Design and implementation of an Algorithm for an Author Disambiguation problem

Universitat de Lleida
Escola Politècnica Superior

Lluís Echeverria Rovira

Robert Garcia Gonzalez

September - 2017

Summary

Person name disambiguation is basic to distinguish persons that share the same name where unique identifiers are not defined. This problem is common in many domains, including digital libraries or data bases with publications, where the same name can refer to multiple unique authors. With the aim to attributing correctly the work, the data bases must be disambiguated.

This project wants to give a possible solution to this problem, designing and implementing an algorithm for the disambiguation of the names. Different techniques and tools, within the scope of the distributed computations, like Spark or Hadoop, will be used in the development, in order to improve the efficiency of the process.

As a base data set, the more than 8 millions of publications from the AGRIS (International System for Agricultural and Technology) repository will be used in the disambiguation process.

Index

Index of Figures

Index of Tables

1. Introduction

Once, one friend told me:

"If you would like to be a good Data Scientist in the Big Data scope, to do so, first you have to be a good Data Analyst and a good Data Architect"

In other words, the above sentence means that when the concepts of data science and Big Data come together, despite the data analysis remains important, also come into play the ways or methods of how to deal to work with the big amount of data.

As a final part of the Big Data master degree, this project aims to look into with all the subjects and knowledge acquired during this two years, and also tries to explore and extend new methods and tools related to them.

Specifically, I wanted to work with distributed tools, for the data science and Big Data, in a cluster scope. I wanted to see how these tools help to improve the performance of the computations in a distributed way, instead than in traditional methods. Tools like Hadoop, with the HDFS distributed file system, or the Spark data processing engine.

Finally one challenge was presented to me, which involved, more or less, all of these ideas. This is an 'Author Name Disambiguation' problem type, and this project is the result of the efforts made in order to give a possible solution for it.

2. The problem: Author name disambiguation

As I have introduced above, this project is involved in the 'Author Name Disambiguation' problem.

This kind of problem is a type of a Record Linkage[1] task that is applied to scholarly documents, where the goal is to find all mentions of the same author and cluster them together.

Users of digital libraries or databases where publication, papers or articles are stored, usually want to know the exact author or authors of a document. The problem comes when different authors may share the same name, either as full names or as initials and last names, or the same author uses different abbreviations of his name in different publications. Generally, there are the following cases:

1) Aliasing problem: an author uses different name variations such as "Albert A. Liaw" and "A. A. Liaw".
2) Common name problem: there is more than one person with the same name.
3) Typographic errors.
4) Different combinations of the previous cases.

Finally, Name Disambiguation can have other goals, like enable a better bibliometric analysis by allowing a more accurate counting and grouping of publications and citations.

[1] Record linkage (RL) is the task of finding records in a data set that refer to the same entity across different data sources.

3. State of art

Author name ambiguity in bibliographic databases has long been recognized as an important problem [Garfield E. British quest for uniqueness versus American egocentrism. Nature. 1969;223:763.], so the community has worked a lot, and has a large documentation and different solutions and implementations.

Fundamental approaches include:

1. Manual assignment by librarians [Scoville CL, Johnson ED, McConnell AL. When A. rose is not A. Rose: The vagaries of author searching. Med. Refer. Serv. Quart. 2003;22:1–11].
2. Community-based efforts[2].
3. Unsupervised clustering that groups articles by similarity [Soler JM. Separating the articles of authors with the same name. Scientometrics. 2007;72:281–290].
4. Supervised methods that utilize manually compiled training sets [Reuther P, Walter B. Survey on test collections and techniques for personal name matching. Int. J. Metadata, Seman. Ontol. 2006;1:89–99].
5. Methods that go behind pairwise analysis of explicit information to analyze graphs and implicit information [Galvez C, Moya-Anegón F. Approximate personal name-matching through finite state graphs. J. Amer. Soc. Inform. Sci. Technol. 2007;58:1960–1976].

These solutions use the author's names and the metadata of the articles in the analysis.

In the latest researches, it's common to see clustering algorithms in no supervised solutions, through random forest ensemble learning methods, which also include distributed implementations to deal with Big Data problems.

The most interesting approaches for our case are the 3 (unsupervised clustering that groups articles by similarity) and the 5 (pairwise analysis). The first two are not in our scope and the 4 (supervised methods) utilizes external data sets to train the supervised algorithm, feature that we cannot use.

[2] https://meta.wikimedia.org/wiki/WikiAuthors

4. Our problem & case

One of these databases, where there is no author management, is the base of AGRIS, the International System for Agricultural Science and Technology, a network of more than 150 institutions from 65 countries, maintained by the Food and Agricultural Organization (FAO) of the UN.

Figure 1. FAO logotype

AGRIS is the world's leading public information service that provides open access to more than 8 million of publications, research and technical information about agriculture. It is the world's only multilingual bibliographic database for agricultural science. Contents in AGRIS include agriculture, forestry, animal husbandry, aquatic sciences, fisheries, human nutrition and extension.

It is unique for its collection of rich bibliographic records, indexed by AGROVOC, the FAO multilingual agricultural thesaurus. Most of the resources are full text (accessed through Google).

AGRIS was developed in the 1970s to allow its users to have free access to knowledge in agricultural science and technology, and was intended to be an international cooperative system to serve both developed and developing countries.

With the arrival of Internet and the promises offered by open access publishing, there was a growing awareness that the management of agricultural science and technology information would have various facets: standards and interoperability and facilitation of knowledge exchange; tools to enable information management specialists to process data; information and knowledge exchange across countries.

On 5th December 2013 AGRIS 2.0 was released becoming the current described service, who allows a mash-up web application that links the bibliographic AGRIS knowledge to related resources on the web (from external data sources, as the World Bank, DBPedia and Nature) using the Linked Open Data methodology.

Access to the AGRIS Repository is provided through the AGRIS Search Engine. As such, it:

6. Enables retrieval of bibliographic records contained in the AGRIS Repository.

7. Allows users to perform either full-text or fielded, parametric and assisted queries.

AGRIS data were converted to RDF and the resulting linked data set created some 200 million triples.

5. Planning

In order to define the steps of the project, we have initially divided it into different sections and we have estimated the times. By this way, at the end of each section, we will be able to document the current state of the project, the results and the problems.

To define this steps we have to take into account that I am working full time, so I can only devote an average of 14 hours per week to the implementation of this project.

At the beginning, there is the “Problem studio & definition” section, where we will analyze the problem and the state of the art, and propose a possible solution. Next, in the “Data collection” section, we will collect all the necessary data for the problem.

The “Algorithm implementation” section will be divided in different iterations, in order to implement the solution in a progressive way. There will be four iterations:

1. Iteration 1: The first iteration will define the base of the algorithm, and will introduce the data related to the authors. This will clean the data and will try to perform the first disambiguation of authors.
2. Iteration 2: The aim of this iteration is to refine the results of the previous, improving the processes and the results.
3. Iteration 3: This iteration will introduce new data to the algorithm, the subjects. This will clean the new data and will try to improve the previous disambiguation.
4. Iteration 4: The last iteration will refine the procedures and the results of the previous, resulting with a consistent and reliable algorithm.

Id.	Nombre de tarea	Duración
1	Problem studio & definition	11d
2	Data collection	12d
3	**Algorithm implementation**	**86d**
4	Iteration 1	20d
5	Iteration 2	23d
6	Iteration 3	22d
7	Iteration 4	21d
8	**Documentation**	**125d**
9	Initial sections	16d
10	Iteration 1	8d
11	Iteration 2	8d
12	Iteration 3	8d
13	Iteration 4	6d
14	Final sections	23d
15	**Lecture**	**6d**
16	Slides	5d
17	Presentation	1d

Figure 2. Gantt chart

The different sections and iterations are not fixed elements, and could suffer some changes during the course of the project, caused by different problems or for not having enough time to do the defined work.

6. Budget

In order to realize an overall estimation of the budget of this project, we have considered the salary of a Junior Data Scientist. After comparing different values, as an average, a Junior Data Scientist earns 22.000 € per year[3] (more or less 10€/h).

Following the timing defined in the previous section, there will be one person working in this project, 14 hours per week, during 25 weeks.

If we perform the calculations we obtain:

$$25\ weeks \cdot\ 14\ hours/week\ \cdot\ 10\ €/hour\ = 3500€$$

[3] https://www.glassdoor.com/Salaries/junior-data-scientist-salary-SRCH_KO0,21.htm

7. Technology Stack

The implemented solution, from collecting the raw data, until the results delivery, uses a different kind of technologies, tools, protocols and standards to give a solution in each step.

7.1. Virtuoso SPARQL Query Service

OpenLink Virtuoso[4] is the first CROSS PLATFORM Universal Server to implement Web, File, and Database server functionality alongside native XML Storage, and Universal Data Access Middleware, as a single server solution.

Virtuoso provides transparent access to existing data sources, which are typically databases from different database vendors.

The Virtuoso SPARQL Query Service[5] originally implemented the SPARQL Protocol for RDF[6], and has been updated to support SPARQL 1.1[7], providing SPARQL query-processing for RDF data.

7.2. RDF

RDF[8] (Resource Description Framework) is a standard model for data interchange on the Web. RDF has features that facilitate data merging even if the underlying schemas differ, and it specifically supports the evolution of schemas over time without requiring all the data consumers to be changed.

RDF extends the linking structure of the Web to use URIs to name the relationship between things as well as the two ends of the link (this is usually referred to as a “triple”). Using this simple model, it allows structured and semi-structured data to be mixed, exposed, and shared across different applications.

This linking structure forms a directed, labeled graph, where the edges represent the named link between two resources, represented by the graph nodes. This graph view is the easiest possible mental model for RDF and is often used in easy-to-understand visual explanations.

[4] http://docs.openlinksw.com/virtuoso/
[5] http://vos.openlinksw.com/owiki/wiki/VOS/VOSSparqlProtocol
[6] http://www.w3.org/TR/rdf-sparql-protocol/
[7] http://www.w3.org/TR/sparql11-overview/
[8] https://www.w3.org/RDF/

7.3. SPARQL

SPARQL[9] is an RDF query language, that is, a semantic query language for databases, able to retrieve and manipulate data stored in Resource Description Framework (RDF) format.

7.4. Docker

Docker[10] is a software technology that provides an additional layer of abstraction and automation of *Operating-system-level virtualization*[11] on Windows and Linux. Docker uses the resource isolation features of the Linux kernel such as cgroups and kernel namespaces, and a union-capable file system to allow independent "containers" to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines.

7.5. Hadoop

The Apache Hadoop[12] project develops open-source software for reliable, scalable and distributed computing.

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

The project includes these modules:

- Hadoop Common: The common utilities that support the other Hadoop modules.
- Hadoop Distributed File System (HDFS): A distributed file system that provides high-throughput access to application data.
- Hadoop YARN: A framework for job scheduling and cluster resource management.
- Hadoop MapReduce: A YARN-based system for parallel processing of large data sets.

[9] https://www.w3.org/TR/rdf-sparql-protocol/
[10] https://www.docker.com/
[11] Operating-system-level virtualization is a computer virtualization method in which the kernel of an operating system allows the existence of multiple isolated user-space instances, instead of just one. Such instances, which are sometimes called containers, may look like real computers from the point of view of programs running in them.
[12] http://hadoop.apache.org/

7.6. Apache Spark

Spark[13] is an open-source fast and general engine for large-scale data processing and analytics. It has an advanced DAG execution engine that supports acyclic data flow and in-memory computing. Offers over 80 high-level operators that make it easy to build parallel apps and is supported in the Scala, Python and R languages.

Spark powers a stack of libraries including SQL and DataFrames[14], MLlib[15] for machine learning, GraphX[16] , and Spark Streaming[17].

GraphFrames

GraphFrames[18] is a package for Apache Spark which provides DataFrame-based Graphs. It provides high-level APIs in Scala, Java, and Python. It aims to provide both the functionality of GraphX[19] and extended functionality taking advantage of Spark DataFrames. This extended functionality includes motif finding, DataFrame-based serialization, and highly expressive graph queries.

7.7. Python

Python[20] is a widely used high-level programming language for general-purpose programming. An interpreted language, Python has a design philosophy that emphasizes code readability (notably using whitespace indentation to delimit code blocks rather than curly brackets or keywords), and a syntax that allows programmers to express concepts in fewer lines of code than might be used in languages such as C++ or Java. The language provides constructs intended to enable writing clear programs on both a small and large scale.

Python features a dynamic type system and automatic memory management and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles. It has a large and comprehensive standard library.

Python interpreters are available for many operating systems, allowing Python code to run on a wide variety of systems. CPython, the reference implementation of Python, is open source

[13] http://spark.apache.org/
[14] http://spark.apache.org/sql/
[15] http://spark.apache.org/mllib/
[16] http://spark.apache.org/graphx/
[17] http://spark.apache.org/streaming/
[18] https://graphframes.github.io/
[19] GraphX is a new component in Spark for graphs and graph-parallel computation. At a high level, GraphX extends the Spark RDD by introducing a new Graph abstraction: a directed multigraph with properties attached to each vertex and edge http://spark.apache.org/docs/latest/graphx-programming-guide.html
[20] https://www.python.org/

software and has a community-based development model, as do nearly all of its variant implementations. CPython is managed by the non-profit Python Software Foundation.

7.8. IPython

IPython[21] is a command shell that provides a rich architecture for interactive computing with:

- A powerful interactive shell.
- A kernel for Jupyter[22].
- Support for interactive data visualization and use of GUI toolkits.
- Flexible, embeddable interpreters to load into your own projects.
- Easy to use, high-performance tools for parallel computing.

7.9. Miniconda

Miniconda[23] is a small "bootstrap" version that includes only Conda[24], Python, and the packages they depend on. Over 720 scientific packages and their dependencies can be installed individually from the Continuum[25] repository.

7.10. Spyder

Spyder[26] is the Scientific Python Development Environment. Spyder is a powerful interactive development environment for the Python language with advanced editing, interactive testing, debugging and introspection features and a numerical computing environment thanks to the support of IPython (enhanced interactive Python interpreter) and popular Python libraries such as NumPy (linear algebra), SciPy (signal and image processing) or matplotlib (interactive 2D/3D plotting).

Spyder may also be used as a library providing powerful console-related widgets for your PyQt-based applications – for example, it may be used to integrate a debugging console directly in the layout of your graphical user interface.

[21] https://ipython.org/
[22] https://jupyter.org/
[23] https://conda.io/docs/
[24] Conda is an open source package management system and environment management system for installing multiple versions of software packages and their dependencies and switching easily between them. It works on Linux, OS X and Windows, and was created for Python programs but can package and distribute any software.
[25] https://www.continuum.io/
[26] https://pythonhosted.org/spyder/

8. Development environment

The virtuoso server, where we have the base data, was deployed in a GRIHO's (UdL) machine. This machine does not need many resources because the data collection process is light and only is performed one time per data kind or query.

On the other hand, the algorithm was initially developed and tested in a simple, but powerful, laptop, with the following features:

- MacBook Pro
- O.S.: macOS Sierra 10.12.4
- Processor: 4 Cores, 2.5 GHz Intel Core i7
- Memory: 16 GB 1600 MHz DDR3
- SSD disk

Through the Conda software, the Spyder IDE (for the python code development) together with the python environment and the Pyspark and GraphFrames libraries were installed in the laptop for the algorithm implementation.

