
mongoDB

Universidad Nacional Autónoma de México

UNAM

RED UNIVERSITARIA DE COLABORACIÓN
EN INGENIERÍA DE SOFTWARE Y BASES DE DATOS

RED IS&BD

Thinking in Documents

Alejandro Mancilla
Sr. Solutions Architect, LATAM
alejandro.mancilla@mongodb.com
@alxmancilla

Agenda

- What is NoSQL?
- Flexible Data Model
- MongoDB Schema Design

What is NoSQL?

What is NoSQL?

- NoSQL encompasses a wide variety of different database technologies that were developed in response to the demands presented in building modern applications:
- New modern applications demand:.
  - Massive volumes of new, rapidly changing data
  - Structured, semi-structured, unstructured and polymorphic data types
  - Agile development methodologies
  - High Availability
  - High Scalability

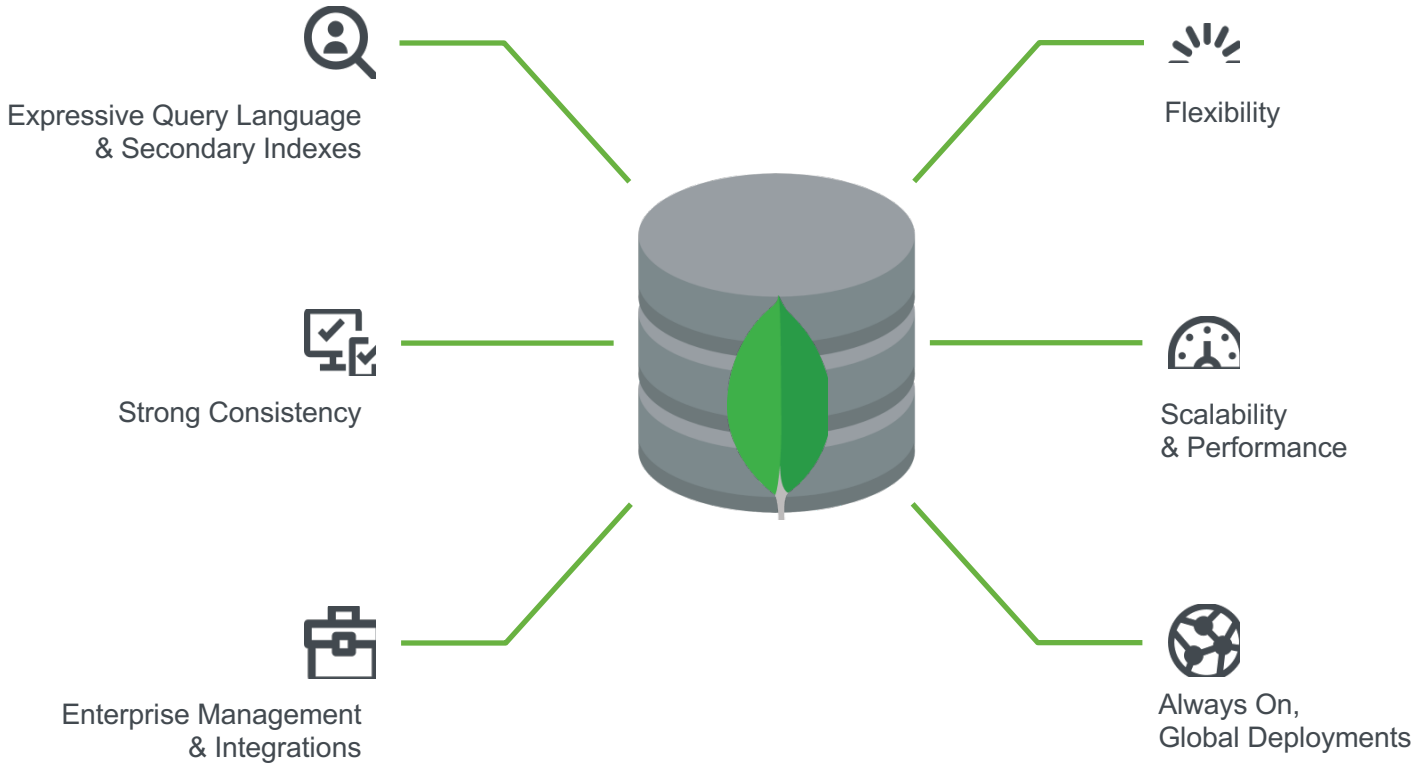
NoSQL Database Types

- **Key-Value store**
  - The simplest NoSQL databases (key : value)
  - Riak & Berkeley DB
- **Graph store**
  - Used to store information about networks of data, such as social connections
  - Neo4J & Giraph
- **Wide-column store**
  - Queries over large datasets, and store columns of data together, instead of rows
  - Ejemplos: Cassandra y HBase
- **Document store**
  - Pair each key with a complex data structure known as a document
  - MongoDB y Couchbase

The Benefits of NoSQL

- When compared to relational databases, NoSQL databases are more scalable and provide superior performance
- Their data model addresses several issues that the relational model is not designed to address:
  - Large volumes of rapidly changing structured, semi-structured, and unstructured data
  - Agile sprints, quick schema iteration, and frequent code pushes
  - Object-oriented programming that is easy to use and flexible
  - Geographically distributed scale-out architecture instead of expensive, monolithic architecture

