



Large-scale and Multi-structured Databases

JobAdvisor

Aizdi Leena
Arezoumandan Morteza
Dallatomasina Erica

[Click here for the Github repository of the project](#)

Table of contents

INTRODUCTION	2
MAIN ACTORS.....	3
FUNCTIONAL REQUIREMENTS.....	3
NON FUNCTIONAL REQUIREMENTS	4
DATASET DESCRIPTION	4
UML CLASS DIAGRAM	5
USE CASE DIAGRAM.....	6
DATABASE ORGANIZATION	7
MONGODB ORGANIZATION	7
NEO4J ORGANIZATION.....	9
CONSISTENCY BETWEEN NEO4J AND MONGODB	10
SOFTWARE ARCHITECTURE: DESIGN AND IMPLEMENTATION	11
SOFTWARE ARCHITECTURE	11
IMPLEMENTATION PHASE.....	12
MOST RELEVANT QUERIES USING MONGO DB.....	14
ANALYTIC 1: RANKING OF THE TOP 10 CITIES	14
ANALYTIC 2: RANKING OF THE TOP 10 SKILLS	15
GET JOB OFFERS BY CITY	16
GET JOB OFFERS BY COMPANY.....	17
GET JOB OFFER BY SALARY	18
MOST RELEVANT QUERIES USING NEO4J	20
GRAPH CENTRIC AND DOMAIN-SPECIFIC QUERIES	20
1 st RECOMMENDATION SYSTEM: RECOMMEND COMPANIES TO USERS	22
2 nd RECOMMENDATION SYSTEM: RECOMMEND JOB OFFERS TO USERS.....	24
INDEXES.....	26
MONGODB INDEXES	26
NEO4J INDEXES.....	27
TEST ON AVAILABILITY OF THE APPLICATION	28

INTRODUCTION

Job Advisor is an employment-oriented application whose core purpose is to provide a platform not only to job seekers to search for job offers, browse through lists of employers¹ and job offers, and save the posts they like, but also to employers who post new job offers and meanwhile search for relevant job seekers. The users first log in with their existing credentials (i.e. username and password) or sign up as job seeker or as employer to benefit from the functionalities of the application.

The core functions of a job seeker include searching for job offers with respect to a few parameters, saving job offers they are interested in and browsing them later, searching for an employer, following an employer and browsing through list of recommended job offers and recommended employers. On the other hand, an employer can mainly perform tasks such as publishing a new job offer and browsing through list of its already published job offers, browsing through the list of its followers and searching for job seekers matching a certain skill. Other trivial functions of both job seekers and employers are updating account details like password and deleting account. The third user of the application is an admin user whose core purpose is to manage and perform analysis of the application. The admin can delete Job offers and users (both job seekers and employers). Moreover, this user can find the most common skills found in job seekers, browse through a list of most followed employers and look for the cities with the greatest number of job offers.

¹ In the following, we will use the two words “employer” and “company” to refer to the same entity

MAIN ACTORS

The application has **three main actors**:

- **Job seeker.** He is one type of user of the application, in particular it is the person who searches for a job
- **Employer.** He is one type of user of the application, in particular he is the owner or the responsible of a company, he offers job requests
- **Admin.** He is the administrator of the application. He has special privileges that allows him, for example, to delete user accounts or to do actions that the other users cannot do.

FUNCTIONAL REQUIREMENTS

- An **anonymous user** can
 - Register as a company/job seeker
 - Login as a company/job seeker/admin
- A **standard user** can
 - Log out
 - Delete its account
 - Browse/update information associated to his account
 - Search job offers by title, by city, by company, by job type or by salary and browse them
 - Search a company by its name and browse the information related to it and the job offers it published
 - Follow/unfollow a company
 - Browse the companies he follows
 - Browse the posts published by the companies he follows
 - Browse all the offers he saved
 - Save/remove a job offer from his favourites
- An **employer** can
 - Log out
 - Delete its account
 - Browse/update information associated to its account
 - Publish a new job offer
 - Browse the job offers he published
 - Search users by skill or by location and browse them
 - Browse its followers
 - Delete/update a job offer he published
- The **administrator** can
 - Log out
 - Search an account (of job seeker or employer) by its username and delete it
 - Search a job offer by its id and delete it
 - Find top 10 cities depending on the job type (where top 10 means the ones that published more job offers that have this job type)
 - Find the most repeated skills between job seekers
 - Find the top 10 companies (where top 10 means the ones that have more followers)
 - View the number of job seekers accounts, employer accounts and published job offers

NON-FUNCTIONAL REQUIREMENTS

Here there is a list of the non-functional requirements of the application:

- **Performance:** the response time of the application to user's actions under certain workload should be minimum, in order to offer to the user a good experience while interacting with it
- **Usability:** the application must be easy to use and to understand for a common user
- **AP Solution (Availability and Partition Tolerance):** The application should be accessible to the users at any given point in time. It needs to continue to function regardless of system failures and network partitions. It may result in inconsistency at some points however, this is a small cost comparing to the benefit of availability in the case of this application

What we have done in order to fulfil these requirements?

- To guarantee usability we implement a graphical user interface
- To guarantee the availability of the application, we used two databases implemented as clusters: a document database (mongoDB) and a graph database (Neo4J).
- To guarantee performance and fast response:
 - we implement some indexes (see section INDEXES)
 - we use these settings to connect to mongoDB:

```
ConnectionString uriString= new ConnectionString("mongodb://172.16.3.151:27020,172.16.3.123:27020, 172.16.3.124:27020");
MongoClientSettings mcs = MongoClientSettings.builder().applyConnectionString(uriString)
    .readPreference(ReadPreference.nearest())
    .retryWrites(true)
    .writeConcern(WriteConcern.W1)
    .build();
```

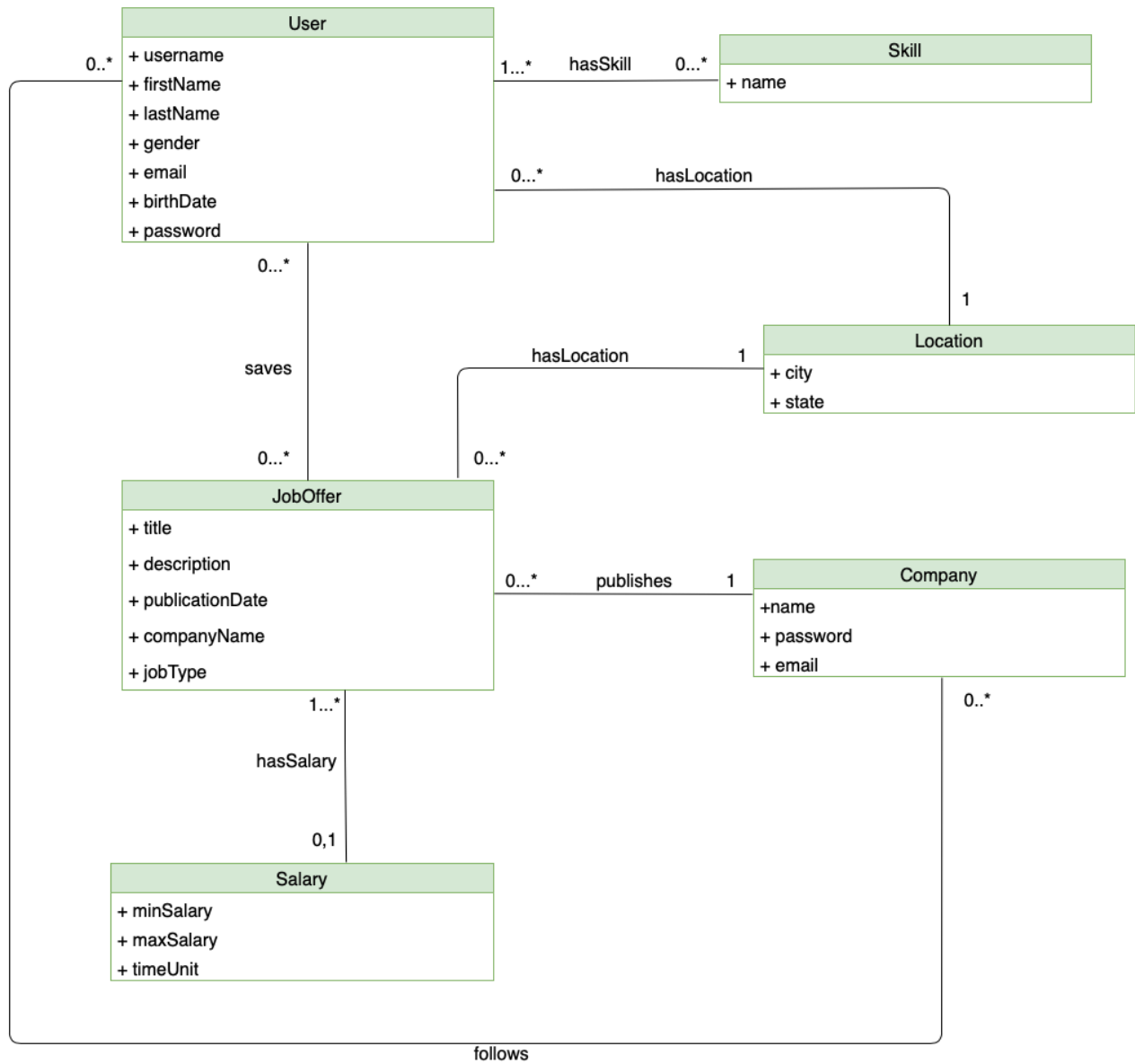
As we can see, we set the write concern equal to W1 and for the read preference we set the nearest (both these settings guarantee a better response time)

