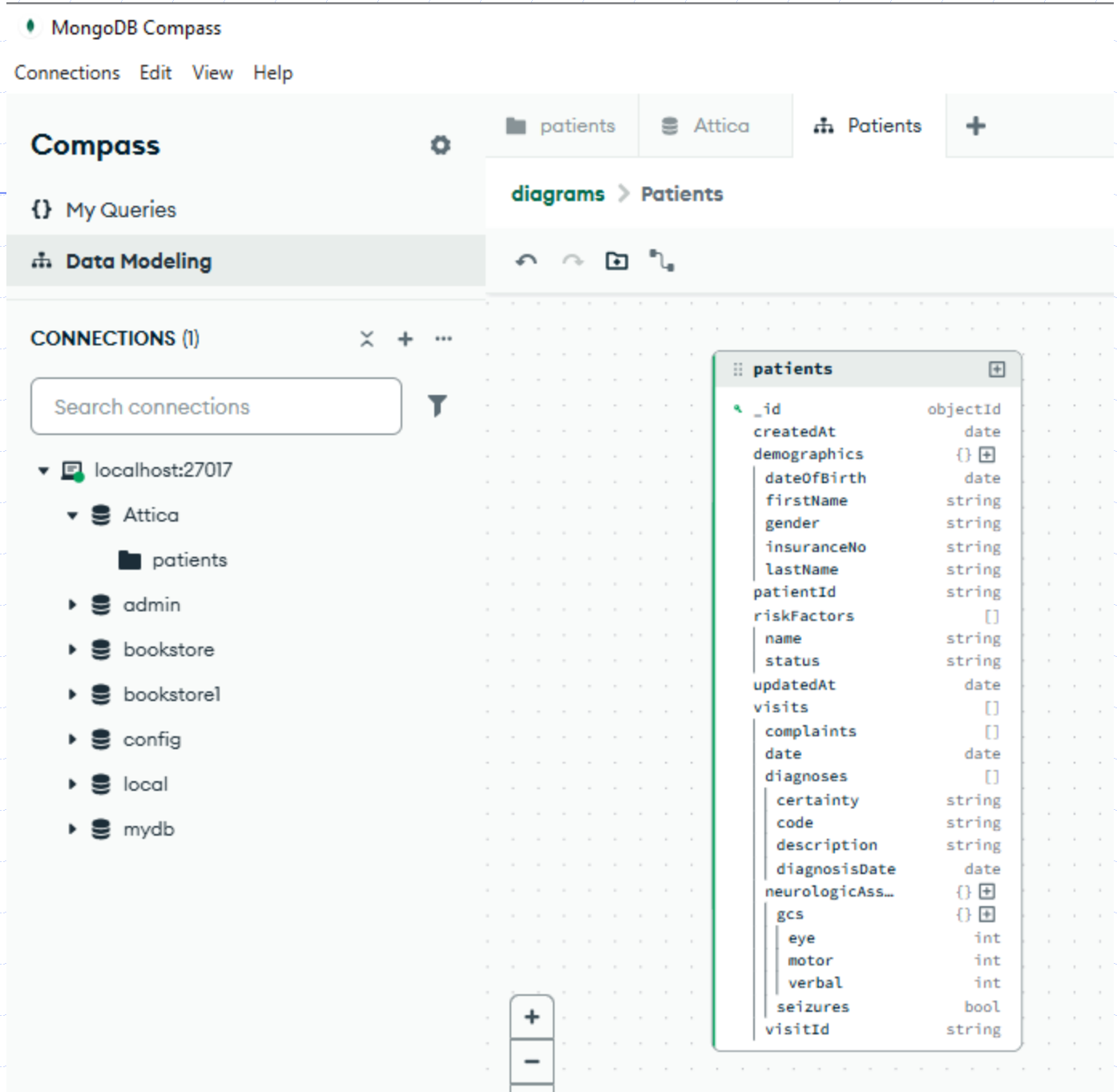
index

- db.patients.createIndex({ patientId: 1 }, { unique: true })
- db.patients.createIndex({ "demographics.insuranceNo": 1 }, { unique: true })
- db.patients.createIndex({ "visits.diagnoses.code": 1 })
- db.patients.createIndex({ "riskFactors.name": 1 })