MongoDB - Nexus Architecture



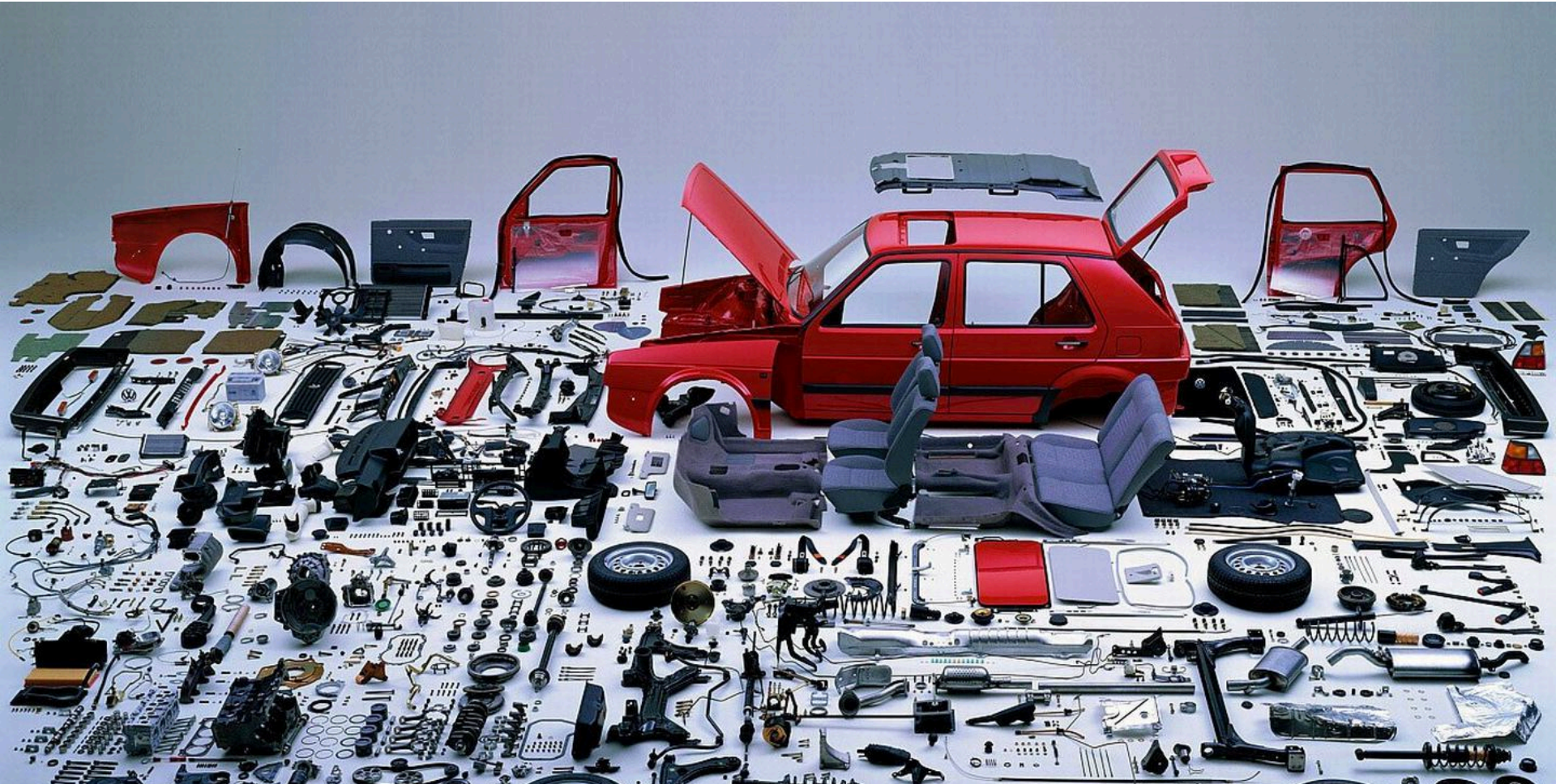
Flexible Data Model

Object: Car

Properties:

- Make
- Model
- Year
- Doors (count)
- Color
- Wheels
- Tires
- Options:
  - Engine:
    - Cylinders (count)
    - Valves (count)
    - CI/Liters
  - Transmission
  - Exhaust
- …

Impedance Mismatch



Terminology

RDBMS		MongoDB
Database	→	Database
Table	→	Collection
Index	→	Index
Row	→	Document
Column	→	Field
Join	→	Embedding & Linking

Document Model

RDBMS

PERSON

Pers_ID	Surname	First_Name	City
0	Miller	Paul	London
1	Ortega	Alvaro	Valencia
2	Huber	Urs	Zurich
3	Blanc	Gaston	Paris
4	Bertolini	Fabrizio	Rome

NO RELATION

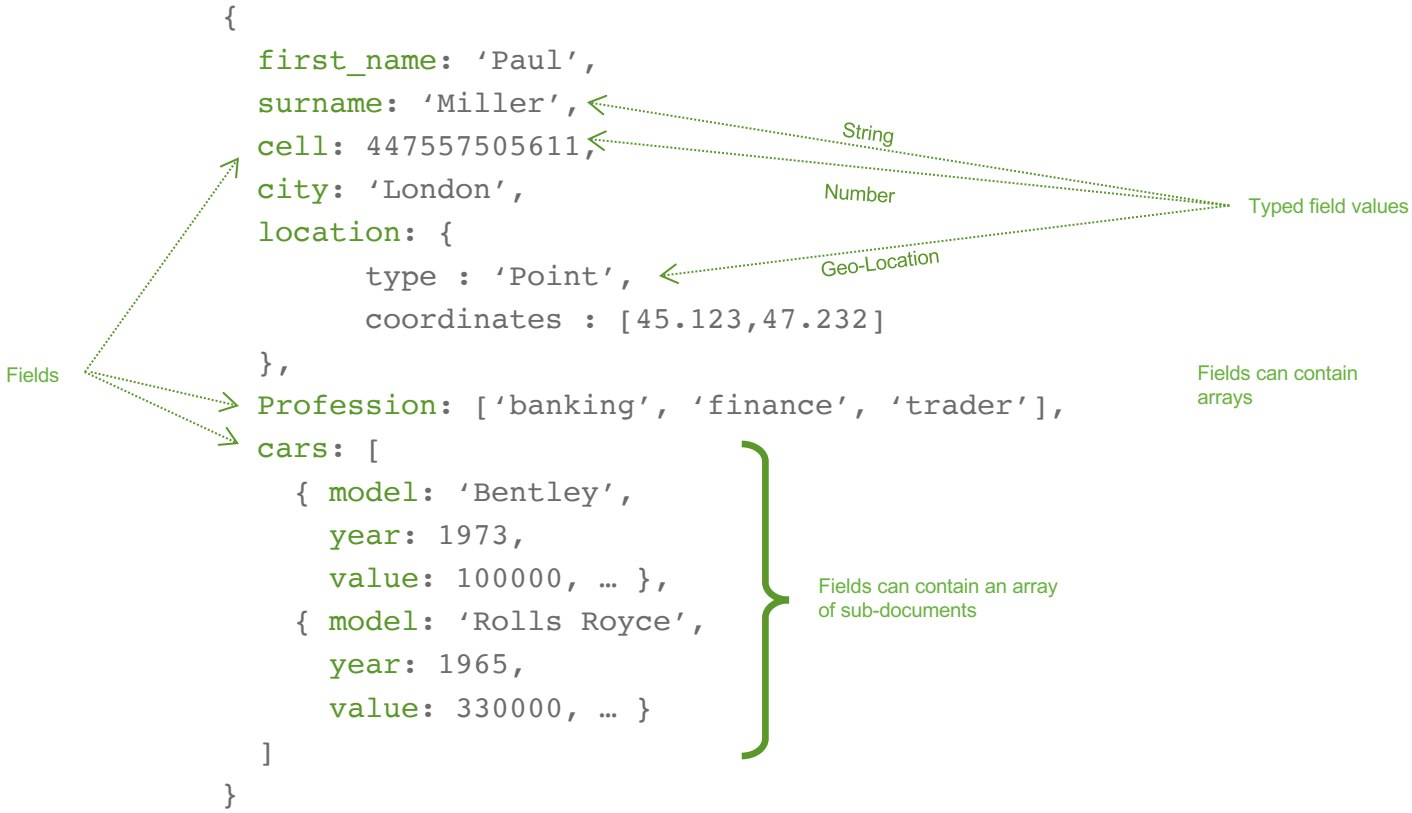
CAR

Car_ID	Model	Year	Value	Pers_ID
101	Bently	1973	100000	0
102	Rolls Royce	1965	330000	0
103	Peugeot	1993	500	3
104	Ferrari	2005	150000	4
105	Renault	1998	2000	3
106	Renault	2001	7000	3
107	Smart	1999	2000	2

