Server-side
Web Development

Unit 15. Annex MongoDB

Fidel Oltra, Ricardo Sánchez

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2023-24

Index

1 The NoSQL movement

The predominant technology in the market today, in terms of data storage systems, is still DBMS (relational databases), based on the use of tables to store data and the relationships between them, as well as the use of the query language and SQL data manipulation.

In any case, over time other database paradigms have appeared, such as those oriented to objects or based on XML storage, whose characteristics have been absorbed by the DBMSs themselves.

For some time now, the **NoSQL** movement has been gaining momentum, with the emergence of new alternative databases to traditional relational systems.

Some of the reasons that supporters of the **NoSQL** movement argue for the use of this type of database are:

- less complexity
- performance improvements
- horizontal scalability
- less hardware requirements
- relational mapping is eliminated, since the stored data structures are much closer to those of OOP languages, particularly in applications with key-value structures
- better performance in distributed databases

In this unit we will focus on the document-oriented database **MongoDB**, for being one of the most used today.

2 MongoDB

MongoDB is a document-oriented database, based on the storage of its data structures in JSON-type documents with a dynamic schema. Although it started development by the *10gen* company, today it's an open source project with a large community of users.

A MongoDB server can have several databases, and each of them has a set of collections, which we could equate to tables in a relational DB. Each collection stores a set of JSON documents, made up of key-value attributes, which would become the records of a relational database.

Broadly speaking, we could establish the following relationships:

Relational model	MongoDB
Relational BD	BD Oriented to documents
Table	Collection
Record/Queue	JSON Document
Attributes/Columns	JSON Document Keys

Let's see an example:

```
{
    _id: 1,
    episode: "IV",
    title: "A new Hope",
    year: 1977,
    director: {
        name: "George",
        surname:  "Lucas",
        birthdate: 1944
    }
},
{
     _id: 5,
    title: "Rogue One. A Star Wars Story.",
    year: 2016,
    director: {
        name: "Gareth",
        surname:  "Edwards",
        birthday: 1975,
        country: "UK"
    }
}
```

As we can see, each movie type document has its own attributes, which do not have to match each other. Another characteristic is that, as we can see, we have the information about the director in the document itself, and not in another related document, as would be the case with SQL tables.

2.1 MongoDB features

The main features of MongoDB are:

- **document-oriented** database: different documents or database objects can be easily mapped to application objects. In addition, the fact of being able to have documents embedded within others (like a director within a film), means that we don't need JOINs in queries. The dynamic scheme (that a document can have different fields to others) makes polymorphism simpler
- **high performance** as nested documents make reads and writes faster, indexes can include nested document keys and vectors, and in addition *streaming* (real-time data intake) writes can be performed when commits are not required the end of writing
- **high availability**, with replicated servers with automatic system failure management
- **easy scalability** with automatic *sharding* that distributes a collection's data across multiple machines, and reads can be distributed across replicated servers

Differences between MongoDB and Relational Databases:

- **greater flexibility**, not being subject to the strict definition of a schema. For example, when in a DBMS we need a new field for a specific row, we need to incorporate the field in all rows, while in MongoDB when we need to add a field to a document, we only need to add it to the document itself, without the others being altered
- **SQL language is not used** to make queries, while a JSON is passed as a parameter that describes what you want to retrieve
- **Mongo does not support JOINS**, offering multidimensional data types instead, such as arrays or other documents within the document itself. For example, a document can have an array with all related objects in another collection
- **Mongo does not offer transactions**, as each independent operation is performed atomically

2.1.1 Example

Let's say we want to have a database with information about movies and their directors. A movie can be directed by some directors, and a director may have directed a lot of movies. So, we have a M:M relation between the two entities, `Movie` and `Director`.

In a relational database we would had `Movie` and `Director` tables, with a third intermediate table representing the many-to-many relationship. This intermediate table would be composed of the two primary keys of the `Movie` and `Director` tables, being foreign keys to these tables.

In MongoDB we could have two collections: `Movies` and `Directors`, and depending on how we are going to access the data, we can directly save one, the other or both.

