INSTITUTO POLITÉCNICO DE BRAGANÇA

Bases de Dados

Engenharia Informática (2ºano)

Tecnologias Digitais e Gestão (2ºano)

- 5/6.MongoDB -

João Paulo Pereira | jprp@ipb.pt
Marisa Ortega | marisa.ortega@ipb.pt
David Dias |davide.dias@ipb.pt
Tiago Santos | tiago.santos@ipb.pt

2025

INSTITUTO POLITÉCNICO DE BRAGANÇA

Índice

- Conteúdo
  - 1. Introdução aos ambientes de Base de Dados
    - 1.1 Noção de Sistema de Informação
    - 1.2 A Informação nas Organizações
    - 1.3 Tecnologias de Informação
    - 1.4 Gestão de Informação
  - 2. Sistemas de Gestão de Bases de Dados
    - 2.1 Abordagem e Vantagens
    - 2.2 Arquitetura de um SGBD
    - 2.3 Tipos de Utilizadores num SGDB
  - 3. Modelação e Normalização de Dados
    - 3.1 Manutenção da Integridade
    - 3.2 Redundância e Chaves
    - 3.3 Diagramas E-R
    - 3.4 Modelo Relacional
  - 4. Álgebra Relacional e SQL - (MySQL)
    - 4.1 Conceitos e aplicação de Álgebra Relacional
    - 4.2 Ferramentas de Administração MySQL
    - 4.3 Comandos DDL
    - 4.4 Comandos DML
  - 5. Introdução às Bases de Dados NoSQL
    - Bases de Dados NoSQL,
    - Tipos de Bases de Dados NoSQL,
    - Bases de dados orientadas a documentos
  - 6. MongoDB
    - Estruturas JSON e BSON,
    - Modelação de dados,
    - Criação de coleções e documentos,
    - Operações CRUD e agregação,
    - Indexação e transações

INSTITUTO POLITÉCNICO DE BRAGANÇA

Books

- MongoDB: The Definitive Guide: THIRD EDITION / Shannon Bradshaw, Eoin Brazil, and Kristina Chodorow
- Cosmos DB for MongoDB Developers: Migrating to Azure Cosmos DB and Using the MongoDB API/ Manish Sharma

- https://beginnersbook.com/2017/09/mongodb-create-database/

INSTITUTO POLITÉCNICO DE BRAGANÇA

NoSQL Databases

- A database Management System provides the mechanism to store and retrieve the data. There are different kinds of database Management Systems:
  1. RDBMS (Relational Database Management Systems)
  2. OLAP (Online Analytical Processing)
  3. NoSQL (Not only SQL)

- NoSQL databases are different than relational databases.
  - ➔ In relational database you need to create the table, define schema, set the data types of fields etc before you can actually insert the data.
  - ➔ In NoSQL you don't have to worry about that, you can insert, update data on the fly.

- One of the advantage of NoSQL database is that they are really easy to scale and they are much faster in most types of operations that we perform on database.

- There are certain situations where you would prefer relational database over NoSQL, however when you are dealing with huge amount of data then NoSQL database is your best choice.

INSTITUTO POLITÉCNICO DE BRAGANÇA

NoSQL Databases

- We are familiarized to structure information such that it can be represented in tabular form.
- But not all information can follow that structure, hence the existence of NULL values.
  - ➔ The NULL value represents cells without information.
- To avoid NULLs, we must split one table into multiples, thus introducing the concept of normalization.
  - ➔ In normalization, we split the tables, based on the level of normalization we select. These levels are 1NF (first normal form), 2NF, 3NF, BCNF (Boyce–Codd normal form, or 3.5NF), 4NF, 5NF, ...
  - ➔ Every level dictates the split, and, most commonly, people use 3NF, which is largely free of insert, update, and delete anomalies.
  - ➔ To achieve normalization, one must split information into multiple tables and then, while retrieving, join all the tables to make sense of the split information.
  - ➔ This concept poses few problems, and it is still perfect for online transaction processing (OLTP).
- Working on a system that handles data populated from multiple data streams and adheres to one defined structure is extremely difficult to implement and maintain.
- The volume of data is often humongous and mostly unpredictable.
  - ➔ In such cases, splitting data into multiple pieces while inserting and joining the tables during data retrieval will add excessive latency.

NoSQL Databases

- We can solve this problem by inserting the data in its natural form.
  - ➔ As there is no or minimal transformation required, the latency during inserting, updating, deleting, and retrieving will be drastically reduced.

- With this, scaling up and scaling out will be quick and manageable. Given the flexibility of this solution, it is the most appropriate one for the problem defined.
  - ➔ The solution is NoSQL, also referred to as not only, or non-relational, SQL.

- One can further prioritize performance over consistency, which is possible with a NoSQL solution and defined by the CAP (consistency, availability, and partition tolerance) theorem.

- Summary:
  - ➔ Create documents without having to first define their structure
  - ➔ Each document can have its own unique structure
  - ➔ The syntax can vary from database to database
  - ➔ Can be added fields as necessary

NoSQL Databases

- Advantages of NoSQL
  - ➔ There are several advantages of working with NoSQL databases such as MongoDB. The main advantages are high scalability and high availability.

  - ➔ High scalability:
    - ⇨ NoSQL database such as MongoDB uses sharding for horizontal scaling.
      - → Sharding is partitioning of data and placing it on multiple machines in such a way that the order of the data is preserved.
      - → Vertical scaling means adding more resources to the existing machine while horizontal scaling means adding more machines to handle the data.
      - → Vertical scaling is not that easy to implement, on the other hand horizontal scaling is easy to implement. Horizontal scaling database examples: MongoDB, Cassandra etc.
      - → Because of this feature NoSQL can handle huge amount of data, as the data grows NoSQL scale itself to handle that data in efficient manner.

  - ➔ High Availability:
    - ⇨ Auto replication feature in MongoDB makes it highly available because in case of any failure data replicates itself to the previous consistent state.

Types of NoSQL

- In NoSQL, data can be represented in multiple forms. The most commonly used ones are:
  - ➔ key-value,
  - ➔ Columnar (column-oriented),
  - ➔ document, and
  - ➔ graph.

The types of NoSQL databases and the name of the databases system that falls in that category.
Key Value Store: Memcached, Redis, Coherence
Tabular: Hbase, Big Table, Accumulo
Document based: MongoDB, CouchDB, Cloudant

- Key-Value Pair
  - ➔ This is the simplest data structure form but offers excellent performance.
  - ➔ All the data is referred only through keys, making retrieval very straightforward.
  - ➔ The most popular database in this category is Redis Cache.
  - ➔ The keys are in the ordered list, and a HashMap is used to locate the keys effectively.

Key	Value
C1	XXX XXXX XXXX
C2	123456789
C3	10/01/2005
C4	ZZZ ZZZZ ZZZZ

- Columnar (column-oriented)
  - ➔ This type of database stores the data as columns instead of rows (as RDBMS do) and are optimized for querying large data sets.
  - ➔ This type of database is generally known as a wide column store.
  - ➔ Some of the most popular databases in this category include Cassandra, Apache Hadoop's HBase, etc.
  - ➔ Unlike key-value pair databases, columnar databases can store millions of attributes associated with the key forming a table, but stored as columns.
  - ➔ However, being a NoSQL database, it will not have any fixed name or number of columns, which makes it a true schema-free database.

INSTITUTO POLITÉCNICO DE BRAGANÇA

Types of NoSQL

- Document
  - ➔ This type of NoSQL database manages data in the form of documents.
  - ➔ Many implementations exist for this kind of database, and they have different various types of document representation.
  - ➔ Some of the most popular store data as JSON, XML, BSON, etc.
  - ➔ The basic idea of storing data in document form is to retrieve it faster, by matching to its meta information
  - ➔ Documents can contain many different forms of data key-value pairs, key-array pairs, or even nested documents.
  - ➔ One of the popular databases in this category is MongoDB.