MongoDB

```
{
  first_name: ‘Paul’,
  surname: ‘Miller’,
  city: ‘London’,
  location: {
        type : ‘Point’,
        coordinates : [45.123,47.232]
  },
  cars: [
    { model: ‘Bentley’,
      year: 1973,
      value: 100000, … },
    { model: ‘Rolls Royce’,
      year: 1965,
      value: 330000, … }
  ]
}
```

Documents are Rich Data Structures



MongoDB really speaks JSON

- From the very first version it was a native JSON database
- Understands and can index the sub-structures
- Stores JSON as an serialized binary format called BSON
- Efficient for encoding and decoding for network transmission
- MongoDB can create indexes on any document field

Documents are Flexible

Documents in the same product catalog collection in MongoDB

```
{
  product_name: ‘Acme Paint’,
  color: [‘Red’, ‘Green’],
  size_oz: [8, 32],
  finish: [‘satin’, ‘eggshell’]
}
```

```
{
  product_name: ‘T-shirt’,
  size: [‘S’, ‘M’, ‘L’, ‘XL’],
  color: [‘Heather Gray’ … ],
  material: ‘100% cotton’,
  wash: ‘cold’,
  dry: ‘tumble dry low’
}
```

```
{
  product_name: ‘Mountain Bike’,
  brake_style: ‘mechanical disc’,
  color: ‘grey’,
  frame_material: ‘aluminum’,
  no_speeds: 21,
  package_height: ‘7.5x32.9x55’,
  weight_lbs: 44.05,
  suspension_type: ‘dual’,
  wheel_size_in: 26
}
```