Spark distributes the jobs between different workers and, in a worker level, it's also able to parallelize the jobs between the machine cores. This is why we can use Spark on a single machine, taking into account that is not the same than a cluster because, in terms of performance in the shuffle operations (which involve disk I/O, data serialization, and network I/O among the executors), the steps and communication between cores on a single machine are not the same (are really faster) than between the machines in a cluster.

As we advanced with the solution implementation, we identified the requirement of more computational resources since, despite was not a Big Data problem, the machine which we were working was not a data processing engine and the amount of data and the operations began to be considerable.

To tackle this problem we analyzed different possible solutions and, the most suitable option was to deploy the ecosystem (Python, Spark…) in a processing machine in the cluster of the company where I work, Eurecat[27].

The machine resources consist on:

- O.S.: Ubuntu 16.04.2 LTS

[27] https://eurecat.org

- Processor: 16 Cores, 2.3 GHz Intel Xenon E5
- Memory: 48 Gb
- SSD Disk

With the aim of not install (in the new machine) all the tools and libraries needed in this project, since these only would be used in it, we began to work with the Docker technology. By this means, we were able to isolate from the host machine all the components, and build the computational architecture in a separated and controlled context.

Docker also enables the possibility to share data between the host machine and the executed container, through the Docker volumes[28]. This helped us to reuse the downloaded data resulting from the SPARQL queries.

A Dockerfile[29] was defined with all the required components:

- The centos:6.6 as image base[30].
- Basic centos components
- Miniconda for Python and other libraries
- Java 1.8
- Hadoop 2.7.1
- Spark 2.2.0

In order to practice with distributed tools, Hadoop was added to the Docker image, getting in this way the distributed file system HDFS and other software like YARN[31], etc. Hadoop provides a web interface for the Resource Manager and other for the HDFS health.

Spark was installed in standalone mode[32] and, unlike only working with Spark as a library, this mode also offers a web-based user interface to monitor the cluster and check the jobs evolution and status.

We only had one (big) machine so we couldn't build a cluster of nodes but, in order to test the previous (distributed) tools in a fully-distributed scenario, through different Docker images and

[28] Volumes are the preferred mechanism for persisting data generated by and used by Docker containers.
[29] Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.
[30] https://hub.docker.com/_/centos/
[31] Yet Another Resource Negotiator: A framework for job scheduling and cluster resource management.
[32] https://spark.apache.org/docs/latest/spark-standalone.html

the Docker Compose[33] tool, we could start up a virtualized cluster of four nodes (1 master and 3 workers), testing different configurations for the tools and getting some experience with them.

Obviously this configuration had not the best efficiency, because all the virtualized nodes were on the same machine and were sharing the same resources since, in terms of performance, is better to give all the resources to one worker, in a single host, and this worker will manage all the processes distribution and the resources scheduling.

As I said, this was only for testing. The final configuration consisted of a single machine (container), which had one Hadoop node (working as a Namenode and Datanode, and a data replication factor fixed to 1) and two Spark processes, one for the master and other for one worker.

The last problem, as far as the development environment, was to deal with this remote machine. This means that we wanted to keep working and codifying the algorithm in our laptop, for comfort, but launching and testing it in the new processing machine. This was solved through the Ipython kernels[34] and the Spyder IDE.

To do so, is necessary to start an Ipython kernel on the remote machine. When Ipython starts a kernel, it passes to the kernel a connection file, so this specifies how to set up communications with the front end. The next step is to configure Spyder to work with this kernel. You have to configure a user on the remote machine, and add your public ssh key for the remote access. Then specify all of these communications parameters to the IDE, together with the connection file and, finally, you will be able to run your local code on the remote machine.

8.1. Cluster start up

Through the Dockerfile file, we define the required components of the system, and also configure different parameters:

- New environment variables.
- New filesystem folders.
- Specify which configuration files have to be copied into the image, needed by the installed tools.
- Define the exposed ports.

[33] Compose is a tool for defining and running multi-container Docker applications. https://docs.docker.com/compose/

[34] A 'kernel' is a program that runs and introspects the user's code. IPython includes a kernel for Python code.

- Specify the entry point file.

In the cluster test case, where four containers were deployed, there were two different Dockerfiles, one for the master and other for the workers.

Distributed tools have a default configuration, only for development and tests purposes. The tools configuration is managed through different configuration files:

- Hadoop:
  - hadoop-env.sh: Environment variables that are used in the scripts to run Hadoop.
  - core-site.xml: Configuration settings for Hadoop Core such as I/O settings that are common to HDFS and MapReduce.
  - hdfs-site.xml: Configuration settings for HDFS daemons, the Namenode, the secondary Namenode and data nodes.
  - mapred-site.xml: Configuration settings for MapReduce daemons: the job-tracker and the task-trackers.
  - masters: A list of machines that each run a secondary node.
  - slaves: A list of machines that each run a Datanode and a Task-Tracker.
- Spark:
  - spark-env.sh: Certain Spark settings can be configured through environment variables. In Standalone and Mesos modes, this file can give machine specific information such as hostnames. It is also sourced when running local Spark applications or submission scripts.
  - spark-defaults.conf: key-value file with the Spark configuration variables.

The entry point file is executed every time when the image is deployed in a new container, and helps to initialize the different tools. The cluster tests case also had two different entry point files.

Entry point steps:

1) Initialize the ssh daemon and set up passphrase-less keys for nodes communication.
2) Format the HDFS file system:
   `$HADOOP_HOME/bin/hdfs namenode -format ProcessingCluster`
   $HADOOP_HOME is an environment variable, defined in the Dockerfile, which stores the Hadoop /bin path.
3) Start the HDFS file system:
   `$HADOOP_HOME/sbin/start-dfs.sh`
4) Start YARN:
   `$HADOOP_HOME/sbin/start-yarn.sh`
5) Start Spark master:
   `$SPARK_HOME/sbin/start-master.sh`
   $SPARK_HOME is an environment variable, defined in the Dockerfile, which storage the Spark /bin path.
6) Start Spark workers (in each node worker):

```
$SPARK_HOME/sbin/start-slave.sh spark://master:7077
```

7) Initialize HDFS folders:
```
hadoop fs -mkdir -p /Projects/AuthorsDisambiguation/Data/
```
8) Update necessary data to HDFS:
```
hadoop fs -put /home/AuthorsDisambiguation/Data/*.csv
/Projects/AuthorsDisambiguation/Data/
```

Once the DockerFile and entry point files are defined, to build the image is necessary to execute the following Docker command:

```
docker build -t spark_cluster .
```

Where the –t option sets the image name, and the final point indicates the Dockerfile path (current path).

Then, in order to execute the image in a new container, we have to execute the Docker command:

```
docker run -itd --name spark_cluster -v
/home/bigcat/Projects/Others/AuthorsDisambiguation:/home/AuthorsDisambiguation
-h master -p 10000:8088 -p 10001:50070 -p 10002:8080 -p 10003:7077 -p 10010-
10020:4040-4050 spark_cluster
```

Where the following options mean:

- -i: keep STDIN open if not attached.
- -t: allocate a pseudo-TTY
- -d: run container in background and print container ID.
- --name: container name.
- -v: Bind mount a volume.
- -h: container host name.
- -p: publish a container port to the host.
- spark_cluster: image.

In the cluster test case, as I have explained few lines above, we used the Docker Compose tool, which uses a specific file (docker-compose.yml), where all the parameters of each container are defined to deploy the four containers.

If everything goes right (if not, you should check the logs), you must be able to interact with Hadoop and Spark through the UIs. Next, screenshots of the Fully-distributed cluster UIs:

- Hadoop Resource Manager:

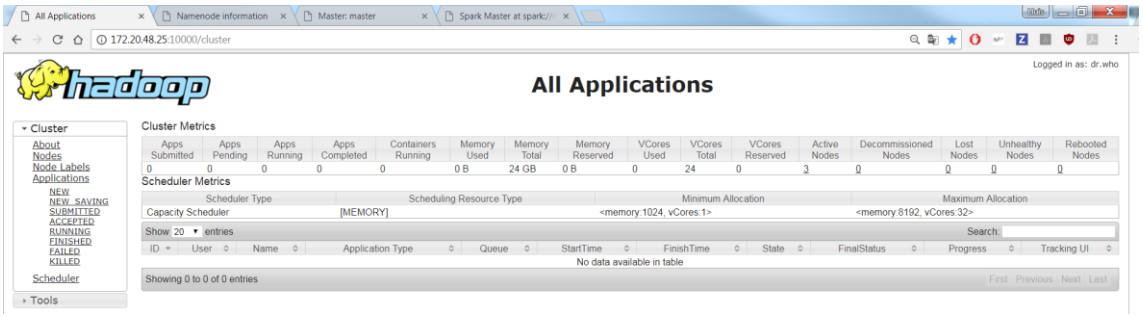

Figure 3. Hadoop Resource Manager (cluster environment)

- Hadoop DFS health:

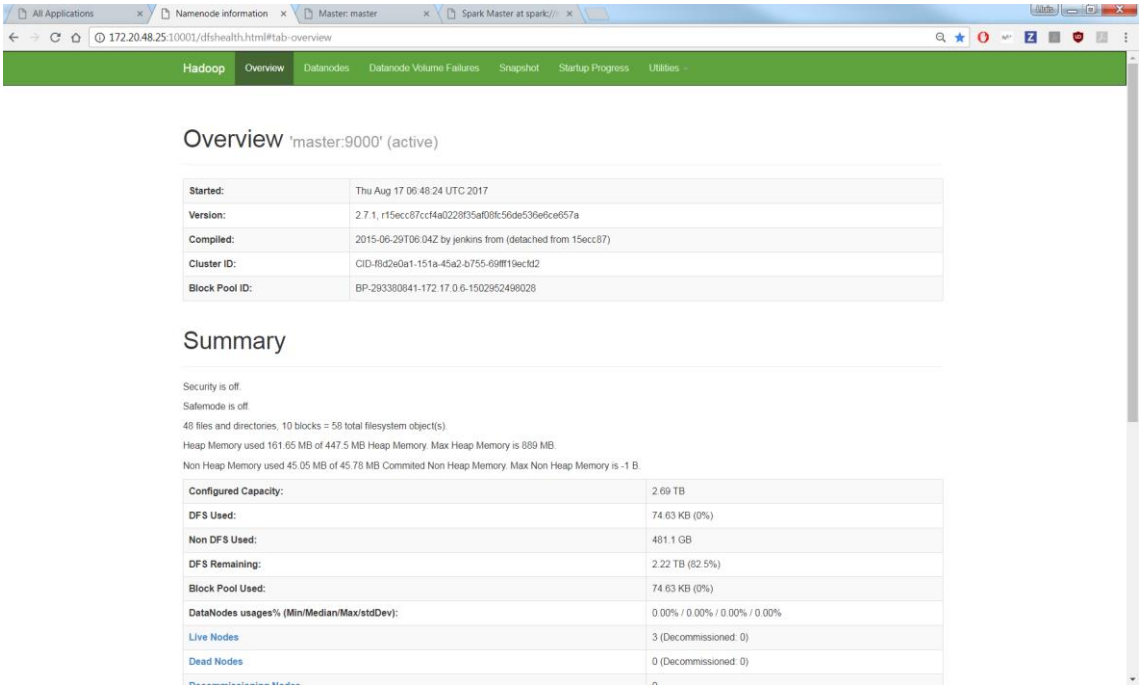

Figure 4. Hadoop DFS health (cluster environment)

- Spark master UI:

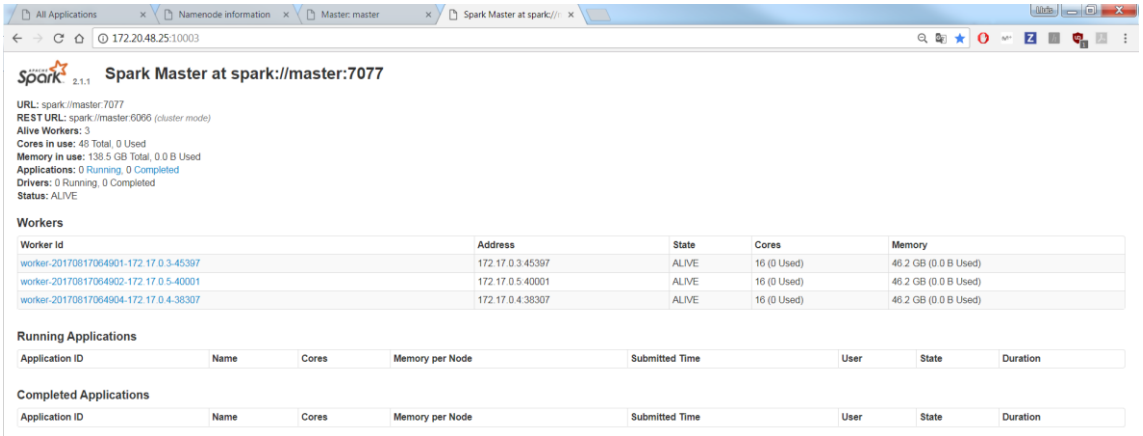

Figure 5. Spark master UI (cluster environment)

We can see, in the three screenshots, the 3 worker nodes. In the Hadoop case each of these nodes has a Datanode daemon and in the Spark case, each node has a Spark worker process.