For example, we could have a structure for movies, which contains a list of people who have participated in its direction:

Collection Movies

```
{
    _id: 10,
    Title: "Solo. A Star Wars Story",
    Year: "2018",
    Director: [
        {
            Name: "Irvin Kershner",
            Birthday: 1923
        },
        {
            Name: "George Lucas",
            Birthday: 1944
        },
    ]
}
```

This way, we would access the information about the directors of a movie using dot notation; for example: `movie.Director[0].Name`

On the other hand, we could also do the storage in reverse: store the directors, and inside them an array with the movies:

Collection Directors

```
{
    _id: 1,
    Name: "George Lucas",
    Birthday: "1944",
    Movies:[
        {
            Title: "A new hope",
            Year: 1977
        },
        {
            Title: "Return of the Jedi",
            Year: 1984
        },
        ...
    ]
}
```

2.2 Getting started with MongoDB

Mongo is available on several platforms: Linux, Solaris, MacOS X, and Windows. The project website is www.mongodb.org. There is a web learn.mongodb.com with free courses to train in MongoDB. You can also find some MongoDB courses in Open Webinars.

There are two main MongoDB versions: The *Community*, with which we will work, and the *Enterprise* version, which requires a subscription and provides more advanced features and support. On the web, we are also told about *MongoDB Atlas*, a cloud computing service for MongoDB. We can try it for free in the MongoDB web.

To work in our local system, we have two options: install MongoDB or use a Docker image.

2.2.1 Install MongoDB on Ubuntu 22.04

We are going to install MongoDB Community Edition using the apt package manager.

From a terminal, install gnupg and `curl` if they are not already available:

```
sudo apt-get install gnupg curl
```

Import the MongoDB public GPG key:

```
curl -fsSL https://pgp.mongodb.com/server-7.0.asc | \
   sudo gpg -o /usr/share/keyrings/mongodb-server-7.0.gpg \
   --dearmor
```

Create the list file `/etc/apt/sources.list.d/mongodb-org-7.0.list`:

```
echo "deb [ arch=amd64,arm64
 ↪  signed-by=/usr/share/keyrings/mongodb-server-7.0.gpg ]
 ↪  https://repo.mongodb.org/apt/ubuntu jammy/mongodb-org/7.0 multiverse" |
 ↪  sudo tee /etc/apt/sources.list.d/mongodb-org-7.0.list
```

Update the local repository and install MongoDB:

```
sudo apt update
sudo apt-get install -y mongodb-org
```

For some reason, you need to restart the MongoDB service to make it available:

```
sudo systemctl restart mongod
```

The detailed steps are at:

Install MongoDB in Ubuntu

2.2.2 Install MongoDB on Windows

Download the MongoDB Community Windows x64 `.msi` installer from the following link:

MongoDB Download Center

And follow the instructions.

During the installation choose the options: Complete installation, install as a service and run as a Network service user.

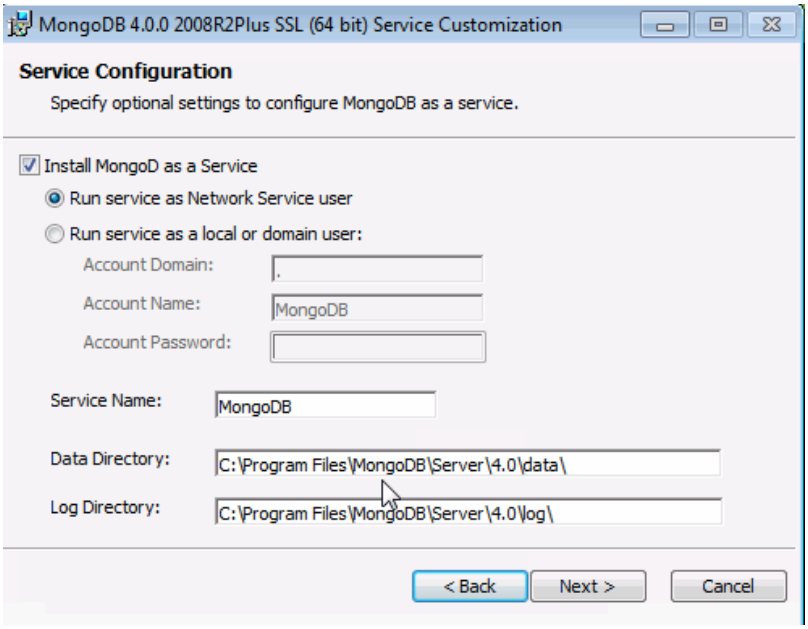

Figure 1: Windows installation

You also can install Mongo Shell from the following link: mongosh

2.2.3 Using a Docker image

If you want to use a Docker image. Follow these steps:

- Pull the MongoDB Docker image

```
docker pull mongodb/mongodb-community-server
```

- Create a Docker volume to persist the MongoDB data

```
docker volume create mongodata
```

- Run the image

```
docker run --name mongo -v mongodata:/data/db -p 27017:27017 -d
↪  mongodb/mongodb-community-server:latest
```

- Check that the container is running

```
docker ps
```

Now that the container is running, we can access to it with

```
sudo docker exec -it mongo mongosh
```

We are now in the mongo shell of our container.