Drivers & Frameworks

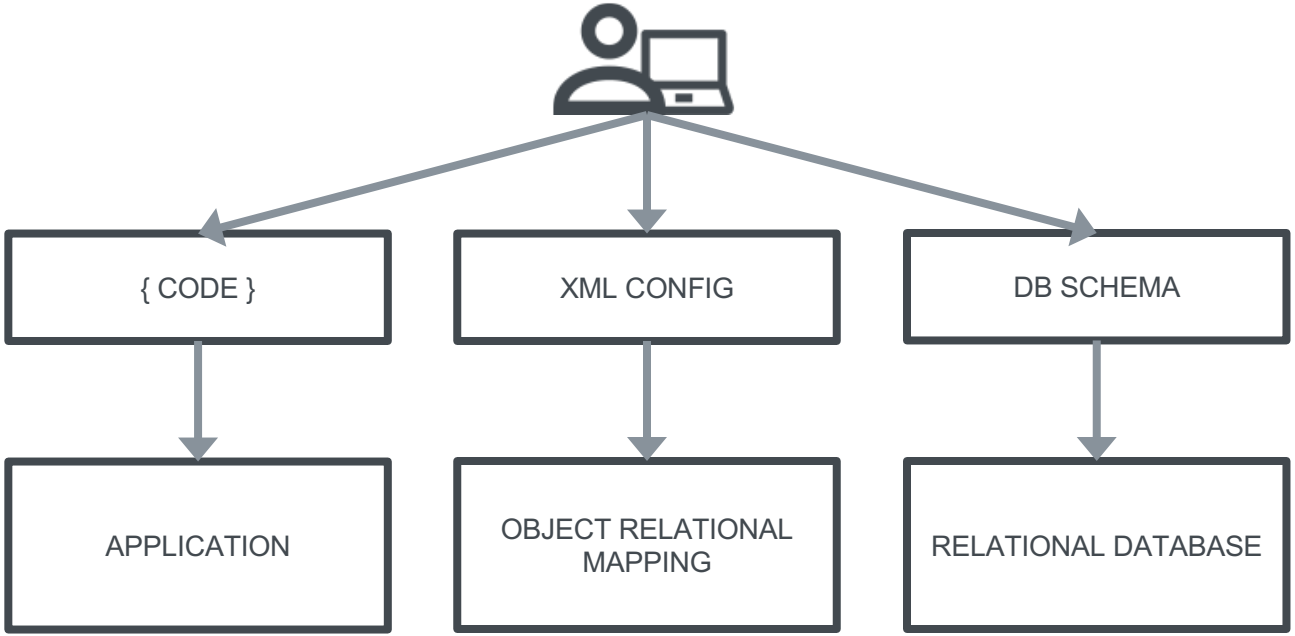




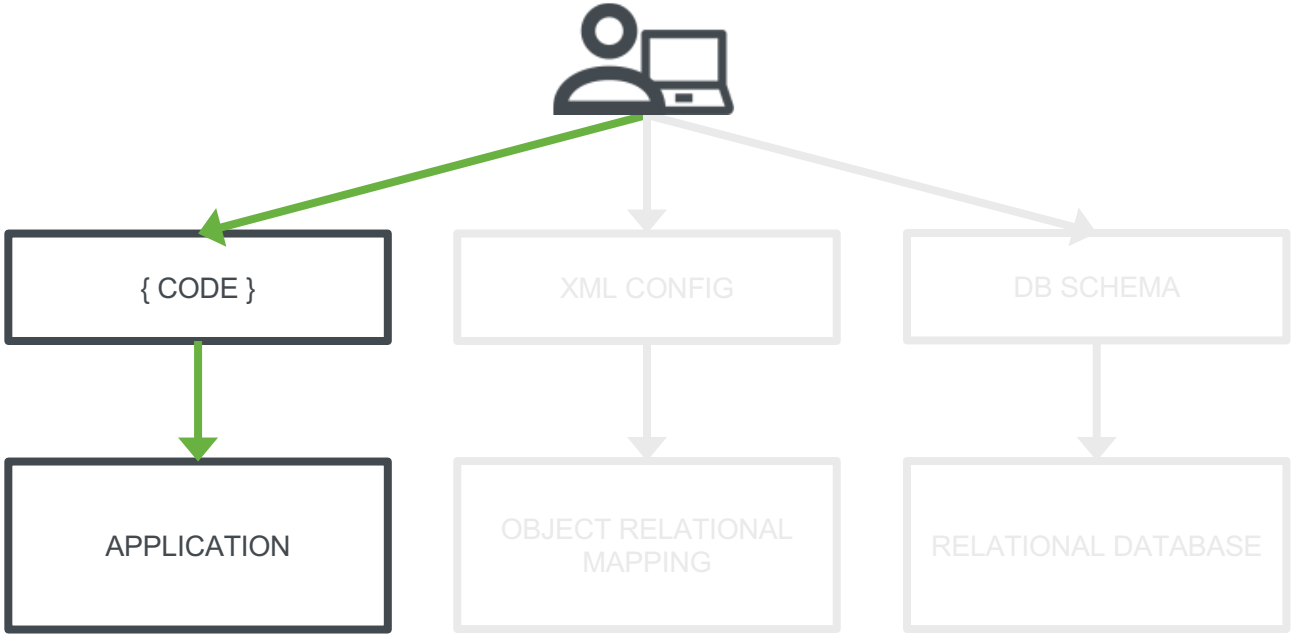
MEAN Stack

Morphia

Development – The Past



Development – With MongoDB



MongoDB 3.4 - Multi-Model

Document
Rich JSON Data Structures
Flexible Schema
Global Scale

Graph
Graph & Hierarchical
Recursive Lookups

Relational
Left-Outer Join
Views
Schema Validation

Search
Text Search
Multiple Languages
Faceted Search

Key/Value
Horizontal Scale
In-Memory

GeoSpatial
GeoJSON
2D & 2DSphere

Binaries
Files & Metadata
Encrypted

MongoDB Schema Design

Theme #1:
Great Data Design involves much more than the database

Theme #2:
Today's solutions need to accommodate tomorrow's needs

Theme #3:
MongoDB offers you
choice

Terminology

RDBMS		MongoDB
Database	→	Database
Table	→	Collection
Index	→	Index
Row	→	Document
Column	→	Field
Join	→	Embedding & Linking

What is a Document?

```
{
     _id: “123”,
     title: "MongoDB: The Definitive Guide",
     authors: [
          { _id: "kchodorow", name: "Kristina
Chodorow“ },
          { _id: "mdirold", name: “Mike Dirolf“ }
     ],
     published_date: ISODate(”2010-09-24”),
     pages: 216,
     language: "English",
     thumbnail: BinData(0,"AREhMQ=="),
     publisher: {
          name: "O’Reilly Media",
          founded: 1980,
          locations: ["CA”, ”NY” ]
     }
}
```

Documents Map to Language Constructs

```
// Java: maps
DBObject query = new
BasicDBObject("publisher.founded", 1980));
Map m = collection.findOne(query);
Date pubDate = (Date)m.get("published_date"); //
java.util.Date


// Javascript: objects
m = collection.findOne({"publisher.founded" :
1980});
pubDate = m.published_date;    // ISODate
year = pubDate.getUTCFullYear();


# Python: dictionaries
m = coll.find_one({"publisher.founded" : 1980 });
pubDate = m["pubDate"].year   # datetime.datetime
```

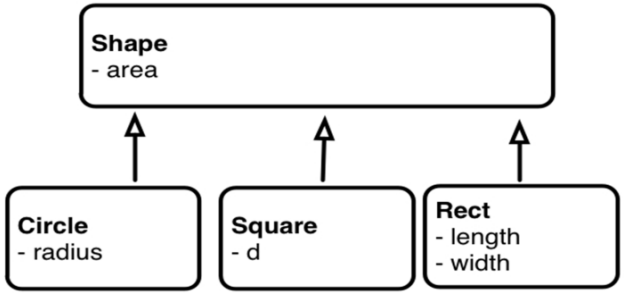
Traditional Data Design

- Static, Uniform Scalar Data
- Rectangles
- Low-level, physical representation

Document Data Design

- Flexible, Rich Shapes
- Objects
- Higher-level, business representation

Thing are not always relational



id	type	area	radius	d	length	width
1	circle	3.14	1	2		
2	square	4			2	
3	rect	10			5	2

- Single Table Inheritance
  - Ver imagen
  - Un “kludge”
- Class Table Inheritance
  - Pesadilla de consultas
  - SQL UNIONs complejos
- Blobs
  - Can’t query into
  - No indexing of secondary attributes

When to use MongoDB vs. RDBMS

Datos

- ❑ Variables o semi-estructurados
- ❑ Jerarquías
- ❑ Coordenadas geo-espaciales
- ❑ Fuentes separadas
- ❑ Constante cambio de esquemas

Consultas

- ❑ Análisis y agregaciones en tiempo real
- ❑ Por ubicación
- ❑ Menor latencia
- ❑ Respuesta de tiempo / experiencia del usuario
- ❑ Relaciones conocidas entre entidades / colecciones
- ❑ Lecturas y escrituras locales o globales

Otros aspectos

- ❑ Desarrollo ágil
- ❑ Crecimiento rápido
- ❑ Alto rendimiento
- ❑ Disponibilidad (~99.999%)
- ❑ Mejorar TTM
- ❑ TCO más bajo
- ❑ Infraestructura de nube
- ❑ Retos con el uso de RDMBS

Model Your Data The Way You Use It

RDBMS

PERSON

Pers_ID	Surname	First_Name	City
0	Miller	Paul	London
1	Ortega	Alvaro	Valencia
2	Huber	Urs	Zurich
3	Blanc	Gaston	Paris
4	Bertolini	Fabrizio	Rome

NO RELATION

CAR

Car_ID	Model	Year	Value	Pers_ID
101	Bently	1973	100000	0
102	Rolls Royce	1965	330000	0
103	Peugeot	1993	500	3
104	Ferrari	2005	150000	4
105	Renault	1998	2000	3
106	Renault	2001	7000	3
107	Smart	1999	2000	2

MongoDB

```
{
  _id:0
  first_name: 'Paul',
  surname: 'Miller',
  city: 'London',
  location: {
        type : 'Point'
        coordinates : [45.123,47.232]
  },
  cars: [
    { model: 'Bentley',
      year: 1973,
      value: 100000, … },
    { model: 'Rolls Royce',
      year: 1965,
      value: 330000, … }
  ]
}
```

Number

String

Geo-Coordinates

Typed field values

Fields can contain an array of sub-documents

1-to-Many relationships

ORM Layer removed – Data is already an object!

mongoDB

Model Your Data The Way You Use It

Referencing

user document

```
{
  _id: <ObjectId1>,
  username: "123xyz"
}
```

contact document

```
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone: "123-456-7890",
  email: "xyz@example.com"
}
```

access document

```
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```

Embedding

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
             phone: "123-456-7890",
             email: "xyz@example.com"
           },
  access: {
            level: 5,
            group: "dev"
          }
}
```

Embedded sub-document

Embedded sub-document

mongoDB

Modeling and Cardinality

- One to One
  - Author to blog post
- One to Many
  - Blog post to comments
- One to Millions
  - Blog post to site views (e.g. Huffington Post)

One To One Relationships

- “Belongs to” relationships are often embedded
- Holistic representation of entities with their embedded attributes and relationships.
- Great read performance

Most important:

- **Keeps simple things simple**
- **Frees up time to tackle harder schema issues**

