

INDIVIDUAL PROJECT_ DATABASES

Student: ANA MARÍA DEL CACHO TENA

Master in Data Science (Assembler Institute of Technology)

README.

This individual project comprises three components: MongoDB exercises, Neo4j exercises (both of them done in this PDF file), and an optional report in Jupyter Notebooks using the library "pymongo". MongoDB exercises will involve working with the popular NoSQL database, focusing on data retrieval, manipulation, and querying. Neo4j exercises will explore the concepts of graph databases, emphasizing graph traversal, pattern matching, and graph analytics. Finally, an optional report has been created from a new database ("movies", also attached) using Jupyter Notebooks, showing an EDA with "pymongo" and Mongo-Atlas chart visualizations.

This project aims to provide hands-on experience with different database technologies and foster a deeper understanding of data management and analysis.

MongoDB EXERCISES

```
test> use mongo_exercise  
  
switched to db mongo_exercise
```

These commands are used to select and switch to the context of a database named "mongo_exercise" in MongoDB. This allows you to work within the chosen database, i.e. "mongo_exercises".

1. CREATE DATA BASE

All the documents were introduced in MongoDB as a file using the information in the original document from Classroom.

2. QUERIES / SEARCH DOCUMENTS

2.1. Get all documents

```
mongo_exercise> db.movies.find()

// SOLUTION

[
  {
    _id: ObjectId("6486d580ef06210180ab211f"),
    title: 'Fight Club',
    writer: 'Chuck Palahniuk',
    year: 1999,
    actors: [ 'Brad Pitt', 'Edward Norton' ]
  },
  {
    _id: ObjectId("6486d580ef06210180ab2120"),
    title: 'Pulp Fiction',
    writer: 'Quentin Tarantino',
    year: 1994,
    actors: [ 'John Travolta', 'Uma Thurman' ]
  },
  {
    _id: ObjectId("6486d580ef06210180ab2121"),
    title: 'Inglorious Basterds',
    writer: 'Quentin Tarantino',
    year: 2009,
    actors: [ 'Brad Pitt', 'Diane Kruger', 'Eli Roth' ]
  },
  {
    _id: ObjectId("6486d580ef06210180ab2122"),
    title: 'The Hobbit: An Unexpected Journey',
    writer: 'J.R.R. Tolkien',
    year: 2012,
    franchise: 'The Hobbit'
  },
  {
    _id: ObjectId("6486d580ef06210180ab2123"),
    title: 'The Hobbit: The Desolation of Smaug',
    writer: 'J.R.R. Tolkien',
    year: 2013,
    franchise: 'The Hobbit'
  },
  {
    _id: ObjectId("6486d580ef06210180ab2124"),
    title: 'The Hobbit: The Battle of the Five Armies',

```

```

    writer: 'J.R.R. Tolkien',
    year: 2012,
    franchise: 'The Hobbit',
    synopsis: 'Bilbo and Company are forced to engage in a war against an array of com
batants and keep the Lonely Mountain from falling into the hands of a rising darknes
s.'
  },
  {
    _id: ObjectId("6486d580ef06210180ab2125"),
    title: "Pee Wee Herman's Big Adventure"
  },
  { _id: ObjectId("6486d580ef06210180ab2126"),
    title: 'Avatar'
  }
]

```

This command searches and retrieves all documents stored in the "movies" collection of the current database.

Specifically, the empty "find()" function displays all documents in the "movies" collection.

2.2. Get documents with "writer" equal to "Quentin Tarantino"

```

mongo_exercise> db.movies.find({"writer":"Quentin Tarantino"})

// SOLUTION

[
  {
    _id: ObjectId("6486d580ef06210180ab2120"),
    title: 'Pulp Fiction',
    writer: 'Quentin Tarantino',
    year: 1994,
    actors: [ 'John Travolta', 'Uma Thurman' ]
  },
  {
    _id: ObjectId("6486d580ef06210180ab2121"),
    title: 'Inglorious Basterds',
    writer: 'Quentin Tarantino',
    year: 2009,
    actors: [ 'Brad Pitt', 'Diane Kruger', 'Eli Roth' ]
  }
]

```

In this case, however, since we have specified what we want to find in the "find()", we are shown a very specific solution, i.e., the two movies whose screenwriter is

Quentin Tarantino.