The MongoDB shell is based on javascript, so we can run javascript code directly from it, use its libraries or define functions.

```
> let a=10;
> a
10
> let b=a+5
> b
15
> Math.sqrt(100)
10
> let today=new Date();
> today
ISODate("2023-12-14T18:20:42.522Z")
> function add(a, b){
... return a+b;
... }
> add(4,4)
8
```

```
fidel@fidel-DWES:~$ sudo docker exec -it mongo mongosh
Current Mongosh Log ID: 657c969a09c01ab325821c69
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=tr
Using MongoDB:          7.0.4
Using Mongosh:          2.1.1

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

------
   The server generated these startup warnings when booting
   2023-12-15T17:57:38.441+00:00: Using the XFS filesystem is strongly
p://dochub.mongodb.org/core/prodnotes-filesystem
   2023-12-15T17:57:39.008+00:00: Access control is not enabled for th
ion is unrestricted
   2023-12-15T17:57:39.009+00:00: vm.max_map_count is too low
------

test>
```

Figure 2: MongoDB shell

2.3 Basic instructions

By default, we work with the database `test`. We can change the database (or create one if it doesn't exist) with the instruction `use`.

```
test> use cinema
switched to db cinema
cinema>
```

The prompt has changed to `cinema>`

- Check the collections in our database:

```
db.getCollectionNames();
```

We don't have any collection in the database. Let's create one called `movies`

```
db.createCollection("movies");
```

If we run `db.getCollectionNames()` again, we'll get the name of our new collection.

Collections are the equivalent of tables in the relational model, with the difference that collections have a dynamic schema, whereby documents in the same collection can have different structure from each other.

Collection names must be subject to certain restrictions:

- We cannot use the empty string (“”), or the null character
- We cannot create collections starting with “system.”, as this is a prefix for internal system collections
- A collection name can't contain the $ character

2.4 Working with documents

The unit of information inside a MongoDB collection is the **document**, which would be the equivalent of a row in a relational model. These are JSON documents, made up of key-value pairs, and which represent information in a fairly intuitive way.

In order to form a JSON document for MongoDB, there are a few things to keep in mind:

- **Regarding keys**:
  - They cannot be null,
  - They can be formed by any UTF-8 character, except the characters . or $.
  - They are case-sensitive
  - They must be unique within the same document
- **Regarding their values**:
  - They can be of any permitted type

Type	Description
null	Represents both the null value and a field that does not exist
boolean	It allows the values *true* and *false*,

Type	Description
number	They represent numerical values in floating point (Float). If we want to use integer or float types, we must use our own classes: `NumberInt("3")` or `NumberLong("3")`
string	They represent any valid UTF-8 text string.
date	Represents dates, expressed in milliseconds
array	These are lists of values that are represented as vectors in Mongo
embedded documents	Documents can have other documents embedded in them
ObjectId	This is the default type for _id fields, and is designed to easily generate unique values globally

- **Regarding the document**:
  - It must have a `_id` field, with a unique value, which will act as the ID of the document. If we do not specify this key, MongoDB will generate it automatically, with an object of type ObjectId.

2.4.1 Some considerations about data types

Let's look at some interesting peculiarities about some of these types.

The Date Type

Mongo uses Javascript's `Date` type. When we generate a new object of type `Date`, it is necessary to make use of the operator `new`, because otherwise we would get a representation of the date in the form of *string*:

```
cinema> today = new Date()
cinema> today
ISODate('2023-12-15T18:33:21.946Z')
cinema> today=Date()
Fri Dec 15 2023 18:53:08 GMT+0000 (Coordinated Universal Time)
cinema> today
Fri Dec 15 2023 18:53:08 GMT+0000 (Coordinated Universal Time)
cinema>
```

Arrays

Arrays can be used to represent ordered collections, such as lists or queues, or unordered, such as sets. As in Javascript, and unlike other languages such as Java, each element of the vector can have a different data type, even other objects of type array.