Sample document structure (JSON) code

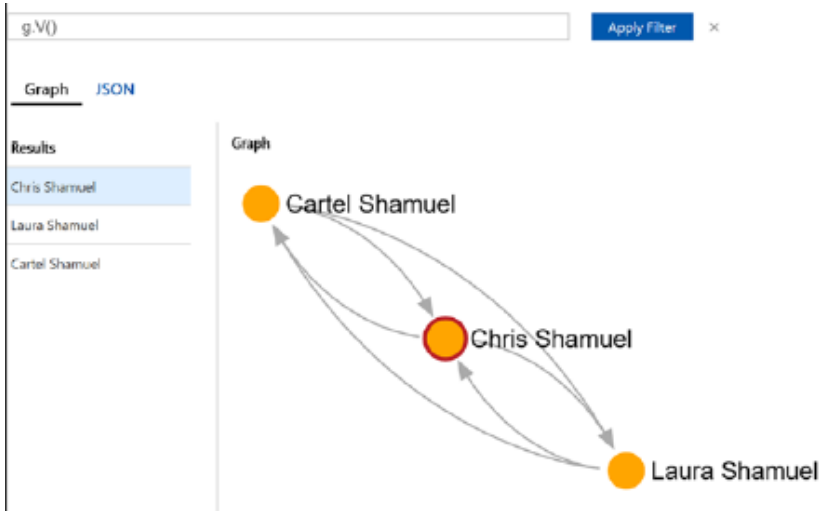
```
{
   "FirstName": "David",
   "LastName": "Jones",
   "EmployeeId": 10
}
```

Sample document structure (XML) code

```
<employee>
   <firstname>David</firstname>
   <lastname>Jones</lastname>
   <employeeId>10</employeeId>
</employee>
```

Types of NoSQL

- Graph
  - ➔ This type of database stores data in the form of networks, e.g., social connections, family trees, etc.
  - ➔ Its beauty lies in the way it stores the data: using a graph structure for semantic queries and representing it in the form of edges and nodes.
  - ➔ Nodes are leaf information that represent the entity, and the relationship (or relationships) between two nodes is defined using edges.
  - ➔ In the real world, our relationship to every other individual is different which can be distinguished by various attributes, at the edges level.



SQL vs NoSQL

- Terminology:
  - ➔ collection vs. table
  - ➔ document vs. row and
  - ➔ field vs. column

SQL	MongoDB
Table	Collection
Row	Document
Column	Field
Primary key	ObjectId
Index	Index
View	View
Nested table or object	Embedded document
Array	Array

- Core difference
  - ➔ The core difference comes from the fact that relational databases define columns at the table level whereas a document-oriented database defines its fields at the document level.
  - ➔ That is to say that each document within a collection can have its own unique set of fields.
  - ➔ As such, a collection is a dumbed down container in comparison to a table, while a document has a lot more information than a row.

INSTITUTO POLITÉCNICO DE BRAGANÇA

SQL vs NoSQL

- RDBMS: It is a structured data that provides more functionality but gives less performance.
- NoSQL: Structured or semi structured data, less functionality and high performance
  - ➔ We can't have constraints in NoSQL
  - ➔ Joins are not supported in NoSQL

- SQL vs NoSQL
  - ➔ SQL databases are relational, NoSQL are non-relational.
  - ➔ SQL databases use structured query language and have a predefined schema. NoSQL data bases have dynamic schemas for unstructured data.
  - ➔ SQL data bases are vertically scalable, NoSQL data bases are horizontally scalable.
  - ➔ SQL databases are table based, while NoSQL databases are document, key-value, graph or wide-column stores.
  - ➔ SQL databases are better for multi-row transactions, NoSQL are better for unstructured data like documents or JSON.

- When to choose NoSQL over relational database:
  - ➔ When you want to store and retrieve huge amount of data.
  - ➔ The relationship between the data you store is not that important
  - ➔ The data is not structured and changing over time
  - ➔ Constraints and Joins support is not required at database level
  - ➔ The data is growing continuously and you need to scale the database regular to handle the data

MongoDB

- Simple concepts we need to understand:
  1. MongoDB has the same concept of a database with which you are likely already familiar (or a schema in Oracle).
     - ⇨ Within a MongoDB instance you can have zero or more databases, each acting as high-level containers for everything else.
  2. A database can have zero or more collections. A collection shares enough in common with a traditional table that you can safely think of the two as the same thing.
  3. Collections are made up of zero or more documents. Again, a document can safely be thought of as a row.
  4. A document is made up of one or more fields, which you can probably guess are a lot like columns.
  5. Indexes in MongoDB function mostly like their RDBMS counterparts.
  6. Cursors are different than the other five concepts.
     - ⇨ The important thing to understand about cursors is that when you ask MongoDB for data, it returns a pointer to the result set called a cursor, which we can do things to, such as counting or skipping ahead, before actually pulling down data.
- Summary:
  - ➔ MongoDB is made up of databases which contain collections.
  - ➔ A collection is made up of documents. Each document is made up of fields.
  - ➔ Collections can be indexed, which improves lookup and sorting performance.
  - ➔ Finally, when we get data from MongoDB we do so through a cursor whose actual execution is delayed until necessary.

MongoDB: Databases and Collections

- Databases and Collections

  MongoDB stores data records as documents (specifically BSON documents) which are gathered together in collections. A database stores one or more collections of documents.

1. Databases

➔ In MongoDB, databases hold one or more collections of documents. To select a database to use, in the mongo shell, issue the use <db> statement, as in the following example:

```
use myDB
```

➔ Create a Database

⇨ If a database does not exist, MongoDB creates the database when you first store data for that database. As such, you can switch to a non-existent database and perform the following operation in the mongo shell:

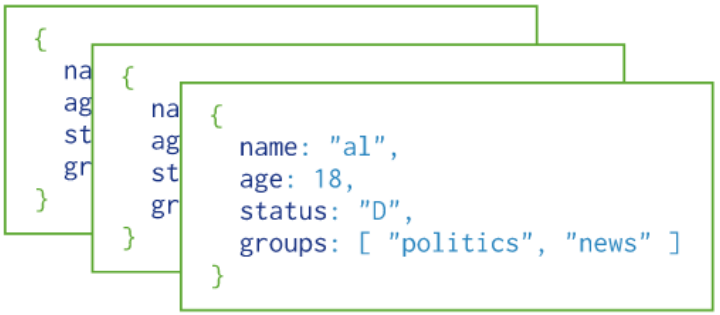
```
use myNewDB
db.myNewCollection1.insertOne( { x: 1 } )
```

⇨ The insertOne() operation creates both the database myNewDB and the collection myNewCollection1 if they do not already exist. Be sure that both the database and collection names follow MongoDB Naming Restrictions.

For example I am creating a database “dbtest” so the command should be:
`use dbtest`
Show database name:
`db`
To list down all the databases, use the command `show dbs`
Now we are creating a collection **user** and inserting a document in it:
`db.user.insert({name: "Joana", age: 30})`

MongoDB: Databases and Collections

- Collections
  - ➔ MongoDB stores documents in collections.
  - ➔ Collections are analogous to **tables in relational databases**.



Collection

  - ➔ Create a Collection
    - ⇨ If a collection does not exist, MongoDB creates the collection when you first store data for that collection.