One To One Relationships

```
{
    "title" : "This is a blog post",
    "author"  : {
                  name : "Alejandro Mancilla",
                  login : "alejandro.mancilla@mongodb.com",
                  },
   …
}
```

We can index on `"title"` **and** `"author.login"`**.**

One to Many - Embedding

```
{
    "_id"         : ObjectID( "ZZZZ" ),
    "Title"       : "A Blog Title",
    "Body"        : "A blog post",
    "comments"    : [{
                name     : "Juan Perez",
                email    : "jperez@mongodb.com",
                comment :"I love your writing style",
        }
        {
                name     : "Pedro Sanchez",
                email    : "psanchez@mongodb.com",
                comment :"I hate your writing style",
        }]
}
```

Where we expect a small number of sub-documents we can embed them in the main document

Key Concerns

- ***What are the write patterns?***
  - Comments are added more frequently than posts
  - Comments may have images, tags, large bodies of text
- ***What are the read patterns?***
  - Comments may not be displayed
  - May be shown in their own window
  - People rarely look at all the comments

One to Many – Linking I

- Keep all comments in a separate comments collection
- Add references to posts IDs
- Requires two queries to display blog post and associated comments

```
{
    "_id"       : ObjectID( "ZZZZ" ),
    "Title"     : "A Blog Title",
    "Body"      : "A blog post"
}
{
    "_id"       : ObjectID( "ZZZZ" ),
    "Title"     : "Another Blog Title",
    "Body"      : "Another blog post",
}
```

```
{
    _id      : ObjectID( "AAAA" ),
    post_id  : ObjectID( "ZZZZ" ),
    name     : "Juan Amores",
    email    : "jamores@mongodb.com",
    comment  :"I love your writing style",
}
{
    _id      : ObjectID( "AAAB" ),
    post_id  : ObjectID( "ZZZZ" ),
    name     : "Pedro Víbora",
    email    : "pvivora@mongodb.com",
    comment  :"I hate your writing style",
}
```

One to Many – Linking II

- Keep all comments in a separate comments collection
- Add references to comments as an array of comment IDs
- Requires two queries to display blog post and associated comments
- Requires two writes to create a comments

```
{
   "_id"      : ObjectID( "ZZZZ" ),
   "Title"    : "A Blog Title",
   "Body"     : "A blog post",
   "comments" : [ ObjectID( "AAAA" ),
                  ObjectID( "AAAB" )]
}
{
   "_id"      : ObjectID( "ZZZZ" ),
   "Title"    : "A Blog Title",
   "Body"     : "A blog post",
   "comments" : []
}
```

```
{
   _id     : ObjectID( "AAAA" ),
   name    : "Joe Drumgoole",
   email   : "Joe.Drumgoole@mongodb.com",
   comment :"I love your writing style",
}
{
   _id     : ObjectID( "AAAB" ),
   name    : "John Smith",
   email   : "Joe.Drumgoole@mongodb.com",
   comment :"I hate your writing style",
}
```

One To Many – Hybrid Approach

```
{
    _id          : ObjectID( “ZZZZ” ),
    Title        : “A Blog Title”,
    Body         : “A blog post”,
    last_comments  : [{
        _id    : ObjectID( “AAAA” )
        name   : “Juan Perez”,     comment
:“I love your writing style”,
    },
    {
        _id      : ObjectID( “AAAB” ),
        name     : “Pedro Sanchez”,
        comment :“I hate your writing
    style”,
    }]
}
```

```
{
    “_id”      : ObjectID( “AAAA” ),
    “post_id” : ObjectId( “ZZZZ”),
    “name”     : “Juan Perez”,
    “email”    : “jperez@mongodb.com”,
    “comment” :“I love your writing
    style”,
    }
    {...},{...},{...},{...},{...},{...}
    ,{..},{...},{...},{...} ]
```

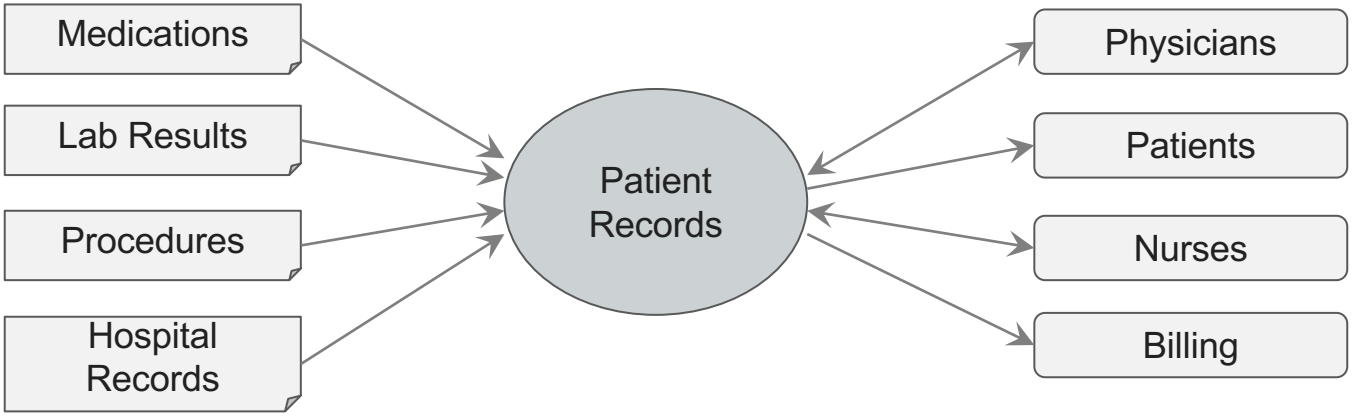
Linking vs. Embedding

- Embedding
  - Terrific for read performance
    - Webapp “front pages” and pre-aggregated material
  - Writes can be slow
  - Data integrity needs to be managed

- Linking
  - Flexible
  - Data integrity is built-in
  - Work is done during reads