The current environment UIs, where one node has all the processes:

- Hadoop Resource Manager:

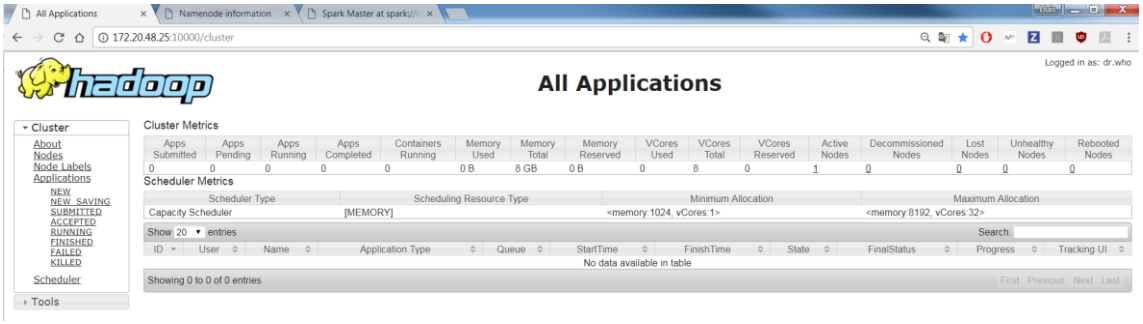

Figure 6. Hadoop Resource Manager (current environment)

- Hadoop DFS health:

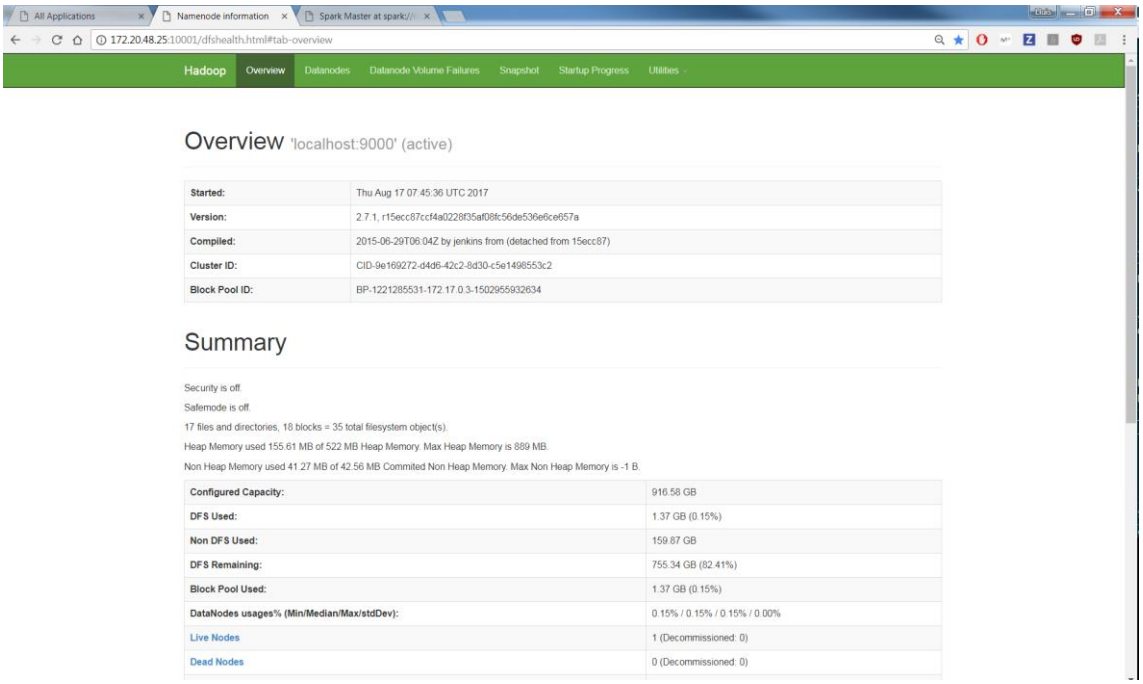

Figure 7. Hadoop DFS health (current environment)

- Spark master UI:

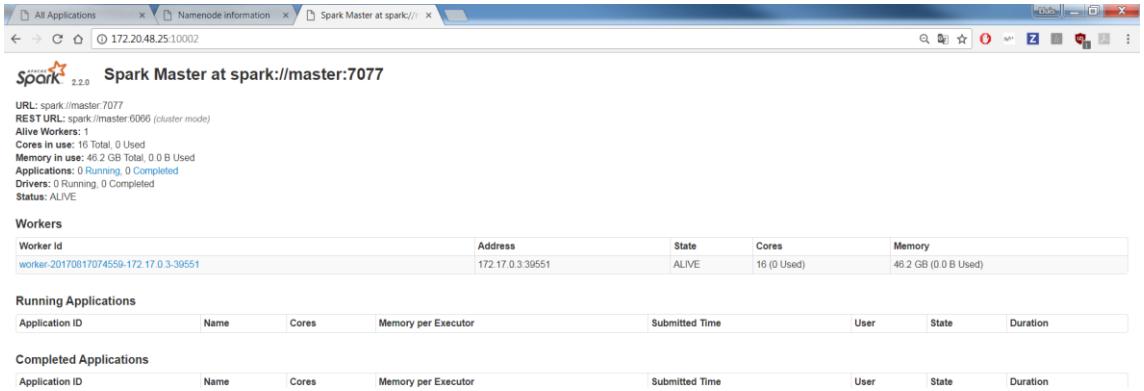

Figure 8. Spark master UI (current environment)

Finally, to interact with these containers, the following Docker command gives us a shell[35]:

```
docker exec –it spark_cluster /bin/bash
```

[35] A Unix shell is a command-line interpreter that provides a traditional Unix-like command line user interface.

9. Data collection

As described in previous points, the solution is based on the AGRIS's data, stored in the AGRIS's repositories and accessible through a SPARQL endpoint. Following this endpoint we are able to access all the required data, but having to work with some technical limitations, like the query processing speed and the query results dimension, which are limited.

To avoid these limitations we asked the AGRIS's repositories managers the possibility to obtain all the base dataset, in order to deploy it in a local database, managed by us, and without limitations. Finally, after sharing some information, they sent us the dataset, in RDF format, and we deploy it in a local Virtuoso server.

Other public datasets had been uploaded into the Virtuoso, like AGROVOC, which contains a controlled vocabulary covering different areas of interest, also published by FAO, and managed by a community of experts; including food, nutrition, agriculture, fisheries, forestry, environment, etc.

Once all the data was in the Virtuoso, we were able to generate the necessary queries, with the SPARQL protocol (an RDF query language), through the Virtuoso SPARQL Query Service.

Despite having avoided the main limitations, the queries results were too large so, to minimize the big volume of data resulting in each query, we limited the returning records with the SPARQL commands 'LIMIT 1000000' and 'OFFSET X', thereby the query time was most acceptable and more files with the data were obtained.

- Query 1. To obtain the issued date, submitted date, conference title, journal title and journal ISSN of each article:

```
SELECT ?articleId, ?issued, ?submitted, ?conferenceTitle, ?journalTitle, ?journalIssn WHERE {
        ?article a bibo:Article ;
        dct:identifier ?articleId;
        dct:type "Journal Article".
        OPTIONAL {?article dct:issued ?issued}.
        OPTIONAL {?article dct:dateSubmitted ?submitted}.
        OPTIONAL {?article bibo:presentedAt ?conference.
                ?conference dct:title ?conferenceTitle}.
        OPTIONAL {?article dct:isPartOf ?journal.
                ?journal dct:title ?journalTitle.
                ?journal bibo:ISSN ?journalIssn}.
} LIMIT 1000000 OFFSET 0
```

- Query 2. To obtain the authors names of each article:

```
SELECT ?articleId ,group_concat(?authorName;separator='||') as ?authorNames WHERE {
```

```
        ?article a bibo:Article ;
        dct:identifier ?articleId;
        dct:creator ?author;
        dct:type "Journal Article".
        ?author foaf:name ?authorName .
} LIMIT 1000000 OFFSET 0
```

- Query 3. To obtain the subjects of each article:

```
SELECT ?articleId, group_concat(?subject;separator='||') as ?subjects WHERE {
        ?article a bibo:Article ;
        dct:identifier ?articleId;
        dct:type "Journal Article";
        dc:subject ?subject.
} LIMIT 1000000 OFFSET 0
```

- Query 4. To obtain the language of each article:

```
SELECT ?articleId, group_concat(?language;separator='|') as ?language WHERE {
        ?article a bibo:Article ;
        dct:identifier ?articleId;
        dct:type "Journal Article".
        OPTIONAL {?article dct:language ?language}.
} LIMIT 1000000 OFFSET 0
```

- Query 5. To obtain the subjects hierarchy.

```
SELECT ?s ?o WHERE {
        ?s a skos:Concept .
        ?o a skos:Concept .
        ?s ?p ?o .
        FILTER(?p = skos:narrower)
}
SELECT ?s ?o WHERE {
        ?s a skos:Concept .
        ?o a skos:Concept .
        ?s ?p ?o .
        FILTER(?p = skos:broader)
}
```

- Query 6. To obtain the subjects of the jounals.

```
SELECT ?articleId, ?journal, group_concat(?s;separator='||') as ?journalSubjects WHERE {
        ?article a bibo:Article ;
        dct:identifier ?articleId;
        dct:type "Journal Article".
        OPTIONAL {?article dct:isPartOf ?journal.
                ?journal dct:subject ?s}.
}
```

Through the SPARQL command ‘group_concat’ we avoided extra lines in the results.

Finally, the queried data are downloaded into 19 CSV files, with a total of 2.2 GB, prepared for the data cleaning steps.

10. The approach

In order to give one possible solution to the previous problem, we have designed a specific algorithm. The algorithm is composed of different steps or stages, where each one of these uses a different kind of data to improve the classification obtained by the previous step, being the first the most important one.

This data is related to the publications and, in order to use them in the algorithm, we have to be able to establish a way to compare this data throughout the publications.

10.1. First Stage

10.1.1. Theory

The first is the most important because performs the initial disambiguation of authors and uses the main data of the problem, the authors themselves and her publications.

The procedure of this step takes as starting point the fact that the same author usually submits the publications with a common group of co-authors. This is not always true, but in most of the cases this is the rule, these are the exceptions:

- When two different authors, with the same name, have in their co-authors network one author also with the same name. This will join the two networks but is difficult to happen.
- When the same author publishes with different groups of co-authors, without any relation between the co-authors. This will separate the same author in various co-authors networks.
- Publications with only one author make impossible the classification of this author via this method.

In order to fix these exceptions, and to improve this first disambiguation, it's possible to add in this first step certain methods to avoid these inconsistencies but, the best way to solve it is to include new data, related to the authors, in the algorithm. This is the aim of the next steps.

Then, in summary, this stage attempts to search for relations or groups of co-authors between the authors' publications.

10.1.2. Practical cases

Case 1

There is two publication with the following authors:

- P1 -> Authors: X, A, B, C, D.
- P2 -> Authors: X, D, E, F.

Throw this first stage, the author X has this network of co-authors:

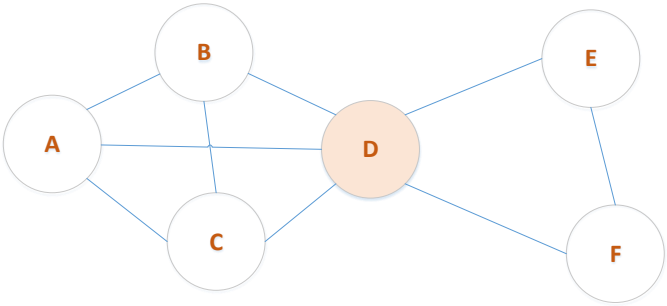

Figure 9. Network of co-authors of author X (case 1)

As we can see, these publications match in two authors, X and D. This shows that both authors could be the same person in each publication.

Case 2

There is two publication with the following authors:

- P3 -> Authors: X, G, H, I, J.
- P4 -> Authors: X, I, J, K, L.

The author X has this co-authors network:

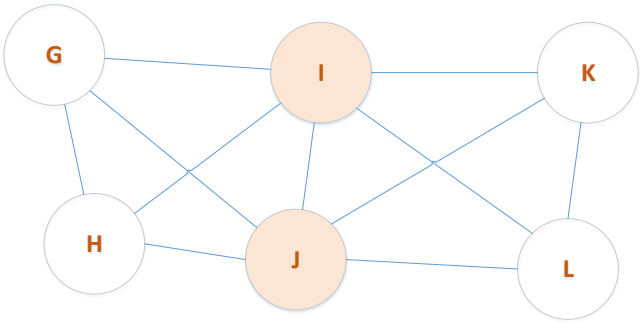

Figure 10. Network of co-authors of author X (case 2)

This relation is most powerful than the previous, where there are 3 matches, X, C and D.

One interesting point in this stage will be the names processing because, as I introduced in the firsts points of the document, the authors can publish with different name combinations or abbreviations.

To deal with this problem some techniques will be applied. These techniques consist in string processing where, for instance, only the name's letters will be considered (no points, commas or special characters).

Once this stage is finished, we will have some structures like the following, for the author X (in the previous examples):

Co-author	Publications	Disambiguation group
A	[P1]	1
B	[P1]	1
C	[P1]	1
D	[P1,P2]	1
E	[P2]	1
F	[P2]	1
G	[P3]	2
H	[P3]	2
I	[P3,P4]	2
J	[P3,P4]	2
K	[P4]	2
L	[P4]	2

In summary, from 4 publications where the person X is an author, the results of the algorithm are separated this author in other two:

- X_1 from the publications P1 and P2, with a low linkage.
- X_2 from the publications P3 and P4, with a higher linkage.

10.2. Second Stage

10.2.1. Theory

In the second stage, we have introduced the publications subjects, because we think these are an identifying feature specific from each author and, after the authors' classification done in the first stage, this could be the best option to improve the author's disambiguation.

We start from the fact that if an author publishes a document that has associated some subjects, this author works in this subjects' scope.

Here, the problem is how to compare two subjects in order to establish if are similar or not. In the previous stage this was easier, because we wanted to search the same name, expressed in different ways, and we solved this through string matching operations. Now we work with subject's IDs, which allow us to ensure that subject A is not the subject B. Instead, we have to implement a method to say how similar two subjects are.

To solve this point, we will use a good feature of the AGRIS repository. The subjects stored in it have a hierarchy structure having, for each one, the broader and narrower subjects. This will enable us to build a tree graph where compute the distances between subjects.

10.2.2. Practical case

If we have the following hierarchy tree of subjects:

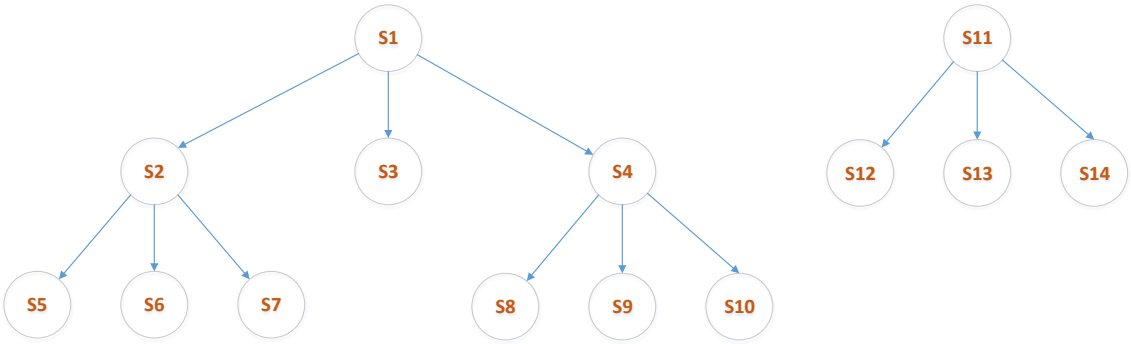

Figure 11. Hierarchy tree

We can say:

- The subject S5 is more specific than S1.
- The subject S6 is at distance 3 from S3.
- The subject S7 has no relation with S12.

Also, we can compute distances between groups of subjects (short distance):

- The group of subjects [S5, S3] is at distance 3 from S8.
- The group of subjects [S6, S12] is at distance 1 from the group [S11, S13].

Following the examples introduced in stage one, and the previous hierarchy of subjects, imagine:

- P1 -> S5, S2, S12.
- P2 -> S11, S14, …

We can say that the union performed in stage one, through the author D, has more possibilities to be real because of the distance between the groups of subjects, of each publication, is 1. If this distance would have been greater, this would show the two publications have not subjects in common, so the author D could be different in each publication.

We will use this stage to validate the previous disambiguation, checking if the disambiguated authors work in similar subjects, obtained from different publications.

10.3. Next Stages

Not implemented in this project, the next stages can use other data like:

- Issued date: comparing the years, for example.
- Conference: checking if an author, or group of authors, have published in the same conference.
- Journal: checking if an author, or group of authors, have published in the same journal.
- Language.
- Others.

Always trying to improve the results of the previous disambiguation.