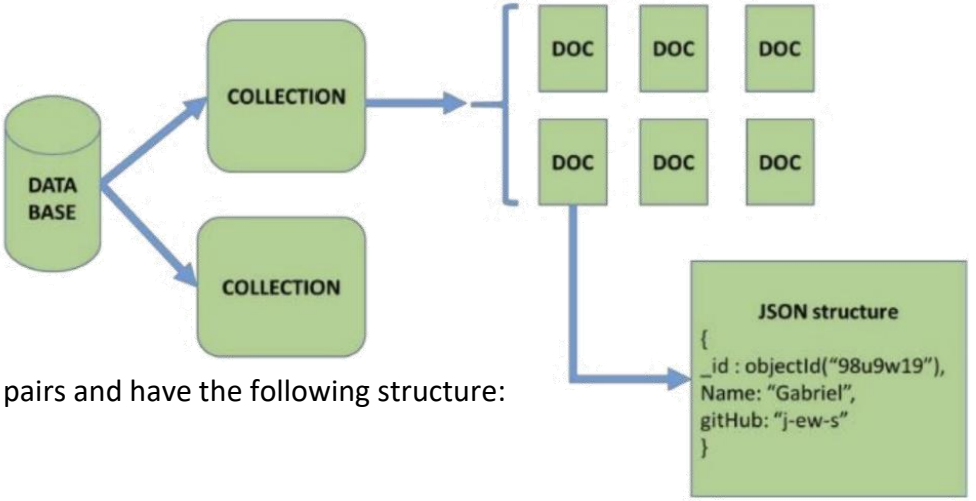
```
db.myNewCollection2.insertOne( { x: 1 } )
db.myNewCollection3.createIndex( { y: 1 } )
```

    - ⇨ Both the insertOne() and the createIndex() operations create their respective collection if they do not already exist. Be sure that the collection name follows MongoDB Naming Restrictions.
  - ➔ Explicit Creation
    - ⇨ MongoDB provides the db.createCollection() method to explicitly create a collection with various options, such as setting the maximum size or the documentation validation rules.
    - ⇨ If you are not specifying these options, you do not need to explicitly create the collection since MongoDB creates new collections when you first store data for the collections.

MongoDB: Documents

- Documents
  - ➔ MongoDB stores data records as BSON documents: BSON is a binary representation of JSON documents, though it contains more data types than JSON.
    - ⇨ Example of a document (rows in RDBMS)

```
{
  name: “Joana",
  age: 30,
  website: “www.estig.ipb.pt",
  hobbies: ["Teaching", "Watching TV"]
}
```



  - ➔ Document Structure
    - ⇨ MongoDB documents are composed of field-and-value pairs and have the following structure:

```
{
    field1: value1,
    field2: value2,
...
    fieldN: valueN
}
```

    - ⇨ The value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents. For example, the following document contains values of varying types:

```
var mydoc = {
     _id: ObjectId("5099803df3f4948bd2f98391"),
     name: { first: "Alan", last: "Turing" },
     birth: new Date('Jun 23, 1912'),
     contribs: [ "Turing machine", "Turing test", "Turingery" ],
     views : NumberLong(1250000)
            }
```

Fields have the following data types:
- _id holds an ObjectId.
- name holds an embedded document that contains the fields first and last.
- birth hold values of the Date type.
- contribs holds an array of strings.
- views holds a value of the NumberLong type.

MongoDB: JSON and XML

- JSON
  - ➔ JSON: JavaScript Object Notation.
  - ➔ JSON is a syntax for storing and exchanging data.
  - ➔ JSON is text, written with JavaScript object notation.

  - ➔ Exchanging Data
    - ⇨ When exchanging data between a browser and a server, the data can only be text.
    - ⇨ JSON is text, and we can convert any JavaScript object into JSON, and send JSON to the server.
    - ⇨ We can also convert any JSON received from the server into JavaScript objects.
    - ⇨ This way we can work with the data as JavaScript objects, with no complicated parsing and translations.
  - ➔ Example (https://www.w3schools.com/js/js_json_intro.asp):

```
{"employees":[
  { "firstName":"John", "lastName":"Doe" },
  { "firstName":"Anna", "lastName":"Smith" },
  { "firstName":"Peter", "lastName":"Jones" }
]}
```

- **BSON is a binary serialization format used to store documents and make remote procedure calls in MongoDB.**

MongoDB: JSON and XML

- XML
  - ➔ XML stands for eXtensible Markup Language.
  - ➔ XML was designed to store and transport data.
  - ➔ XML was designed to be both human- and machine-readable.
  - ➔ XML is a software- and hardware-independent tool for storing and transporting data.

  - ➔ What is XML?
    - ⇨ XML stands for eXtensible Markup Language
    - ⇨ XML is a markup language much like HTML
    - ⇨ XML was designed to store and transport data
    - ⇨ XML was designed to be self-descriptive
    - ⇨ Syntax:

```
<root>
  <child>
     <subchild>.....</subchild>
  </child>
</root>
```

  - ➔ https://www.w3schools.com/xml/default.asp

```
<?xml version="1.0" encoding="UTF-8"?>
<breakfast_menu>
<food>
    <name>Belgian Waffles</name>
    <price>$5.95</price>
    <description>
   Two of our famous Belgian Waffles with plenty
of real maple syrup
   </description>
    <calories>650</calories>
</food>

<food>
    <name>French Toast</name>
    <price>$4.50</price>
    <description>
    Thick slices made from our homemade
sourdough bread
    </description>
    <calories>600</calories>
</food>
<food>
    <name>Homestyle Breakfast</name>
    <price>$6.95</price>
    <description>
    Two eggs, bacon or sausage, toast, and our
ever-popular hash browns
    </description>
    <calories>950</calories>
</food>
</breakfast_menu>
```

MongoDB: JSON and XML

- JSON vs XML

The following JSON and XML examples both define an employees object, with an array of 3 employees:

➔ JSON Example

```
{"employees":[
  { "firstName":"John", "lastName":"Doe" },
  { "firstName":"Anna", "lastName":"Smith" },
  { "firstName":"Peter", "lastName":"Jones" }
]}
```

➔ XML Example

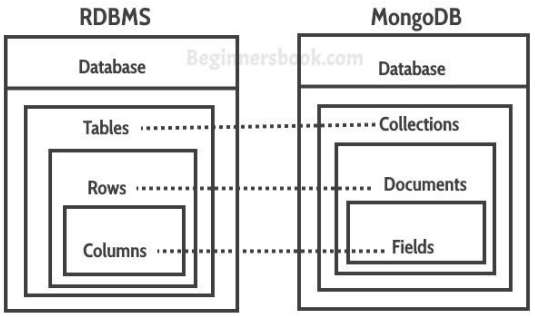
```
<employees>
  <employee>
    <firstName>John</firstName> <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName> <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName> <lastName>Jones</lastName>
  </employee>
</employees>
```


BSON

- BSON means Binary JSON, which, in other words, means binary-encoded serialization for JSON documents.
- If we compare BSON to the other binary formats, BSON has the advantage of being a model that allows you more flexibility. Also, one of its characteristics is that it's lightweight—a feature that is very important for data transport on the Web.
- The BSON format was designed to be easily navigable and both encoded and decoded in a very efficient way for most of the programming languages that are based on C. This is the reason why BSON was chosen as the data format for MongoDB disk persistence.
- The types of data representation in BSON are:
  - ➔ String UTF-8 (string)
  - ➔ Integer 32-bit (int32)
  - ➔ Integer 64-bit (int64)
  - ➔ Floating point (double)
  - ➔ Document (document)
  - ➔ Array (document)
  - ➔ Binary data (binary)
  - ➔ Boolean false (\x00 or byte 0000 0000)
  - ➔ Boolean true (\x01 or byte 0000 0001)
  - ➔ UTC datetime (int64)—the int64 is UTC milliseconds since the Unix epoch
  - ➔ Timestamp (int64)—this is the special internal type used by MongoDB replication and sharding; the first 4 bytes are an increment, and the last 4 are a timestamp
  - ➔ Null value ()
  - ➔ Regular expression (cstring)
  - ➔ JavaScript code (string)
  - ➔ JavaScript code w/scope (code_w_s)
  - ➔ Min key()—the special type that compares a lower value than all other possible
  - ➔ BSON element values
  - ➔ Max key()—the special type that compares a higher value than all other possible

Mapping Relational Databases to MongoDB

SQL	MongoDB
Table	Collection
Row	Document
Column	Field
Primary key	ObjectId
Index	Index
View	View
Nested table or object	Embedded document
Array	Array



- Table vs Collection

How a table in relational database looks in MongoDB.