Medical Records

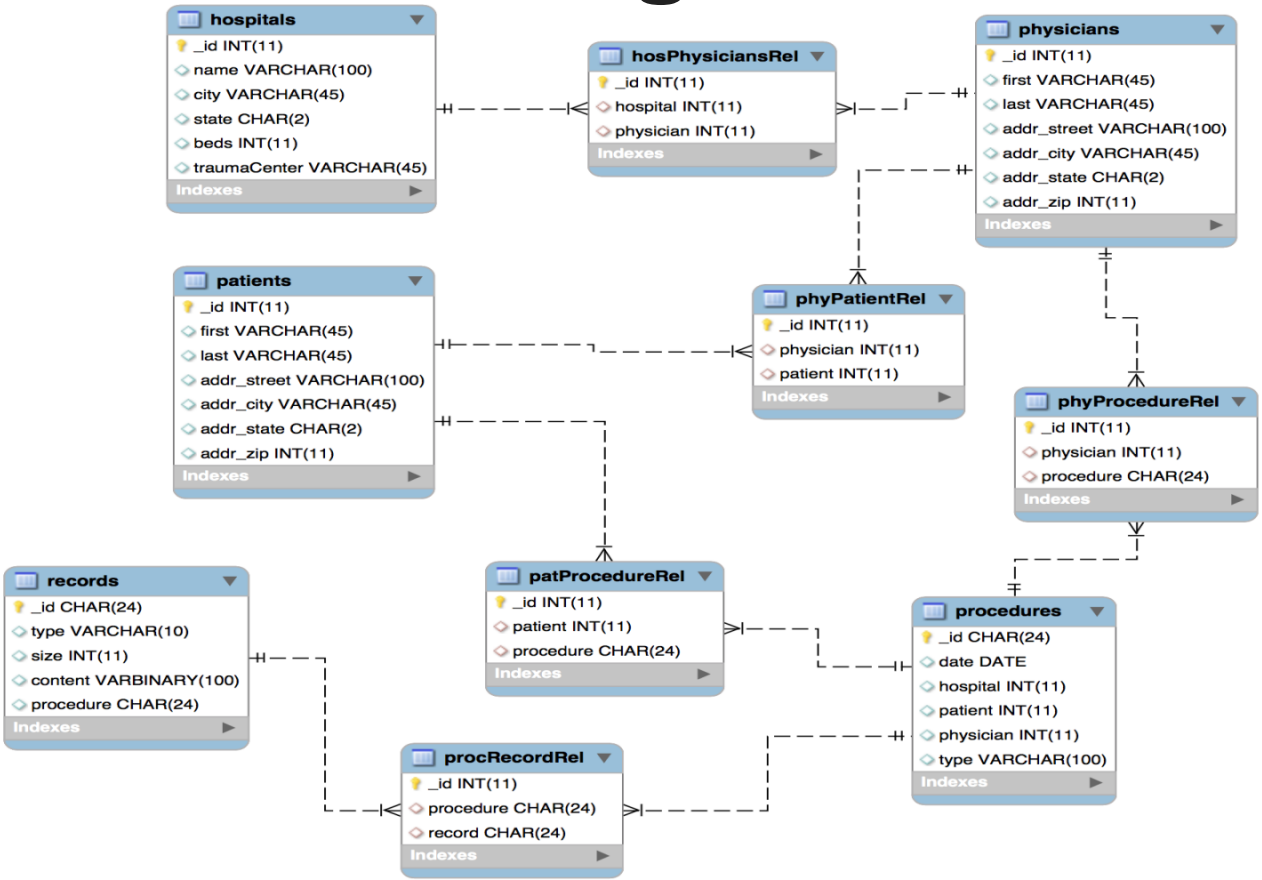
- Collects all patient information in a central repository
- Provide central point of access for
  - Patients
  - Care providers: physicians, nurses, etc.
  - Billing
  - Insurance reconciliation
- Hospitals, physicians, patients, procedures, records



Medical Record Data

- **Hospitals**
  - have physicians
- **Physicians**
  - Have patients
  - Perform procedures
  - Belong to hospitals
- **Patients**
  - Have physicians
  - Are the subject of procedures
- **Procedures**
  - Associated with a patient
  - Associated with a physician
  - Have a record
  - Variable meta data
- **Records**
  - Associated with a procedure
  - Binary data
  - Variable fields

Relational Starting Point

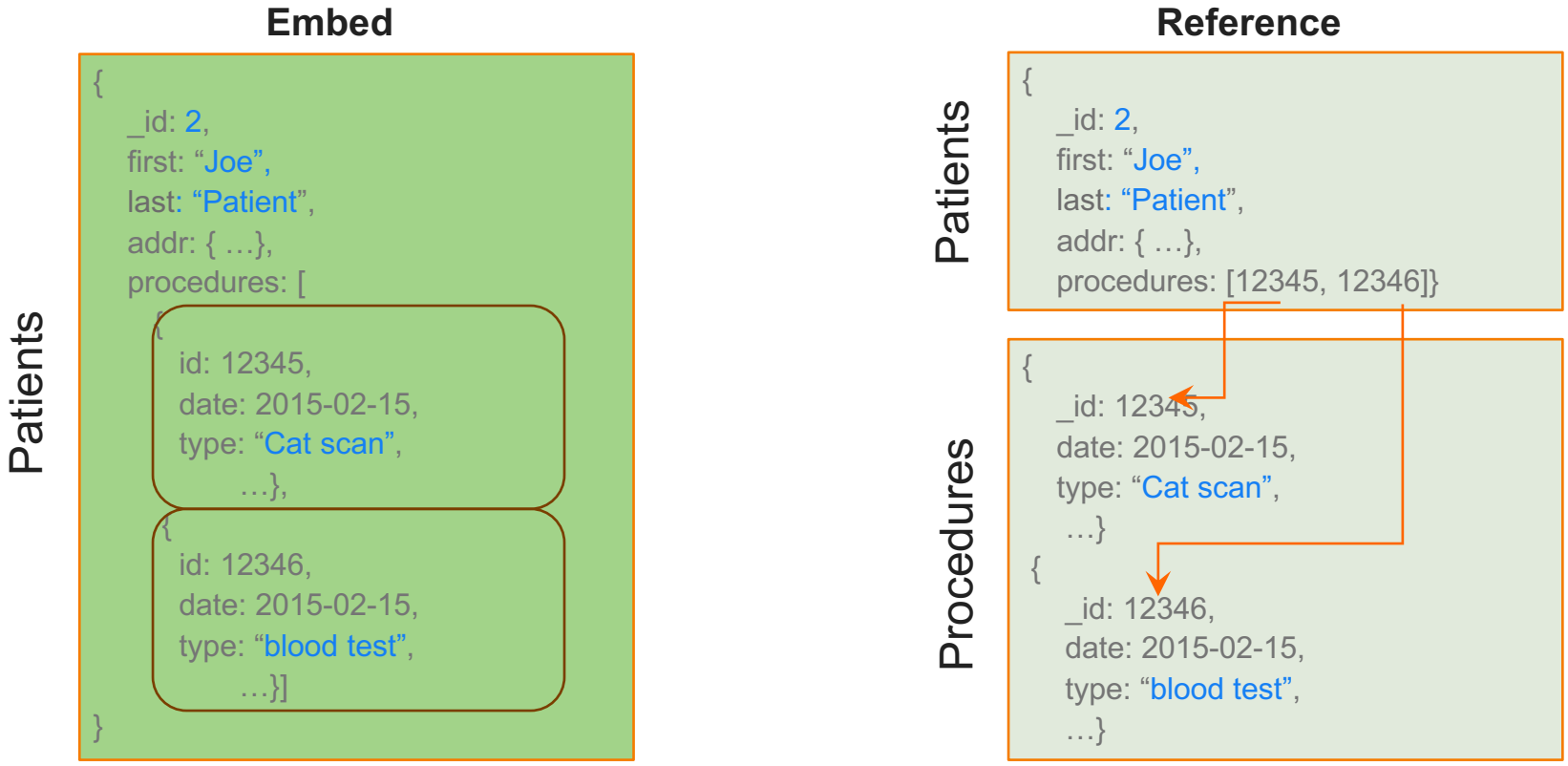


Schema Design

- Relational schema is typically not a good fit for MongoDB's document model
- Leverage document model to gain
  - Agility
  - Scalability
  - Performance
- Denormalized schema
  - Query workload will determine best schema
  - Document schema should contain all related information to:
    - Avoid application-level "joins"
    - Leverage atomicity of document updates