Test that validation works

- This insert has gender: "X" (invalid) → should be rejected:

```
> db.patients.insertOne({
    patientId: "P9999",
    demographics: {
      firstName: "Test",
      lastName: "Invalid",
      dateOfBirth: new Date("1990-01-01"),
      gender: "X",
      insuranceNo: "INS-9999"
    },
    visits: [],
    createdAt: new Date(),
    updatedAt: new Date()
  })
✖ ▸ MongoServerError: Document failed validation
```


MongoDB Compass
Connections Edit View Help
Compass
My Queries
Data Modeling
CONNECTIONS (1)
Search connections
localhost:27017
Attica
patients
admin
bookstore
bookstore1
config
local
mydb
patients
Attica
Patients
diagrams > Patients
patients
_id objectId
createdAt date
demographics {}
dateOfBirth date
firstName string
gender string
insuranceNo string
lastName string
patientId string
riskFactors []
name string
status string
updatedAt date
visits []
complaints []
date date
diagnoses []
certainty string
code string
description string
diagnosisDate date
neurologicAss... {}
gcs {}
eye int
motor int
verbal int
seizures bool
visitId string

Crud operation

- Insert new patient

```
db.patients.insertOne({
  patientId: "P3001",
  demographics: {
    firstName: "Erion",
    lastName: "Basha",
    dateOfBirth: new Date("1990-04-18"),
    gender: "M",
    insuranceNo: "INS-3001"
  },
  riskFactors: [{ name: "Smoking", status: "Active" }],
  visits: [],
  createdAt: new Date(),
  updatedAt: new Date()  })
```

Add a new visit for an existing patient

```
db.patients.updateOne(
  { patientId: "P2001" },
  {
    $push: {
      visits: {
        visitId: "V2001-2",
        date: new Date("2024-10-01"),
        complaints: ["Memory loss"],
        neurologicAssessment: {
          seizures: false,
          gcs: { eye: 4, verbal: 5, motor: 6 }
        },
        diagnoses: []
      }
    },
    $set: { updatedAt: new Date() }
  } )
```

- db.patients.find()
- db.patients.findOne({ patientId: "P2001" })
- db.patients.find({ "visits.diagnoses.code": { $in: ["I63", "I64"] } }, { patientId: 1, demographics: 1 })
- db.patients.find({}, { _id: 0, patientId: 1, demographics: 1 })
- db.patients.updateOne({ patientId: "P2002" }, { $set: { "demographics.insuranceNo": "INS-UPDATED-2002" } })

Add a new diagnosis to an existing visit

```
db.patients.updateOne(
  { patientId: "P2001", "visits.visitId": "V2001-1" },
  {
    $push: {
      "visits.$.diagnoses": {
        code: "G45",
        description: "Transient Ischemic Attack",
        certainty: "suspected",
        diagnosisDate: new Date()
      }
    },
    $set: { updatedAt: new Date() }
  }
)
```

Delete the patient

- db.patients.deleteOne({ patientId: "P3001" })

Delete a diagnosis from a visit (nested delete)

```
db.patients.updateOne(
  { patientId: "P2001",
"visits.visitId": "V2001-1" },
  {
    $pull: {
      "visits.$.diagnoses": { code:
"G45" }
    }
  }
)
```

MongoDB Aggregation Commands

- **Aggregation** in MongoDB is used to:
  - analyze data
  - transform documents
  - compute summary statistics
  - generate derived results
- Unlike find(), aggregation:
  - processes data through a **pipeline**
  - applies transformations **stage by stage**
  - returns **computed and reshaped results**

Filtering (SQL WHERE)

```
db.patients.aggregate([
  { $match: { "visits.diagnoses.code": "I63" } }
])
```

Aggregation and Grouping (SQL GROUP BY)

```
{
  $group: {
    _id: "$visits.diagnoses.code“,
    totalOccurrences: { $sum: 1 }
  }
}
```

Aggregation vs SQL

SQL	MongoDB Aggregation
SELECT	`$project`
WHERE	`$match`
GROUP BY	`$group`
HAVING	`$match` (after `$group`)
ORDER BY	`$sort`
COUNT()	`$count`

Conclusion

- A complete non relational IS design was presented
- Supports both retrieval and maintenance
- This presentation demonstrated the design of a non-relational Information System based on MongoDB, highlighting how document-oriented databases effectively support complex, hierarchical data structures and query-driven access patterns in modern information systems.

Questions

- **Thank you for your attention**
- Questions?