➔ Columns are represented as key-value pairs (JSON Format),

➔ rows are represented as documents.

MongoDB automatically inserts a unique _id (12-byte field) field in every document, this serves as primary key for each document.

Relational Database

Student_Id	Student_Name	Age	College
1001	Chaitanya	30	Beginnersbook
1002	Steve	29	Beginnersbook
1003	Negan	28	Beginnersbook

MongoDB

```
{
 "_id": ObjectId("....."),
 "Student_Id": 1001,
 "Student_Name": "Chaitanya",
 "Age": 30,
 "College": "Beginnersbook"
}
{
 "_id": ObjectId("....."),
 "Student_Id": 1002,
 "Student_Name": "Steve",
 "Age": 29,
 "College": "Beginnersbook"
}
{
 "_id": ObjectId("....."),
 "Student_Id": 1003,
 "Student_Name": "Negan",
 "Age": 28,
 "College": "Beginnersbook"
}
```

Mapping Relational Databases to MongoDB

```
[
        {
          first_name: “Paul”,
          surname: “Miller”,
          city: “London”,
          cars: [
                        {
                         model: “Bentley”,
                         year: 1973,
                         value: 100000
                        },
                        {
                         model: “RollsRoyce”,
                         year: 1965,
                         value: 330000
                        }
                        ]
},
        {
         first_name: “Urs”,
         surname: “Huber”,
         city: “Zurich”,
         cars:
                        {
                         model: “Smart”,
                         year: 1999,
                         value: 2000
                        }
        }
]
```

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 CRUD Operations

- MongoDB CRUD Operations: create, read, update, and delete documents

1. Create Operations
   - ➔ Create or insert operations add new documents to a collection. If the collection does not currently exist, insert operations will create the collection.
   - ➔ MongoDB provides the following methods to insert documents into a collection:

     `db.collection.insertOne()` New in version 3.2

     `db.collection.insertMany()` New in version 3.2
   - ➔ In MongoDB, insert operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

```
db.products.insertMany( [
   { item: "card", qty: 15 },
   { item: "envelope", qty: 20 },
   { item: "stamps" , qty: 30 }
] )
```

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

Use `insertOne` to insert only one record:

```
db.people.insertOne({name: 'Tom', age: 28});
```

Use `insertMany` to insert multiple records:

```
db.people.insertMany([{name: 'Tom', age: 28},{name: 'John', age: 25}, {name: 'Kathy', age: 23}])
```

MongoDB CRUD Operations

2. Read Operations

➔ Read operations retrieve documents from a collection; i.e. query a collection for documents. MongoDB provides the following methods to read documents from a collection:

```
db.collection.find()
```

➔ We can specify query filters or criteria that identify the documents to return.

```
db.users.find(                          ← collection
   { age: { $gt: 18 } },                ← query criteria
   { name: 1, address: 1 }              ← projection
).limit(5)                              ← cursor modifier
```

Query for all the docs in the `people` collection that have a `name` field with a value of `'Tom'`:

```
db.people.find({name: 'Tom'})
```

Or just the first one:

```
db.people.findOne({name: 'Tom'})
```

You can also specify which fields to return by passing a field selection parameter. The following will exclude the `_id` field and only include the `age` field:

```
db.people.find({name: 'Tom'}, {_id: 0, age: 1})
```

MongoDB CRUD Operations

INSTITUTO POLITÉCNICO DE BRAGANÇA

```
db.student.update(
    {name: 'Tom'}, // query criteria
    {$set: {sex: 'F'}} // update action
);
```

```
db.student.update(
    {name: 'Tom'}, // query criteria
    {$set: {sex: 'F', age: 40}} // update action
);
```

3. Update Operations

➔ Update operations modify existing documents in a collecti methods to update documents of a collection:

`db.collection.updateOne()` New in version 3.2

`db.collection.updateMany()` New in version 3.2

`db.collection.replaceOne()` New in version 3.2

➔ In MongoDB, update operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

➔ You can specify criteria, or filters, that identify the documents to update. These filters use the same syntax as read operations.

Update the **entire** object:

```
db.people.update({name: 'Tom'}, {age: 29, name: 'Tom'})

// New in MongoDB 3.2
db.people.updateOne({name: 'Tom'},{age: 29, name: 'Tom'}) //Will replace only first matching
document.

db.people.updateMany({name: 'Tom'},{age: 29, name: 'Tom'}) //Will replace all matching documents.
```

Or just update a single field of a document. In this case age:

```
db.people.update({name: 'Tom'}, {$set: {age: 29}})
```

You can also update multiple documents simultaneously by adding a third parameter. This query will update all documents where the name equals Tom:

```
db.people.update({name: 'Tom'}, {$set: {age: 29}}, {multi: true})

// New in MongoDB 3.2
db.people.updateOne({name: 'Tom'},{$set:{age: 30}}) //Will update only first matching document.

db.people.updateMany({name: 'Tom'},{$set:{age: 30}}) //Will update all matching documents.
```

```
db.users.updateMany(                          ← collection
   { age: { $lt: 18 } },                      ← update filter
   { $set: { status: "reject" } }             ← update action
)
```

```
db.restaurant.updateOne(
   { "name" : "Central Perk Cafe" },   → Filter
   { $set: { "violations" : 3 } }      → Update document
)
```

If a document is needed to be replaced,

```
db.collection.replaceOne({name:'Tom'}, {name:'Lakmal',age:25,address:'Sri Lanka'})
```

MongoDB CRUD Operations

3. Update Operations (Update of embedded documents)

- For the following schema:

```
{name: 'Tom', age: 28, marks: [50, 60, 70]}
```

➔ Update Tom's marks to 55 where marks are 50 (Use the positional operator $):

```
db.people.update({name: "Tom", marks: 50}, {"$set": {"marks.$": 55}})
```

- For the following schema:

```
{name: 'Tom', age: 28, marks: [{subject: "English", marks: 90},{subject: "Maths", marks: 100},
{subject: "Computes", marks: 20}]}
```

➔ Update Tom's English marks to 85 :

```
db.people.update({name: "Tom", "marks.subject": "English"},{"$set":{"marks.$.marks": 85}})
```

- Explaining above example:

➔ By using {name: "Tom", "marks.subject": "English"} you will get the position of the object in the marks array, where subject is English. In "marks.$.marks", $ is used to update in that position of the marks array

MongoDB CRUD Operations

3. Update Operations (Update Values in an Array)

- The positional $ operator identifies an element in an array to update without explicitly specifying the position of the element in the array.
- Consider a collection students with the following documents:

```
{ "_id" : 1, "grades" : [ 80, 85, 90 ] }
{ "_id" : 2, "grades" : [ 88, 90, 92 ] }
{ "_id" : 3, "grades" : [ 85, 100, 90 ] }
```

➔ To update 80 to 82 in the grades array in the first document, use the positional $ operator if you do not know the position of the element in the array:

```
db.students.update(
   { _id: 1, grades: 80 },
   { $set: { "grades.$" : 82 } }
                )
```