2.3. Get documents with `actors` that include "Brad Pitt"

```
mongo_exercise> db.movies.find({actors:"Brad Pitt"})

// SOLUTION

[
  {
    _id: ObjectId("6486d580ef06210180ab211f"),
    title: 'Fight Club',
    writer: 'Chuck Palahniuk',
    year: 1999,
    actors: [ 'Brad Pitt', 'Edward Norton' ]
  },
  {
    _id: ObjectId("6486d580ef06210180ab2121"),
    title: 'Inglorious Basterds',
    writer: 'Quentin Tarantino',
    year: 2009,
    actors: [ 'Brad Pitt', 'Diane Kruger', 'Eli Roth' ]
  }
]
```

Similar to the previous one, but changing the "find()" key, we search for those movies that include Brad Pitt as an actor. Also, in subsection 2.4, we do the same but with the "franchise" item.

2.4. Get documents with `franchise` equal to "The Hobbit"

```
mongo_exercise> db.movies.find({franchise:"The Hobbit"})

// SOLUTION

[
  {
    _id: ObjectId("6486d580ef06210180ab2122"),
    title: 'The Hobbit: An Unexpected Journey',
    writer: 'J.R.R. Tolkien',
    year: 2012,
    franchise: 'The Hobbit'
  },
  {
    _id: ObjectId("6486d580ef06210180ab2123"),
    title: 'The Hobbit: The Desolation of Smaug',

```

```

    writer: 'J.R.R. Tolkien',
    year: 2013,
    franchise: 'The Hobbit'
  },
  {
    _id: ObjectId("6486d580ef06210180ab2124"),
    title: 'The Hobbit: The Battle of the Five Armies',
    writer: 'J.R.R. Tolkien',
    year: 2012,
    franchise: 'The Hobbit',
    synopsis: 'Bilbo and Company are forced to engage in a war against an array of com
batants and keep the Lonely Mountain from falling into the hands of a rising darknes
s.'
  }
]

```

2.5. Get all the 90s movies.

```

mongo_exercise> db.movies.find({ $and: [ {year: {$gt:1990} } , {year:{ $lt:2000} } ]
})

// SOLUTION

[
  {
    _id: ObjectId("6486d580ef06210180ab211f"),
    title: 'Fight Club',
    writer: 'Chuck Palahniuk',
    year: 1999,
    actors: [ 'Brad Pitt', 'Edward Norton' ]
  },
  {
    _id: ObjectId("6486d580ef06210180ab2120"),
    title: 'Pulp Fiction',
    writer: 'Quentin Tarantino',
    year: 1994,
    actors: [ 'John Travolta', 'Uma Thurman' ]
  }
]

```

Of course, in MongoDB you can also use the "find()" function to filter documents. Here we have done it by years, i.e., we only want the movies developed between the 1990s and 2000s. To do this, we use the logical operator "\$and" (which works with arrays), and with the operators "\$gt" (greater than) and "\$lt" (less than).

2.6. Obtain the films released between the year 2000 and 2010.

```

mongo_exercise> db.movies.find({$and: [{year: {$gt: 2000}}, {year: {$lt: 2010}}]})

// SOLUTION

[
  {
    _id: ObjectId("6486d580ef06210180ab2121"),
    title: 'Inglorious Basterds',
    writer: 'Quentin Tarantino',
    year: 2009,
    actors: [ 'Brad Pitt', 'Diane Kruger', 'Eli Roth' ]
  }
]

```

3. UPDATE DOCUMENTS

3.1. Add synopsis to "The Hobbit: An Unexpected Journey": "A reluctant hobbit, Bilbo Baggins, sets out to the Lonely Mountain with a spirited group of dwarves to reclaim their mountain home - and the gold within it - from the dragon Smaug ."

```

mongo_exercise> db.movies.updateOne({title:"The Hobbit: An Unexpected Journey"},{$set:
{synopsis:"A reluctant  hobbit, Bilbo Baggins, sets out to the Lonely Mountain with a
  spirited group of dwarves to reclaim their mountain home - and the gold within it - f
rom the dragon Smaug."}})

// SOLUTION

{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}

```

The method “updateOne” is used to update a single document that matches the specified filter in the "movies" collection. Additionally, the “\$set” operator is used to act on this update.