MongoDB Schema Design in 2 Slides

Relationships modeled in 2 possible ways

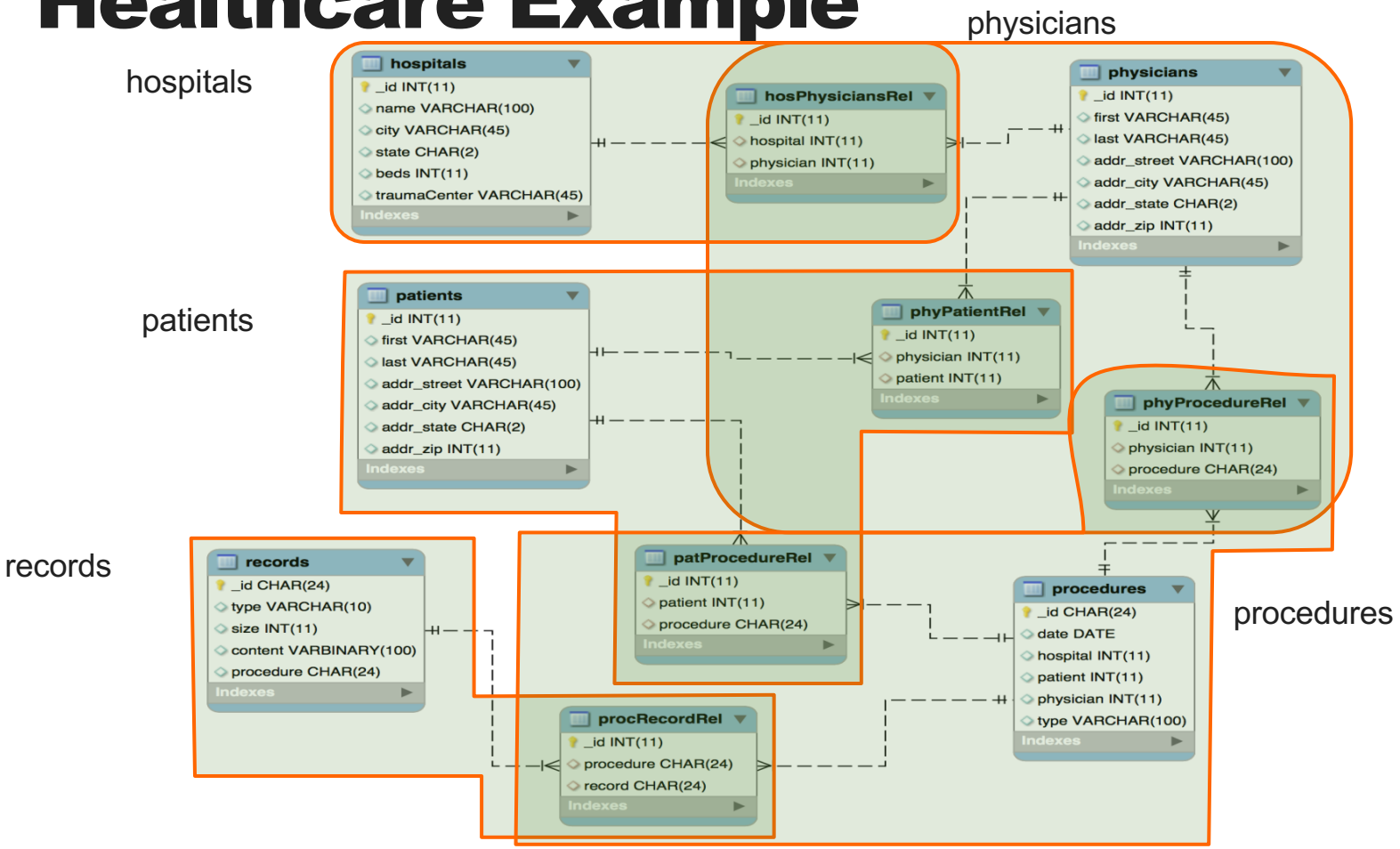


Relationships

Relationship	Modeling approach
1 to 1	Typically embed
1 to Many	Often embed
Many to Many	?

- Embedding is preferred
  - A single query returns all relevant data
  - Related information can be updated in an atomic operation
- Query workload determines the best approach
  - Performance
  - “transactions”
- Hybrids approaches often make sense too
  - Embed some of the data
  - Reference the rest