MongoDB CRUD Operations

3. Delete Operations

- Delete operations remove documents from a collection. MongoDB provides the following methods to delete documents of a collection:

```
db.collection.deleteOne() New in version 3.2
db.collection.deleteMany() New in version 3.2
```

- In MongoDB, delete operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.
- You can specify criteria, or filters, that identify the documents to remove. These filters use the same syntax as read operations.

```
db.orders.deleteOne( { "expiryts" : {$lt: ISODate("2015-11-01T12:40:15Z")}})
```

Deletes all documents matching the query parameter:

```
// New in MongoDB 3.2
db.people.deleteMany({name: 'Tom'})

// All versions
db.people.remove({name: 'Tom'})
```

Or just one

```
// New in MongoDB 3.2
db.people.deleteOne({name: 'Tom'})

// All versions
db.people.remove({name: 'Tom'}, true)
```

```
db.users.deleteMany(        <-- collection
  { status: "reject" }      <-- delete filter
)
```

MongoDB's `remove()` method. If you execute this command without any argument or without empty argument it will remove all documents from the collection.

```
db.people.remove();
```

or

```
db.people.remove({});
```

MongoDB: Data models

- Document Database
  - ➔ A record in MongoDB is a document, which is a data structure composed of field and value pairs.
  - ➔ MongoDB documents are similar to JSON objects.
  - ➔ The values of fields may include other documents, arrays, and arrays of documents.

  - ➔ The advantages of using documents are:
    - ⇨ Documents (i.e. objects) correspond to native data types in many programming languages.
    - ⇨ Embedded documents and arrays reduce need for expensive joins.
    - ⇨ Dynamic schema supports fluent polymorphism.

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

MongoDB: Data models

- Document Structure
  - ➔ The key decision in designing data models for MongoDB applications revolves around the structure of documents and how the application represents relationships between data.
  - ➔ MongoDB allows related data to be embedded within a single document.

- Embedded Data
  - ➔ Embedded documents capture relationships between data by storing related data in a single document structure.
  - ➔ MongoDB documents make it possible to embed document structures in a field or array within a document.
  - ➔ These denormalized data models allow applications to retrieve and manipulate related data in a single database operation.

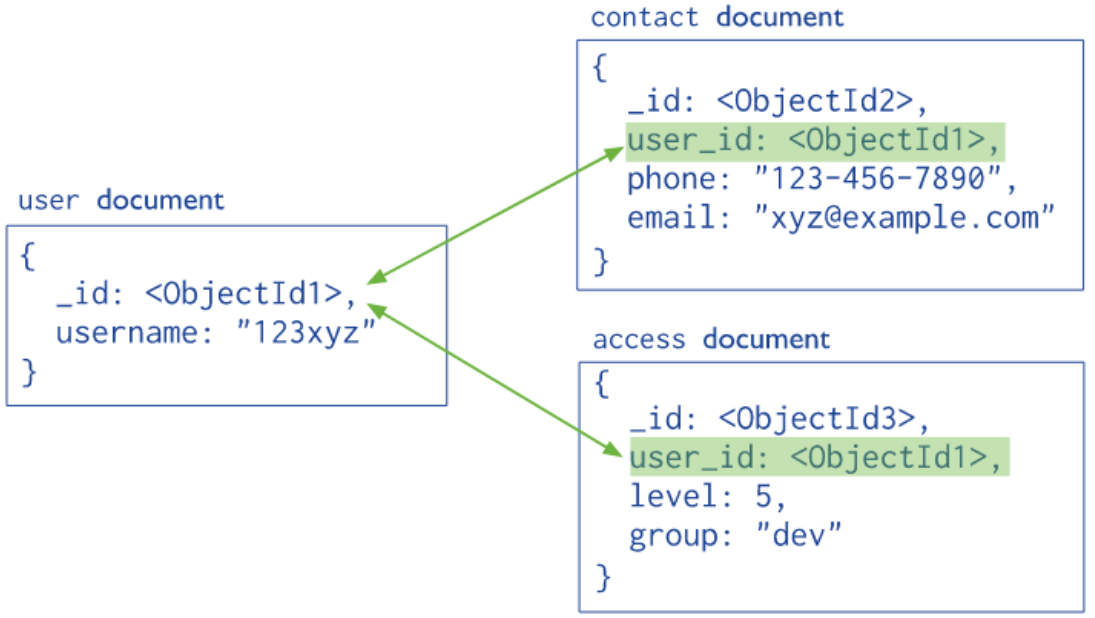
```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
             phone: "123-456-7890",
             email: "xyz@example.com"
           },
  access: {
            level: 5,
            group: "dev"
          }
}
```

Embedded sub-document

Embedded sub-document

MongoDB: Data models

- References (Normalized Data Models)
  - ➔ References store the relationships between data by including links or references from one document to another.
  - ➔ Applications can resolve these references to access the related data.
  - ➔ Broadly, these are normalized data models: Normalized data models describe relationships using references between documents.



MongoDB: Data models

- **Model One-to-One Relationships with Embedded Documents**

In the normalized data model, the address document contains a reference to the patron document.

```
// patron document
{
  _id: "joe",
  name: "Joe Bookreader"
}
// address document
{
  patron_id: "joe", // reference to patron document
  street: "123 Fake Street",
  city: "Faketon",
  state: "MA",
  zip: "12345"
}
```

- **Model One-to-Many Relationships with Embedded Documents**

In the normalized data model, the address documents contain a reference to the patron document

```
// patron document
{
  _id: "joe",
  name: "Joe Bookreader"
}
// address documents
{
  patron_id: "joe", // reference to patron document
  street: "123 Fake Street",
  city: "Faketon",
  state: "MA",
  zip: "12345"
}
{
  patron_id: "joe",
  street: "1 Some Other Street",
  city: "Boston",
  state: "MA",
  zip: "12345"
}
```

MongoDB: Data models

- MongoDB structure

```
{
  "first_name": "Paul",
  "surname": "Miller",
  "cell": "447557505611",
  "city": "London",
  "location": [45.123, 47.232],
  "profession": ["banking", "finance", "trader"],
  "cars": [
  {
    "model": "Bentley",
    "year": 1973
  },
  {
    "model": "RollsRoyce",
    "year": 1965
  }
  ]
}
```

Users

ID	first_name	surname	cell	city	location_x	location_y
1	Paul	Miller	447557505611	London	45.123	47.232

Professions

ID	user_id	profession
10	1	banking
11	1	finance
12	1	trader

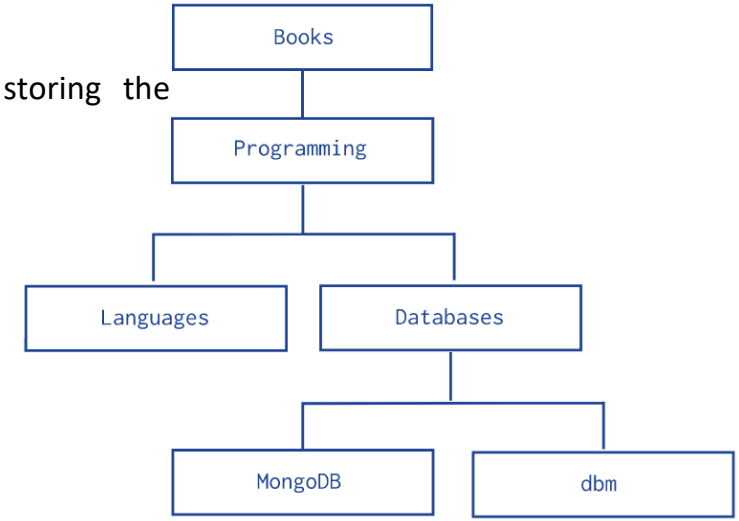
Cars

ID	user_id	model	year
20	1	Bentley	1973
21	1	Rolls Royce	1965

MongoDB: Data models

- Model Tree Structures with Parent References
  - ➔ Use a data model that describes a tree-like structure in MongoDB documents by storing references to "parent" nodes in children nodes.
  - ➔ Pattern
    - ⇨ The Parent References pattern stores each tree node in a document; in addition to the tree node, the document stores the id of the node's parent.
    - ⇨ Consider the hierarchy of categories.
    - ⇨ The following example models the tree using Parent References, storing the reference to the parent category in the field parent:

```
db.categories.insertMany( [
  { _id: "MongoDB", parent: "Databases" },
  { _id: "dbm", parent: "Databases" },
  { _id: "Databases", parent: "Programming" },
  { _id: "Languages", parent: "Programming" },
  { _id: "Programming", parent: "Books" },
  { _id: "Books", parent: null }
] )
```