3.2. Add synopsis to "The Hobbit: The Desolation of Smaug" : "The dwarves, along with Bilbo Baggins and Gandalf the Grey, continue their quest to reclaim Erebor, their

homeland, from Smaug. Bilbo Baggins is in possession of a mysterious and magical ring."

```
mongo_exercise> db.movies.updateOne({title:"The Hobbit: The Desolation of Smaug"},{$set:{ synopsis:"The dwarves, along with Bilbo Baggins and Gandalf the Grey, continue their quest to reclaim Erebor, their homeland, from Smaug. Bilbo Baggins is in possession of a mysterious and magical ring."}})
```

```
// SOLUTION
```

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

3.3. Adding an actor named "Samuel L. Jackson" to the movie "Pulp Fiction"

```
mongo_exercise> db.movies.updateOne({title:"Pulp Fiction"},{$push:{actors:"Samuel L. Jackson"}})
```

```
// SOLUTION
```

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

In this exercise, we used the operator “\$push” to add elements (“Samuel L. Jackson”) to an array field (“actors”).

Neo4j EXERCISES

CREATE DATABASE

1.1.

- COURSE: JAVA, course: “Standard Java Programming”, duration: 120, price: 80
- COURSE: ANGULAR, course: “Angular”, duration: 30, price: 110
- COURSE: SPRING, course: “Spring”, duration: 80, price: 200
- STUDENT: PEPE, name: “Pepe”, age: 20
- STUDENT: ANA, name: “Ana”, age: 40
- STUDENT: ELENA, name: “Elena”, age: 34
- STUDENT: MARIO, name: “Mario”, age: 19

```
CREATE (java:course {name:'Standard Java Programming', duration:120, price:80}), (angular:course {name:'Angular', duration:30, price:110}), (spring:course {name:'Spring', duration:80, price:200})
```

1.2.

- PEPE DOES THE JAVA COURSE IN THE MORNING SCHEDULE
- PEPE CARRIES OUT THE ANGULAR COURSE IN THE AFTERNOON
- ELENA TAKES THE JAVA COURSE IN THE AFTERNOON SCHEDULE
- ANA CARRIES OUT THE ANGULAR COURSE IN THE MORNING SCHEDULE
- MARIO DOES THE SPRING COURSE IN THE MORNING SCHEDULE

```
CREATE (pepe:student {name:'Pepe', age:20}), (elena: student { name:'Elena', edad:34}), (ana:student { name:'Ana', age:40}), (mario: student { name:'Mario', age:19})
```

In order to create a DataBase, we use the keyword “CREATE” to create nodes and relationships, that will be used to establish relationships or do some queries, as we will see later.

1.3. Create matchs

Here, we use “MATCH” in order to find nodes based on specific criteria [i.e., (pepe:student {name:'Pepe', age:20}), (java:course {name:'Standard Java

Programming', duration:120, price:80}]], and “CREATE” to build a relationship [i.e., “Pepe does the Java course in the morning”].