DATASET DESCRIPTION

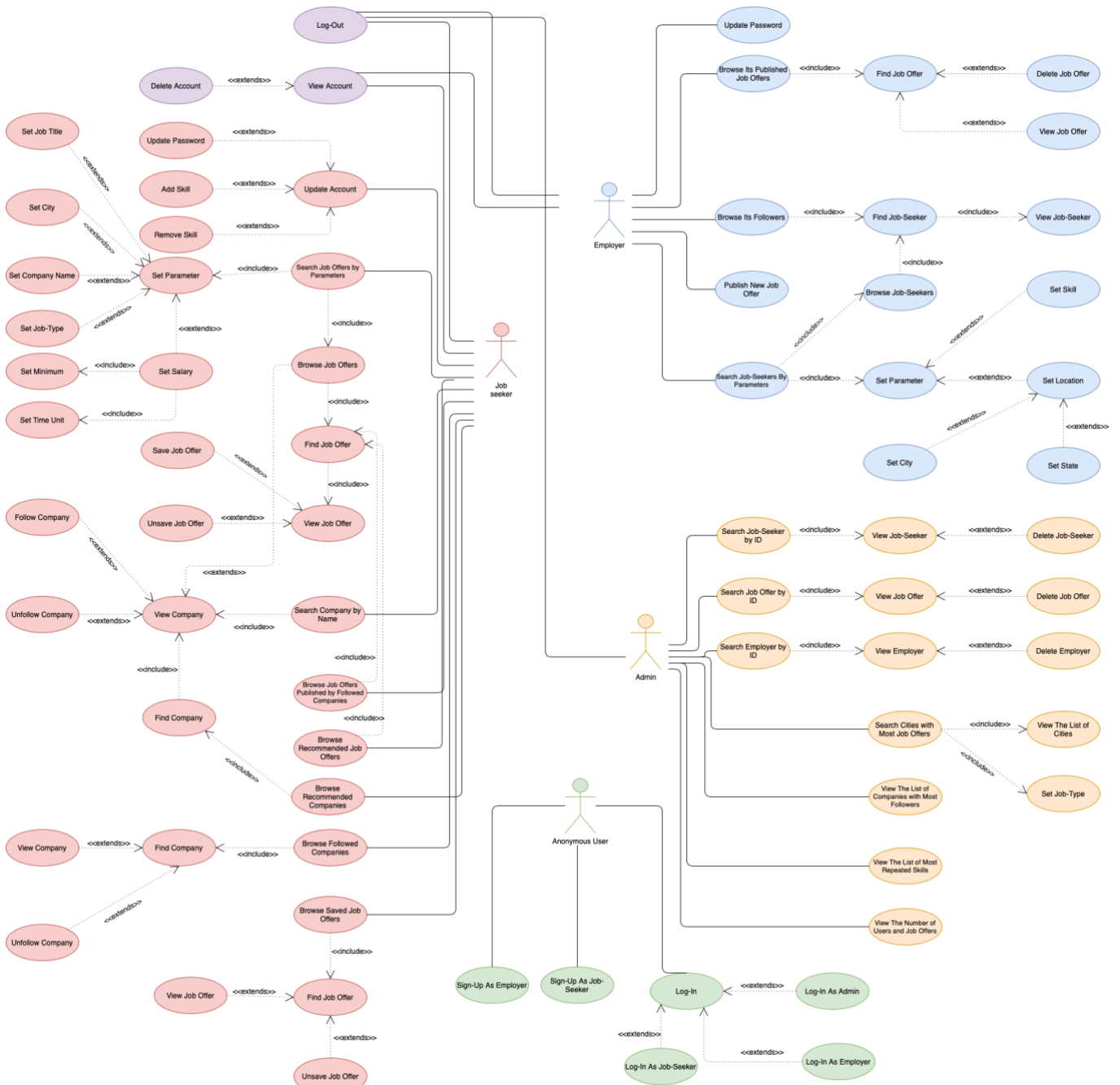
The data for the database of the application comes from different datasets as described below:

- **Job Offers:** Combining two different datasets, one from job offers of [Careerbuilder](#) website and one from job offers of [Dice](#) website.
- **Companies:** Constructed based on the job offers dataset (taking the name of company who published the job offers) and a password dataset.
- **Job-Seekers:** Merging two different datasets. One for [Resumes](#) and one for [Human Resources](#).

UML CLASS DIAGRAM



USE CASE DIAGRAM



DATABASE ORGANIZATION

MONGODB ORGANIZATION

We have three collections as follows:

- **Job Seekers:** In this collection, we store a document for each job seeker in the database. The structure of the document is the following:
 - `_id`: Representing the username.
 - `password`: Representing the password.
 - `first_name`: Representing the first name.
 - `last_name`: Representing the last name.
 - `birthdate`: Representing the date of birth.
 - `gender`: Representing the gender in a single character.
 - `email`: Representing the email.
 - `location`: Representing the location in an embedded document with the following fields:
 - `state`: Representing the state of user's location.
 - `city`: Representing the city of user's location.
 - `skills`: Representing the skills as an array. **This is an optional field**; hence, it might not be present in some documents.

Here is an example of a job-seeker document:

```
{
  "_id": "avlynde",
  "password": "C~GWnJlKIKruo!2",
  "first_name": "Ahmad",
  "last_name": "Lynde",
  "birthdate": "10/21/63",
  "gender": "M",
  "email": "ahmad.lynde@gmail.com",
  "location": {
    "city": "Chillicothe",
    "state": "IL"
  },
  "skills": [
    "Veterinary Medicine",
    "Agricultural Production",
    "Zoology",
    "Anesthesiology"
  ]
},
```

- **Job Offers:** In this collection, we store a document for each published job offer in the database. The structure of the document is the following:
 - `_id`: Representing the unique id for each job offer.
 - `job_title`: Representing the title of job.
 - `company_name`: Representing the name of the company that published the job offer.
 - `location`: Representing the location in an embedded document with the following fields:
 - `state`: Representing the state of job's location.
 - `city`: Representing the city of job's location.

- **salary**: Representing the salary offered by the company for the job offer. **This is an optional field**; hence, it might not be present in some documents. The structure of salary document is the following:
 - **from**: Representing the minimum possible salary for the job in US dollar.
 - **to**: Representing the maximum possible salary for the job in US dollar.
 - **time_unit**: Representing the time unit which the minimum and maximum for the possible salary is based on. In the application, salary can be presented either based on hour or based on year.
- **post_date**: Representing the date that the job posted by the company.
- **job_type**: Representing the type of the position offered by the company such as full-time, part-time, etc.
- **job_description**: Representing the description for the job.

Here is an example of a job offer document:

```
{
  "_id": "5ac916defe77c609099e4c79bc3dd7bc",
  "job_title": "Amazon Warehouser (Part-Time)",
  "company_name": "Amazon Fulfillment",
  "location": {
    "city": "Garwood",
    "state": "NJ"
  },
  "post_date": ISODate("2020-02-10T00:00:00Z"),
  "job_type": "Part-Time",
  "job_description": "sFulfillment Centers \u2013 Work inside an Amazon warehouse, selecting, packing and shipping customer orders. If you like a fast-paced, physical position that gets you up and moving, then come help bring orders to life. Work a set, full-time schedule. Shift options include overnight and days, and usually at least one weekend day. *Full-time and part-time roles with set schedules may also be available. Apply now.Start as soon as 7 days. No resume or previous work experience required. Amazon is committed to a diverse and inclusive workplace. Amazon is an equal opportunity employer and does not discriminate on the basis of race, national origin, gender, gender identity, sexual orientation, protected veteran status, disability, age, or other legally protected status. For individuals with disabilities who would like to request an accommodation, please visit [ Link Removed ]"
},
```

- **Companies**: In this collection, we store a document for each company in the database. The structure of the document is the following:
 - **_id**: Representing the name of the company. We used id because the name of each company should be unique through the entire database.
 - **password**: Representing the password.
 - **email**: Representing email.