11. The Solution

As already seen in the planning point, the solution is implemented iteratively, trying to solve the problem step by step. In each iteration, the necessary data is collected and cleaned, and finally processed, in order to get the information in which contributes.

11.1. Initialization

There are some steps to do before start programming with python and spark.

11.1.1. Spark Session

In Spark 2.0 (we are working with Spark 2.2), we had a new entry point for DataSet and Dataframe APIs called as Spark Session.

SparkSession is essentially a combination of the old SQLContext, HiveContext and StreamingContext. All the API's available on those contexts are available on Spark Session also. Spark Session internally has a Spark Context for actual computation.

So, we have to start defining the SparkSession:

```
spark = SparkSession.builder
        .appName('Authors_Disambiguation')
        .master('spark://master:7077')
        .getOrCreate()
```

'Authors_Disambiguation' is the application name, and the string 'spark://master:7077' is the URL of the spark master.

11.1.2. File system paths

We have defined two paths:

1) basePath: Path where results, images and graphics will be stored.

```
basePath = "/home/AuthorsDisambiguation/"
```

2) hdfsbasePath: HDFS path where queries data are stored.

```
hdfsbasePath = "hdfs:///Projects/AuthorsDisambiguation/Data/"
```

11.2. First Iteration

The first iteration works in the first stage, defined in the previous chapter, where we have to deal with the authors of the publication to give a first name disambiguation.

This basically consists of two points, load and clean the data of the authors, and define the algorithm flow to obtain a graph structure with the network of coauthors. As I have explained in

the 'Technology Stack' chapter, we will use the GraphFrames library to create and work with these graphs.

In Pyspark we can't apply a 'map' operation in a DataFrame, the 'map' is only supported in the RDDs API. So, since we have to perform many operations over strings in these initial steps, we have started using the RDDs. Despite this, the step from the RDD to DataFrame, or vice versa, is immediate.

While we are operating over the all the publications data, we can consider it as a 'Big Data' problem but, when we work at the author level, the data volume is considerably reduced. Despite this, we continue using Spark since it is the goal of the project, taking into account the overheads generated by these distributed frameworks, which are despicable working with Big Data sets.

11.2.1. The data & cleaning

The author's data, in the CSV files, has the following structure:

"US201300966508","Tarvainen, O.|Markkola, A.M.|Ahonen-Jonnarth, U.|Jumpponen, A.|Strommer, R."

In each row there are two strings, separated by a comma. The first string is the publication ID, and the second contains the authors' names, separated by the pipe character ('|').

The data is loaded into a DataFrame (as a collection of Row objects), from the HDFS file system, specifying the format (CSV) and the path. In order to load all the files at the same time, we use the character '*' in the path expression.

```
authors_raw_DF = spark.read.format("csv")
        .load(hdfsbasePath+"Authors-*.csv",header = True)
```

If we had worked in a real distributed cluster, the data of the partitions of the DataFrame would match with the data of each partition in the HDFS file system.

The entry point of the algorithm consists in an RDD where each element has two items, the publication ID and a list with the authors' names.

So, in order to prepare the data for the algorithm, we have to apply some 'map' operations in the initial DataFrame. The first operation wants to obtain, for each row, all the values of the Row object and, the second operation, generates the requested element structure, putting in the first position the publication Id and, in the second, the authors' names list, obtained through a 'split' operation over the string of names.

```
authors = authors_raw.rdd
    .map(lambda r:list(r.asDict().values()))
    .map(lambda r: (r[0]], r[1].split('|')))
```

The other entry point of the algorithm is the name of the author which we want to disambiguate. As an example in this document I want to disambiguate the author named 'Yamaoka. Y.', from now called target.

```
target = 'Yamaoka, Y.'
```

11.2.2. The algorithm

Once we have the cleaned data, we are able to start with the algorithm implementation. The first thing that we need are all the publications which have our target as an author. To do so, we will go over all the publications, applying a filter function finding the target among the authors. Hereunder we drop the target from the filtered results, obtaining a new RDD with the publications and the co-authors.

```
def filterAuthor(elem):
    if target in elem[1]:
        return True
def mapExtractAuthor(elem):
    elem[1].remove(target)
    return elem
authorsByArticle = authors
    .filter(filterAuthor)
    .map(mapExtractAuthor)
```

At this point, we have introduced a validation point for the cases where no publications are found for the target name, finalizing the execution of the algorithm.

```
if(coauthorsByArticle.count() == 0):
    print("No possible publications for the input author: ", target)
    spark.stop()
    sys.exit()
```

The following operation is one of the keys of this stage, it consist of building tuples between each pair of authors in each publication, having the results in a new RDD through a *flatMap* operation. In summary, if we have the following *authorsByArticle* RDD:

```
[
        ('pub_ID_1',[author_name_1, author_name_2, author_name_3, author_name_4]),
        ('pub_ID_2',[author_name_1, author_name_2, author_name_6, author_name_7]),
        ('pub_ID_3',[author_name_6, author_name_7, author_name_3, author_name_4])
]
```

Then, the new RDD will be:

```
[
        ((author_name_1, author_name_2), ['pub_ID_1']),
        ((author_name_1, author_name_3), ['pub_ID_1']),
        ((author_name_1, author_name_4), ['pub_ID_1']),
        ((author_name_2, author_name_3), ['pub_ID_1']),
        ((author_name_2, author_name_4), ['pub_ID_1']),
        …
        ((author_name_1, author_name_2), ['pub_ID_2']),
        ((author_name_1, author_name_6), ['pub_ID_2']),
        ((author_name_1, author_name_7), ['pub_ID_2']),
        …
        ((author_name_6, author_name_7), ['pub_ID_3']),
        ((author_name_6, author_name_3), ['pub_ID_3']),
        ((author_name_6, author_name_4), ['pub_ID_3']),
        …
]
```

The code uses the *combinations* method, from the itertools[36] library, to get all the 2 length subsequences (pairs) of elements from the iterable input.

```
coauthorsByArticlePairs = coauthorsByArticle.flatMap(
        lambda elem:
                [
                        (tuple(sorted(coauth)),[elem[0]])
                                for coauth in itertools.combinations(elem[1], 2)
                ]
        )
```

Apart of the pairs calculation, the above process prepare the data for the following *reduceByKey* operation, creating a key-value structure for each value. In this operation we will obtain, for each pair of authors, the publications where both have worked with our target. As we can see, the key is the sorted authors pair and the value is the publication where they have worked together. In the reduce operation we will group the publications in a list and, with another *map* operation, we obtain a new RDD with a special structure. Let's see how it works.

```
coauthorsByArticles = coauthorsByArticlePairs
        .reduceByKey(lambda x,y: x + y)
        .map(lambda x: (x[0][0],x[0][1],x[1]))
```

On other side, we want to have one RDD with all the co-authors and the publications where they have worked with our target. Again we will use *reduceByKey* function, like the previous, but first we have to prepare the data.

[36] Python module that implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python.
https://docs.python.org/3.6/library/itertools.html

```
def yieldFunc(elem):
    for item in elem[1]:
        yield (item,[elem[0]])
coauthors = coauthorsByArticle.flatMap(yieldFunc).reduceByKey(lambda x,y: x + y)
```

In this case we use the yield[37] expression for the data preparation step, which defines a generator function. The reduce function takes the authors' names as a key, and the values are the publications.

We have done all these previous steps in order to build a graph with the first network of co-authors. The *coauthors* RDD has the nodes, and the *coauthorsByArticles* RDD has the edges, plus other relevant information. So the last step in this iteration consists of building this graph through the GraphFrames library.

```
v = spark.createDataFrame(coauthors,['id','articles'])
e = spark.createDataFrame(coauthorsPairsByArticles,['src','dst','articles'])
g = GraphFrame(v, e)
```

11.2.3. Results

Once finished the first iteration, we have defined the flow of the first stage, starting with the cleaning of the author's data and continuing with the basic operations to build the network of co-authors (graph).

Analyzing the resulting network for the target author, we have 81 vertexes, corresponding to 81 co-authors among 31 publications where the target has appeared.

Through the library toyplot[38] we are able to print these graphs and have an idea of how these networks are composed.

[37] https://docs.python.org/3/reference/expressions.html#yieldexpr
[38] http://toyplot.readthedocs.io/en/stable/index.html

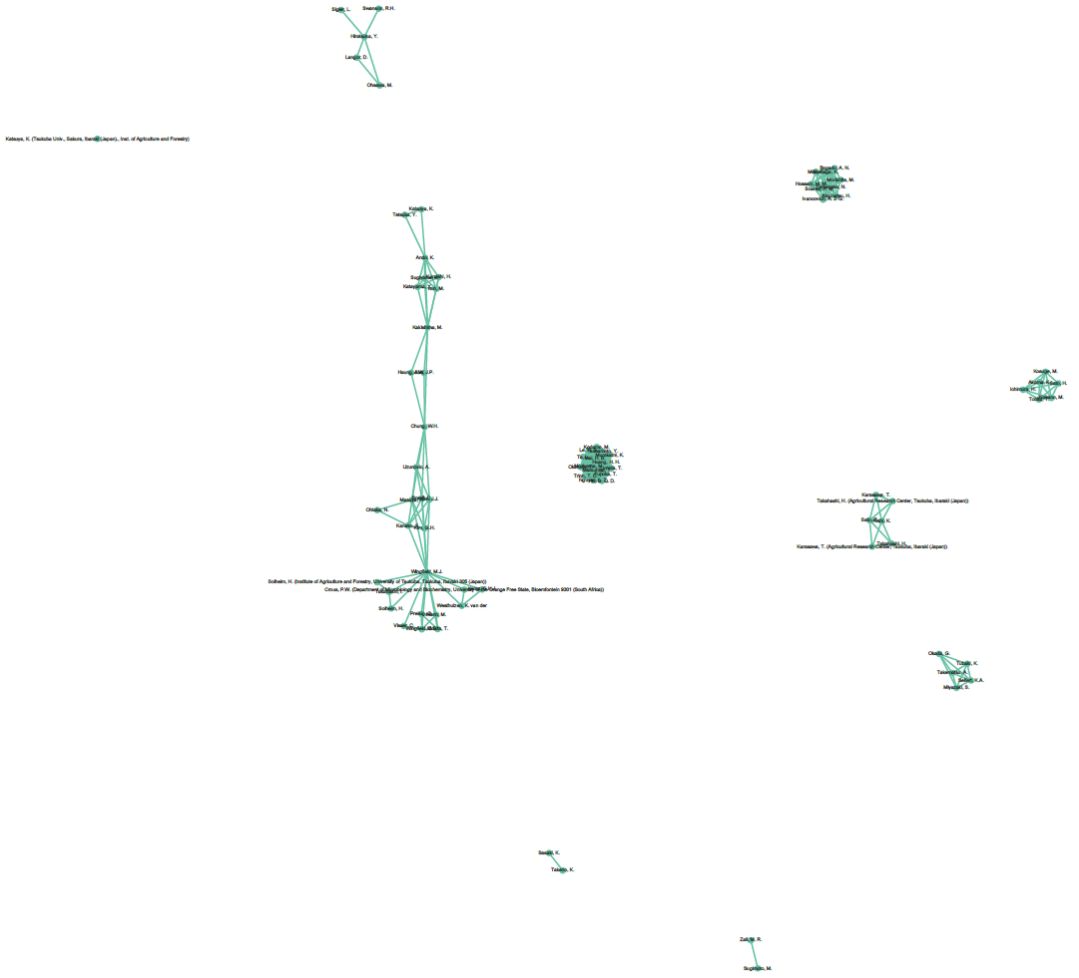

Figure 12. Results of the disambiguation process (first iteration)

The previous screen shows the different subgraphs, the first results of the disambiguation process. We can see 10 subgraphs, one of them with only one node, two with two nodes, and the rest, are more complex. This means that our first result says that there are 10 different target authors.

Analyzing, for example, the bigger one, the lowest part is better, because there are more connections between the nodes, supporting the fact that the target author from these publications is the same. The medium and upper parts have weaker connections, being the possibility that the authors which connect them could be different, affecting the outcome of the disambiguation.

11.3. Second iteration

The second iteration works on the same stage, the first. Here the goal is to improve the strings processing, both on the target filtering side and the co-authors matching side because, as is explained above, one name can be expressed in different ways (first the name followed the surname, first the surname or even using abbreviations). Also add some new features of the

GraphFrames library to the algorithm and finally create new RDDs, since, for example, we have realized aware that we don't have controlled the publications where the target is the unique author.

11.3.1. Data & cleaning

We won't add new data to the algorithm, we will keep working with the authors' data introduced in the first iteration.

Analyzing the data we have detected some cases where the names are followed by some information (locations, universities, etc), between parenthesis. Here one example of different names:

```
'Shant, P.S.', 'Hunek, T. (Polska Akademia Nauk, Warszawa (Poland). Inst. Rozwoju Wsi i Rolnictwa)', 'Belsham, G.J. (AFRC Institute for Animal Health, Pirbright, Woking, Surrey GU24 0NF (United Kingdom))', 'Hambelton, W.W.', 'Franco, Christopher M. M.', 'Labeda, David P.', 'Lassauce, Aurore', 'Paillet, Yoan', 'Jactel, Hervé', 'Bouget, Christophe', 'Borisenko, P.T.', 'Vasilenko, E.D.', 'Biriulina, K.A.', 'Janzen, H.G.'
```

In the algorithm, we will separate and store this extra data for a future processing, because it can be helpful in the disambiguation process.

The following function will help us in this task:

```
def separateNameInfo(string):
        pos = string.find('(')
        if pos == -1:
            return (string,'')
        last = string.rfind(')') if string.rfind(')') > 0 else len(string)
        return (string.replace(string[pos:last+1],'').strip(),
               string[pos+1:last].strip())
```

11.3.2. The algorithm

First we improve the target search among the publications, keeping only the letters and building different patterns (regular expressions) with the name of the target. We can't apply algorithms which compute sets similarities (cosine similarity[39], Jaccard index[40], etc), because two names can be similar but different ('Karl Karltron' is very similar to 'Karl Karltre', but are different people).

For instance, for the target (`'Yoshio Yamaoka'`), we will obtain two patterns, or regular expressions:

[39] https://en.wikipedia.org/wiki/Cosine_similarity
[40] https://en.wikipedia.org/wiki/Jaccard_index

`'.*Yamaoka.*Y.*'`: match all the names that have the following pattern: starts with any character sequence + the word 'Yamaoka' + any character sequence + the letter 'Y' + any character sequence.

`'.*Y.*Yamaoka.*'`: match all the names that have the following pattern: starts with any character sequence + the letter 'Y' + any character sequence + the word 'Yamaoka' + any character sequence.

To introduce these patterns in the process, we have to redefine the *filterAuthor* and *mapExtractAuthor* functions.

```
def filterAuthor(elem):
    for pattern, author in itertools.product(patternToSearch,elem[1]):
        if re.search(pattern, separateNameInfo(author)[0]):
            return True
    return False
def mapExtractAuthor(elem):
    for pattern, author in itertools.product(patternToSearch,elem[1]):
        if re.search(pattern, separateNameInfo(author)[0]):
            elem[1].remove(author)
            return elem
```

In the above functions we have used the *product* method, from the intertools library, to get the Cartesian product between the inputs; the *re*[41] python module, to work with regex expressions and, also, we have used the *separateNameInfo* method previously defined.