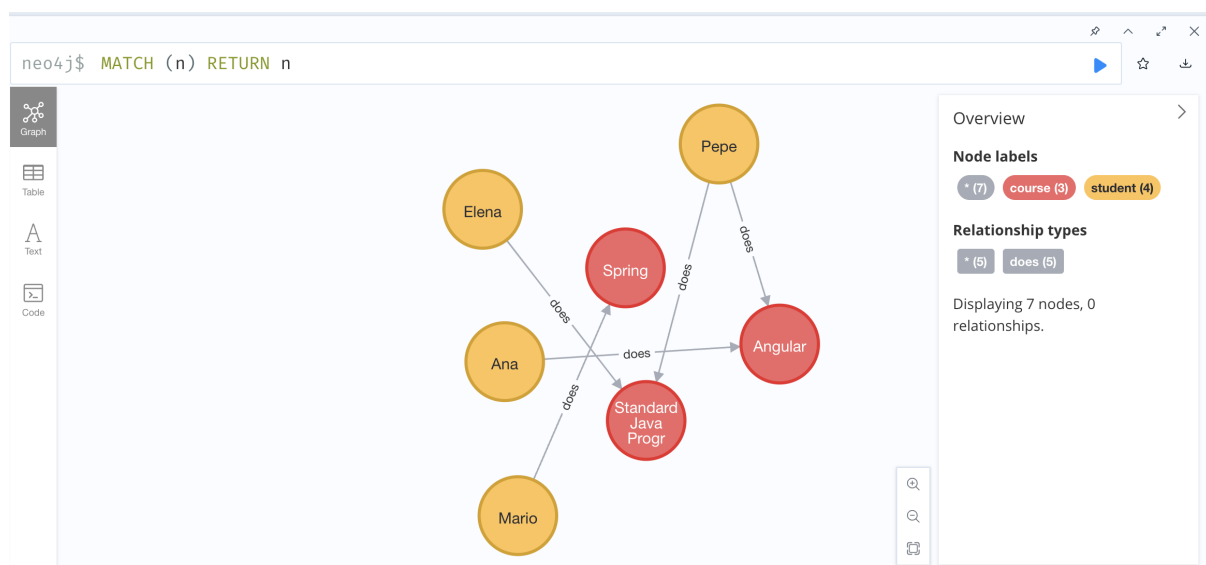
```
MATCH
(pepe:student {name:'Pepe', age:20}), (java:course {name:'Standard Java Programming',
duration:120, price:80})
CREATE (pepe)-[: does {schedule:'Morning'}]->(java);

MATCH
(elena: student { name:'Elena'}), (java:course {name:'Standard Java Programming'})
CREATE (elena)-[: does {Schedule:'Afternoon'}]->(java);

MATCH
(pepe:student {name:'Pepe'}), (angular:course {name:'Angular'})
CREATE (pepe)-[: does {Schedule:'Afternoon'}]->(angular);

MATCH
(ana:student { name:'Ana'}), (angular:course {name:'Angular'})
CREATE (ana)-[: does {schedule:'Morning'}]->(angular);

MATCH
(mario: student { name:'Mario'}), (spring:course {name:'Spring'})
CREATE (mario)-[: does {schedule:'Tomorrow'}]->(spring);
```



OPERATIONS

2.1. Show the name of the students who take courses:

```
MATCH (student)-[:does]-> (course) RETURN student.name
```



neo4j\$ MATCH (student)-[:does]→ (course) RETURN student.name

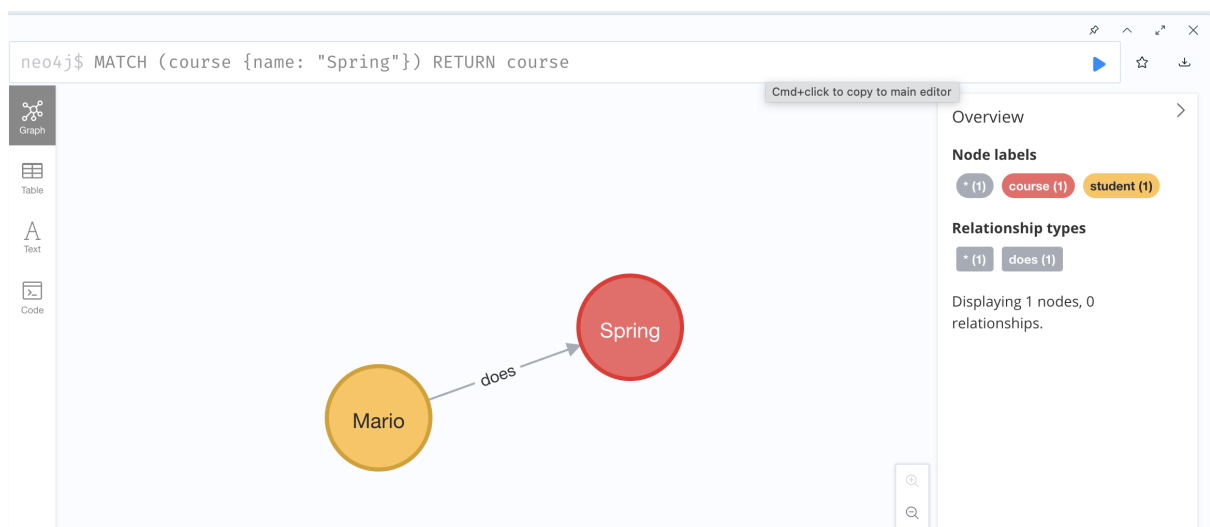
	student.name
1	"Pepe"
2	"Elena"
3	"Pepe"
4	"Ana"
5	"Mario"

Started streaming 5 records after 2 ms and completed after 7 ms.

With this code, a MATCH query is performed to find all "student" nodes connected to "course" nodes through a "does" relationship, and then returns the "name" property of each student.