The way to access its elements will be as we have seen with vectors and objects with Javascript. Let's see some examples:

```
cinema> v={objects: ["temperature", 10, {text: "hello"}, false] }
{ objects: [ 'temperature', 10, { text: 'hello' }, false ] }
cinema> v
{ objects: [ 'temperature', 10, { text: 'hello' }, false ] }
cinema> v.objects
[ 'temperature', 10, { text: 'hello' }, false ]
cinema> v.objects[0]
temperature
cinema> v.objects[1]
10
cinema> v.objects[2]
{ text: 'hello' }
cinema> v.objects[2].text
hello
cinema>
```

About ObjectIds

The *ObjectId* class uses 12 bytes, organized as follows:

0	1	2	3	4	5	6	7	8	9	10	11
Timestamp				Machine			PID		Increment		

Figure 3: ObjectId structure

- **Timestamp (bytes 0-3)**: The timestamp in seconds since January 1, 1970.
- **Machine (bytes 4-6)**: The machine's unique identifier, usually a hash of its hostname
- **PID (bytes 7-8)**: Identifier of the process that generates the ObjectID, to guarantee uniqueness within the same machine
- **Increment (bytes 9-11)**: Auto-incremental value, in order to guarantee uniqueness in the same second, machine and process

As we can see, this is a more robust mechanism than a simple autoincremental field as in MySQL. This corresponds to the distributed nature of MongoDB, so that objects can be generated in a multi-host environment.

2.4.2 Document operations

Operator	Meaning	Example
insertOne(document)	Add a document to the collection	`db.movies.insertOne(movie)`
insertMany([<document 1> , <document 2>, ...])	Add multiple documents to the collection	`db.movies.insertMany(movie1, movie2, movie3...)`
replaceOne(filter, document, options)	Replaces a single document within the collection based on the filter. If in the options we set the parameter `upsert` to `true`, the document will be inserted if it doesn't exist	`db.movies.replaceOne(filter, movie, {upsert: true})`
updateOne(filter, changes, options)	Updates a single all documents in the collection that match the filter. The parameter `changes` might contain a full document or only the changes we want to make	`db.movies.updateOne({Title: "My movie"}, changes)`
updateMany(filter, changes, options)	Updates a document in the collection based on the filter. The parameter `changes` might contain a full document or only the changes we want to make	`>db.movies.updateMany({Title: "My movie"}, changes)`
find(pattern)	Gets all documents in a collection that match the given pattern	`> db.movies.find({title: "My first document"});`
find()	Gets all documents in a collection	`> db.movies.find();`
findOne(pattern)	Gets an element of the collection matching the pattern	`db.movies.findOne();`

Operator	Meaning	Example
deleteOne(pattern)	Deletes a document from a collection that matches the pattern	> `db.movies.deleteOne({title: "My post"})`
deleteMany(pattern)	Deletes all the documents from a collection that match the pattern	> `db.movies.deleteMany({title: "My post"})`

2.4.3 Adding a new document to the collection

The natural way to add items to the database is through the `insert` method, available in all collections.

Let's add a film to our `movies` collection. First, we create the JSON document:

```
cinema> let newMovie = {
...     _id: 10,
...     Title: "Solo. A Star Wars Story",
...     Year: "2018",
...     Director: [
...         {
...             Name: "Irvin Kershner",
...             Birthday: 1923
...         },
...         {
...             Name: "George Lucas",
...             Birthday: 1944
...         },
...     ]
... }
cinema>
```

Now we can add the document `newMovie` to the `movies` collection:

```
> db.movies.insertOne(newMovie);
{ acknowledged: true, insertedIds: { '0': 11 } }
```

In this case, we have provided the _id, but it may be generated automatically. Let's try this one:

```
cinema> let newMovie = {
... Title: "The Phantom Menace",
... Year: "1999",
... Director: [
...  {
...   Name: "George Lucas",
...   Birthday: 1944
...  },
... ],
... }

cinema>db.movies.insertOne(newMovie);
  acknowledged: true,
  insertedId: ObjectId('659d7d643c2e412cb7592dab')
```

The movie has no predefined identifier, so Mongo has assigned a random one.

If we list the movies of our collection we can see the new movie with the automatic id:

```
cinema> db.movies.find();
[
  {
    _id: 10,
    Title: 'Solo. A Star Wars Story',
    Year: '2018',
    Director: [
      { Name: 'Irvin Kershner', Birthday: 1923 },
      { Name: 'George Lucas', Birthday: 1944 }
    ]
  },
  {
    _id: 11,
    Title: 'The Empire strikes back',
    Year: '1980',
    Director: [ { Name: 'Irvin Kershner', Birthday: '1923' } ]
  },
  {
    _id: ObjectId('659d7d643c2e412cb7592dab'),
    Title: 'The Phantom Menace',
    Year: '1999',
    Director: [ { Name: 'George Lucas', Birthday: 1944 } ]
  }
```

```
]
cinema>
```

We can assign the ids manually as long as we don't repeat them (the _id is the "primary key"). If we try to add the same movie again, we will get a duplicate key message.