Healthcare Example



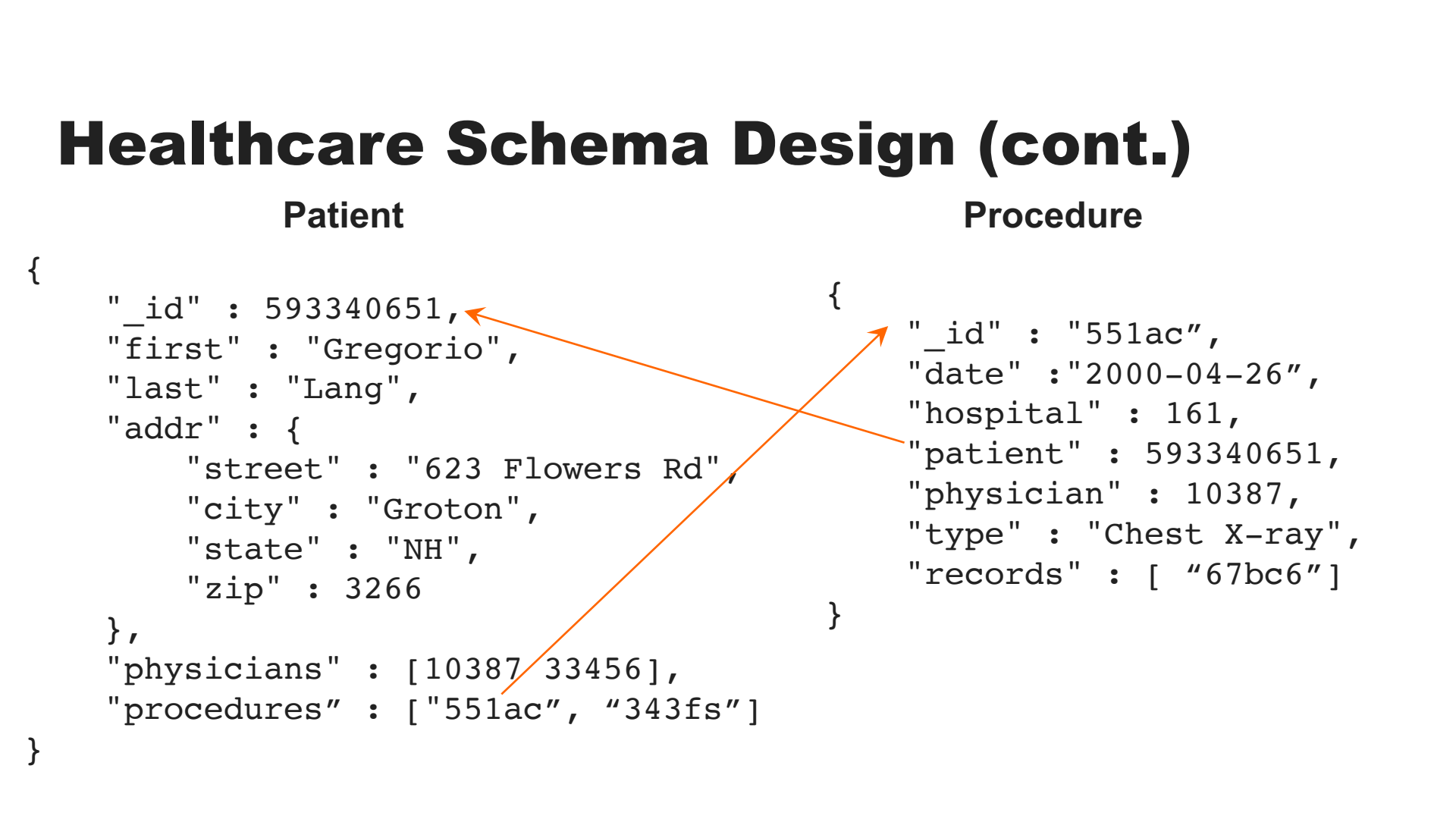
Healthcare Schema Design (cont.)

Patient

```
{
    "_id" : 593340651,
    "first" : "Gregorio",
    "last" : "Lang",
    "addr" : {
        "street" : "623 Flowers Rd",
        "city" : "Groton",
        "state" : "NH",
        "zip" : 3266
    },
    "physicians" : [10387 33456],
    "procedures" : ["551ac", "343fs"]
}
```

Procedure

```
{
    "_id" : "551ac",
    "date" :"2000-04-26",
    "hospital" : 161,
    "patient" : 593340651,
    "physician" : 10387,
    "type" : "Chest X-ray",
    "records" : [ "67bc6"]
}
```



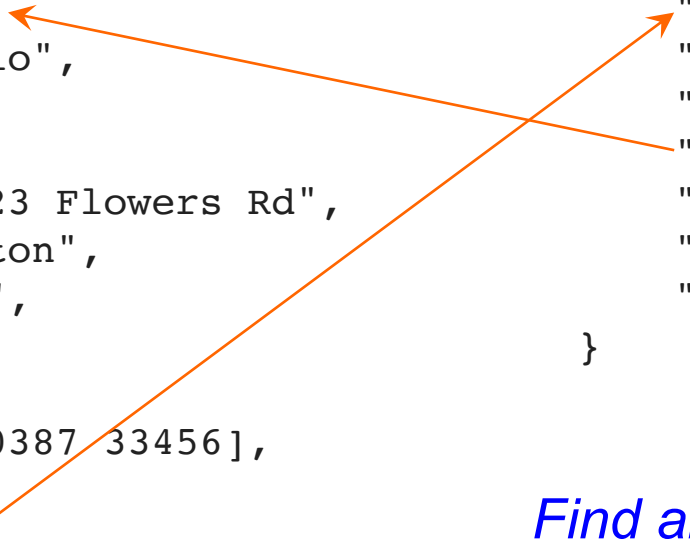
Healthcare Schema Design (cont.)

Patient

```
{
    "_id" : 593340651,
    "first" : "Gregorio",
    "last" : "Lang",
    "addr" : {
        "street" : "623 Flowers Rd",
        "city" : "Groton",
        "state" : "NH",
        "zip" : 3266
    },
    "physicians" : [10387, 33456],
    "procedures" : [
       {id : "551ac",
        type : "Chest X-ray"},
       {id : "343fs",
        type : "Blood Test"}]
}
```

Procedure

```
{
    "_id" : "551ac",
    "date" :"2000-04-26",
    "hospital" : 161,
    "patient" : 593340651,
    "physician" : 10387,
    "type" : "Chest X-ray",
    "records" : [ "67bc6"]
}
```



Find all patients from NH that have had chest x-rays

Let’s do crazy things!

- What is we were tracking mouse position for heat tracking?
  - Each user will generate hundreds of data points per visit
  - Thousands of data points per post
  - Millions of data points per blog site
- Relational-like model
  - Store a blog ID per event
  - Be polymorphic, my friend!

```
{
  “post_id”    : ObjectID(“ZZZZ”),
  “timestamp” : ISODate("2005-01-02T16:35:24Z”),
  “event”      : {
                     type: click,
                     position  : [240, 345]} }
{
  “post_id”    : ObjectID(“ZZZZ”),
  “timestamp” : ISODate("2005-01-02T16:35:24Z”),
  “event”      : {
                     type: close}
}
```

What if we use the structure?

```
{
   post_id   : ObjectID ( “ZZZZ” ),
   cookie_id : “R34oitwrFWt945tw34t4569tiwemrti”,
   timeStamp : ISODate("2005-01-02T16:00:00Z”),
   events : {
     0  : { 0 : { event }, 1 : { event }, … 59: { event }},
     1  : { 0 : { event }, 1 : { event }, … 59: { event }},
     2  : { 0 : { event }, 1 : { event }, … 59: { event }},
     3  : { 0 : { event }, 1 : { event }, … 59: { event }},
     ...
     59 :{ 0 : { event }, 1 : { event }, … 59: { event }}
}
```

What if we build buckets?

```
{
    post_id : ObjectID ( “ZZZZ” ),
    cookie_id : “R34oitwrFWt945tw34t4569tiwemrti”,
    count : 98,
    events :  [ { event }, { event }, { event } ... ]
}
```

Summary

- Schema design is different in MongoDB
  - But basic data design principles stay the same
- Focus on how an application accesses/manipulates data
- Seek out and capture belongs-to 1:1 relationships
- Don’t get stuck in “one record” per item thinking
  - Embrace the hierarchy and think about cardinality
- Evolve the schema to meet requirements as they change
- Be polymorphic!
- Document updates are transactions
- Use validation in your advantage

¿Preguntas?


mongoDB