2.2. Search for the spring course

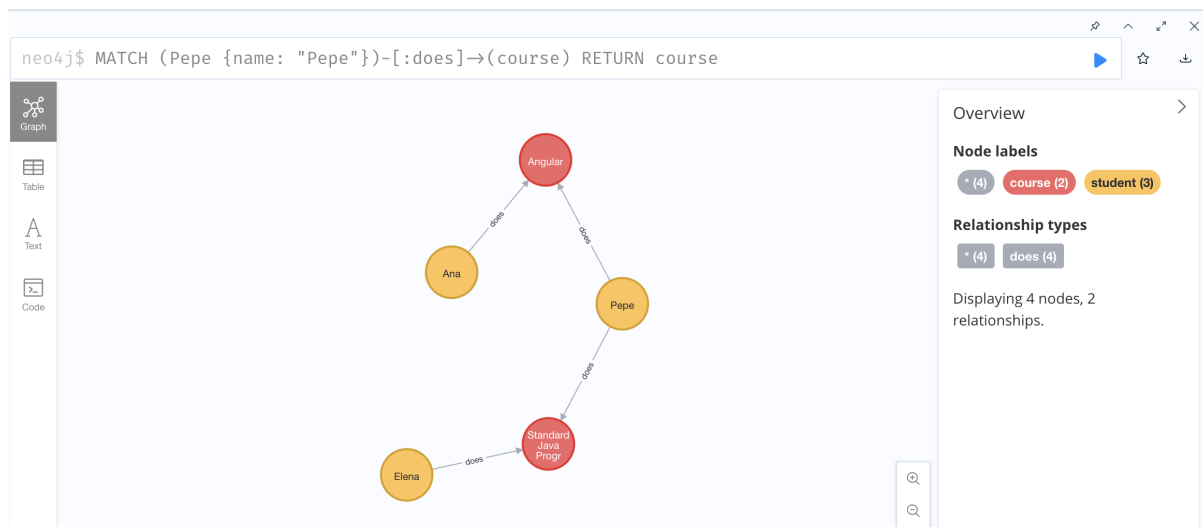
```
MATCH (course {name: "Spring"}) RETURN course
```



In this case, the code is very similar to the previous one, but it performs a MATCH query to find a "course" node with the name "Spring" and then returns the complete course node (including the relationship).

2.3. Show the courses that Pepe takes.

```
MATCH (Pepe {name: "Pepe"})-[:does]->(course) RETURN course
```



This code allows us to see which courses Pepe does, specifying the query with “{name: “Pepe”}” and the action “does” on the “course” nodo.

2.4. Show Pepe's data and the number of courses where he has enrolled

```
MATCH (n {name:'Pepe'})-[r]->() RETURN labels(n), n.age, count(*)
```

The Neo4j interface displays a table query result. The query is `MATCH (n {name:'Pepe'})-[r]->() RETURN labels(n), n.age, count(*)`. The table shows one record for Pepe with labels ['student'], age 20, and count 2.

	labels(n)	n.age	count(*)
1	["student"]	20	2


Started streaming 1 records after 22 ms and completed after 28 ms.

On the one hand, the first part of the code (`MATCH (n {name:'Pepe'})-[r]`) matches a node with the name "Pepe" and any outgoing relationship. The `(n)` is a placeholder for the matched node, and `[r]` represents the outgoing relationship.

On the other hand, the rest of the code represents what to return. So, `labels(n)` gives the labels of the matched node, which represent its node type or categories, `n.age` returns the age property of the matched node, and `count(*)` shows the count of the matched nodes.

2.5. Show the name of the students who take the angular course

```
MATCH (student)-[:does]-> (course{name:"Angular"}) RETURN student.name
```



The screenshot shows the Neo4j Cypher query interface. The query entered is `neo4j$ MATCH (student)-[:does]-> (course{name:"Angular"}) RETURN student.name`. The results are displayed in a table view with the column header `student.name`. There are two rows of results: the first row has the value `"Pepe"` and the second row has the value `"Ana"`. The interface also shows a status message at the bottom: "Started streaming 2 records after 1 ms and completed after 2 ms."

	student.name
1	"Pepe"
2	"Ana"

And, if we want to know the amount?

```
MATCH (student)-[:does]-> (course{name:"Angular"}) RETURN count(student)
```

neo4j\$ `MATCH (student)-[:does]→
(course{name:"Angular"}) RETURN
count(student)`

	count(student)
1	2

Table
Text
Code

2.6. Modify Ana's age from 40 to 25 years. Show the person whose age =25

```
MATCH (n{name:'Ana'}) set n.age=25 RETURN n.name,n.age
```

neo4j\$ `MATCH (n{name:'Ana'}) set n.age=25 RETURN n.name,n.age`

	n.name	n.age
1	"Ana"	25

Table
Text
Code

Set 1 property, started streaming 1 records after 1 ms and completed after 12 ms.