Second, we work with the coauthors' names, because can happens the same as the target's name, where one name can be expressed in different ways. Here there are two possible approaches:

- The implemented in the first iteration, where only the same exact names are grouped in the *reduceByKey* operation, when we compute the *coauthors* RDD. For instance, if in the network of co-authors there are these two names 'Albert One Rey' and 'Albert O.R.', this approach won't group them. Maybe the second author is not the first.
- The second approach will join these names using regular expressions operations.

Personally, after a long time thinking on it, even implementing the both approaches, I think the first is better because following it you keep separated the same author (written in different ways) in two different co-authors and you can join it in a future stages of the algorithm using new information, like the extra information stored in this stage, or the subjects, etc. Otherwise, the second approach can't ensure that 'Albert O.R.' is 'Albert One Rey' or 'Albert Onth Roy',

[41] This module provides regular expression matching operations similar to those found in Perl. https://docs.python.org/3.6/library/re.html

joining this 3 names into one and resulting in a mistake, even though this coincidence is difficult to happen.

The both approaches are implemented, but only the first remains in the algorithm. The second has interesting operations and I have opted to introduce some Spark theory with it.

First co-authors matching approach

As I said, this approach is implemented in the first iteration, we will only apply an operation to remove the non-letters characters, and keep applying the *reduceByKey* operation.

```
def parseCoAuthors(elem):
        return (elem[0], [" ".join(sorted(re.findall("[a-zA-Z()]+", author)))
                for author in elem[1]])
coauthorsByArticle = coauthorsByArticle.map(parseCoAuthors)
coauthors = coauthorsByArticle
       .flatMap(yieldFunc)
       .reduceByKey(lambda x,y: x + y)
```

Second co-authors matching approach

We will use the *cartesian* method of the RDD API to obtain a new RDD with all the possible pairs but, before, we will store in each element his position in the collection (through the method *zipWithIndex* from the RDD API), in order to create one structure to obtain only the all possible combinations (not the pairs). For example, if we have the [A, B, C] RDD, the Cartesian product is [[A, A],[A, B],[A, C],[B, A],[B, C],[C, A],[C, B],[C, C]], and we want to obtain [[A, B],[A, C],[B, C]].

```
coauthors = coauthorsByArticle.flatMap(yieldFunc)
coauthorsWithIndex = coauthors.zipWithIndex()
coauthorsWithIndex.persist()
def authorSimilarity(elem):
    words = re.findall("[a-zA-Z]+", elem[0][0][0])
    patternToSearch = [".*" + x + ".*"
                       for x in map(".*".join, itertools.permutations(words))]
    if any(re.search(pattern, author)
        for pattern, author in itertools.product(patternToSearch,elem[1][0][0])):
        return (elem[0][0][0],elem[0][0][1],elem[1][0][0],elem[1][0][1],True)
    words = re.findall("[a-zA-Z]+", elem[1][0][0])
    patternToSearch = [".*" + x + ".*"
                       for x in map(".*".join, itertools.permutations(words))]
    if any(re.search(pattern, author)
        for pattern, author in itertools.product(patternToSearch,elem[0][0][0])):
        return (elem[0][0][0],elem[0][0][1],elem[1][0][0],elem[1][0][1],True)
```

```
    return (elem[0][0][0],elem[0][0][1],elem[1][0][0],elem[1][0][1],False)

coauthorsAllPairsSimilarity = coauthorsWithIndex
    .cartesian(coauthorsWithIndex)
    .filter(lambda a: a[0][2] < a[1][2])
    .map(authorSimilarity)
```

We use the *persist* method from the RDD API to store the *coauthorsWithIndex* RDD in memory because, since Spark works as a pipelined flux and there are Actions and Transformations, if you call twice the same RDD obtained through some transformation operations, these operations will be done twice.

The Transformation operations create a new dataset from an existing one (*map*, *filter*, *flatMap*…), and the Actions operations return a value to the driver program after running a computation on the data set (*reduce*, *collect*, *count*…).

Once we have the combinations, we can compare the pair elements through the regex operations, implemented in the *authorSimilarity* function.

Continuing with the new features in the second iteration, once we have built the graph (GraphFrame) our goal is to have another structure where we will have grouped each network of co-authors, corresponding each group to a different target person.

This can be done through the *connectedComponents* method, from the GraphFrames library, which computes the connected component[42] membership of each vertex and return a graph with each vertex assigned a component ID. We can use this ID to identify each disambiguated (separated) target author.

[42] https://en.wikipedia.org/wiki/Connected_component_(graph_theory)

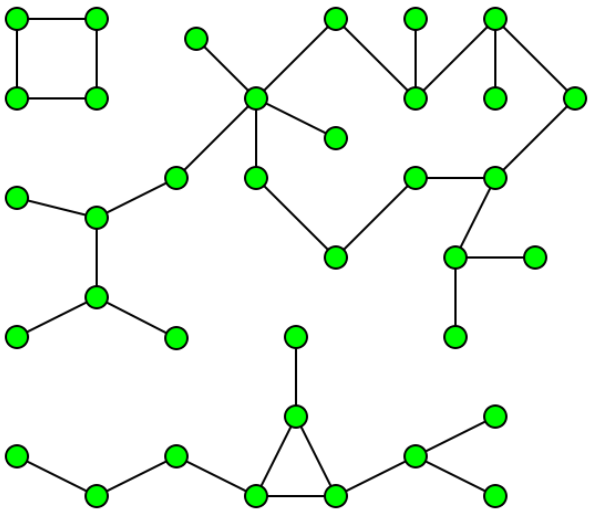

Figure 13. Connected components

And finally, if we group the results by this ID, we obtain all the co-authors in each sub-network, and the publication where he has collaborated with the disambiguated target.

```
spark.sparkContext.setCheckpointDir("hdfs:///Projects/AuthorsDisambiguation/temp/")
components = g.connectedComponents()
def reduceFunc(x, y):
    z = x.copy()
    z.update(y)
    return z
coauthorsNetwork = components.rdd
        .map(lambda x: (x[2],{x[0]:x[1]}))
        .reduceByKey(reduceFunc)
coauthorsNetwork_Article = coauthorsNetwork
        .map(lambda r: (r[0],set(itertools.chain.from_iterable(r[1].values()))))
```

In order to apply the *connectedComponents* method is mandatory, previously, to call the *setCheckpointDir* method, from the SparkContext API, which set the directory under which RDDs are going to be checkpointed. The directory must be a HDFS path if running on a cluster.

Using a new the *yieldInfoFunction,* we can store the authors' extra information in another RDD.

```
def yieldInfoFunc(elem):
    for auth in elem[1]:
        authorInf = separateNameInfo(auth)
        yield (authorInf[0],authorInf[1],[elem[0]])
coauthorsWithInfo = coauthorsByArticle.flatMap(yieldFunc)
```

The last obtained dataset wants to storage the publications where the target is the unique author.

```
articlesOnlyWithGoalAuthor = coauthorsByArticle.filter(lambda r: len(r[1]) == 0)
```

Spark performance

As an example in the development phases, we have tested the algorithm with a specific target that we force in order to obtain a large amount of publications. By this way we can see how Spark works in the heavy operations, like the Cartesian product.

In the *coauthorsAllPairsSimilarity* computation, we have obtained an RDD with 106276 elements (co-authors), distributed among 289 partitions. When we execute a *count* operation to get the number of elements, we have captured the following screenshot, where we can see the distribution of tasks.

Figure 14. Spark Task distribution

Each green block is a *count* task over each partition and, since our machine has 16 cores, at each moment we have, at maximum, 16 parallel tasks.

In the following image (obtained from the htop[43] console command), at the same moment than the previous, we can see how the resources of the machine are working.

[43] https://linux.die.net/man/1/htop

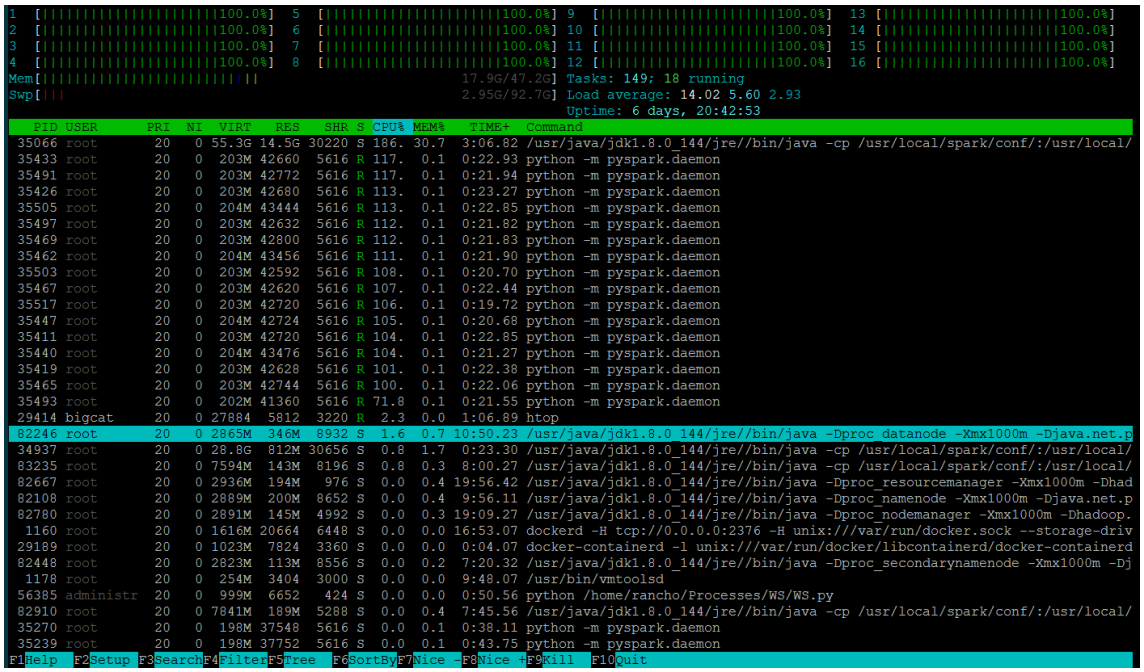

Figure 15. HTop

All the cores are working at 100% (the details say even more) in the Pyspark processes. Finally, we can see other processes related to the distributed tools like the Hadoop Datanode, Spark cluster, Hadoop ResourceManager, Hadoop Namenode, Hadoop NodeManager, etc.

11.3.3. Results

Analyzing the publications obtained filtering with the target's name, we can see how the new method is able to find even the publications which have the target's name modified. For example, filtering with the current target, we have obtained matches like:

['Yamaoka, Y.', 'Yamaoka Y', 'Yamaoka, Yoshio', 'Yamaoka, Yuichi',…]

Thinking about these results, we have realized that in this kind of solutions, actually, the user wants to search the publications of an author of which he knows the name and, probably, one surname. So, one important change is the target's name, since we got one sample where there is a name abbreviation ('Yamaoka, Y.'). We have updated:

```
target = 'Yoshio Yamaoka'
```

With this change we have realized another fact, the current method is not able to catch the possible abbreviations of the names (the previous target's name case). The old target has 77 publications, and this has 16. This will be solved in the next iteration.

The co-authors matching process has already been explained in the details of the algorithm. Just comment that the second approach is much more expensive, in terms of resources, than the first, and could result in an error, despite the probability is low.

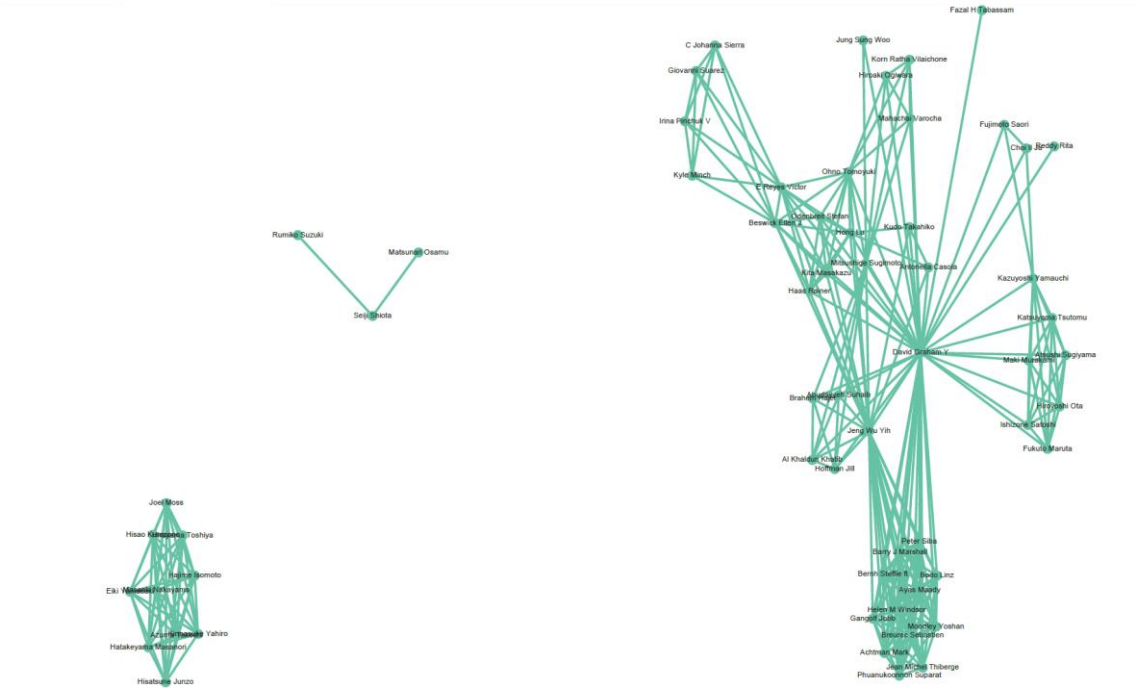

Figure 16. Results of the disambiguation process (second iteration)

The result of the connected components analysis says that there are 3 separated components, the same that we can see in the image.

From the 16 publications obtained filtering with the target author:

- 2 compose the smallest graph, which has 3 nodes (authors). One of them joins the 2 publications.
- 1 composes the left graph, with 10 nodes. These authors do not match with the other, we will try to classify them in the following stages.
- 13 composes the big graph, with 46 authors. There are authors, like 'David Graham Y', which appear in 11 publications. This is a strong graph, good for the disambiguation process, and don't need many improvements.

In respect to the new RDDs, not all the authors will have an associated information, but in the affirmative cases, this could be really helpful, because in many cases this information surely will help to the disambiguation process.

The RDD with the publications where the target is the unique author can't be processed in this stage, because we don't have any way to classify it. In others steps with another kind of data could be processed.

11.4. Third iteration

As I have said in the results of the second iteration, we will implement the definitive method to find the publications of the target author.

Next, we will introduce new data in the algorithm, so we will start working in the second stage.

This data are the subjects of the publications and their hierarchy. We will try to build a tree structure, with the hierarchy, in order to compute the distance between subjects and be able to compare them.

Once we have the first disambiguation, from the stage one, we can work in the second, with two different policies:

- Try to join separated components.
- Try to separate unified components.

