Building Wikipedia Evolution Graph

Daniele Bissoli - 197810
University Of Trento
daniele.bissoli@studenti.unitn.it

1 INTRODUCTION

Wikipedia is an enormous free multi-language encyclopedia that contains a lot of records and facts regarding people's culture, history and science. It has been created by a community of volunteers who has continuously increased the stored knowledge year by year. One interesting aspect of these updates, it that they are all logged, which means that for each Wikipedia page exists a number of different versions, one for each update that has been made to that page.

Since it is a web-based encyclopedia, its structure high resembles a graph, which is a particular data structure where data are arranged as a collection of objects interconnected among them. Formally, a graph G is a pair composed by set of vertices V and a set of edges E. Similarly, Wikipedia can be seen as a graph W where vertices are the pages P and edges are the links L between pages.

$$G = (V,\ E) \equiv W = (P,\ L)$$

In the case of Wikipedia, the edges are directional, which means that if exists a connection between page p_1 and page p_2, it might not exist the opposite relation. Due to its structure and openness, Wikipedia can be easily exploited to conduct a high number of different researches, such as natural language processing or data mining tasks.

In the last few year, it has been possible to observe a change in how Wikipedia can be considered as data source. Indeed, its size has grown exponentially, every day thousand of updates have been performed to its pages and a large variety of different layouts characterize its pages. All these elements (volume, velocity and variety respectively) make Wikipedia belonging to a category of data, called Big Data. As a result, processing Wikipedia has become a complex task, because traditional tools and methods are no more suitable for coping with this kind of data. For this reason the **MapReduce** [3] programming model has been introduced to solve Big Data challenges.

Given these introduction notions, the goal of this project was to produce a tool capable of visualizing the evolution across the time of a portion of the Wikipedia graph (only for English pages). In particular, the objective requires not only to show the graph, such as nodes and edges, but also to show how similar (or close) are the pages that are linked together. This is done by defining a metric of similarity and using it as a weight attached to each edge.

In order to achieve this goal, some of MapReduce concepts have been used, in conjunction with a particular kind of database for storing the results.

2 THE PROJECT

It is divided into the following three main parts:

- **The processing of raw data**
  It is the part responsible of converting the text of Wikipedia pages into something meaningful for the system, that is extract features from the pages to enable their comparison.
- **The storage of processed data**
  In this section are explained the issues, methodologies and technologies employed for storing processed data, so that they can be later accessed by each components in the most efficient and easy manner.
- **The visualization of stored data**
  It is the part in charge of retrieving the stored data and showing them to the user in a significant way.

Each stage will be further analyzed throughout this report to explain which are their objectives and what are the main choices that has been taken to implement them.

3 DATA PROCESSING

This project section is in charge of loading the raw data (pages' text), discover meaningful features from them and finally transform them so that they can be later exploited in the visualization module. This part has been written in Scala [11] programming language adopting the Apache Spark [1] processing framework for MapReduce computation, leveraging the DataFrame API of Spark 2.x to further improve the processing work flow. In the following paragraphs will be explained each subcomponents of this module, highlighting their functionalities, input and output and which technologies have been employed.

3.1 Data Loading

The first step for processing data is to collect them and make them suitable to be later processed. To gather Wikipedia pages, a special export tool, offered by Wikipedia [12], has been used. It let users download any specified page, together with its history of versions.

Afterwards, the retrieved pages are loaded into Spark application by means of the `spark-xml` [16] library. This library is necessary to enable Spark reading methods to parse XML files. Using this component, a first approach has been taken to read XML data, which was based on letting the library inferring the schema by itself. However, this method is not able to scale for processing humongous files, since it requires to read in memory at least a portion of the initial data just to extract their schema. For this reason, a different procedure has been adopted to satisfies this requirement and speed-up the loading phase: instead of asking the system to infer the XML schema, a built-in one has been provided. This built-in schema is based on the XML schema published by Wikipedia [15], which regulates how its pages are structured.

After this procedure, each page is expanded to represent all its versions, so that the concept of evolution across the time is well-represented by the data. Eventually, each row of the DataFrame

```
{{Infobox Galaxy
  | name = [[New General Catalogue|NGC]] 4457
  | image = NGC 4457.jpg
  | image_size = 250px
  | caption = The galaxy NGC 4457
  | epoch = [[J2000]]
  ...
  | dist_ly = 54,801,600 [[Light-year|ly]]
  | group_cluster = [[Virgo II Groups]]
  | appmag_v = 11.76 <ref name="ned">
  | size_v = 2.7 x 2.3 <ref name="ned">
  | constellation name = [[Virgo (constellation)]]}}
```

Figure 1: Example of Wikipedia page Infobox

that contains Wikipedia pages has the following schema:

$$< title, timestamp, page\_text >$$

3.2 Data Preprocessing

Once pages (and their versions) are loaded into the Spark application, the next step is to discover meaningful information from the data. The first point that have to observed is that Wikipedia pages' text is composed both by structured and unstructured text, which means that it is possible to retrieve different information from those parts. The former is represented by a set of key-value attributes that summarize page information, while the latter contains just simple text.

