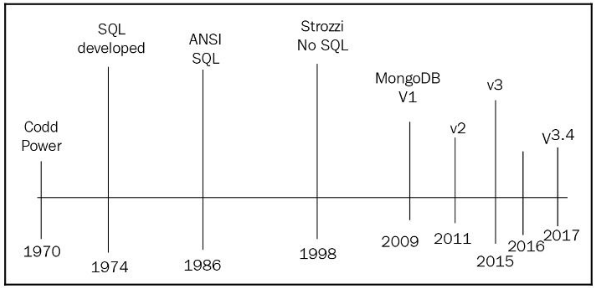
MongoDB

Table of contents

Chapter 1

SQL and NoSQL evolution

- 1970 : Dr. EF Codd published the paper "A Relation Model of Data for Large Shared Data Banks"
- 1974 : IBM developed SQL
- 1979 : Oracle provides the first RDBMS commercially available
- 1986 : The first SQL Standard is published
- 1998 : First time the term NoSQL is coined by Carlo Strozzi for an open source database that relaxes some SQL constraints (ACID) in favor of availability and scalability

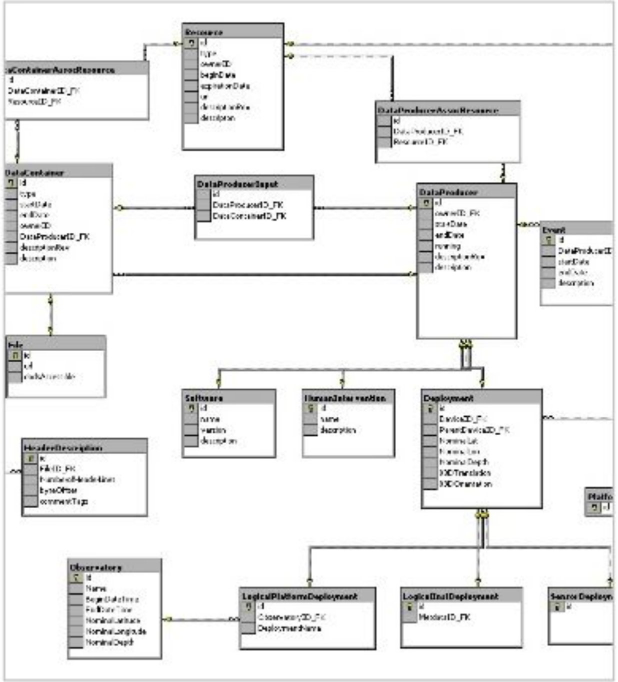


Timeline of SQL and NoSQL evolution

MongoDB key characteristics and use cases

Document-oriented database

RDBMS



MongoDB

```
{
    _id : ObjectId("4c4ba5e5e8aabf3"),
    employee_name: "Dunham, Justin",
    department : "Marketing",
    title : "Product Manager, Web",
    report_up: "Neray, Graham",
    pay_band: "C",
    benefits : [
            {  type :   "Health",
               plan : "PPO Plus" },
            {  type :    "Dental",
               plan : "Standard" }
                  ]
}
```

MongoDB key characteristics and use cases

Key characteristics

- General purpose database
  - Not the case for many other NoSQL databases (e.g. graph-oriented)
- Flexible schema design
  - Document oriented approaches with non-defined attributes
- Build for high availability
- Build to scale
- Aggregation framework
  - Provides powerful transformation capabilities
- Native replication
- Security features
- JSON based (BSON)
- Support MapReduce

MongoDB key characteristics and use cases

What is the use case for MongoDB?

- Integration of data providing a single view of them
- Internet of Things
- Mobile applications
- Real-time analytics
- Personalization
- Catalog management
- Content management

MongoDB key characteristics and use cases

MongoDB criticism

- Schema-less nature
- Lack of proper ACID guarantees

⇒ Hard to ensure consistency, isolation...