The first wants to find common subjects between the different components and, the second, finds the nodes which have worked in different publications and check the subjects from each one, trying to disambiguate them. This iteration is focused in the second case.

The GraphFrames API is based on the DataFrames API of Spark and, in order to analyze the graphs and get the different measures, we will work with the Datafames and the SQL support and, also, the RDDs.

11.4.1. Data & cleaning

The data of the subjects, in the CSV files, has the following structure (is similar to that of the authors):

"US201302263148","http://aims.fao.org/aos/agrovoc/c_12714||http://aims.fao.org/aos/agrovoc/c_10195"

In each row there are two strings, separated by a comma. The first string is the publication ID, and the second is the URL of the subjects of the publications, separated by two pipe character ('||'). Like the data of the authors, this is loaded into a DataFrame and, finally, parsed and stored into other DataFrame. From the URL of the subjects, we need the last part, the subject ID, obtained through a regular expression operation.

```
subjects_raw = spark.read.format("csv").load(hdfsbasePath+"Sub-*.csv",header = True)
subjects = spark.createDataFrame(
        subjects_raw.rdd\
            .map(lambda r:list(r.asDict().values()))\
            .map(lambda r: Row(articleId=r[0], subjects=
```

```
                    re.findall(r'_([^_|]*)\|?',r[1], overlapped=True))
                )
            )
```

In addition, to build the hierarchy of the subjects, we also need the extra data which relate the subjects between them. There are two files, in one we have the broader data and, in the other, the narrower. This data has the following structure:

"http://aims.fao.org/aos/agrovoc/c_3728","http://aims.fao.org/aos/agrovoc/c_1555"

In the case of the broader data file, this row means that the subject 3728 is less specific than 1555. In other words, in the tree structure, the 1555 node is a sub node of the 3728.

The same flow is applied to this new data.

```
subjects_broader_raw = spark.read.format("csv")
       .load(hdfsbasePath+"Subjects_broader.csv",header = True)
subjects_narrower_raw = spark.read.format("csv")
       .load(hdfsbasePath+"Subjects_narrower.csv",header = True)
```

In order to build the graph with the hierarchy, we have to parse this data and join the two Dataframes.

```
subjects_narrower = spark.createDataFrame(
        subjects_narrower_raw.rdd
            .map(lambda r: Row(id=r.s.split('_')[-1],narrower=r.o.split('_')[-1]))
        )
subjects_broader = spark.createDataFrame(
        subjects_broader_raw.rdd
            .map(lambda r: Row(id=r.s.split('_')[-1],broader=r.o.split('_')[-1]))
        )
subjects_hierachy_DF =
    spark.createDataFrame(
        subjects_narrower.join(
               subjects_broader, subjects_narrower.id == subjects_broader.id,'outer')
       .rdd.map(lambda r: Row(
               narrower=r.narrower,
               broader=r.broader,
               id=r[0] if r[0] != None else r[3])
       )
)
v_subjects = subjects_hierachy_DF.select("id").distinct()
e_subjects = spark.createDataFrame(
            subjects_hierachy_DF.rdd
                .map(lambda r:
                    Row(src=r.broader,dst=r.id) if r.narrower == None
                    else Row(src=r.id,dst=r.narrower))
            ).dropDuplicates()
e_subjects = e_subjects
```

```
        .union(e_subjects
                .withColumnRenamed("src","aux")
                .withColumnRenamed("dst","src")
                .withColumnRenamed("aux","dst")
                .select("dst","src")
        )
g_subjects = GraphFrame(v_subjects, e_subjects)
g_subjects.vertices.persist()
g_subjects.edges.persist()
```

The edges of the graph have to be bidirectional, in order to be able to compute distances between nodes in any direction.

11.4.2. The algorithm

The last improvement, in reference the target name filtering, is more complex than the previous implementations, and consists of three steps:

i) Generate a pattern, for the matching phase, with the first letter of each word of the target's name.

ii) Find all the names, among the publications, that match with one of the previous patterns.

iii) For each obtained name, check if match with the completed target's name.

```
def targetSearch(elem):
    # Posible targets with info
    trgts = [author
        for pattern, author in itertools.product(patternToSearch,elem[1])
            if re.search(pattern, separateNameInfo(author)[0])]
    # Targets with info
    tgt = [el for el in set(trgts) if all(word in target for word
           in re.findall("[a-zA-Z]+", separateNameInfo(el)[0]))]
    return (elem[0],
            [el for el in elem[1] if el not in tgt],
            tgt)
```

By this way we can obtain even the target's name abbreviations.

We have included in the new RDD the matched name, in the last field. The *filterAuthor* and *mapExtractAuthor* functions are replaced by:

```
coauthorsByArticle = authors
        .map(targetSearch)
        .filter(lambda x: len(x[2]) > 0)
coauthorsByArticle.persist()
```

In order to start with the second stage, once we have solved the cleaning process, we have added the subjects after the connected components analysis.

To try to separate the components, we only can work with these authors that have worked in more than one publication, and compare the subjects in each one.

In one side we have the *components* DataFrame, which has the articles grouped by author and, in the other, we have the *subjects* DataFrame, with the subjects of each article. To put them together, we will expand (through the method *explode*, from the pyspark.sql.functions package) the articles list in each row of the *components* DataFrame and drop the duplicates.

```
articlesSubjectsComponent = components
    .filter(size(components.articles) > 1)
    .select("id",explode("articles").alias("articleId"),"component")
    .dropDuplicates()
    .join(subjects,"articleId","left_outer")
    .filter(size("subjects")>0)
```

Finally, we have to apply a filter to ignore the articles that do not have subjects, since we are comparing them. This fact will cause problems, so in the next iteration we will try to minimize them in order to try to remove this filter.

Now, we have implemented a complex function that returns the authors that have to be separated. For example, if in the first stage we had an author A that works in the publications P1 and P2, with the subjects S100 and S200 respectively (these subjects have no relation), this function will return two new names for this author (A1 and A2), in order to represent a different authors in a new connected components analysis.

```
def separateAuthors(articlesSubjectsComponent):
    authorArticlesSubjects =  articlesSubjectsComponent
        .groupBy("id")
        .agg(
                collect_list("articleId").alias("articles"),
                collect_list("subjects").alias("subjects")
            )
    authorArticlesSubjectsLocal = authorArticlesSubjects
        .filter("subjects IS NOT NULL").rdd.map(lambda r:
(r.id,r.articles,r.subjects)).collect()
    distAuth = [(
                    author[0],
                    c[0],c[1],

applyBFS(author[2][author[1].index(c[0])],author[2][author[1].index(c[1])])
                )
                for author in authorArticlesSubjectsLocal
                    for c in itertools.combinations(author[1],2)
               ]
    df = spark.createDataFrame(distAuth,['author','src','dst','dist'])
```

```
    df.persist()
    modifications = []
    for author in df.select("author").distinct().rdd.map(lambda r: r.author).collect():
        g = GraphFrame(
                df.filter("author = '" + author + "'")
                    .select("src")
                    .union(df.filter("author = '" + author + "'")
                        .select("dst")
                    ).distinct()
                    .withColumnRenamed("src","id"),
                df.filter("author = '" + author + "' AND dist >= 0")
                    .select("src","dst"))
         spark.sparkContext
                .setCheckpointDir("hdfs:///Projects/AuthorsDisambiguation/temp/")
        cc = g.connectedComponents().rdd
        .map(lambda r: (r.component,[r.id]))
        .reduceByKey(lambda x,y: x+y)
        cc.persist()
        if cc.count() > 1:
            modifications +=
         [(author,article, component[0])
                for component in cc.collect() for article in component[1]]
        cc.unpersist()
    return modifications
```

In a few words, this function takes the *articlesSubjectsComponent* DataFrame, applies a *groupBy* and *aggregate* operations (similar to the *reduceByKey* operation in the RDDs), grouping by author name and storing the articles and the subjects of each one, thus generating a structure for the next operation. Next, we have to collect this structure in a local mode, in the driver, where we will compute the distances between the subjects of each publication of each author.

The distance between the subject groups will be computed over the hierarchy graph of subjects, through the *BFS* [44] method, from the GraphFrames library. This method, as the name suggests (Breath-first search), finds the shortest paths from one vertex (or set of vertices) to another vertex (or a set of vertices)[45]. We have defined another function to implement this.

```
def applyBFS(sGroup1,sGroup2, maxPathLength):
    res = g_subjects.bfs(
            "id IN ('" + "','".join(str(x) for x in sGroup1) + "')",
            "id IN ('" + "','".join(str(x) for x in sGroup2) + "')",
            maxPathLength= maxPathLength
            )
```

[44] https://graphframes.github.io/user-guide.html#breadth-first-search-bfs
[45] https://en.wikipedia.org/wiki/Breadth-first_search

```
    if res.count() == 0:
        return -1
    if len(res.columns) == 2:
        return 0
    return int((len(res.columns) - 1)/2)
```

We will consider that two subjects are similar if the distance between them is less than 7 (defined in the parameter `maxPathLength` of the *BFS* method).

The *BFS* method returns a DataFrame with one row for each shortest path between the matching vertices. For example, if we have the groups of subjects:

['5851', '13325', '1463', '15733', '4195', '6596']

['26593', '5962', '7631', '35120', '11042', '35118', '4810', '5630']

The *BFS* algorithm will return:

```
+-------+------------+-------+--------------+--------+--------------+-------+-------------+-------+
|   from|          e0|     v1|            e1|      v2|            e2|     v3|           e3|     to|
+-------+------------+-------+--------------+--------+--------------+-------+-------------+-------+
|[15733]|[7280,15733]| [7280]|  [49904,7280]| [49904]|  [4807,49904]| [4807]|  [5630,4807]| [5630]|
| [1463]|[31667,1463]|[31667]|[331215,31667]|[331215]|[31669,331215]|[31669]|[26593,31669]|[26593]|
+-------+------------+-------+--------------+--------+--------------+-------+-------------+-------+
```

This means that the minimum distance between these two groups is 4, and there are two nodes in the first group at distance 4 from other two nodes in the second group.

If the same node is in both groups we will obtain:

```
+-------+-------+
|   from|     to|
+-------+-------+
|[32395]|[32395]|
+-------+-------+
```

And, if the nodes are in separated components or the distance is greater than 6, the result will be empty:

```
+---+
| id|
+---+
+---+
```

The last part of the *separateAuthors* function takes these distances (between publications), from each author, and build a graph in order to apply a connected components analysis. In the graph,

the nodes are the different publications, and there will be an edge between two nodes if the distance between them it's between 0 and 6. The connected components analysis will show the disambiguated authors, in terms of subjects.

Since we are working in the driver, and these auxiliary graphs (for each author) are small, we could have worked with simpler structures from Python, like the based on sparse matrix representations and the algorithms from the SciPy libraries[46]. Probably more effective in this case, but we have preferred to continue working with GraphFrames avoiding the introduction of new libraries.

Only remains to apply these results in the authors' names of the first stage, and recalculate the global connected components with these new disambiguated authors. The modifications will be applied through the function:

```
def updateNames(a):
        items = [x for x in modifications if x[1] == a[0]]
        if len(items) > 0:
            authors = [x[0] for x in items]
            return (a[0],
               [n if n not in authors
                      else str(n) + " " + str(items[authors.index(n)][2])
                      for n in a[1]],a[2])
        else:
            return a
```

In order to reutilize the first stage code, this has been organized in different functions, allowing the possibility of being called again.

11.4.3. Results

First, the results of the first stage. Through the new improvements, we have obtained 62 possible publications (6 of them with the target as a unique author) and 175 co-authors. The connected components analysis has obtained 20 separated components in the graph.

We have improved the graph plot, adding colors to represent different measures. If a pair of co-authors has worked more than once with the target, the edge is green; if they have worked more than twice, the edge is red and, if they only have worked once, is gray. The nodes are printed in a yellow scale, being darker when the co-author has worked more times with the target.

[46] https://docs.scipy.org/doc/scipy/reference/sparse.csgraph.html#module-scipy.sparse.csgraph

Figure 17. Results of the disambiguation process (third iteration, stage one)

In this plot, we only can see the different components of the graph, and how bad the plot library is drawing a large number of separated components.

Enlarging the bigger components for the analysis:

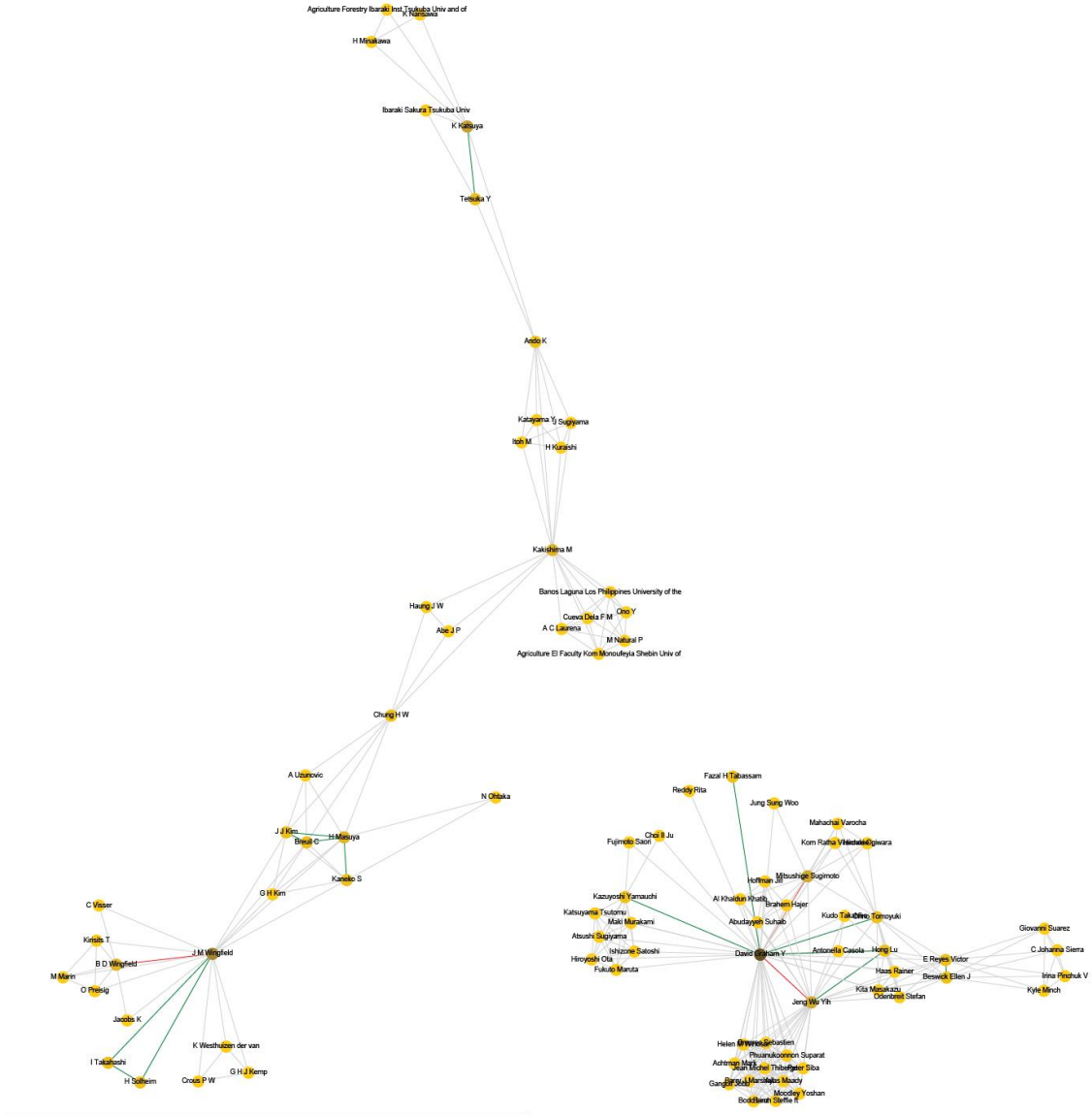

Figure 18. Enlarging the results of the disambiguation process (third iteration, stage one)

We have seen the same right component in the results of the second iteration and a similar left component, in the first.