Considering the structured piece of the page, its key-value pairs are arranged according to a meta-template, called Infobox [13, 14], which provides a convenient way to get page's quick access information. An example of this structure can be found in figure 1. However, there are some issues with this structured data. First, since a Infobox is a meta-template, each category of pages can integrate a different final structure, producing a large variety of templates. This makes parsing Infobox quite complex. Secondly, the integration of Infobox in Wikipedia pages is still quite low, so that it is insufficient to rely only on those information to represent a page.

As a result, it has been decided to use a shallow approach in detecting Infobox presence. Employing a parser with an ad-hoc grammar enables to find whether a page contains an Infobox or not. In particular, it checks that the infobox is positioned as the first element of the page. This method is faster and more suitable for the requirements of this project, compared to using recursive regular expression [4]. In case of positive result, structured data are processed to extract the key-value pairs. Otherwise the page is considered as pure unstructured text. The final result is a list, that might be empty, which contains those tokens, one after the other:

$$[key1, value1, key2, value2, \dots, keyN, valueN]$$

Regarding page's unstructured text, the most interesting piece of information that can be retrieved are its links. Indeed, they represent pages connections and are fundamental to express how a page has evolved across the time. For example, the number of links and their frequency throughout the text (different link instances that all points to the same page) can be used as possible metrics to describe how a page's content might have mutated. However, those metrics does not take into account local changes happened within a page, such as whether links have been moved into different position around the page. For this reason, the concept of *link context* has been introduced. The idea is to consider both links and their surrounding words, instead of just taking links by themselves. In this way, it is possible to capture the notion of position in the page and better distinguish one page's version from another. To have enough space for creating the context, the number of surrounding words has been set to 10 before and after each link. Thus, to extract links context, the links have been first separated by means of a regular expression and a char sequence that is very unlikely to find in English texts (éèé) and then processed to retrieve the surrounding words.

"... because of lack of interest or concerns about the expense of the project. The United Kingdom withdrew from the **[[International Space Station programme|preliminary agreement]]** because of concerns about the expense of the project. Portugal, Luxembourg, Greece, the Czech Republic, Romania and Poland joined ESA after the agreement had been signed. ..."

$$\Downarrow$$

```
[expense, of, the, project., The,
United, Kingdom, withdrew, from, the,
[[International Space Station programme
| preliminary agreement]], because, of,
concerns, about, the, expense, of, the,
project., Portugal,]
```

Figure 2: Example of link context extraction

The final result of computing this procedure over a link is a list containing all the selected words and the wrapped link. An example of it can be seen in figure 2, where the words inside the double square brackets represent the link, while the blue words are the ones that compose the link context.

After all the previous elaboration, before proceeding to next step, it was necessary to specifically arrange all these gathered information into a specific schema in order to later process them. For this reason, the list of link contexts has been flattened and then, the resulting list has been concatenated to the one containing the extracted Infobox structure. Consequently, each Wikipedia page content is now represented by a single vector of meaningful strings. This list will be useful for the computation in the next module.

Besides, it is useful to keep track of which are the pages each page is pointing at, so that future computation can be restricted to only those ones. Therefore, for each page its links, contained in both the structured and unstructured sections, are analyzed, returning a list of page titles that represent the pages which are neighbours to the considered one. The final outcome of this stage is hence structured according to the below schema:

$$< title, timestamp, page\_representation, page\_neighbours >$$

3.3 Feature Extraction

Now that a smaller yet meaningful page representation has been reached, it is possible to transform it into a vectorial representation. This step is needed to later compute the similarity between Wikipedia pages. To achieve this aim, it has been chosen to use a distributed vector representation of words or phrases, with their weighs based on the context where they appear rather than just checking how much terms are related, such as using TF-IDF. Indeed, while the latter focus on providing a term relevance, the former it able to exploit the semantic of the text, providing a better representation of Wikipedia pages. Thus, this concept is quite relevant for the project, since in this way it possible to retain and exploit the information recovered by links context.

To perform this transformation, it has been decided to employ the Word2Vec (*W2V*) model [8], which takes advantage of machine learning techniques to compute words embeddings (vectorial representation). In the Spark application it has been deployed the *W2V* model provided by Spark ML library, which implements the *Skip-Gram* version.

During the development of this stage, two variants were designed to come out with a final transformation process. The first one consisted in training the model on the given data, that were the currently selected Wikipedia pages. This choice has brought to a straightforward implementation due to the easiness of the library and the fact that it was based only on the input data. However, similarly to the data loading section, to perform such training operation all the pages have to be loaded into the main memory and this is unlikely working for very large datasets. Consequently, it has been pondered to use a pre-trained model, still on Wikipedia pages. The selected pre-trained model is the outcome of a recent work [6], which comes at handy for solving this module task, providing a model for English words made by vectors of 300 features each.