Here is an example of a company document:

```
{
  "_id": "Logic",
  "password": "H$0i7ZB0a4H]y3",
  "email": "apply@logic.com",
},
```

NEO4J ORGANIZATION

Nodes:

We have three types of nodes as follows:

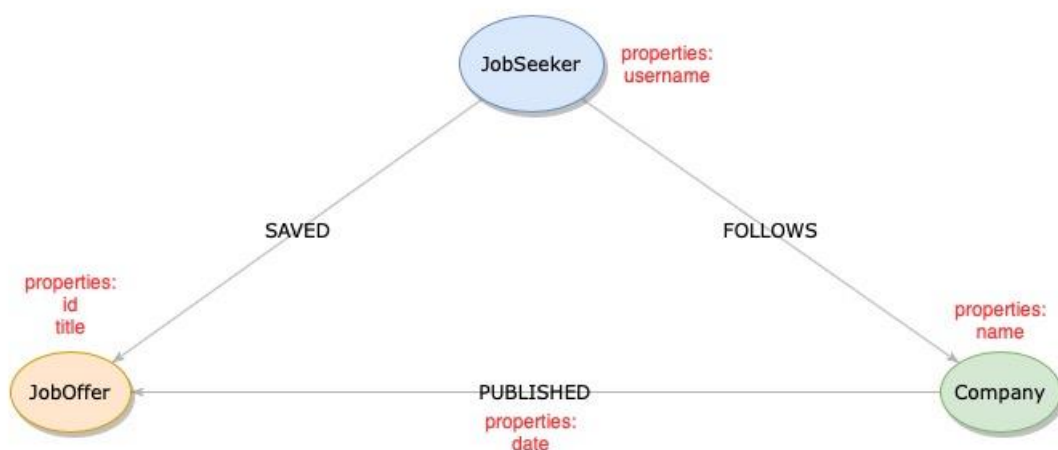
- **JobOffer:** Representing the job offers published in the application. Each job offer node has two properties:
 - **id:** Representing the id of the job which is the same in mongoDB.
 - **title:** Representing the title of the job offer.
- **JobSeeker:** Representing the job seekers registered in the application. Each job seeker node has one property:
 - **username:** Representing the username of the job seeker.
- **Company:** Representing the employers registered in the application. Each company node has one property:
 - **name:** Representing the name of the company.

Relationships:

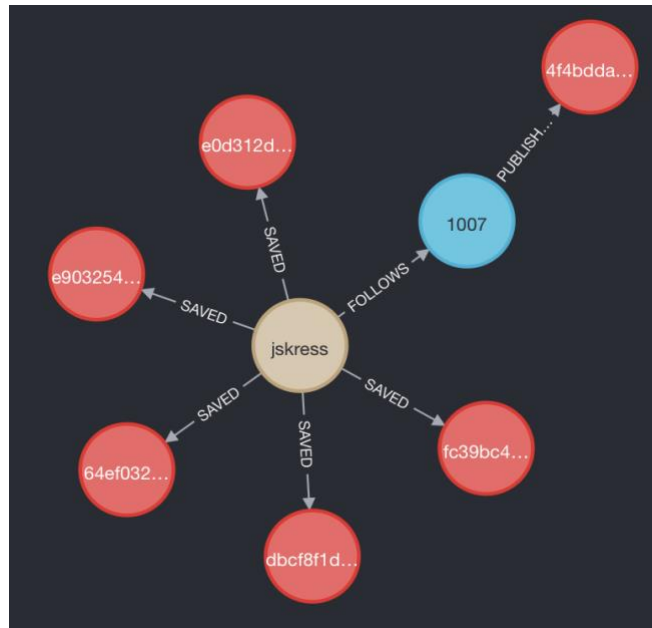
We have three types of relationships as follow:

- **Company → PUBLISHED → JobOffer:** Representing the relationship between companies and their published offers. Each PUBLISHED relationship has one property:
 - **date:** Representing the date which the company published the offer.
- **JobSeeker → FOLLOWS → Company:** Representing the relationship between job seekers and companies that they are following in the application.
- **JobSeeker → SAVED → JobOffer:** Representing the relationship between job seekers and job offers that they have saved in the application.

The structure of the graph database is depicted below:



In the following image, there is an actual snapshot of a small part of the graph database:



In this picture, one job-seeker node (in wheat color), its saved job offers (in red color), its following companies (in blue color), and job offers published by following companies are present. (all these nodes may have other relationships but excluded from the query for the sake of picture's simplicity)

CONSISTENCY BETWEEN NEO4J AND MONGODB

The consistency between document database and graph database needs be kept only during the creation and elimination of the main entities. Therefore, in the following situations, the application manages the consistency:

- **Creation:** After the document successfully inserted in the MongoDB database, the associated node will be created in the Neo4J database. In the case of creating a new job offer, also the relationship to its company will be created.
- **Deletion:** After the document successfully deleted from the MongoDB database, the associated node and all its relationship will be deleted from the Neo4J database. In case of deleting an employer, also all its job offers and their relationships will be deleted.

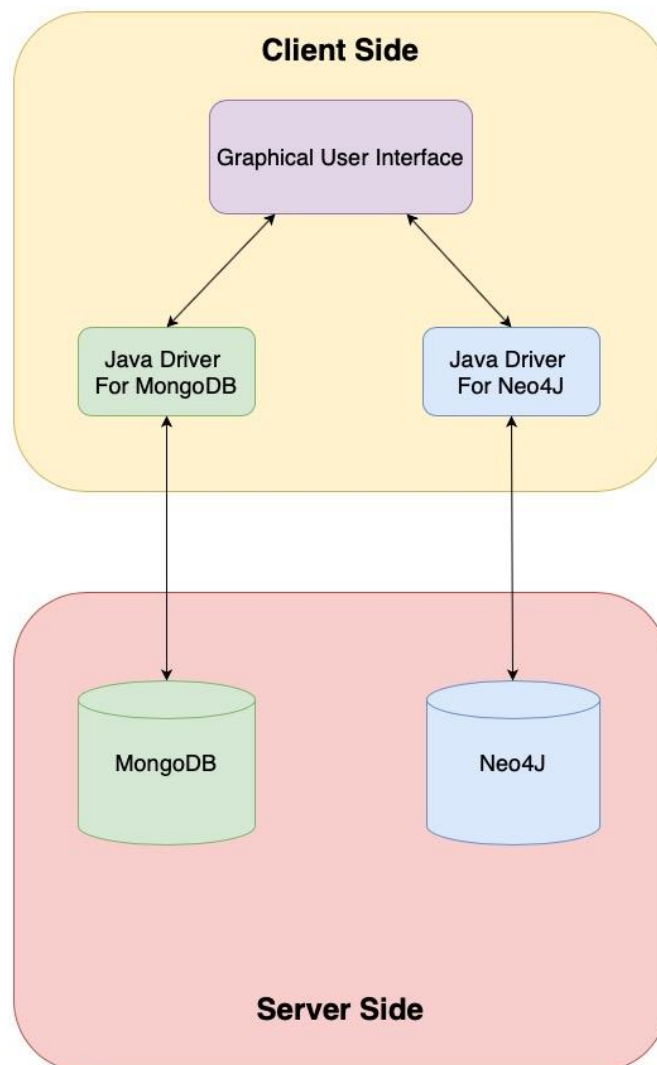
An additional note needs to be done. With the strategy depicted above, we ensure the consistency in case of no failures during writing to the Neo4J database. However, it is still possible that at some point in time, after writing successfully to MongoDB, an issue happens during the write to the Neo4J database and as a result, there will be inconsistent data between these two databases. For this issue, we must apply an additional strategy. The second strategy for ensuring consistency is **logging**. While the application is running, in case of happening any exceptions, these exceptions will be added to the log file. The administrator of the system can review the logs from time to time, to check if failures happened while writing to the databases and fix the issue directly from the database.

SOFTWARE ARCHITECTURE: DESIGN AND IMPLEMENTATION

SOFTWARE ARCHITECTURE

This application is a **client-server application**. On the client side, a graphical user interface based on JavaFx allows users to interact with the application. Moreover, **two Java drivers** has been used to bridge data between the GUI and the server side: one driver for handling MongoDB connections and one for handling Neo4J connections. On the server side, there is a document database and a graph database.