The left component is a clear example of the improvements made in the algorithm, because now there are new nodes that contribute in the consistency of the network, although there are still weak unions.

Grouping by the target, we have 5 possible target's names and, enlarging the plot:

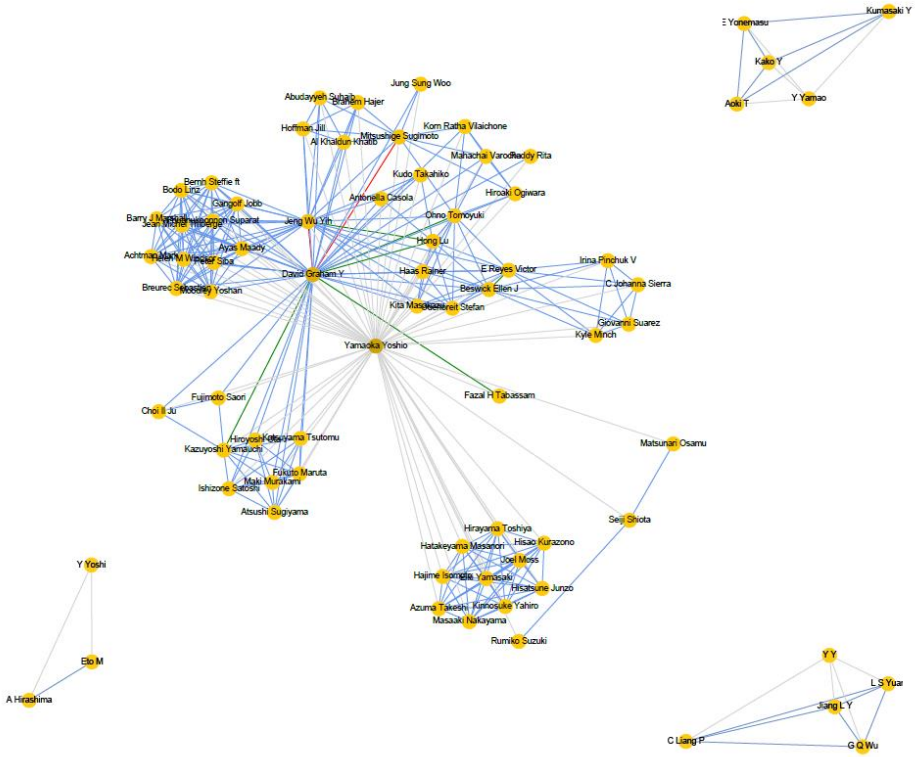

Figure 19. Results of the disambiguation process (third iteration, stage one). Grouping by the target.

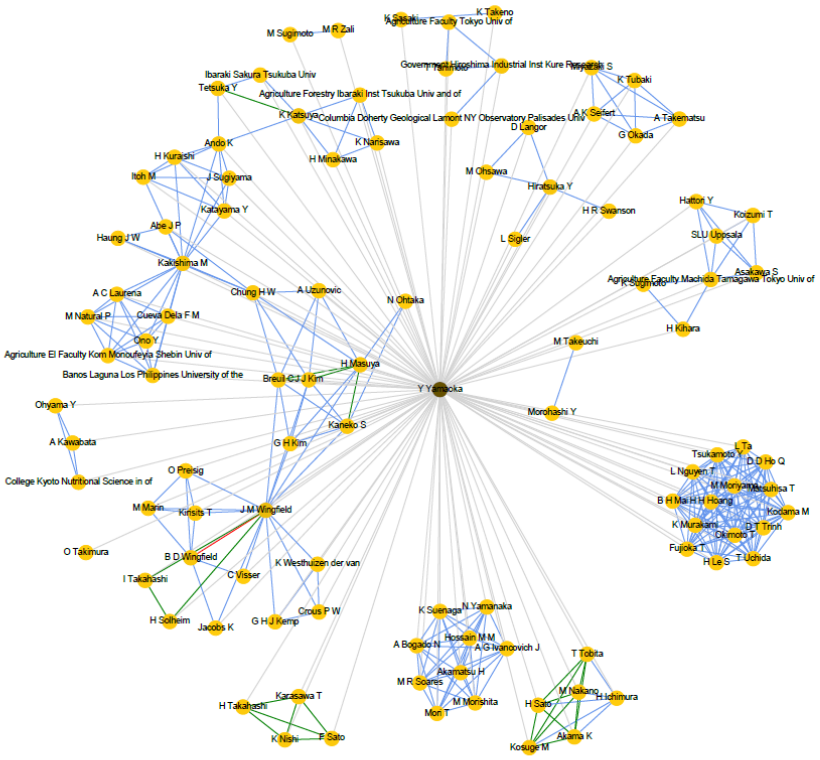

Figure 20. Results of the disambiguation process (third iteration, stage one). Grouping by the target.

This time, the edges in gray are related to the target, the others are blue, green or red.

In these plots, we have grouped the different components by the target's name from which they have been obtained.

As regards the subjects, we have taken for granted that they had a tree hierarchy structure, but this is not correct. While we were trying to plot this structure, through specific methods of the *toyplot* library to print trees, always we got an error saying that this was not a tree. After some research over the data, we could detect that there are subjects that have more than one broader node, breaking the tree structure. So, from then, we have worked with a common graph structure, the theory also works well. Here we can see how big this graph is.

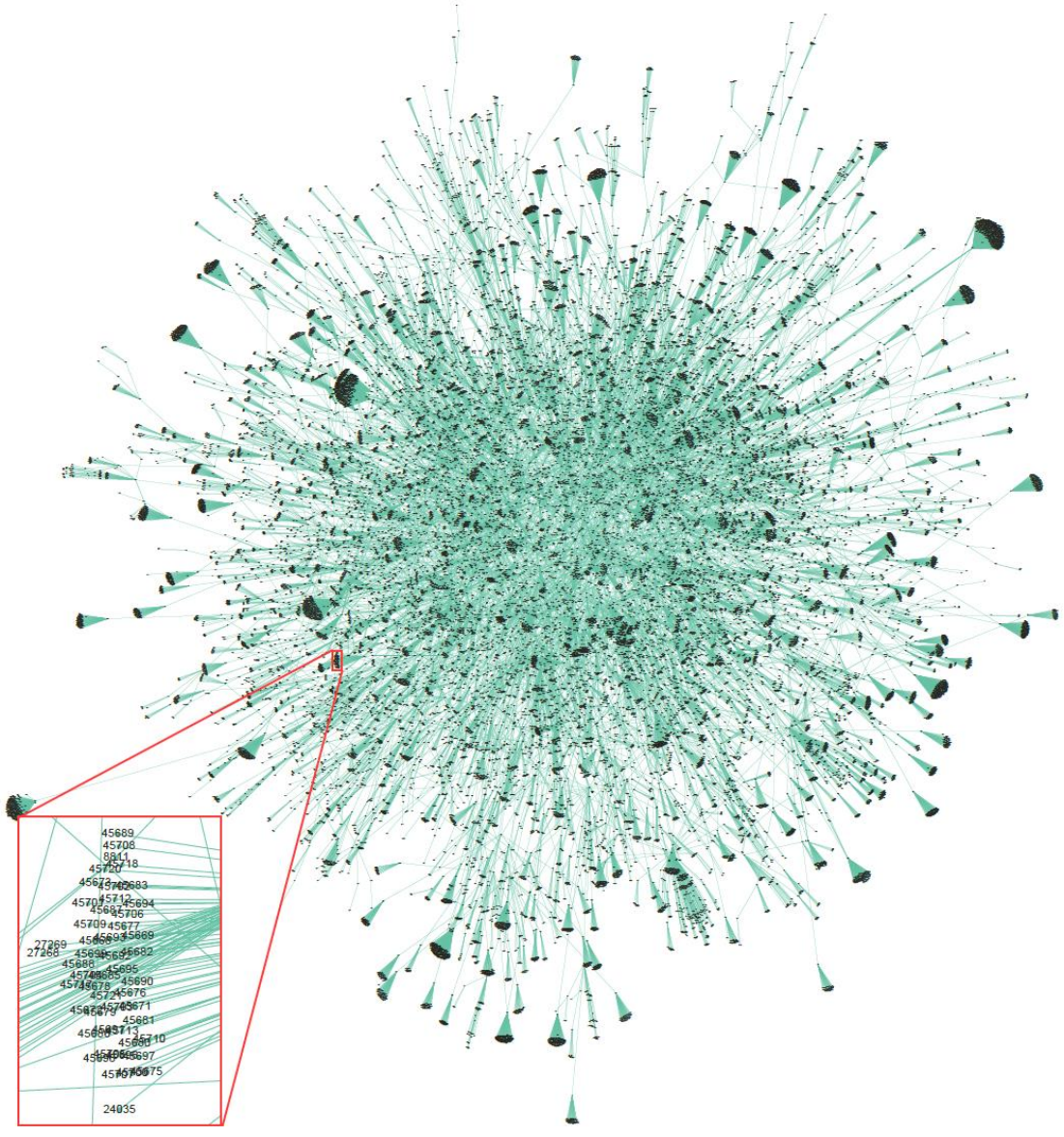

Figure 21. Hierarchy structure of subjects.

Agree with the AGROVOC information, there are more than 32000 concepts (32703 exactly) and almost 34000 edges (33905).

As a SQL query example, the top concepts of the hierarchy can be obtained through the commands:

```
subjects_hierachy_DF.createOrReplaceTempView("subjects_hierachy")
spark.sql("SELECT DISTINCT(id) FROM subjects_hierachy WHERE broader IS NULL")
```

Or:

```
subjects_hierachy_DF.filter("broader IS NULL").select("id").distinct()
```

There are 25 top concepts in the hierarchy:

[7644, 330493, 7778, 330704, 330829, 330919, 330892, 330985, 330834, 330995, 4788, 6211, 331000, 9001017, 330705, 330988, 330979, 50227, 13586, 331061, 330991, 49874, 49904, 331093, 330998]

Applying the connected components procedure, we have found only 3 separated components. If the graph had been a tree, this will have as much components as top concepts.

```
spark.sparkContext.setCheckpointDir("hdfs:///Projects/AuthorsDisambiguation/temp/")
subjectComp_ = g_subjects.connectedComponents()
subjectComp = subjectComp_
        .rdd
        .map(lambda x: (x[1],[x[0]]))
        .reduceByKey(lambda a,b : a + b)
subjectComp.persist()
```

Working with this structure is where we have needed more computational resources.

The currently implemented part of the second stage (where we try to separate the same authors), with the current target, is able to disambiguate two authors, separating them into four. These authors are (the parsed names, from the stage one):

- Agriculture Faculty Machida Tamagawa Tokyo Univ of
- Government Hiroshima Industrial Inst Kure Resh

These are special cases, because are not people, are institutions, but the algorithm is able to separate them through the subjects. The new resulting names are:

- Agriculture Faculty Machida Tamagawa Tokyo Univ of 1271310319616
- Agriculture Faculty Machida Tamagawa Tokyo Univ of 670014898176
- Government Hiroshima Industrial Inst Kure Resh 730144440320
- Government Hiroshima Industrial Inst Kure Resh 412316860416

These results show an important fact, the first stage is able to perform a good classification, because the probability of having two authors with the same (similar) name is really difficult. Classify and separate by author's name is a good choice for the first stage of the algorithm.

Applying the modifications, and recomputing the global connected components analysis, the result shows there are 23 components, plus the 6 publications where the target is the unique author.

The original components, in stage one:

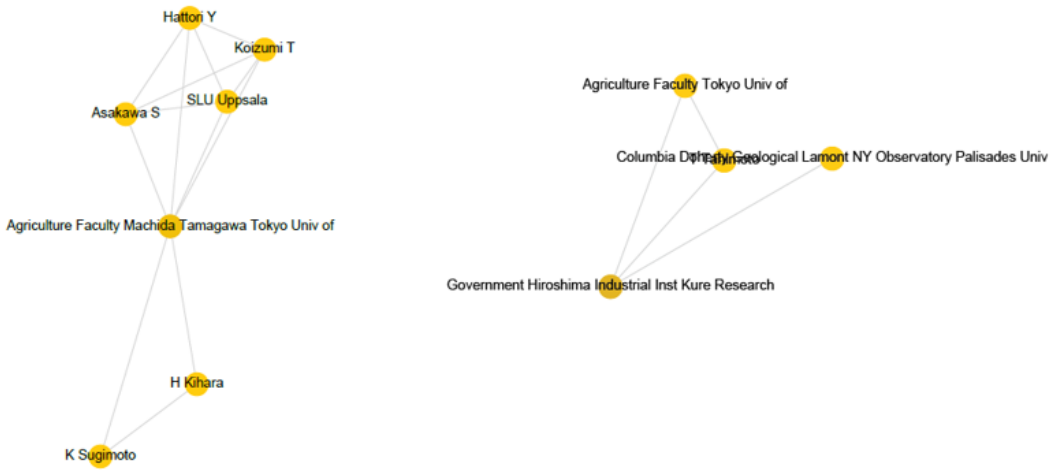

Figure 22. Original components from stage one.

And the new components, in stage two:

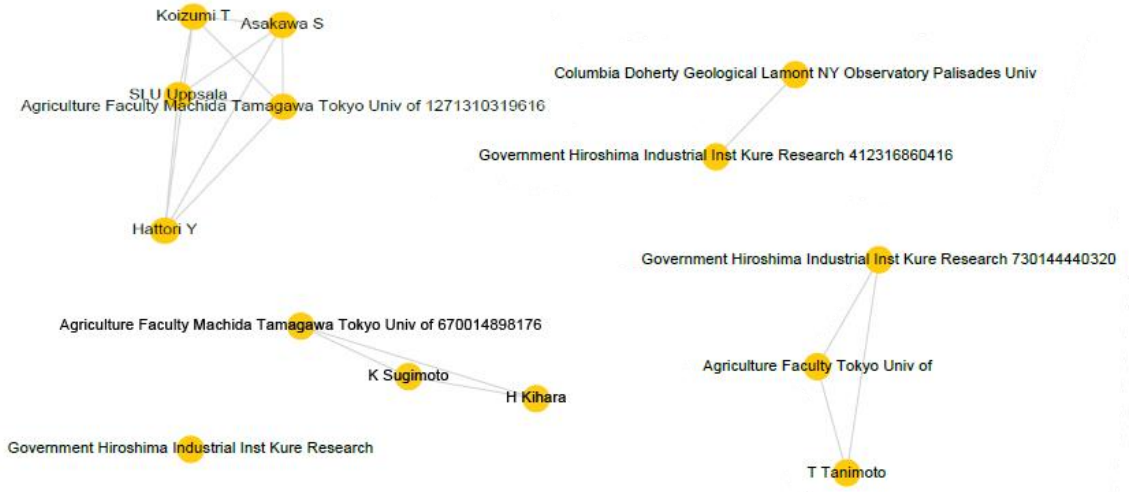

Figure 23. New components in stage two.