As you see in the list, the documents in the collection are not forced to have the same structure. The first movie has two directors, the other two movies have only one. We can even add a movie with no directors.

2.4.4 Batch insert

MongoDB allows to add multiple documents at a time using insertMany instead of insertOne.

```
> let movie1={Title: "Star Wars. A new Hope", Year: "1977"};
> let movie2={Title: "Empire Strikes Back", Year: "1981"};
> let movie3={Title: "Return of the Jedi", Year: "1984"};
> db.movies.insertMany([movie1, movie2, movie3]);
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('659d840395c28fa22d088ba3'),
    '1': ObjectId('659d840395c28fa22d088ba4'),
    '2': ObjectId('659d840395c28fa22d088ba5')
  }
}
```

If one document fails in being added to the collection, the rest of the documents won't be added either.

2.4.5 Deleting documents

To remove a document from a collection we use the `remove` method, passing as a parameter a JSON document with the condition(s) of the documents that will be removed.

Deleting a document using the id (`deleteOne`):

```
cinema> db.movies.deleteOne({_id: ObjectId('659d840395c28fa22d088ba5')});
{ acknowledged: true, deletedCount: 1 }
cinema>
```

Delete the documents using a condition. Let's remove the movies with the word `strikes` in the title:

```
cinema> db.movies.deleteMany({Title: /strikes/});
{ acknowledged: true, deletedCount: 1 }
cinema>
```

As you can see, the movie with the word `Strikes` has not been deleted, because the pattern is case sensitive.

```
cinema> db.movies.find();
[
  {
    _id: 10,
    Title: 'Solo. A Star Wars Story',
    Year: '2018',
    Director: [
      { Name: 'Irvin Kershner', Birthday: 1923 },
      { Name: 'George Lucas', Birthday: 1944 }
    ]
  },
  {
    _id: ObjectId('659d7d643c2e412cb7592dab'),
    Title: 'The Phantom Menace',
    Year: '1999',
    Director: [ { Name: 'George Lucas', Birthday: 1944 } ]
  },
  {
    _id: ObjectId('659d840395c28fa22d088ba3'),
    Title: 'Star Wars. A new Hope',
    Year: '1977'
  },
  {
    _id: ObjectId('659d840395c28fa22d088ba4'),
    Title: 'Empire Strikes Back',
    Year: '1981'
```

```
  }
]
cinema>
```

Deleting all the movies of 1981:

```
cinema> db.movies.deleteMany({Year: "1981"});
{ acknowledged: true, deletedCount: 1 }
cinema>
```

We can remove the whole collection with the method `drop`.

```
cinema> db.getCollectionNames();
[ 'movies' ]
cinema> db.createCollection("test2024");
{ ok: 1 }
cinema> db.getCollectionNames();
[ 'test2024', 'movies' ]
cinema> db.test2024.drop();
true
cinema> db.getCollectionNames();
[ 'movies' ]
cinema>
```

2.4.6 Updating documents

We can update only a document with `updateOne`, or all the documents that match a condition with `updateMany()`.

```
cinema> db.movies.updateOne({"Title": "Solo. A Star Wars Story"},{$set:
↪  {"Title":'Hans Solo. A Star Wars Story'}});
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
cinema>
```

We use `$set` to made the changes. Other operators are

Name	Description
`$currentDate`	Sets the value of a field to current date, either as a Date or a Timestamp
`$inc`	Increments the value of the field by the specified amount
`$min`	Only updates the field if the specified value is less than the existing field value
`$max`	Only updates the field if the specified value is greater than the existing field value
`$mul`	Multiplies the value of the field by the specified amount
`$rename`	Renames a field
`$setOnInsert`	Sets the value of a field if an update results in an insert of a document. Has no effect on update operations that modify existing documents
`$unset`	Removes the specified field from a document

If a field in the document is an array, we can operate over the array too:

Name	Description
`$`	Acts as a placeholder to update the first element that matches the query condition
`$[]`	Acts as a placeholder to update all elements in an array for the documents that match the query condition
`$['<identifier>']`	Acts as a placeholder to update all elements that match the arrayFilters condition for the documents that match the query condition
`$addToSet`	Adds elements to an array only if they do not already exist in the set
`$pop`	Removes the first or last item of an array
`$pull`	Removes all array elements that match a specified query
`$push`	Adds an item to an array
`$pullAll`	Removes all matching values from an array