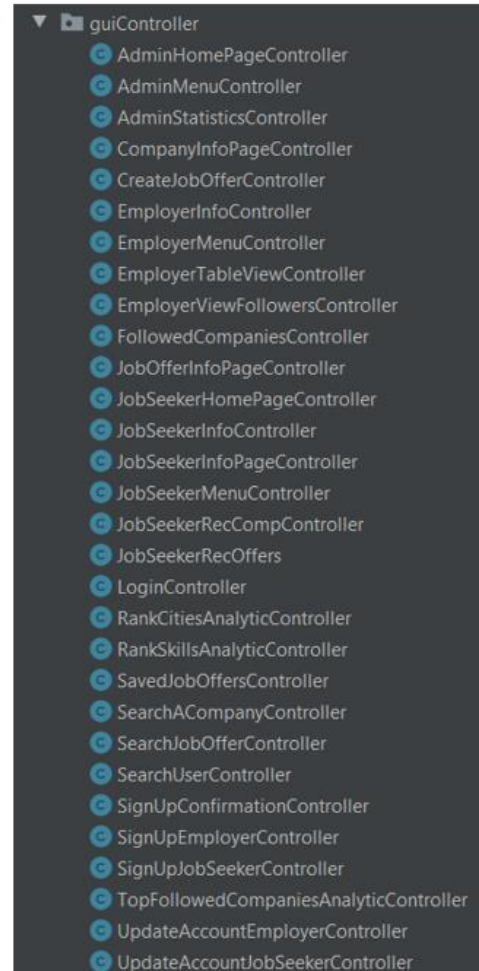
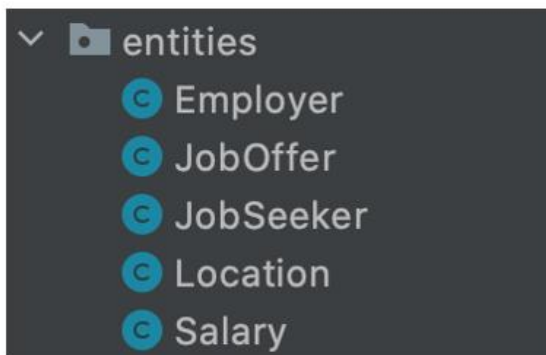
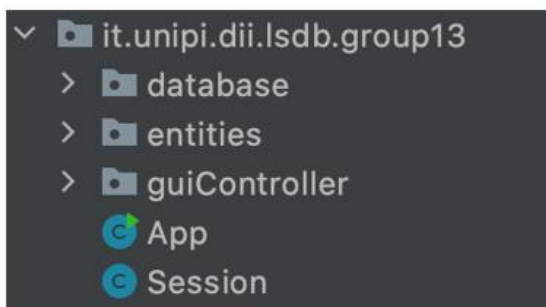
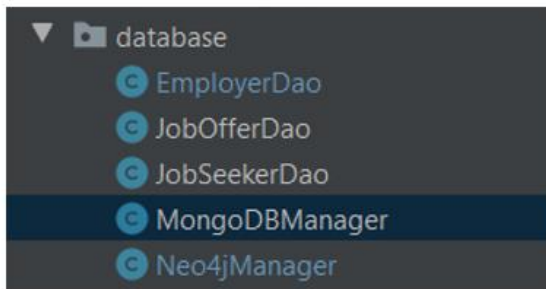
The document database is a MongoDB replica set and the graph database is a Neo4J replica set. The MongoDB replica set consists of one primary and two secondary replicas to ensure eventual consistency. The Neo4J replica set consists of three cores and three replicas to ensure eventual consistency.



IMPLEMENTATION PHASE

The final implementation of the Java application project consists of 3 modules and 42 classes written in Java (in the “java” folder) and some other files (FXML files and one CSS file) to implement the GUI and style it (in the “resources” folder).

These are the java classes:



The application is based on **three main modules** as follows:

- **Entities:** This module corresponds to all the data-related objects that are being transferred between the front-end and the back-end. Entities classes are the following:
 - **JobSeeker:** Representing the job-seeker entity.
 - **Employer:** Representing the employer entity.
 - **JobOffer:** Representing the job offer entity.
 - **Location:** Representing the location entity which is an attribute in JobSeeker and JobOffer classes.
 - **Salary:** Representing the salary entity, which is an attribute in JobOffer class.
- **Database:** This module acts as an interface between the databases and the GUI. It consists of two class that manages connections between database and the application (MongoDbManager and

Neo4JManager), and DAO classes. DAO classes talk to both the databases and the GUI, and move instances of the entities classes between them. Database classes are the following:

- [MongoDBManager](#): Manages connection with the MongoDB database.
 - [Neo4JManager](#): Manages connection with the Neo4J database.
 - [JobSeekerDao](#): This class is used to communicate with the database regarding the operations related to job seekers.
 - [JobOfferDao](#): This class is used to communicate with the database regarding the operations related to job offers.
 - [EmployerDao](#): This class is used to communicate with the database regarding the operations related to employers.
- **GuiController**: This module is used for all the UI logic of the application. GuiController classes are the following:
 - [LoginController](#)
 - [SignUpJobSeekerController](#)
 - [SignUpJobEmployerController](#)
 - [SignUpConfirmationController](#)
 - [JobSeekerHomeController](#)
 - [JobSeekerMenuController](#)
 - [JobSeekerInfoController](#)
 - [JobSeekerInfoPageController](#)
 - [JobOfferInfoPageController](#)
 - [EmployerHomeController](#)
 - [EmployerMenuController](#)
 - [EmployerInfoController](#)
 - [EmployerTableViewController](#)
 - [EmployerViewFollowersController](#)
 - [AdminHomeController](#)
 - [AdminMenuController](#)
 - [AdminStatisticsController](#)
 - [CompanyInfoPageController](#)
 - [TopFollowedCompaniesAnalyticController](#)
 - [CreateJobOfferController](#)
 - [EmployerTableViewController](#)
 - [FollowedCompaniesController](#)
 - [JobSeekerRecCompController](#)
 - [JobSeekerRecOfferController](#)
 - [RankCitiesAnalyticController](#)
 - [RankSkillsAnalyticController](#)
 - [SavedJobOffersController](#)
 - [SearchACompanyController](#)
 - [SearchJobOfferController](#)
 - [SearchUserController](#)
 - [UpdateAccountEmployerController](#)
 - [UpdateAccountJobSeekerController](#)
 - There are two additional classes in the application that are not belonging to these modules:
 - [Session](#): This class is used to maintain information about the session.
 - [App](#): This class is used to create the JavaFx application.

MOST RELEVANT QUERIES USING MONGO DB

In this section, the most relevant queries performed with MongoDB are briefly explained and discussed. We will show only the queries that need an aggregation pipeline to work properly.

ANALYTIC 1: RANKING OF THE TOP 10 CITIES

DESCRIPTION: Find the top 10 cities given a certain job type, where top 10 means the 10 cities that are associated with the greater number of published job offer of the given type.

STAGES:

1. **Match** all the documents in the *job_offers* collection that have the field *job_type* equal to the one given in input
2. **Group** all the founded documents by location (same city and state) and for each group count the occurrences
3. **Sort** the documents founded in the second stage in descending order
4. **Limit** the documents to the first 10 in order to find the top 10 cities

- This is the aggregation pipeline in the mongo shell for the “Full-Time” jobs:

```
db.job_offers.aggregate([
  {$match : {job_type : "Full-Time"}},
  {$group : { _id: "$location", count: {$sum : 1}}},
  {$sort : {"count": -1}},
  {$limit:10}
])
```

- This is the code for MongoDB java driver:

```
public class AdminDao {

    public LinkedHashMap<String, Integer> rankCities(String jobType){
        LinkedHashMap<String, Integer> map = new LinkedHashMap<>();
        // Rank cities based on # of (job_type) jobs
        MongoDBManager mongoDB = MongoDBManager.getInstance();
        mongoDB.getJobOffersCollection();
        Bson m = match(eq( fieldName: "job_type", jobType));
        Bson g = group( id: "$location", Accumulators.sum( fieldName: "count", expression: 1));
        Bson s = sort(Sorts.descending( ...fieldNames: "count"));
        Bson l = limit(10);
        AggregateIterable<Document> aggregate = mongoDB.getJobOffersCollection().aggregate(Arrays.asList(m, g, s, l));
        MongoCursor<Document> iterator = aggregate.iterator();
        while (iterator.hasNext()) {
            Document next = iterator.next();
            Document loc = (Document) next.get("_id");
            map.put(loc.getString( key: "state") + " - " + loc.getString( key: "city"), next.getInteger( key: "count"));
        }
        iterator.close();

        return map;
    }
}
```

ANALYTIC 2: RANKING OF THE TOP 10 SKILLS

Description: Find the top 10 skills, where top 10 means the 10 skills that belong to multiple job seekers.

STAGES:

1. **Unwind** the *skills* array
2. Using the **GroupByCount** operator we group the documents obtained from the first stage that have the same skill², we count their occurrences and we sort them in descending order. Each output document contains two field: an *_id* field containing the distinct grouping value, and a *count* field containing the number of documents belonging to that group.
3. **Limit** the documents to the first 10 in order to find the top 10 skills

- This is the aggregation pipeline in the mongo shell:

```
db.job_seekers.aggregate( [  
  {$unwind: {path: "$skills"}},  
  {$sortByCount: {$toLower: {$trim: {input: "$skills"}}}},  
  {$limit:10}  
] )
```

- This is the code for MongoDB java driver:

```
public LinkedHashMap<String, Integer> rankSkills(){  
    MongoDBManager mongoDB = MongoDBManager.getInstance();  
    MongoCollection collection = mongoDB.getJobSeekersCollection();  
    Bson uw = unwind( fieldName: "$skills");  
    // trim operator to remove the whitespace characters and toLower operator to change all to lower case  
    Bson sb = sortByCount(eq( fieldName: "$toLower", eq( fieldName: "$trim", eq( fieldName: "input", value: "$skills"))));  
    Bson limit = limit(10);  
    AggregateIterable aggregate = collection.aggregate(Arrays.asList(uw, sb, limit));  
    LinkedHashMap<String, Integer> map = new LinkedHashMap<String, Integer>();  
    MongoClientCursor<Document> iterator = aggregate.iterator();  
    while (iterator.hasNext()) {  
        Document next = iterator.next();  
        map.put(next.getString( key: "_id"), next.getInteger( key: "count"));  
    }  
    iterator.close();  
    return map;  
}
```