2.7. Show all ages of students

```
MATCH (n:student)RETURN collect(n.age)
```

neo4j\$ `MATCH (n:student)RETURN collect(n.age)`

	<code>collect(n.age)</code>
1	[20, 25, 19]

Started streaming 1 records after 1 ms and completed after 2 ms.

2.8. Show the price of all courses

```
MATCH (n:course)RETURN collect(n.price)
```

neo4j\$ `MATCH (n:course)RETURN collect(n.price)`

	<code>collect(n.price)</code>
1	[80, 110, 200]

Started streaming 1 records after 1 ms and completed after 2 ms.

2.9. We discharge Mario.

```
MATCH (n:student {name: 'Mario'}) detach DELETE n
```

neo4j\$ MATCH (n:student {name: 'Mario'}) detach DELETE n

Deleted 1 node, deleted 1 relationship, completed after 2 ms.

Deleted 1 node, deleted 1 relationship, completed after 2 ms.

2.10. Obtain the maximum and minimum age and the sum of students

```
MATCH (n:student)RETURN min(n.age),max(n.age),SUM(n.age)
```

neo4j\$ MATCH (n:student) RETURN min(n.age),max(n.age),SUM(n.age)

	min(n.age)	max(n.age)	SUM(n.age)
1	20	25	45

Started streaming 1 records after 1 ms and completed after 2 ms.

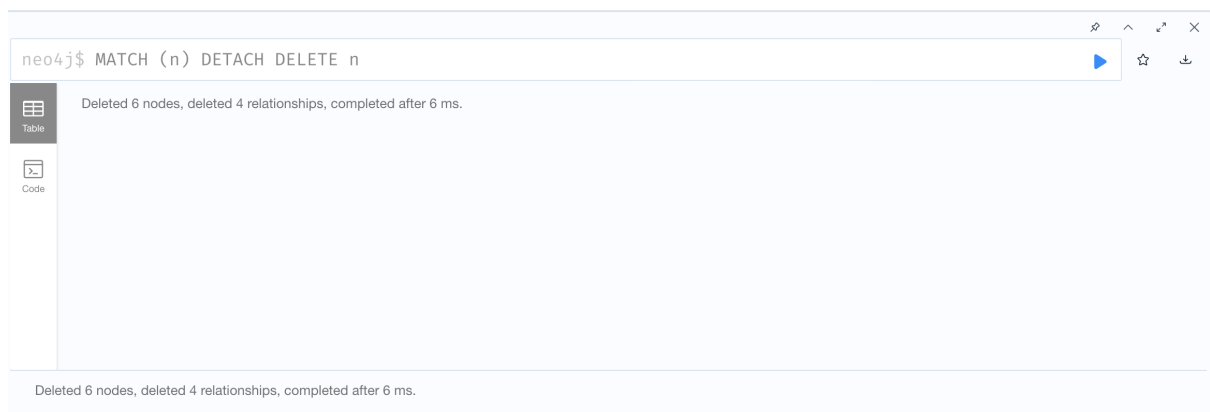
2.11. Modify the relationship (pepe)-[:does{schedule:"Morning"}]->(java), since it happens to perform it in the afternoon

```
MATCH (pepe)-[old:does {schedule:'Morning'}]->(java)
CREATE (pepe)-[new: realiza{schedule:'afternoon'}]->(java)
DELETE old
```