MongoDB key characteristics and use cases

MongoDB binaries

- mongod
  - starts MongoDB
- mongo
  - open the MongoDB shell
- mongodump
  - create a database dump (bson+json files)
- mongorestore
  - restore a database dump

Chapter 2

CRUD using the shell

- List the databases
  - `db`
- Connect to a database (with creation if does not exist)
  - `use <database_name>`
- Insert a document
  - `db.<collection_name>.insert({...MATCHING_OBJECT...})`
  - `db.books.insert({title: 'mastering mongoDB', isbn: '101'})`
  - `db.books.insert({title: 'mongoDB cookbook', isbn: '1331'})`
- Find documents
  - `db.<collection_name>.find({...MATCHING_OBJECT...})`
  - `db.books.find({isbn: '1331'})`
- Delete a document
  - `db.<collection_name>.remove({...MATCHING_OBJECT...})`
  - `db.books.remove({isbn: '101'})`
- Delete a document (2)
  - `db.<collection_name>.remove(OBJECT_ID_STRING)`
  - `db.books.remove('2345678dae345eff')`

CRUD using the shell

- Update a document
  - `db.<collection_name>.update({...MATCHING_OBJECT...},{...UPDATING_OBJECT...})`
  - `db.books.update({isbn: '1331'},{price: 30})`
    => Replace all matched objects with the new object
- Update a document (2)
  - `db.<collection_name>.update({...MATCHING_OBJECT...},{$set: {...UPDATING_OBJECT...}})`
  - `db.books.update({isbn: '1331'},{$set: {price: 30, title:'Mongo'}})`
- Find all documents
  - `db.<collection_name>.find()`
- Find documents and pretty print them in the console
  - `db.<collection_name>.find({...MATCHING_OBJECT...}).pretty()`
  - `db.books.find().pretty()`

Scripting for the mongo shell

Main reason for using the shell : Mongo shell is a Javascript shell

```
> var title = 'MongoDB in a nutshell'
> title
MongoDB in a nutshell
> db.books.insert({title: title, isbn: 102})
WriteResult({ "nInserted" : 1 })
> db.books.find()
{ "_id" : ObjectId("59203874141daf984112d080"), "title" : "MongoDB in a
nutshell", "isbn" : 102 }
```

Scripting for the mongo shell

Using scripts and functions

```
> queryBooksByIsbn = function(isbn) { return db.books.find({isbn: isbn})}
> queryBooksByIsbn("101")
{ "_id" : ObjectId("592035f6141daf984112d07f"), "title" : "mastering
mongoDB", "isbn" : "101", "price" : 30 }
```

Executing script files:

From the shell

```
mongo FILE_PATH
```

From the mongo shell:

```
load(FILE_PATH_STRING)
```

Scripting for the mongo shell

Batch inserts using the shell

```
authorMongoFactory = function() {
     for(loop=0;loop<1000;loop++){
          db.books.insert({name: "MongoDB factory book" + loop})
     }
}
```

Although working, it performs 1000 database inserts (which is not efficient).

Prefer a bulk write:

```
fastAuthorMongoFactory = function() {
     var bulk = db.books.initializeUnorderedBulkOp();
     for(loop=0;loop<1000;loop++) {
          bulk.insert({name: "MongoDB factory book" + loop})
     }
     bulk.execute();
}
```

If you want the data having the same order of insertion than declaration, use

```
     db.books.initializeOrderedBulkOp();
```

Scripting for the mongo shell

Batch inserts using the shell

Alternative

```
db.collection.bulkWrite(
[ <operation 1>, <operation 2>, ... ],
      {
      writeConcern : <document>,
      ordered : <boolean>
      }
)
```

Administration

- db.dropDatabase()
- db.getCollectionNames()
- db.copyDatabase(fromDB, toDB)

Simple Extraction

Simple extraction

find()

```
db.<collection>.find(
    {...SELECTOR_OBJECT...},
    {...PROJECTION_OBJECT...}
)
```

Simple extraction

$exists

```
$exists
allows to filter on properties existence

db.<collection>.find({<FIELD> : {$exists: <BOOL>}},{})
```

Simple extraction

Projection

The projection allows to get specific property from the objects

```
db.<collection>.find({},{<FIELD> : <BOOL>})
```

When <BOOL> is

- true, then shows only the specified properties
- false, then hides the specified properties

Simple extraction

$eq, $gt, $lt, $gte, $lte, $ne

Comparison operators

```
db.<collection>.find({
    <FIELD>: {<OPERATOR>:<VALUE>}
},{})
```

$eq	equal
$gt	greater than
$lt	lower than
$gte	greater equal
$lte	lower equal
$ne	not equal