² Due to the format of the skills' strings we have also to use other two operators: the **\$trim** operator (to remove whitespace characters at the beginning and at the end of the skill' string) and the **\$toLower** operator (to put the skill' string in lower characters)

GET JOB OFFERS BY CITY

Description: Find all the job offers that have the field *city* of the *location* embedded document equal to the city given in input

STAGES:

1. First of all, for each job offer document we **project** all the fields of the document (except the salary) and the field *lower* that is computed as the field *city* of the *location* embedded document written in lower case.
 2. Then we **match** the documents that have the *lower* field equal to the city given in input
- This the aggregation pipeline in the mongo shell for the city “new york”:

```
db.job_offers.aggregate( [
  { $project: {
    job_title : 1,
    company_name : 1,
    job_type : 1,
    location : 1,
    post_date : 1,
    lower : { $toLower : "$location.city" } },
  { $match: { lower : "new york" } } ] )
```

- This is the code for MongoDB java driver:

```
public List<JobOffer> getJobOffersByCity(String city){
    List<JobOffer> jobOffers = new ArrayList<>();
    MongoDBManager mongoDB = MongoDBManager.getInstance();
    MongoCollection<Document> collection= mongoDB.getJobOffersCollection();
    AggregateIterable<Document> founded = collection.aggregate(Arrays.asList(project(fields(include( ...fieldNames: "job_title",
        "job_type", "job_description", "company_name", "location", "post_date"), computed( fieldName: "lower",
        eq( fieldName: "$toLower", value: "$location.city" ))), match(eq( fieldName: "lower", city )))));
    for(Document doc: founded) {
        jobOffers.add(parseJobOffer(doc));
    }
    return jobOffers;
}
```

GET JOB OFFERS BY COMPANY

Description: Find all the job offers that have the field *company_name* equal to the name of the company given in input

STAGES:

1. First of all, for each job offer document we **project** all the fields of the document (except the salary) and the field *lower* that is computed as the field *company_name* written in lower case.
2. Then we **match** the documents that have the *lower* field equal to the company name given in input

- This the aggregation pipeline in the mongo shell for the company "superior group":

```
db.job_offers.aggregate( [  
  { $project: {  
    job_title : 1,  
    company_name : 1,  
    job_type : 1,  
    location : 1,  
    post_date : 1,  
    lower : { $toLower : "$company_name" } } },  
  { $match: { lower : "superior group" } } ] )
```

- This is the code for MongoDB java driver:

```
public List<JobOffer> getJobOffersByCompany(String company){  
    List<JobOffer> jobOffers = new ArrayList<>();  
    MongoDBManager mongoDB = MongoDBManager.getInstance();  
    MongoCollection<Document> collection= mongoDB.getJobOffersCollection();  
    AggregateIterable<Document> founded = collection.aggregate(Arrays.asList(project(fields(  
        include( ...fieldNames: "job_title", "job_type", "job_description", "company_name", "location", "post_date"),  
        computed( fieldName: "lower", eq( fieldName: "$toLower", value: "$company_name" ))), match(eq( fieldName: "lower", company))))  
    for(Document doc: founded) {  
        jobOffers.add(parseJobOffer(doc));  
    }  
    return jobOffers;  
}
```

GET JOB OFFER BY SALARY

Description: Find all the job offers that have the minimum salary greater or equal the value given in input (considering also the time unit that must be equal to the one given in input).

STAGES:

1. First of all, we **match** all the documents in the *job_offer* collection that have the field *time_unit* of the *salary* embedded document equal to the one given in input
 2. Then for each job offer document we **project** its fields excluding the *_id* field and we calculate two new fields:
 - a. the *time* field (that is equal to the *time_unit* field of the *salary* embedded document)
 - b. the *min* field (that is obtained parsing as a double the field *from* of the *salary* embedded document)³
 3. Finally, we **match** all the documents obtained from the second stage that have the *min* field equal or greater than the minimum salary given in input
- This the aggregation pipeline in the mongo shell for job offers with salary time unit equal to “/year” and a minimum salary of 7000:

```
db.job_offers.aggregate([
  {$match : { "salary.time_unit" : "/year"}},
  {$project : {
    job_title : 1,
    company_name : 1,
    job_type : 1,
    location : 1,
    salary : 1,
    post_date : 1,
    time : "$salary.time_unit",
    min : {$convert: {
      input: {$reduce: {
        input: {$split: ["$salary.from", ","]},
        initialValue: "",
        in: {$concat: ["$$value", "$$this"]}
      }},
      to : "double",
      onError : "0"}
    }
  }},
  {$match: { min : {$gte : 7000}}}, {$limit : 10}
])
```

Notes: the **\$convert** operator takes in input a string and parse it as a double. The **\$split** operator splits the field *salary.from* using comma. The **\$reduce** operator applies the function defined in the *in* field to each element of the input array.

³ Due to the format of the *from* field (it is a string and it is written using the English standard, so with commas to divide the integer digits) we also have to use the **\$convert**, the **\$reduce**, the **\$split** and the **\$concat** operators.

- This is the code for MongoDB java driver:

```
public List<JobOffer> getJobOffersBySalary(String timeunit, Double minimum){
    List<JobOffer> jobOffers = new ArrayList<>();
    MongoDBManager mongoDB = MongoDBManager.getInstance();
    MongoCollection collection = mongoDB.getJobOffersCollection();
    Bson m = match(eq( fieldName: "salary.time_unit", timeunit));
    Bson p = project(fields(include( ...fieldNames: "job_title", "company_name", "job_type","location", "salary", "post_date"),
        computed( fieldName: "time", expression: "$salary.time_unit"),
        computed( fieldName: "min",
            eq( fieldName: "$convert",
                new BsonDocument("input", new BsonDocument("$reduce",
                    new BsonDocument("input", new BsonDocument("$split",new BsonArray(Arrays.asList(
                        new BsonString( value: "$salary.from"),
                        new BsonString( value: ","))))))
                    .append("initialValue", new BsonString( value: ""))
                    .append("in", new BsonDocument("$concat", new BsonArray(Arrays.asList(
                        new BsonString( value: "$$value"),
                        new BsonString( value: "$$this"))))))))
                    .append("to",new BsonString( value: "double"))
                    .append("onError",new BsonString( value: "0")))))));
    Bson n = match(gte( fieldName: "min", minimum));
    AggregateIterable<Document> founded = collection.aggregate(Arrays.asList(m, p , n));
    for(Document doc: founded) {
        jobOffers.add(parseJobOffer(doc));
    }
    return jobOffers;
}
```

MOST RELEVANT QUERIES USING NEO4J

In this section, the most relevant queries performed with Neo4J are briefly explained and discussed (both in a graph-centric manner and in a “domain-specific” manner) and the Cypher code to implement them is shown. Then we discuss about the recommendation system (that uses Neo4J database to work) showing some tests performed to verify it is working.

GRAPH CENTRIC AND DOMAIN-SPECIFIC QUERIES

1. FOLLOWED COMPANIES

This query is used to retrieve all the companies followed by a certain user.

To see the actual implementation see the method *followedCompanies(String username)* of *JobSeekerDao* class.

DOMAIN-SPECIFIC QUERIES	GRAPH-CENTRIC QUERIES
Given a job seeker, what are the companies that he follows?	What are the nodes linked to the given job seeker by the FOLLOWS relationship?

This is the Cypher code to implement the query for the job seeker with username “aacosta”:

```
1 MATCH (js:JobSeeker)-[:FOLLOWS]→(c) WHERE js.username = "aacosta"
2 RETURN c.name as Name
```

2. SAVED JOB OFFERS

This query is used to retrieve all the job offers saved by a certain user.

To see the actual implementation see the method *savedOffers(String username)* of *JobSeekerDao* class.

DOMAIN-SPECIFIC QUERIES	GRAPH-CENTRIC QUERIES
Given a job seeker, what are the job offers that he saved?	What are the nodes linked to the given job seeker by the SAVED relationship?

This is the Cypher code to implement the query for the job seeker with username “aacosta”:

```
1 MATCH (js:JobSeeker)-[:SAVED]→(jo) WHERE js.username = "aacosta"
2 RETURN jo.title as title, jo.id AS id
```

3. PUBLISHED JOB OFFERS

This operation is used to when an employer account wants to retrieve the job offers that he published.

To see the actual implementation see the method *findFollowers(String companyName)* of *EmployerDao* class.

DOMAIN-SPECIFIC QUERIES	GRAPH-CENTRIC QUERIES
What are the job offers published by the company?	Given the node with label "Company" and property "name" equal to the company name given in input, what are the nodes linked to it by the PUBLISHED relationship?

This is the Cypher code to implement the query for the company with name "*Tuesday Morning*":

```
1 MATCH (c:Company {name: "*Tuesday Morning*"})←[:FOLLOWS]-(j:JobSeeker)
2 RETURN j.username AS username
```

4. 3rd ANALYTIC FOR ADMIN: FIND TOP 10 COMPANIES

This operation is used by the admin account to retrieve the top 10 companies, where top ten means the ten companies that are followed by more users.