→ The query to retrieve the parent of a node is fast and straightforward:

```
db.categories.findOne( { _id: "MongoDB" } ).parent
```

→ You can create an index on the field parent to enable fast search by the parent node:

```
db.categories.createIndex( { parent: 1 } )
```

→ You can query by the parent field to find its immediate children nodes:

```
db.categories.find( { parent: "Databases" } )
```

MongoDB – Getting database information

- List all collections in database
  - ➔ `show collections`

  or
  - ➔ `show tables`

  or
  - ➔ `db.getCollectionNames()`

- List all databases
  - ➔ `show dbs`

  or
  - ➔ `db.adminCommand('listDatabases')`

  or
  - ➔ `db.getMongo().getDBNames()`

MongoDB Query and Projection Operators

- Comparison

Name	Description
$eq	Matches values that are equal to a specified value.
$gt	Matches values that are greater than a specified value.
$gte	Matches values that are greater than or equal to a specified value.
$in	Matches any of the values specified in an array.
$lt	Matches values that are less than a specified value.
$lte	Matches values that are less than or equal to a specified value.
$ne	Matches all values that are not equal to a specified value.
$nin	Matches none of the values specified in an array.

- Logical

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

- Element

Name	Description
$exists	Matches documents that have the specified field.
$type	Selects documents if a field is of the specified type.

- Evaluation

Name	Description
$expr	Allows use of aggregation expressions within the query language.
$jsonSchema	Validate documents against the given JSON Schema.
$mod	Performs a modulo operation on the value of a field and selects documents with a specified result.
$regex	Selects documents where values match a specified regular expression.
$text	Performs text search.
$where	Matches documents that satisfy a JavaScript expression.

- Array

Name	Description
$all	Matches arrays that contain all elements specified in the query.
$elemMatch	Selects documents if element in the array field matches all the specified $elemMatch conditions.
$size	Selects documents if the array field is a specified size.

https://docs.mongodb.com/manual/reference/operator/query/

MongoDB Database Commands

- Aggregation Commands

Name	Description
aggregate	Performs aggregation tasks such as group using the aggregation framework.
count	Counts the number of documents in a collection or a view.
distinct	Displays the distinct values found for a specified key in a collection or a view.
mapReduce	Performs map-reduce aggregation for large data sets.

- Query and Write Operation Commands

Name	Description
delete	Deletes one or more documents.
find	Selects documents in a collection or a view.
findAndModify	Returns and modifies a single document.
getLastError	Returns the success status of the last operation.
getMore	Returns batches of documents currently pointed to by the cursor.
insert	Inserts one or more documents.
resetError	Deprecated. Resets the last error status.
update	Updates one or more documents.

MongoDB - Querying for Data

- Find()
  - ➔ retrieve all documents in a collection

```
db.collection.find({});
```

  - ➔ retrieve documents in a collection using a condition (similar to WHERE in MYSQL)

```
db.collection.find({key: value});
Example:
        db.users.find({email:"sample@email.com"});
```

  - ➔ retrieve documents in a collection using Boolean conditions (Query Operators)

```
//AND
db.collection.find( {
        $and: [
          { key: value }, { key: value }
        ]
})
//OR
db.collection.find( {
        $or: [
          { key: value }, { key: value }
        ]
})
//NOT
db.inventory.find( { key: { $not: value } } )
```

  - ➔ NOTE: find() will keep on searching the collection even if a document match has been found , therefore it is inefficient when used in a large collection , however by carefully modeling your data and/or using indexes you can increase the efficiency of find()

MongoDB - Querying for Data

- To skip first 3 documents:
  - ➔ db.test.find({}).skip(3)
- To sort descending by the field name (If we want to sort ascending just replace -1 with 1):
  - ➔ db.test.find({}).sort({ "name" : -1})
- To count the results:
  - ➔ db.test.find({}).count()
- Also combinations of this methods are allowed. For example get 2 documents from descending sorted collection skipping the first 1:
  - ➔ db.test.find({}).sort({ "name" : -1}).skip(1).limit(2)

MongoDB - Querying for Data

MongoDB1	mySQL
db.students.find().pretty()	**SELECT** * **FROM** students
db.students.find({firstName:"Prosen"})	**SELECT** * **FROM** students **WHERE** firstName = "Prosen";

db.students.find({
"firstName": "Prosen",
"age": {
"$gte": 23
}
});
	SELECT * **FROM** students
	WHERE firstName = "Prosen"
	AND age >= 23

db.students.find({
"$or": [{
"firstName": "Prosen"
}, {
"age": {
"$gte": 23
}
}]
});
	SELECT * **FROM** students
	WHERE firstName = "Prosen"
	OR age >= 23

db.students.find({ firstName : "Prosen", $or : [{age : 23}, {age : 25}] });	**SELECT** * **FROM** students **WHERE** firstName = "Prosen" **AND** age = 23 **OR** age = 25;
db.students.find(lastName:{$in:["Ghosh", "Amin"]})	SELECT * FROM students WHERE lastName IN ('Ghosh', 'Amin')

MongoDB - Collections

- Create a Collection
  - ➔ First Select Or Create a database. Switch to db mydb:
    use mydb
  - ➔ Using db.createCollection("yourCollectionName") method you can explicitly create Collection.
    db.createCollection("newCollection1")
    ⇨ Message: { "ok" : 1 }
  - ➔ Using show collections command see all collections in the database.
    show collections
    ⇨ Message: newCollection1
- Drop Collection
  - ➔ MongoDB's db.collection.drop() is used to drop a collection from the database. First, check the available collections into your database mydb.
    use mydb
    → switched to db mydb
    show collections
    → newCollection1
    → newCollection2
    → newCollection3
    → system.indexes
  - ➔ Now drop the collection with the name newCollection1.
    db.newCollection1.drop()

MongoDB - examples

- Basic commands:
  - ➔ Commands that you execute against the current database are executed against the db object, such as db.help() or db.stats().
  - ➔ Commands that you execute against a specific collection, which is what we'll be doing a lot of, are executed against the db.COLLECTION_NAME object, such as db.unicorns.help() or db.unicorns.count().
  - ➔ If you enter db.help(), you'll get a list of commands that you can execute against the db object.