Example:

```
db.products.find({price: {$lte: 50}},{})
```

Simple extraction

$in, $nin

Is in or **is not in** operator

```
db.<collection>.find({
    <FIELD>: {<OPERATOR>:<VALUES_ARRAY>}
},{})
```

Example:

```
db.customers.find({City: {$in: ['Orlando', 'Baton Rouge']}})
```

Simple extraction

$and, $or, $nor, $not

Logical operators

```
db.<collection>.find({
    <OPERATOR>: [<EXPR1>, <EXPR2>,...]
},{})
```

Example:

```
db.products.find({$and: [
    {City: 'Orlando'},{Price: {$lt:50}}
]},{})
```

$and	Joins query clauses with a logical AND returns all documents that match the conditions of both clauses.
$or	Joins query clauses with a logical OR returns all documents that match the conditions of either clause.
$nor	Joins query clauses with a logical NOR returns all documents that fail to match both clauses.
$not	

Simple extraction

$type

Selects the documents where the value of the field is an instance of the specified BSON type(s).

```
db.<collection>.find({
    <FIELD>: {$type : <TYPESTRING>}
},{})
```

Example:

```
db.products.find({$and: [
    {Price: {$type:'double'}}
]},{})
```

Simple extraction

$mod

Select documents where the value of a field divided by a divisor has the specified remainder

```
db.<collection>.find({
    <FIELD>: {$mod : [<DIVISOR>,<REMAINDER>]}
},{})
```

Example:

```
db.products.find({$and: [
    {Price: {$mod:[2,0]}}
]},{})
```

Simple extraction

$regex

Provides regular expression capabilities for pattern matching strings in queries. (PCRE 8.41)

```
db.<collection>.find({
    <FIELD>: {$regex : /REGEX/, $options : <STRING_OPTIONS>}
},{})
db.<collection>.find({
    <FIELD>: /REGEX/OPTIONS
},{})
```

Example:

```
db.products.find({productName: /cup/i})
```

i	Case insensitivity to match upper and lower cases.
m	For patterns that include anchors (i.e. ^ for the start, $ for the end), match at the beginning or end of each line for strings with multiline values. Without this option, these anchors match at beginning or end of the string.
x	“Extended” capability to ignore all white space characters in the $regex pattern unless escaped or included in a character class.

Simple extraction

$text

Performs a text search on the content of the fields indexed with a text index.

```
REQUIRED: db.<collection>.createIndex({<FIELD>:”text”})
db.<collection>.find({
    $text: {
        $search: <TEXT_TO_SEARCH>
        $language: <LANGUAGE>
        $caseSensitive: <BOOL>
        $diacriticSensitive: <BOOL>
    }
},{})
```

Simple extraction

$text

Performs a text search on the content of the fields indexed with a text index.

Example:

```
db.products.find({
    $text: {
        $search: ‘fry’,
        $language: ‘english’,
        $diacriticSensitive: true
    }
})
```

Simple extraction

$where

Use the [$where](#) operator to pass either a string containing a JavaScript expression or a full JavaScript function to the query system. (From 3.6, $expr should be prefered).

```
db.products.find({
    $where: <JS_EXPRESSION>
})
```

Example:

```
db.embeddedOrders.find({$where: 'this.Orders.length == 1'})
```

Simple extraction

$expr (3.6)

Allows the use of aggregation expressions within the query language. $expr can build query expressions that compare fields from the same document in a $match stage

```
db.products.find({
    $expr: <EXPRESSION>
})
```

Example:

```
db.monthlyBudget.find({$expr:{$gt:["$spent","$budget"]}})
```

Simple extraction

.distinct()

Finds the distinct values for a specified field across a single collection or view and returns the results in an array.

```
db.collection.distinct(
    field,
    query
)
```

Simple extraction

.count()

Returns the count of documents that would match a `find()` query for the collection or view

```
db.collection.count(
    query,
    options
)
```

Simple extraction

Exercices

1. Get the customers without company
2. List distinctly all the city where customers working for “Yodo” or “Kare” are living
3. Count the number of products which are sold at a price greater than 50
4. Display the first and last name of the customer having a *****@patch.com email adres.
5. Find purple product costing more than 20
6. Find the product having their color in their name

Aggregation

Aggregation

$expr (3.6)

```
db.<collection>.aggregate([
    {<EXPRESSION1>},
    {<EXPRESSION2>},
    {<EXPRESSION3>},
    ...
])
```

Aggregation is performed step-by-step. The document resulting from an expression is used as input for the following expression.