To see the actual implementation see the method *findTopCompanies()* of *EmployerDao* class.

DOMAIN-SPECIFIC QUERIES	GRAPH-CENTRIC QUERIES
What are the 10 companies that are followed by more users?	What are the nodes with label equal to "Company" that have more ingoing links of type FOLLOWS?

This is the Cypher code to implement the query:

```
1 MATCH(co:Company)←[r:FOLLOWS]-(js:JobSeeker)
2 WITH co, count (r) as rels
3 RETURN co.name AS name, rels AS relation ORDER BY rels DESC
4 LIMIT 10
```

1st RECOMMENDATION SYSTEM: RECOMMEND COMPANIES TO USERS

The idea behind this recommendation system is the following: if a job seeker “A” follows a company “C1” and “C1” is followed also by the job seeker “B”, then “A” could be interested in the companies followed by “B”. Moreover, the greater is the number of common companies followed by both the users, the greater is the probability that “A” will be interested in the companies followed by “B” (an in the same manner that “B” will be interested in the companies followed by “A”). We call this probability “strength”.

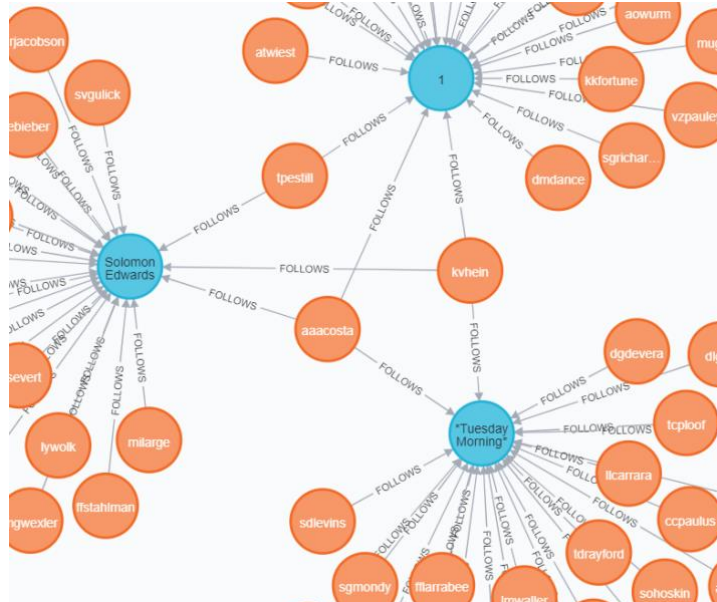
- This is the Cypher code for the user with username "aaacosta":

```

1 MATCH (u1:JobSeeker)-[:FOLLOWS]→(c:Company)←[:FOLLOWS]-(u2:JobSeeker)
2 WHERE u1.username = "aaacosta" AND u1.username <> u2.username
3 WITH u2 AS foundedUser, count(DISTINCT c) AS strenght
4 MATCH (foundedUser)-[:FOLLOWS]→(c1:Company)
5 WHERE NOT EXISTS { (j:JobSeeker {username : "aaacosta"})-[:FOLLOWS]-(c1) }
6 RETURN foundedUser.username, c1.name, strenght ORDER BY strenght DESC LIMIT 10

```

To see that it works, we performed some tests starting from the situation in the following figure.



In the image, the orange nodes are the job seekers, the blue nodes represent the companies. The **FOLLOWS** relationship links them. This is the cypher code to obtain this image:

```

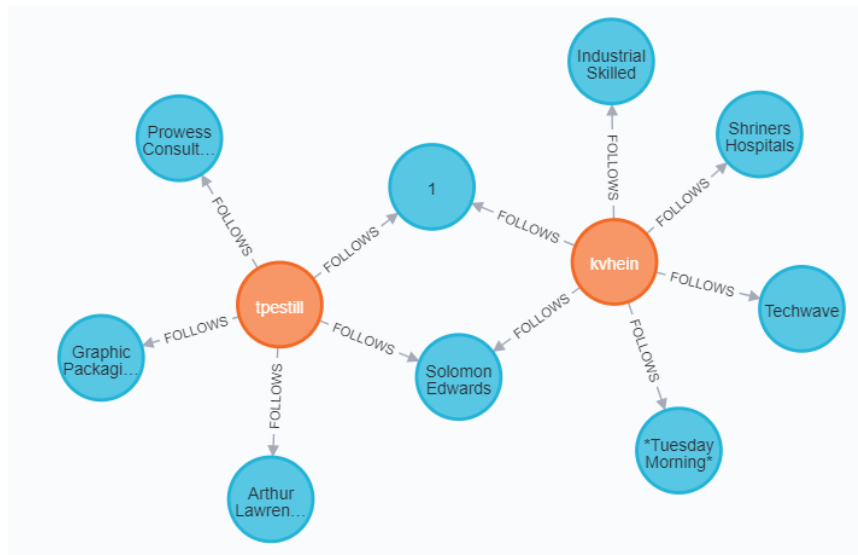
MATCH p = (j1:JobSeeker)-[:FOLLOWS]->(c:Company), q=(c:Company)-[:FOLLOWS]-(j2:JobSeeker)
WHERE j1.username = "aaacosta" AND j1.username<>j2.username
RETURN p,q

```

Error! Reference source not found. As we can see, three job seekers follow common companies:

- “aacosta” and “kvhein” follow three common companies
- “aacosta” and “tpestill” follow two common companies

TError! Reference source not found. following figure shows the companies followed individually by “kyhein” and “tpestill”:



In the image the orange nodes are the job seekers, the blue nodes represent the companies. The FOLLOWS relationship links them. In particular, we focus on two job seekers “tpestill” and “kvhein” and the companies which they follow. This is the cypher code to obtain this image:
MATCH p = (j1:JobSeeker)-[:FOLLOWS]->(c:Company), q=(c1:Company)-[:FOLLOWS]-(j2:JobSeeker)
WHERE j1.username = "tpestill" AND j2.username="kvhein"
RETURN p,q

With the Cypher code showed at the beginning, we obtain the following result:

"foundedUser.username"	"c1.name"	"strenght"
"kvhein"	"Industrial Skilled Trades"	3
"kvhein"	"Shriners Hospitals For Children"	3
"kvhein"	"Techwave Consulting Inc"	3
"tpestill"	"Arthur Lawrence"	2
"tpestill"	"Prowess Consulting, Llc"	2
"tpestill"	"Graphic Packaging"	2
"sohoskin"	"Gemalto Inc."	1
"sdlevins"	"Inframark, LLC"	1
"dlgoff"	"Savantis Group Inc."	1
"sgmondy"	"SARANSH, Inc."	1

So, as we expect, the companies followed by “kvhein” are suggested with an higher strength (strength equal to 3 because he has with “aaacosta” 3 common followed companies), then we have the ones followed by “tpestill” (with strength equal to two because he has 2 common followed companies with “aaacosta”) and then other companies suggested with strength 1.

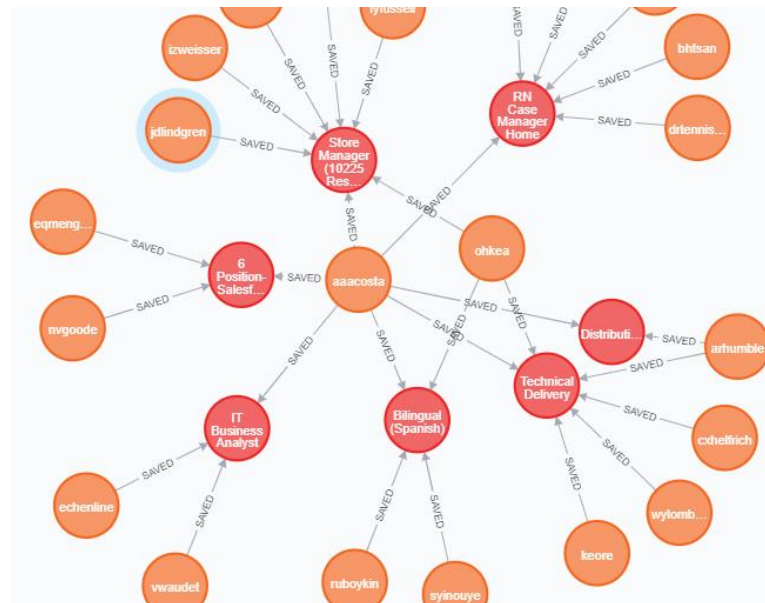
2nd RECOMMENDATION SYSTEM: RECOMMEND JOB OFFERS TO USERS

The idea behind this recommendation system is similar to the one behind the first one: if a job seeker “A” has saved a job offer “J1” and “J1” is saved also by the job seeker “B”, then “A” could be interested in the job offers saved by “B”. Moreover, the greater is the number of common job offers saved by both the users, the greater is the probability that “A” will be interested in the job offers saved by “B” (an in the same manner that “B” will be interested in the job offers saved by “A”). We call this probability “strength”.