As we can see, there is one error, in the 'Government Hiroshima Industrial Inst Kure Resh' case, this is because there is an article without subjects. The next iteration will try to solve it.

11.5. Fourth iteration

In the last iteration of the project, we will continue working in the second stage, in the second part of it. Now we are 'sure' that all of the authors in each component are correct, so we will try to join components.

In the results of the third iteration, we saw there were publications without subjects. In order to try to minimize this problem, we will introduce in the solution the subjects of the journals related to the publications.

On the other hand, we will add some lines of code to improve the processes of data loading, cleaning and the computation of the hierarchy of the subjects, since they only need to be calculated once.

11.5.1. Data & cleaning

In order to add the new data to the algorithm, we proceed as in previous cases. First, we load the data and, second, we clean it.

```
subjects_journal_raw = spark
       .read.format("csv").load(hdfsbasePath+"journalSubjects_*.csv",header = True)
```

The cleaning process takes the articles with subjects and obtains the id of each subject. There are strange URLs like:

'http://dewey.info/class/574.1905/about'

Which will be removed through the regular expression.

```
subjectsJournals = spark.createDataFrame(
        subjects_journal_raw.rdd
            .map(lambda r: list(r.asDict().values()))
            .filter(lambda r: r[1] is not None)
            .map(lambda r: Row(articleId=r[0],
                        subjects=re.findall(r'_([^_|]*)\|?', r[2], overlapped=True)))
            .filter(lambda r: len(r[1]) > 0)
        )
```

11.5.2. The algorithm

The first time the algorithm is run, the graph with the hierarchy of the subjects will be calculated and stored in the HDFS file system in a parquet[47] format. To do so we will store the *vertices* and *edges* Dataframes.

```
g_subjects.vertices.write.mode('overwrite').parquet(hdfsbasePath+"/vertices")
```

[47] Apache Parquet is a columnar storage format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language.

```
g_subjects.edges.write.mode('overwrite').parquet(hdfsbasePath+"/edges")
```

The connected components analysis over this graph is also always the same, so we can store it too in the HDFS file system.

```
subjectComp.saveAsPickleFile(hdfsbasePath+"/subjectComp")
```

The method *saveAsPickleFile* save this RDD as a SequenceFile of serialized objects. The serializer used is pyspark.serializers.PickleSerializer, default batch size is 10.

In the next executions we will only need to load these structures from the file system and build the graph.

```
v_subjects = spark.read.parquet(hdfsbasePath+"/vertices")
e_subjects = spark.read.parquet(hdfsbasePath+"/edges")
g_subjects = GraphFrame(v_subjects, e_subjects)
subjectComp = spark.sparkContext.pickleFile(hdfsbasePath+"/subjectComp")
g_subjects.vertices.persist()
g_subjects.edges.persist()
subjectComp.persist()
```

Finally, we can do the same with the base structures of the *authors* RDD and the *subjects* DataFrame, despite the cleaning processes over them are really fast.

```
authors.saveAsPickleFile(hdfsbasePath+"/authors")
subjects.write.mode('overwrite').parquet(hdfsbasePath+"/subjects")
subjectsJournals.write.mode('overwrite').parquet(hdfsbasePath+"/subjectsJournals")
```

And:

```
authors = spark.sparkContext.pickleFile(hdfsbasePath+"/authors")
subjects = spark.read.parquet(hdfsbasePath+"/subjects")
subjectsJournals = spark.read.parquet(hdfsbasePath+"/subjectsJournals")
```

The processes that create the previous structures (the loading of the files and the cleaning of the data) will also be executed only the first time.

With the new data, we have to modify the creation of the *articlesSubjectsComponent* DataFrame.

```
articlesSubjectsComponent = components
    .filter(size(components.articles) > 1)
    .select("id",explode("articles").alias("articleId"),"component")
    .dropDuplicates()
    .join(subjects.withColumnRenamed("subjects","s1"),"articleId","left_outer")
    .join(subjectsJournals.withColumnRenamed("subjects","s2"),"articleId","left_outer")
    .withColumn("subjects",
       when((size(col("s1")) == 0) & (col("s2").isNotNull()),col("s2"))
       .otherwise(col("s1")))
```

Now we will implement the second part of stage 2, and we will try to join components with common subjects.

First, we have to take into account the articles where the target is the unique author. We will put each one of these in a different component, with a negative ID, to avoid the collision with the resulting IDs of the connected components analysis.

```
articlesOnlyWithGoalAuthorDf = spark
    .createDataFrame(articlesOnlyWithGoalAuthor
        .zipWithIndex()
        .map(lambda r: Row(articleId=r[0][0],component=-1-r[1]))
)
```

This is important for the final result, in order to compare it with the previous ones, where we had not considered this publications as a components.

In this case, we have to work with all components, so we will perform a similar operation than the previous, where we have obtained the *articlesSubjectsComponent* DataFrame but, this time, without the initial filter.

```
articlesSubjectsComponent = components
    .select(explode("articles").alias("articleId"),"component")
    .dropDuplicates()
    .union(articlesOnlyWithGoalAuthorDf)
    .join(subjects,"articleId","left_outer")
    .join(subjectsJournals.withColumnRenamed("subjects","s2"),"articleId","left_outer")\
    .withColumn("subjects",
       when((size(col("s1")) == 0) & (col("s2").isNotNull()),col("s2"))
       .otherwise(col("s1"))
)
```

We have defined a new function to solve this point that has a behaviour similar to the previous *separateAuthors* function. It starts grouping the articles by component, and aggregating the necessary columns through a personalized UDF[48] in order to perform a specific structure for the next step. Next, this structure is collected, in the driver, to compute the distance (through the *BFS* method) between components. Finally, the function build a graph with the components, and set an edge between two nodes if the distance between them it's between 0 and 6. The connected components analysis will join the components with common, or similar, subjects.

[48] User-Defined Functions (aka UDF) is a feature of Spark SQL to define new Column-based functions that extend the vocabulary of Spark SQL's DSL for transforming DataFrames.

```
def joinComponents(articlesSubjectsComponent):
    joinUDF = udf(lambda l:
       np.unique(np.hstack(np.array([x for x in l if x is not None]).flat)).tolist()
              if len(l) > 0
              else []
       ,ArrayType(StringType()))
    subjectsComponent = articlesSubjectsComponent
       .groupBy("component")
       .agg(joinUDF(collect_list("subjects")).alias("subjects"))
    subjectsComponentLocal = subjectsComponent
       .filter(size(subjectsComponent.subjects) > 0)
       .rdd
       .map(lambda r: (r.component,r.subjects))
       .collect()
    distComp = [(
                   c[0][0],c[1][0],
                   applyBFS(c[0][1],c[1][1],6)
                )
                for c in itertools.combinations(subjectsComponentLocal,2)
              ]
    df = spark.createDataFrame(distComp,['src','dst','dist'])
    g = GraphFrame(
            df.select("src")
              .union(df.select("dst"))
              .distinct()
              .withColumnRenamed("src","id"),
            df.filter("dist >= 0")
              .select("src","dst"))
    spark.sparkContext.setCheckpointDir("hdfs:///Projects/AuthorsDisambiguation/temp/")
    return g.connectedComponents()
       .rdd
       .map(lambda r: (r.component,[r.id]))
       .reduceByKey(lambda x,y: x+y)
       .collect()
```

Only remains to apply these modifications to the *components* DataFrame and recalculate the final outputs. The modifications will be applied via the function:

```
def updateComponents(r):
    item = next((x for x in modifications if r.component in x[1]),None)
    if(item != None):
        return Row(id=r.id,articles=r.articles,component=-item[0]-1000)
    else:
        return Row(id=r.id,articles=r.articles,component=r.component)
```

The components ID will be set in negative in order to avoid collisions with the components which are not modified.

11.5.3. Results

The last added subjects, from the journals, are not sufficient and there are still publications without subjects. As we have seen, this can be a problem; so we have to keep working with the filters.

In the first part of stage two, 32 of the 96 articles do not have subjects and, in the second, 9 of the 29 (23 components from the first part plus the 6 publications with the target as a unique author) components neither have subjects. 3 of these 9 are components with the target as a unique author, the other 6 are common components which have publications without subjects.

We have tested this part of the algorithm with different values in the *maxPathLength* variable, of the *BFS* procedure; because, since we are comparing components, and these components have all the subjects from each article that he groups, there are a lot of subjects in each component. So, the probability that the groups of subjects are closer is greater.

maxPathLength	Final Components	Real grouping
6	1 + 6 + 3	From 20 -> to 1
3	1 + 6 + 3	From 20 -> to 1
1	4 + 6 + 3	From 20 -> to 4

Table 1. Results of Stage 2

The third column shows the real grouping of components performed by the algorithm. This has to be analyzed critically, taking into account that there are components without subjects that cannot be joined and cannot be counted as grouped. This column says: from 20 original components with subjects, we have obtained X grouped components.

Finally, the 'Final Components' column, shows how many components are, in total, after the algorithm execution. The +6 +3 are the components without subjects (6 common components and 3 components with the target as a unique author).

12. Conclusions

The results of each stage, or iteration, have already been commented but, making an overall analysis of the stages and their results, we can say that the first stage performs a good first disambiguation, and the improvements done at this point have helped. This is because the probability of publishing with two different people (or more) with the same name is remote and hard to find. Therefore, in most cases, this first stage makes unnecessary the first part of the second stage, where we try to separate the authors with the same name.

The second part of the second stage is important, because tries to put together the components with similar subjects. It's possible that the same author has different groups with which publishes in similar subjects but, what is the probability of two different authors, with the same name (or really similar), publish with the same subjects? In my opinion, not very big.

The problem with the publications without subjects makes the result of the second stage incomplete. This fact is important and has to be taken into account when the results are interpreted.

Now, the execution times of the different parts and stages of the algorithm. The second part of the second stage is represented depending on the variable *maxPathLength*.

Stage	maxPathLength	Time
Data (loading & cleaning) First execution	-	10 min
Data (load cleaned data) Next executions	-	6 seg
Stage 1	-	2 min
Stage 2 – part 1	-	13 min
Stage 2 – part 2	6	56 min
	3	10 min
	1	1 min

Table 2. Algorithm times.

As we can see, the time increases depending on the *maxPathLength* variable of the *BFS* procedure. This is normal, because as the variable increases, there are more possible paths to be calculated. This is where the algorithm spends most of the time, computing the distance between groups of subjects in the graph of the hierarchy.

On the other hand, the system built through Docker has worked perfectly, and it has been able to support the needs of the problem.

We have worked with distributed tools like Hadoop, HDFS and Spark, which was an initial goal, and we have been able to see how they interact with each other. Also, we have used Spark with Python, an interesting combination that puts together the Spark performance with the Python features (agility, velocity…), fact that we cannot find in other languages like Java. I have worked with Java and Spark and is not the same.

The code has been refactored and, as I have said before, it has been organized in functions, in order to be called when is necessary. It can be found in the main.py file, appended to this document, together with Dockerfile and entrypoint.sh files used to deploy the system.

13. Future work

The project remains open, and there are different ways to follow.

The first, and most immediate, is to try to add more subjects to the algorithm. It's difficult, because the data is limited, but the conferences, where the publications have been exposed, remains unexplored.

Another obvious way to follow is to add more stages, with another kind of data, to the algorithm. Like the extra data found in the names of the authors, stored through the solution; the issued date, the submitted date, or the conference and journal features.

A point I would have liked to implement is the validation, through a verified data set, of the algorithm. That is to say, try to disambiguate a data set where we know and we have classified the authors (through an ID, for example) in order to be able to see how the algorithm works.

Also an application with an interface can be built to interact with the algorithm and to return the results, or try to solve the little graphs, used in the second stage, through other libraries (like SciPy) in order to improve the efficiency.

The last way to follow is the most ambitious, it would consist in disambiguating all the authors, of the data set, at the same time or execution.

14. Bibliography

Conda.io. (2017). *Conda — Conda documentation*. [online] Available at: https://conda.io/docs/

Docs.openlinksw.com. (2017). *OpenLink Virtuoso Universal Server Documentation*. [online] Available at: http://docs.openlinksw.com/virtuoso/

Docs.python.org. (2017). *10.1. itertools — Functions creating iterators for efficient looping — Python 3.6.2 documentation*. [online] Available at: https://docs.python.org/3.6/library/itertools.html

Docker. (2017). *Docker*. [online] Available at: https://www.docker.com

En.wikipedia.org. (2017). *AGRIS*. [online] Available at: https://en.wikipedia.org/wiki/AGRIS

En.wikipedia.org. (2017). *Author Name Disambiguation*. [online] Available at: https://en.wikipedia.org/wiki/Author_Name_Disambiguation

En.wikipedia.org. (2017). *Docker (software)*. [online] Available at: https://en.wikipedia.org/wiki/Docker_(software)

En.wikipedia.org. (2017). *IPython*. [online] Available at: https://en.wikipedia.org/wiki/IPython

En.wikipedia.org. (2017). *Operating-system-level virtualization*. [online] Available at: https://en.wikipedia.org/wiki/Operating-system-level_virtualization

En.wikipedia.org. (2017). *Python (programming language)*. [online] Available at: https://en.wikipedia.org/wiki/Python_(programming_language)

En.wikipedia.org. (2017). *Record linkage*. [online] Available at: https://en.wikipedia.org/wiki/Record_linkage

En.wikipedia.org. (2017). *SPARQL*. [online] Available at: https://en.wikipedia.org/wiki/SPARQL

Fao.org. (2017). *Food and Agriculture Organization of the United Nations*. [online] Available at: http://www.fao.org

Hadoop.apache.org. (2017). *Welcome to Apache™ Hadoop®!*. [online] Available at: http://hadoop.apache.org/

Ipython.org. (2017). *Jupyter and the future of IPython — IPython*. [online] Available at: https://ipython.org/

Parquet.apache.org. (2017). *Apache Parquet*. [online] Available at: https://parquet.apache.org/

Pythonhosted.org. (2017). *Spyder - Documentation — Spyder 3 documentation*. [online] Available at: https://pythonhosted.org/spyder/

Python.org. (2017). *Welcome to Python.org*. [online] Available at: https://www.python.org/

Shankhdhar, G. (2017). *Hadoop Cluster Configuration Files | Edureka*. [online] Edureka Blog. Available at: https://www.edureka.co/blog/hadoop-cluster-configuration-files/

Spark.apache.org. (2017). *Apache Spark™ - Lightning-Fast Cluster Computing*. [online] Available at: http://spark.apache.org/

W3.org. (2017). *RDF - Semantic Web Standards*. [online] Available at: https://www.w3.org/RDF/

W3.org. (2017). *SPARQL Protocol for RDF*. [online] Available at: https://www.w3.org/TR/rdf-sparql-protocol/