Aggregation Pipeline Stages

$match

Filters the documents to pass only the documents that match the specified condition(s) to the next pipeline stage.

```
db.<collection>.aggregate([
    {$match: <QUERY>},
    ...
])
```

Example:

```
db.products.aggregate([{$match: {price: {$gt:50}}}])
```

Aggregation Pipeline Stages

$group

Groups documents by some specified expression and outputs to the next stage a document for each distinct grouping. The output documents contain an _id field which contains the distinct group by key.

```
db.<collection>.aggregate([
    {$group: {
        _id: <EXPRESSION>,
        <FIELD1>: { <ACC1> : <EXPRESSION1> },
        <FIELD2>: { <ACC2> : <EXPRESSION2> },
        ...
    }},
    ...
])
```

$avg
$first
$last
$max
$min
$push
$addToSet
$stdDevPop
$sdtDevSamp
$sum

Aggregation Pipeline Stages

$group

Groups documents by some specified expression and outputs to the next stage a document for each distinct grouping. The output documents contain an _id field which contains the distinct group by key.

Example:

```
db.<collection>.aggregate([
    {$group: {
        _id: <EXPRESSION>,
        <FIELD1>: { <ACC1> : <EXPRESSION1> },
        <FIELD2>: { <ACC2> : <EXPRESSION2> },
        ...
    }},
    ...
])
```

Aggregation Pipeline Stages

$unwind

Deconstructs an array field from the input documents to output a document for each element. Each output document is the input document with the value of the array field replaced by the element.

```
db.<collection>.aggregate([
{
  $unwind:
    {
      path: <FIELD_PATH>,
      includeArrayIndex: <STRING>,
      preserveNullAndEmptyArrays: <BOOL>
    }
}
    ,
    ...
])
```

To specify a field path, prefix the field name with a dollar sign $ and enclose in quotes.

Aggregation Pipeline Stages

$lookup

Performs a left outer join to an unsharded collection in the *same* database to filter in documents from the "joined" collection for processing.

```
db.<collection>.aggregate([
{
   $lookup:
     {
       from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection>,
       as: <output array field>
     }
},
    ...
])
```

Aggregation Pipeline Stages

$sort

Sorts all input documents and returns them to the pipeline in sorted order.

```
db.<collection>.aggregate([
{
    $sort: {
        <field1>: <sort order>,
        <field2>: <sort order>,
        ...
    }
},
    ...
])
```

Aggregation Pipeline Stages

$limit

Limits the number of documents passed to the next stage in the pipeline.

```
db.<collection>.aggregate([
{
    $limit: <POSITIVE_NUMBER>
},
    ...
])
```

> NOTE:
>
> When a `$sort` precedes a `$limit` and there are no intervening stages that modify the number of documents, the optimizer can coalesce the `$limit` into the `$sort`. This allows the `$sort` operation to only maintain the top `n` results as it progresses, where `n` is the specified limit, and ensures that MongoDB only needs to store `n` items in memory. This optimization still applies when `allowDiskUse` is `true` and the `n` items exceed the aggregation memory limit.

Aggregation Pipeline Stages

$out

Takes the documents returned by the aggregation pipeline and writes them to a specified collection. The `$out` operator must be *the last stage* in the pipeline.

```
db.<collection>.aggregate([
{
    $out: {<OUTPUT_COLLECTION_STRING>}
},
    ...
])
```