- Example:
  - ➔ Database commands, like db.getCollectionNames().
    - ⇨ If you do so, you should get an empty array ([ ]). Since collections are schema-less, we don't explicitly need to create them.
  - ➔ We can simply insert a document into a new collection. To do so, use the insert command, supplying it with the document to insert:

```
db.students.insert({name: 'Joana', gender: 'f', weight: 450})
```

  - ➔ We can now use the find command against unicorns to return a list of documents:

```
db.students.find()
```

  - ➔ Every document must have a unique _id field:

```
db.students.getIndexes()
```

  - ➔ Insert a totally different document into STUDENTS:

```
db.students.insert({name: 'Joao',
                    gender: 'm',
                    home: 'Lisboa',
                    have_car: false})
```

This line is executing insert against the STUDENTS collection, passing it a single parameter. Internally MongoDB uses a binary serialized JSON format called BSON

MongoDB – Example 1

- Find a count of the documents in the collection:

```
db.postalCodes.count()
```

- Let's find just one document from the postalCodes collection as follows:

```
db.postalCodes.findOne()
```

- Now, we find multiple documents in the collection as follows:

```
db.postalCodes.find().pretty()
```

- The preceding query retrieves all the keys of the first 20 documents and displays them on the shell. At the end of the result, you will notice a line that says Type "it“ for more. By typing "it", the mongo shell will iterate over the resulting cursor.
  - ➔ Let's do a couple of things now; we will just display the city, state, and pincode fields. Additionally, we want to display the documents numbered 91 to 100 in the collection. Let's see how we do this:

```
db.postalCodes.find({}, { _id:0 , city:1 , state:1 , pincode:1 }).
skip(90).limit(10)
```

- Let's move a step ahead and write a slightly complex query where we find the top 10 cities in the state of Gujarat sorted by the name of the city, and, similar to the last query, we just select city, state, and the pincode field:

```
db.postalCodes.find({state:'Gujarat'},{ _id:0 , city:1 , state:1 ,
pincode:1 }).sort( {city:1}).limit(10)
```

MongoDB – Example 1

- We will now execute the following query on the mongo shell:

```
db.postalCodes.find({},  {_id:0,  city:1,  state:1,  pincode:1}).skip(90).
limit(10)
```

- Here, we pass two parameters to the find method:
  - ➔ The first one is {}, which is the query to select the documents, and, in this case, we ask mongo to select all the documents.
  - ➔ The second parameter is the set of fields that we want in the result documents also known as projection. Remember that the _id field is present by default unless we explicitly say _id:0. For all the other fields, we need to say <field_name>:1 or <field_name>:true. The find portion with projections is the same as saying select field1, field2 from table in a relational world, and not specifying the fields to be selected in the find is saying select * from table in a relational world.

- Moving on, we just need to look at what skip and limit do:
  - ➔ The skip function skips the given number of documents from the result set all the way up to the end document
  - ➔ The limit function then limits the result to the given number of documents

- Let's see what this all means with an example:
  - ➔ By doing .skip(90).limit(10), we say that we want to skip the first 90 documents from the result set and start returning from the 91st document.
  - ➔ The limit, however, says that we will be returning only 10 documents from the 91st document.

INSTITUTO POLITÉCNICO DE BRAGANÇA

Indexing and its importance on all database (SQL vs NoSQL)
Aggregation
Sharding and replication
Clustering

Indexing documents

- Indexes are used to improve the performance of the query.
  - ➔ Without indexes, MongoDB must search the entire collection to select those documents that match the query statement.
  - ➔ MongoDB therefore uses indexes to limit the number of documents that it must scan.

- Indexes are special data structures that store a small portion of the collection's data set in an easy-to-transform format.
  - ➔ The index stores a set of fields ordered by the value of the field. This ordering helps to improve the performance of equality matches and range-based query operations.

- MongoDB defines indexes at the collection level and indexes can be created on any field of the document.
  - ➔ MongoDB creates an index for the _id field by default.

- Note MongoDB creates a default unique index _id field, which helps to prevent inserting two documents with the same value of the _id field.

INSTITUTO POLITÉCNICO DE BRAGANÇA

Indexing documents

- But in case you feel that you are not familiar enough with the concept of indexes, an easy way to understand them is to draw a parallel with books.

- Suppose that we have a book with an index like this:

Index

Indexes

- 1) Working with Indexes
  - ➔ a) create an index
    - ⇨ Consider the following employee collection.

```
db.employee.insert({empId:1,empName:"John",state:"KA",country:"India"})
db.employee.insert({empId:2,empName:"Smith",state:"CA",country: "US"})
db.employee.insert({empId:3,empName:"James",state:"FL",country:"US"})
db.employee.insert({empId:4,empName:"Josh",state:"TN",country:"India"})
db.employee.insert({empId:5,empName:"Joshi",state:"HYD",country:"India"})
```

  - ➔ To create single-field index on the empId field, use the following command.

```
db.employee.createIndex({empId:1})
```

Here, the parameter value “1” indicates that empId field values will be stored in ascending order.

- ⇨ Output:

```
{
"createdCollectionAutomatically" : false,
"numIndexesBefore" : 1,
"numIndexesAfter" : 2,
"ok" : 1
}
```

  - ➔ To create an index on multiple fields, known as a compound index, use this command.

```
db.employee.createIndex({empId:1,empName:1})
```

- ⇨ Output:

```
{
"createdCollectionAutomatically" : false,
"numIndexesBefore" : 2,
"numIndexesAfter" : 3,
"ok" : 1
}
```


Indexes

- 1) Working with Indexes
  - ➔ b) Display a list of indexes
    - ⇨ To display a list of indexes, this is the syntax:

```
db.employee.getIndexes()
```

- ⇨ Output:

```
[
        {
                "v" : 2,
                "key" : {"_id" : 1},
                "name" : "_id_",
                "ns" : "test.employee"
        },
        {
                "v" : 2,
                "key" : {"empId" : 1},
                "name" : "empId_1",
                "ns" : "test.employee"
        },
        {
                "v" : 2,
                "key" : {"empId" : 1,"empName" : 1},
                "name" : "empId_1_empName_1",
                "ns" : "test.employee"
        }
]
```


Indexes

- 1) Working with Indexes
  - ➔ C) drop a compound index
    - ⇨ To drop a compound index, use the following command:

      db.employee.dropIndex({empId:1,empName:1})

      Output:

      { "nIndexesWas" : 3, "ok" : 1 }

    - ⇨ To drop all the indexes, use this command:

      db.employee.dropIndexes()

      Output:

      {
      "nIndexesWas" : 2,
      "msg" : "non-_id indexes dropped for collection",
      "ok" : 1
      }

    - ⇨ Note: We can't drop _id indexes. MongoDB creates an index for the _id field by default.

Indexes

- 2) Indexing Strategies
  - ➔ A) Create an Index to Support Your Queries
    - ⇨ Creating the right index to support the queries increases the query execution performance and results in great performance.
    - ⇨ Create a single-field index if all the queries use the same single key to retrieve the documents.

      db.employee.createIndex({empId:1})
    - ⇨ Create a multifield compound index if all the queries use more than one key (multiple filter condition) to retrieve the documents.

      db.employee.createIndex({empId:1,empName:1})
  - ➔ B) Using an Index to Sort the Query Results
    - ⇨ Sort operations use indexes for better performance. Indexes determine the sort order by fetching the documents based on the ordering in the index.
    - ⇨ Sorting can be done in the following scenarios:
      - → 1) Sorting with a single-field index.
      - → 2) Sorting on multiple fields.

Aggregation in MongoDB

- Aggregation is MongoDB's framework for performing advanced data analysis.
- It allows transformations, filtering, grouping, and computations across documents.
- Aggregation is similar to SQL's GROUP BY and JOIN operations, but more flexible.
- What is aggregation:
  - ➔ MongoDB aggregation processes data and returns computed results.
  - ➔ Key concepts:
    - ⇨ Documents flow through stages
    - ⇨ Each stage transforms the data
    - ⇨ You can filter, compute, reshape, lookup, unwind, and more
  - ➔ Primary aggregation methods:
    - ⇨ aggregate()
    - ⇨ mapReduce()
    - ⇨ single-purpose aggregation commands (count, distinct)

Aggregation Pipeline Overview

- The aggregation pipeline is a sequence of stages.