More info in the official MongoDB documentation

2.4.7 Upserts

If no document matches the condition, no update will be done. There is an option to create a new document, when doing an update, if no document matches the condition. We must use **upserts**. The easy way is to put a third parameter equals to `true` in the update. For instance:

```
cinema> db.movies.updateMany({"Title":"The return of the Jedi"},{$set:
 ↪  {"Title":"The Return of the Jedi","Year":"1984"}});
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 0,
  modifiedCount: 0,
  upsertedCount: 0
}
cinema>
```

No document has been updated because the movie with the given title doesn't exist. If we put the upsert to true:

```
cinema> db.movies.updateOne({"Title":"The return of the Jedi"},{$set:
 ↪  {"Title":"The Return of the Jedi","Year":"1984"}},{"upsert":true});
{
  acknowledged: true,
  insertedId: ObjectId('659d92b602135f6a9d9ed74a'),
  matchedCount: 0,
  modifiedCount: 0,
  upsertedCount: 1
}
cinema>
```

A new document has been inserted to the collection.

2.4.8 Comparison operators

Operator	Description
$eq	Matches values that are equal to a specified value

Operator	Description
`$gt`	Matches values that are greater than a specified value
`$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

Let's find the movies filmed after 1990:

```
cinema> db.movies.find({"Year":{$gte: "1990"}});
[
  {
    _id: 10,
    Title: 'Hans Solo. A Star Wars Story',
    Year: '2018',
    Director: [
      { Name: 'Irvin Kershner', Birthday: 1923 },
      { Name: 'George Lucas', Birthday: 1944 }
    ]
  },
  {
    _id: ObjectId('659d7d643c2e412cb7592dab'),
    Title: 'The Phantom Menace',
    Year: '1999',
    Director: [ { Name: 'George Lucas', Birthday: 1944 } ]
  }
]
cinema>
```

Let's find the movies filmed between 1980 and 1990:

```
cinema> db.movies.find({"Year":{$gte: "1980",$lte:"1990"}});
[
```

```
  {
    _id: ObjectId('659d92b602135f6a9d9ed74a'),
    Title: 'The Return of the Jedi',
    Year: '1984'
  }
]
cinema>
```

When we apply two conditions, by default we are asking for the documents that match both conditions (AND). We can use an OR operator instead.

Let's find the movies filmed before 1980 or after 2000.

```
cinema> db.movies.find({$or: [{"Year":{$gte:
↪  "2000"}},{"Year":{$lte:"1980"}}]});
[
  {
    _id: 10,
    Title: 'Hans Solo. A Star Wars Story',
    Year: '2018',
    Director: [
      { Name: 'Irvin Kershner', Birthday: 1923 },
      { Name: 'George Lucas', Birthday: 1944 }
    ]
  },
  {
    _id: ObjectId('659d840395c28fa22d088ba3'),
    Title: 'Star Wars. A new Hope',
    Year: '1977'
  }
]
cinema>
```

Let's find the movies filmed in 1977 or in 1999. In this case we can use the operator `$in`:

```
cinema> db.movies.find({"Year":{$in: ["1977","1999"]}});
[
  {
    _id: ObjectId('659d7d643c2e412cb7592dab'),
    Title: 'The Phantom Menace',
    Year: '1999',
```

```
    Director: [ { Name: 'George Lucas', Birthday: 1944 } ]
  },
  {
    _id: ObjectId('659d840395c28fa22d088ba3'),
    Title: 'Star Wars. A new Hope',
    Year: '1977'
  }
]
cinema>
```

2.4.9 Asking for conditions in an array

There are a lot of options. Let's find the movies directed by George Lucas. We will use the operator `$elemMatch`.

```
cinema> db.movies.find({Director: { $elemMatch: { Name: "George Lucas" } } });
[
  {
    _id: 10,
    Title: 'Hans Solo. A Star Wars Story',
    Year: '2018',
    Director: [
      { Name: 'Irvin Kershner', Birthday: 1923 },
      { Name: 'George Lucas', Birthday: 1944 }
    ]
  },
  {
    _id: ObjectId('659d7d643c2e412cb7592dab'),
    Title: 'The Phantom Menace',
    Year: '1999',
    Director: [ { Name: 'George Lucas', Birthday: 1944 } ]
  }
]
cinema>
```