- This is the Cypher code for the user with username “aacosta”:

```
1 MATCH (u1:JobSeeker)-[:SAVED]->(j1:JobOffer)<-[:SAVED]-(u2:JobSeeker)
2 WHERE u1.username = "aacosta" AND u1.username <> u2.username
3 WITH u2 AS foundedUser, count(DISTINCT j1) AS strength
4 MATCH (foundedUser)-[:SAVED]->(j2:JobOffer)<-[:PUBLISHED]-(c:Company)
5 WHERE NOT EXISTS { (u:JobSeeker {username : "aacosta"})-[:SAVED]-(j2) }
6 RETURN foundedUser.username, j2.id as id, j2.title AS title, date(p.date) as
   postDate, c.name as company, strength ORDER BY strength DESC LIMIT 15
```

To see that if works, we performed some tests starting from the situation showed in the following figure.



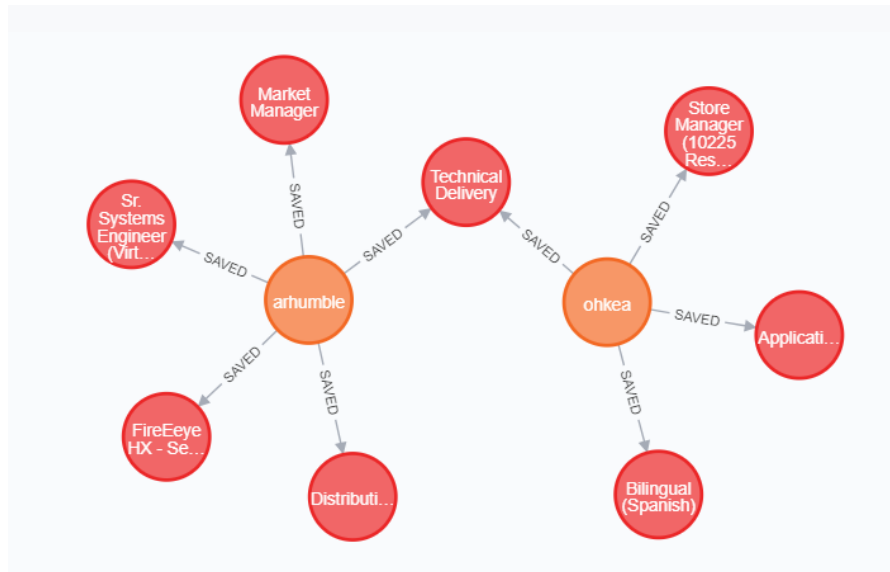
In the image, the orange nodes are the job seekers, the red ones represent the job offers. The SAVED relationship links them. This is the cypher code to obtain this image:

```
MATCH p = (u1:JobSeeker)-[:SAVED]->(j1:JobOffer)<-[:SAVED]-(u2:JobSeeker)
WHERE u1.username = "aacosta" AND u1.username <> u2.username
RETURN p
```

As we can see, there are some job seekers with common saved job offer:

- “aacosta” and “ohkea” saved three common job offers
- “aacosta” and “arhumble” saved two common job offers

The following figure shows the job offers saved by “ohkea” and “arhumble”:



In the image, the orange nodes are the job seekers, the red ones represent the job offers. The SAVED relationship links them. In particular, we focus on two job seekers “arhumble” and “ohkea” and their saved job offers. This is the cypher code to obtain this image:

```
MATCH p = (u1:JobSeeker)-[:SAVED]->(j1:JobOffer), q = (u2:JobSeeker)-[:SAVED]->(j2:JobOffer)
WHERE u1.username = "arhumble" AND u2.username="ohkea"
RETURN p,q
```

With the Cypher code shown at the beginning, we obtain the following result:

"foundedUser.username"	"id"	"title"	"postDate"	"company"	"strength"
"ohkea"	"d6ba918aea4f52318a9443afbbf3c5e5"	"Application Support Engineer"	"2020-11-15"	"Kforce Inc."	3
"arhumble"	"41fae25f1a170b16216e034a0dd99f06"	"Sr. Systems Engineer (Virtualization) (Security Clearance and IAT Level 3 a must) (ID#002)"	"2020-12-15"	"SRG Government Services"	2
"arhumble"	"6f9471658d524167c4bec7045fd1d82d"	"Market Manager"	"2020-02-23"	"Empire Today"	2
"arhumble"	"4a3f8742fae151757eb290a774968371"	"FireEye HX - Security Engineer"	"2020-12-21"	"Turnberry Solutions"	2
"cxhelfrich"	"a537bccf99eb732d139a0ed7ad1d7cfb"	"Software Architect (Android/JAVA)"	"2020-12-28"	"NCR Corporation"	1
"keore"	"7bfd4412edb14a3231bf94d2f0310f15"	"PT Sales Associate 7124"	"2020-03-30"	"The Sherwin-Williams Company"	1
"keore"	"d398d408dded4030b8a4af0091db478a"	"Part Time Teller - Cal State Long Beach"	"2020-03-07"	"Wells Fargo"	1
"wylombard"	"341839c43b90795d9dc7bb4c4cc377e3"	"Project Manager"	"2021-01-12"	"Platinum Resource Group"	1
"keore"	"a348830c9ab983bd22e4096a9022b8fe"	"Object Oriented Software Engineer - \$120K + stock options"	"2021-01-26"	"CyberCoders"	1

So as we expect the job offers saved by “ohkea” but not by “aacosta” are recommended to him with strength equal to 3 (they have in common three saved job offers). Then we have the job offers saved by “arhumble” recommended with strength equal to 2. Finally, we have the others offers saved by users that have with “aacosta” only one common saved job offer (strength equal to 1)

INDEXES

One of the non-functional requirements for the application is the fast response. In this case, indexing is of great importance, because it aids in an efficient execution of read queries. In the final version of our project we decided to have two single property indexes:

- An **index for the `post_date` in the `job_offer` collection**, sorted in descending order
- An **index for the `username` property of the nodes with label `JobSeeker`**

In the following subsections, we briefly explain the choice of introducing these indexes and we show some tests done in order to verify their benefits.

MONGODB INDEXES

In all the queries on `job_offers` collection, we sort the retrieved job offers from the newest to old published job offer, so it proved to be a good approach to index `post_date` in a descending order.

Attached below is an example test query used to experiment our approach:

```
> db.job_offers.find({"job_type":"Full-Time"}).sort({"post_date":-1}).explain("executionStats")
```

We performed this simple query to test the performance of execution of query with and without indexing.

The execution time without indexing during sort operation is 1375 ms:

```
"executionStages" : {
  "stage" : "SORT",
  "nReturned" : 33171,
  "executionTimeMillisEstimate" : 1375,
```

Let us make the `post_date` field indexed now:

```
> db.job_offers.createIndex({post_date:-1})
```

With indexing the execution time reduced by more than 90 percent of the time before indexing. The attached picture supports our choice:

```
"inputStage" : {
  "stage" : "IXSCAN",
  "nReturned" : 50148,
  "executionTimeMillisEstimate" : 92,
  "works" : 50149,
  "advanced" : 50148,
  "needTime" : 0,
  "needYield" : 0,
  "saveState" : 51,
  "restoreState" : 51,
  "isEOF" : 1,
  "keyPattern" : {
    "post_date" : -1
  },
  "indexName" : "post_date_-1",
```

NEO4J INDEXES

In the graph database operations, most of the queries are related to the job seekers. In these queries, the search operation is on the username of the job seekers. Therefore, the only useful indexing could be for the job seekers nodes. Moreover, most of the operation in graph database are read operations, hence the cost of slower writes is trivial comparing to the value of more efficient searches.

In the following, we try to validate the use of indexing on the username of job seekers. First, we perform some operations before creating indexes.

In the following image, there is one of the actual operations that is done on a random user in Neo4J browser:

```
1 MATCH (:JobSeeker{username:"jskress"})-[:FOLLOWS]→(c)-[r:PUBLISHED]→(j)
2 RETURN c,r,j ORDER BY r.date DESC
```

The performance of the result is the following:

Started streaming 1 records after 3 ms and completed after 226 ms.

Let us create the index on the property *username* of the *JobSeeker* nodes:

```
neo4j$ CREATE INDEX jobseeker_index FOR (j:JobSeeker) ON (j.username)
```

After the creation of index, this is the performance of our query:

Started streaming 1 records after 2 ms and completed after 69 ms.

As can be seen, the time of the operation after indexing is approximately one-third of time of the operation before indexing. This is almost the same in all other similar queries. As a result, we are convinced that indexing username can be beneficial for the sake of performance in the application.