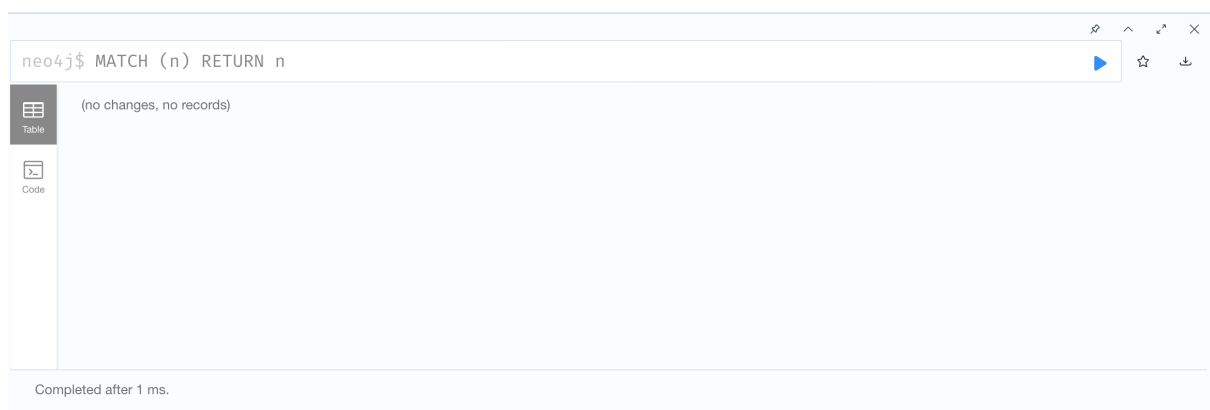


2.12. We remove the tree

MATCH (n) DETACH DELETE n



We check that the database does not exist



NOTES:

Although we know that you can work with Jupyter Notebooks and use Python for both MongoDB and Neo4j, in this project we did not use it because my colleagues were already implementing it. I wanted to do a different job and I chose this platform to detail everything. However, let's go into a few details.

1. For installation, we should write on Jupyter Notebook the following commands:

MongoDB:

```
!pip install pymongo
```

Neo4j:

```
!pip install neo4j
```

2. Create Databases and to define the queries:

MongoDB:

```
from pymongo import MongoClient

# Configure the database connection
uri = "mongodb://localhost:27017"
client = MongoClient(uri)

# Get a reference to the database and to the collection
db = client.mongo_exercises
collection = db.movies

# Define the documents to be inserted
documents = [
    {
        'name': 'Standard Java Programming',
        'duration': 120,
        'price': 80
    },
    {
        'name': 'Angular',
```

```

        'duration': 30,
        'price': 110
    },
    {
        'name': 'Spring',
        'duration': 80,
        'price': 200
    }
]

# Insert documents into the collection
result = collection.insert_many(documents)

# Print the IDs of the inserted documents
for id in result.inserted_ids:
    print(f"Documento insertado con ID: {id}")

```

Neo4j:

```

from neo4j import GraphDatabase

# Configure the database connection
uri = "bolt://localhost:7687"
username = "individualProject"
password = "individualProject1"

# Establishing the connection
driver = GraphDatabase.driver(uri, auth=(username, password))

# Define queries and execute them
with driver.session() as session:
    session.run("CREATE (java:course {name:'Standard Java Programming', duration:120, price:80}), "
                "(angular:course {name:'Angular', duration:30, price:110}), "
                "(spring:course {name:'Spring', duration:80, price:200})")

    session.run("CREATE (pepe:student {name:'Pepe', age:20}), "
                "(elena:student {name:'Elena', edad:34}), "
                "(ana:student {name:'Ana', age:40}), "
                "(mario:student {name:'Mario', age:19})")

    session.run("MATCH (pepe:student {name:'Pepe', age:20}), "
                "(java:course {name:'Standard Java Programming', duration:120, price:80}) "
                "CREATE (pepe)-[:does {schedule:'Morning'}]->(java)")

    session.run("MATCH (elena:student {name:'Elena'}), "
                "(java:course {name:'Standard Java Programming'}) "
                "CREATE (elena)-[:does {schedule:'Afternoon'}]->(java)")

```

```
session.run("MATCH (pepe:student {name:'Pepe'}), "  
            "(angular:course {name:'Angular'}) "  
            "CREATE (pepe)-[:does {schedule:'Afternoon'}]->(angular)")  
  
session.run("MATCH (ana:student {name:'Ana'}), "  
            "(angular:course {name:'Angular'}) "  
            "CREATE (ana)-[:does {schedule:'Morning'}]->(angular)")  
  
session.run("MATCH (mario:student {name:'Mario'}), "  
            "(spring:course {name:'Spring'}) "  
            "CREATE (mario)-[:does {schedule:'Tomorrow'}]->(spring)")
```