- General flow:
  - ➔ 1. Start with a collection
  - ➔ 2. Pass documents through stages
  - ➔ 3. Each stage outputs transformed documents
  - ➔ 4. Final stage returns results to the user

- Benefits:
  - ➔ Efficient: Uses MongoDB query engine
  - ➔ Flexible: Complex transformations
  - ➔ Expressive: Dozens of operators and stages

Common Pipeline Stages in Detail

- Important pipeline stages:
  - ➔ $match – Filters documents (similar to WHERE)
  - ➔ $group – Groups documents and performs accumulations ($sum, $avg, $max, $push)
  - ➔ $project – Reshapes documents (include/exclude fields, compute new fields)
  - ➔ $sort – Sorts documents by specified fields
  - ➔ $limit / $skip – Pagination
  - ➔ $lookup – Performs left-outer joins between collections
  - ➔ $unwind – Flattens arrays into individual documents
  - ➔ $addFields – Adds new computed fields

Detailed Example: Sales Aggregation

- Example: Summing customer purchases

```
db.sales.aggregate([
  {
    $match: { status: "complete" }
  },
  {
    $group: {
      _id: "$customer_id",
      totalSpent: { $sum: "$amount" },
      avgPurchase: { $avg: "$amount" },
      purchaseCount: { $sum: 1 }
    }
  },
  {
    $sort: { totalSpent: -1 }
  }
])
```

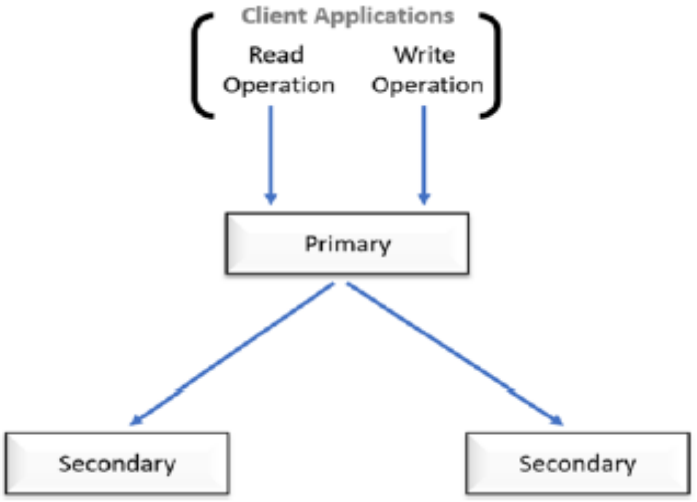
- Explanation:
  - ➔ $match filters only completed orders
  - ➔ $group aggregates totals per customer
  - ➔ $sort ranks customers by total spent

Replication

- Replication is the process of creating and managing a duplicate version of a database across servers to provide redundancy and increase availability of data.

- In MongoDB, replication is achieved with the help of a replica set, a group of mongod instances that maintain the same data set.
  - ➔ A replica set contains one primary node that is responsible for all write operations and
  - ➔ one or more secondary nodes that replicate the primary's oplog and apply the operations to their data sets to reflect the primary's data set.

- Next figure is an illustration of a replica set.



Sharding (Fragmentação)

- Sharding is a method for distributing data across multiple machines. There are two methods for addressing system growth: vertical and horizontal scaling.
  - ➔ Vertical scaling: We need to increase the capacity of a single server such as using a more powerful CPU, adding more RAM, or increasing the amount of storage space.
  - ➔ Horizontal scaling: We need to divide the data set and distribute the workload across the servers by adding additional servers to increase the capacity as required.

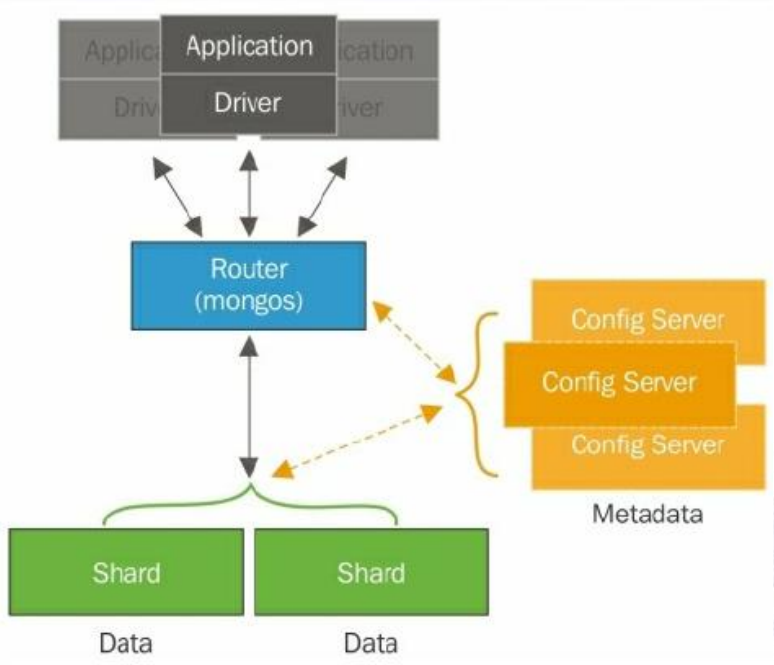
- Increasing the storage capacity, CPU, or memory can be very expensive and sometimes impossible due to our service provider's limitations. On the other hand, increasing the number of nodes in a system can also increase complexity both conceptually and operationally.

- MongoDB supports horizontal scaling through sharding. A MongoDB sharded cluster consists of the following components:
  - ➔ 1. Shard: Each shard contains a subset of the sharded data. Each shard can be deployed as a replica set.
  - ➔ 2. mongos: The mongos acts as a query router, providing an interface between client applications and the sharded cluster.
  - ➔ 3. Config servers: Config servers store metadata and configuration settings for the cluster.

INSTITUTO POLITÉCNICO DE BRAGANÇA

Sharding (Fragmentação)

- Sharding works in MongoDB at the collections level, which means we can have collections with sharding and without sharding enabled in the same database. To set sharding in a collection, we must configure a sharded cluster. The elements for a sharded cluster are shards, query routers, and configuration servers:
  - ➔ A **shard** is where a part of our data set will be allocated. A shard can be a MongoDB instance or a replica set
  - ➔ The **query router** is the interface offered for the database clients that will be responsible for directing the operations to the correct shard
  - ➔ The **config server** is a MongoDB instance that is responsible for keeping the sharded cluster configurations or, in other words, the cluster's metadata



INSTITUTO POLITÉCNICO DE BRAGANÇA

Sharding

- Sharding is a method to achieve horizontal scaling of a database via data partitioning between several machines.
  - ➔ This allows data and load to be spread among the various machines, providing high-throughput and supporting very large datasets, bigger than what could fit on a single server.

- Because of the focus on horizontal scalability these systems are useful for use with very large datasets (that don't fit in a single machine's memory) or loads bigger than one machine can easily handle by itself.
  - ➔ They're therefore good candidates for computational analysis with Geographical Information Systems (GIS) or other use cases where consistency isn't a hard requirement and capacity and performance are priorities.

- Some well-known examples of NoSQL DBMSs are MongoDB (used by several big tech companies such as Google, Facebook and ebay), Cassandra (created and used by Facebook), DynamoDB (part of Amazon Web Services) and BigTable (part of the Google Cloud platform).

INSTITUTO POLITÉCNICO DE BRAGANÇA

Sharding and replication

- While replication can help performance somewhat (by isolating long running queries to secondaries, and reducing latency for some other types of queries), its main purpose is to provide high availability.

- Sharding is the primary method for scaling MongoDB clusters. Combining replication with sharding is the prescribed approach to achieve scaling and high availability.