Aggregation Pipeline Stages

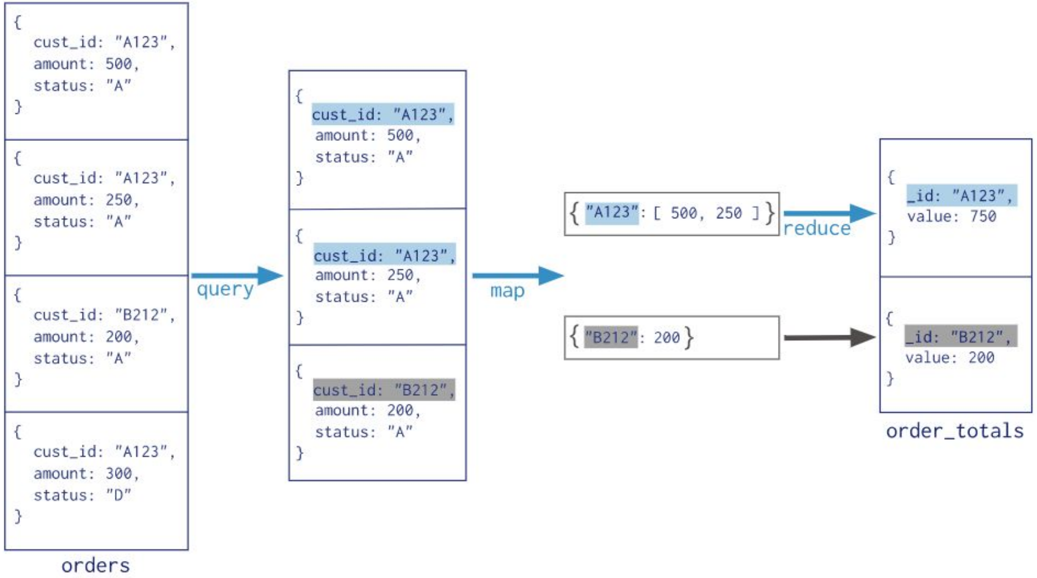
Exercices

1. *Create a new collection which is denormalized where:*
   a. *Customers have Orders that are composed of OrderLine*
2. Count the number of orders for each customer working at Yodo
   a. Display only the first name, the last name and the orders number
3. Display the sales amount and the quantity ordered for each product ordered by Yodo employees
4. *What is the product name of the product the most bought by women*
5. What is the average order value for the people living in Atlanta
6. *Determine the sales amount both by color and by city*
7. *For each customer living in Orlando, determines how many orders included leaf related products.*
8. *Compare the sales amount of cherry and tomato related products*

MapReduce

MapReduce

```
           Collection
db.orders.mapReduce(
    map    →  function() { emit( this.cust_id, this.amount ); },
    reduce →  function(key, values) { return Array.sum( values ) },
              {
    query  →    query: { status: "A" },
    output →    out: "order_totals"
              }
)
```



MapReduce

Exercice

1 - Find the average number of product (orderlines) each customer of Yodo ordered per order

+ Exercices 3 and 6 from *Aggregation Pipeline Stages* exercices

Chapter 3

Data Modelling

Flexible schema

Unlike SQL databases, MongoDB collections do not require the documents to have the same schema.

- Fields, data type… can vary from one document to another.

In practice, the documents of a particular collection should share a similar schema

- Those similarities can be ensured by rules. This is called schema validation.

Data Modelling

Document structure

- Relational databases are designed on the basis of the real-world relations
- Document-oriented database are designed on the basis of how that application will use those data

MongoDB allows both the use of embedded data (denormalisation) or references (normalisation).

Data Modelling

Embedded data

Embedded data is a kind a denormalisation.

Purpose: allowing the application to retrieve and manipulate related-data in a single operation.

Use:

- One-to-one “contains” relationship between data
  Ex: ID card for a person...
- When a child document is always viewed in the context of the parent
  Ex: A customer list of delivery address

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

Embedded sub-document

Embedded sub-document

Data Modelling

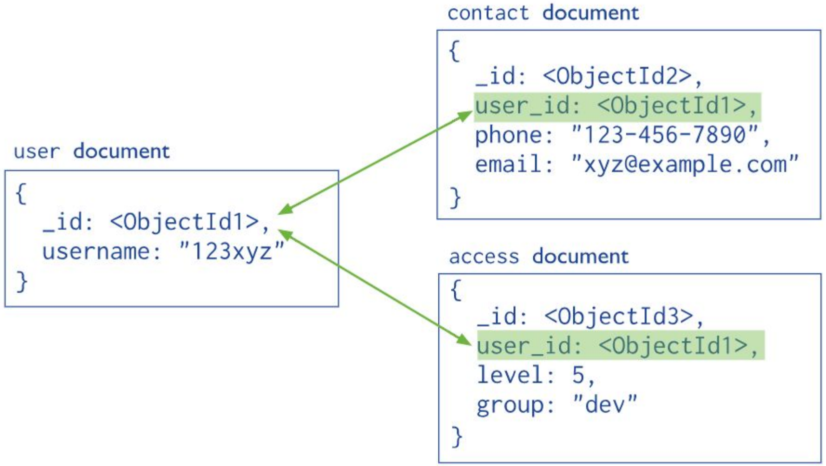
References

References are used to normalise.