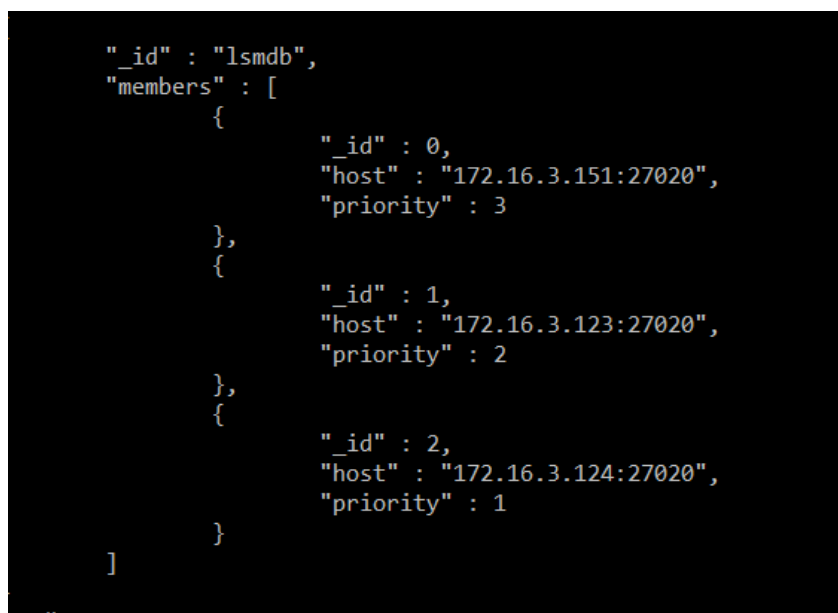
TEST ON AVAILABILITY OF THE APPLICATION

One of the main non-functional requirements of our application, is the availability. In particular, we want that even in case of failure of the primary, the application continues working in a properly manner and a user can still use it.

To prove that our requirement is satisfied, we performed a test shutting down the primary and making read requests with the application. **We proved that even if the primary fails, the application continues to offer its service to the user. Applying the same reasoning for the other virtual machines, we can say that, since we have three replicas, we can handle two server crashes. In this last case, the application will still work using only one server.** A little note must be done: it could happen that the primary fails when a writing operations is being done. In this case, if the write operation isn't still be replicated among the secondary servers, the write update will be lost.

In the following lines, we will briefly discuss about the test we performed.

First of all, we should talk about the configuration of our mongoDB replica set. We set it as is shown in the following image:

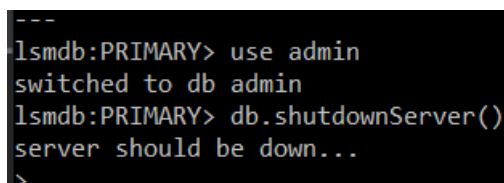


```
    "_id" : "lsmdb",
    "members" : [
      {
        "_id" : 0,
        "host" : "172.16.3.151:27020",
        "priority" : 3
      },
      {
        "_id" : 1,
        "host" : "172.16.3.123:27020",
        "priority" : 2
      },
      {
        "_id" : 2,
        "host" : "172.16.3.124:27020",
        "priority" : 1
      }
    ]
```

So, as we can see, the virtual machine with IP address 172.16.3.151 has the greater priority and, if available, will be always chosen as the primary during the leader election of the replica set.

Now we start the application using a job seeker account and we perform some reading operations. Checking the status of the replica set, as we expect, we have as primary the virtual machine with highest priority.

Now, while the application is still running, we shut down the primary:



```
lsmdb:PRIMARY> use admin
switched to db admin
lsmdb:PRIMARY> db.shutdownServer()
server should be down...
>
```

The application is still working and we can correctly retrieve data of our database using it. **So, when a failure occurs in the primary, the application can still offer its service to the user.**

Connecting to the virtual machine with IP address 172.16.3.123 we can see (through the `rs.status()` command) that now it has been elected as the primary, and that the machine at 172.16.3.151 has a not reachable state:

```
    "members" : [
      {
        "_id" : 0,
        "name" : "172.16.3.151:27020",
        "health" : 0,
        "state" : 8,
        "stateStr" : "(not reachable/healthy)",
        "uptime" : 0,
        "optime" : {
          "ts" : Timestamp(0, 0),
          "t" : NumberLong(-1)
        },
        "optimeDurable" : {
          "ts" : Timestamp(0, 0),
          "t" : NumberLong(-1)
        },
        "optimeDate" : ISODate("1970-01-01T00:00:00Z"),
        "optimeDurableDate" : ISODate("1970-01-01T00:00:00Z"),
        "lastHeartbeat" : ISODate("2021-02-15T17:52:22.647Z"),
        "lastHeartbeatRecv" : ISODate("2021-02-15T17:52:01.176Z"),
        "pingMs" : NumberLong(0),
        "lastHeartbeatMessage" : "Error connecting to 172.16.3.151:27020 :: caused by :: Connection refused",
        "syncSourceHost" : "",
        "syncSourceId" : -1,
        "infoMessage" : "",
        "configVersion" : 2,
        "configTerm" : 8
      },
    ],
```

```
    {
      "_id" : 1,
      "name" : "172.16.3.123:27020",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 1714,
      "optime" : {
        "ts" : Timestamp(1613411540, 1),
        "t" : NumberLong(9)
      },
      "optimeDate" : ISODate("2021-02-15T17:52:20Z"),
      "syncSourceHost" : "",
      "syncSourceId" : -1,
      "infoMessage" : "",
      "electionTime" : Timestamp(1613411520, 1),
      "electionDate" : ISODate("2021-02-15T17:52:00Z"),
      "configVersion" : 2,
      "configTerm" : 9,
      "self" : true,
      "lastHeartbeatMessage" : ""
    },
    {
      "_id" : 2,
      "name" : "172.16.3.124:27020",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 1671,
      "optime" : {
        "ts" : Timestamp(1613411540, 1),
        "t" : NumberLong(9)
      },
      "optimeDurable" : {
        "ts" : Timestamp(1613411540, 1),
        "t" : NumberLong(9)
      },
      "optimeDate" : ISODate("2021-02-15T17:52:20Z"),
      "optimeDurableDate" : ISODate("2021-02-15T17:52:20Z"),
      "lastHeartbeat" : ISODate("2021-02-15T17:52:22.635Z"),
      "lastHeartbeatRecv" : ISODate("2021-02-15T17:52:22.641Z"),
      "pingMs" : NumberLong(0),
      "lastHeartbeatMessage" : "",
      "syncSourceHost" : "172.16.3.123:27020",
      "syncSourceId" : 1,
      "infoMessage" : "",
      "configVersion" : 2,
      "configTerm" : 9
    }
  ],
```

We can now restart our mongod instance at 172.16.3.151 and, due to its priority, as we expect it will be elected as the primary:

```
},
"members" : [
  {
    "_id" : 0,
    "name" : "172.16.3.151:27020",
    "health" : 1,
    "state" : 1,
    "stateStr" : "PRIMARY",
    "uptime" : 11,
    "optime" : {
      "ts" : Timestamp(1613411700, 3),
      "t" : NumberLong(10)
    },
    "optimeDurable" : {
      "ts" : Timestamp(1613411700, 3),
      "t" : NumberLong(10)
    },
    "optimeDate" : ISODate("2021-02-15T17:55:00Z"),
    "optimeDurableDate" : ISODate("2021-02-15T17:55:00Z"),
    "lastHeartbeat" : ISODate("2021-02-15T17:55:01.778Z"),
    "lastHeartbeatRecv" : ISODate("2021-02-15T17:55:00.766Z"),
    "pingMs" : NumberLong(0),
    "lastHeartbeatMessage" : "",
    "syncSourceHost" : "",
    "syncSourceId" : -1,
    "infoMessage" : "",
    "electionTime" : Timestamp(1613411700, 2),
    "electionDate" : ISODate("2021-02-15T17:55:00Z"),
    "configVersion" : 2,
    "configTerm" : 10
  },
],
```

```
{
  " _id" : 1,
  "name" : "172.16.3.123:27020",
  "health" : 1,
  "state" : 2,
  "stateStr" : "SECONDARY",
  "uptime" : 1872,
  "optime" : {
    "ts" : Timestamp(1613411700, 3),
    "t" : NumberLong(10)
  },
  "optimeDate" : ISODate("2021-02-15T17:55:00Z"),
  "syncSourceHost" : "172.16.3.151:27020",
  "syncSourceId" : 0,
  "infoMessage" : "",
  "configVersion" : 2,
  "configTerm" : 10,
  "self" : true,
  "lastHeartbeatMessage" : ""
},
{
  " _id" : 2,
  "name" : "172.16.3.124:27020",
  "health" : 1,
  "state" : 2,
  "stateStr" : "SECONDARY",
  "uptime" : 1829,
  "optime" : {
    "ts" : Timestamp(1613411700, 3),
    "t" : NumberLong(10)
  },
  "optimeDurable" : {
    "ts" : Timestamp(1613411700, 3),
    "t" : NumberLong(10)
  },
  "optimeDate" : ISODate("2021-02-15T17:55:00Z"),
  "optimeDurableDate" : ISODate("2021-02-15T17:55:00Z"),
  "lastHeartbeat" : ISODate("2021-02-15T17:55:01.778Z"),
  "lastHeartbeatRecv" : ISODate("2021-02-15T17:55:00.780Z"),
  "pingMs" : NumberLong(0),
  "lastHeartbeatMessage" : "",
  "syncSourceHost" : "172.16.3.123:27020",
  "syncSourceId" : 1,
  "infoMessage" : "",
  "configVersion" : 2,
  "configTerm" : 10
}
```