Purpose: provides more flexibility and avoid data duplication.

Use:

- when embedding would result in duplication of data
- to represent more complex many-to-many relationships
- to model large hierarchical data sets



Data Modeling

Rules of Thumb for MongoDB Schema Design

Modeling **One-to-Few** : Embedded documents
Example: Person ↔ Address
Addresses are embedded in the person

Modeling **One-to-Many** : References (Many ~ hundreds)
The parent is referencing the children
Example: Product ↔ Parts
Product having an array with references to all parts

Modeling **One-to-Squillions** : References (Many ~squillions)
Each child has a reference to its parent (otherwise the size of the document will explode)
Example: Host ↔ Log
Each Log document has a reference to its host

Data Modeling

JSON Schema

Add validation at collection creation:

```
db.createCollection(<collection_name>, {
   validator: {
      $jsonSchema: {
}}})
```

Add validation when the collection exists:

```
db.runCommand( { collMod: <collection_name>, validator: {
      $jsonSchema: {

}}})
```

Documentation: https://docs.mongodb.com/v3.6/core/schema-validation/index.html#json-schema

Library System

Exercice

Assume there is a library system with the following properties.

The library contains one or several copies of the same book. Every copy of a book has a copy number and is located at a specific location in a shelf. A copy is identified by the copy number and the ISBN number of the book. Every book has a unique ISBN, a publication year, a title, an author, and a number of pages. Books are published by publishers. A publisher has a name as well as a location.

Within the library system, books are assigned to one or several categories. A category can be a subcategory of exactly one other category. A category has a name and no further properties. Each reader needs to provide his/her family name, his/her first name, his/her city, and his/her date of birth to register at the library. Each reader gets a unique reader number. Readers borrow copies of books. Upon borrowing the return date is stored.

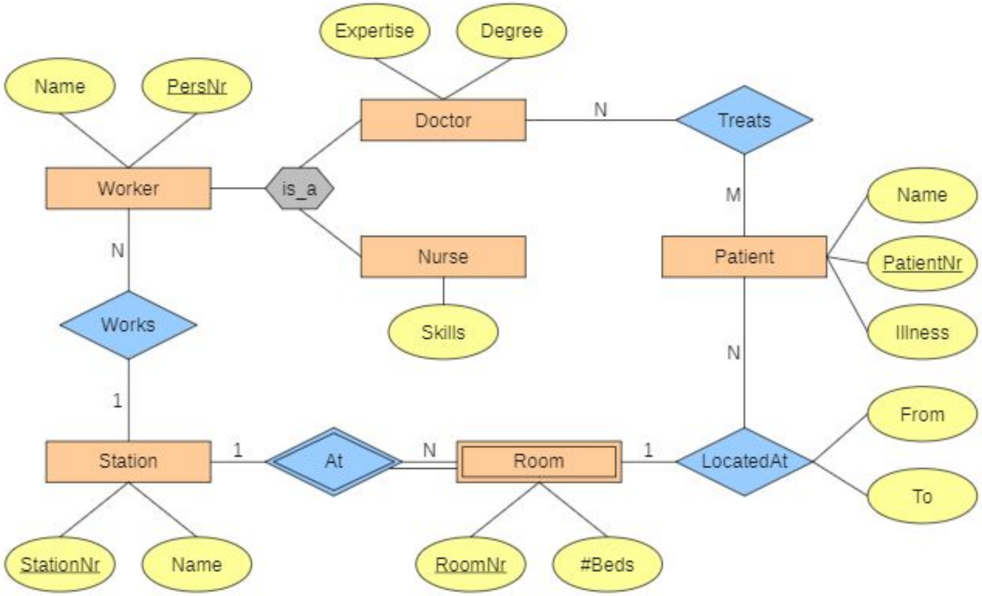
How would you model this situation in a document-oriented database ?

Source: ETH Zurich - Data Modelling and Databases Lectures – Spring 2018

Data Modeling

Exercice 2

Transform this ER Model into document schemas



Data Modeling

Exercice 3

Implement the following document schema:

Customers are always known by their first name and last name that are strings. We can know their address which consists of a street, a zip code (int varying from 1000 to 9999) and a city.

There are no other possible field.