Despite the simplicity of the idea, it was necessary to perform some tricks in order to be able to load the model into the Spark application and actually use it. Indeed, the given pre-trained model was saved as plain text, whereas Spark Word2Vec library was expecting to read a model saved as a set of *parquet* files [10], with some additional metadata. For this reason, it was necessary to convert the plain text model into parquet files and then add metadata cloned from a sample W2V model previously saved. In this way it was possible to create a loadable W2V model for Spark application. Still, it was not possible to utterly load the complete model, since it did not fit into neither cluster machines nor available computers' memory (see section 6 for more information about the hardware). As result, it was necessary to load only a small portion of the model (about 25% of the total pre-trained vectors), leading to poorer results than using the full model to compute Wikipedia pages' vector. To further improve application performances, the W2V model is loaded from the Spark application only once and then is reused across different preprocessing input.

Finally, before saving the preprocessing results, the norm of each features vector has been computed, so that it saves execution time for the next phase (allow disk usage to save later computation time). Therefore the final outcome is saved on the disk, partitioned by the title page with the following schema:

$$< title,\ timestamp,\ features\_vector,\ norm,\ page\_neighbours >$$

```
foreach page p of stored pages do
    currentPage ← load in memory page p;
    links ← getLinks(currentPage);
    neighbours ← foreach page l of links do
        load in memory page l;
    end
    pairs ← join(currentPage, neighbours);
    foreach row r of pairs do
        cosineSimilarity(r.f1, r.f2);
    end
    write results on disk;
end
```

Algorithm 1: Similarity Computation

For further details regarding disk storage, please refer to section 4.

3.4 Similarity

In this project section the similarity between pages is computed. This is a really memory-intensive task because it first requires to compute all the possible combination between pages and their versions. This generate a huge amount of pairs that can not fit all in the memory. Therefore a different approach has to be taken, since it is not possible to work directly on the whole dataset and Spark may not automatize the data offloading (due to its preference towards in-memory computation).

A general description of the similarity algorithm designed to solve this task can be observed in algorithm 1. The main idea is, instead of having all the pages loaded into the memory, to load only the pages required for creating the pairs between selected page and the ones that are linked by it. To know which pages have to be loaded, the application exploits neighbours'page information previously saved during the preprocessing phase. Once needed pages are loaded, it is then possible to compute the set of all possible pairs between those pages. Besides, in order to efficiently compute those pairs, some restrictions have been applied to the join task. In this manner it is possible to avoid calculating similarity between useless pairs, such as different versions of the same page or between pages that have a later timestamp than the one currently selected (can not connect a version that exists in the future from the past).

When all the pairs are computed, it is possible to calculate the similarity between Wikipedia pages. For this task it has been decided to use the *Cosine Similarity*, which exploits the distributed representation of pages as features vector. The Cosine Similarity between two pages $p1$ and $p2$ is therefore computed as follow:

$$similarity(p1,\ p2) = cos(\theta) = \frac{F1 * F2}{||F1|| * ||F2||}$$

where $F1$ and $F2$ represents the features vectors of $p1$ and $p2$ respectively while θ represents the angle between the two features vectors. Another reason to use the Cosine Similarity is its ability to retain the notion of position within the features. In this way, pages with similar content, but that it is located in different position will probably get a lower yet more accurate similarity than using other metrics. After that the Cosine Similarity has been computed, the results are saved back to the disk according to the following schema:

$$< title1,\ timestamp1,\ title2,\ timestamp2,\ similarity >$$

With this simple schema, the preprocessed data can be now easily employed to create a graph that represent a portion of Wikipedia pages.

3.5 Data Filtering

Before concluding the processing phase, it has been introduced a further level meant to reduce the number of rows that are finally returned. The reason for performing this kind of pruning is the fact that the visualization tool, that will be described in section 5, does not need to show all the possible version of a connection between two pages, but just the latest according to the date provided in input by the user.

Now that the processing procedure has been described, it is possible to observe the whole pipeline elaboration in figure 3.

4 DATA STORAGE

In this section it is explained both how and where input data are stored for being processed and where outcomes are saved. Then it will be shown what category of databases can be used to store processed data in a natural way and provide a simple interface for the visualization tool.

For what concerns input data, intermediate results and final outcomes, they are all stored on a HDFS (Hadoop Distributed File System), a component of the Hadoop framework [7]. In this way, data replication is guaranteed by the system and it allows the data to be shared across the cluster. Regarding the output format, both the intermediate results 3.3, obtained after having computed the features vectors, and the output of the similarity phase 3.4 are saved as `parquet` files [10]. The main reason that drives this choice is the fact that parquet files consume less space on the disk when stored and they encapsulate the schema of the data, so it is convenient to be later read for other elaboration. On the other hand, the final output, resulting from the data filtering process is saved directly as `csv` (Comma Separated Values) file, since this format is a lot easier and supported to import into other data stores.

Afterwards, when the processing phase is completed and all the data are safely stored on the HDFS, it is possible to import those data in other data stores better suited for retrieving data from applications. Since the stored data represents a portion of Wikipedia graph, it is straightforward to transfer those information into a graph database, such as Neo4j [9]. The advantage of using this kind of database, which natively integrate the notion of graph, is the fact that queries to retrieve data organized as a graph will be easier to be expressed and faster than using a traditional data store such as RDBMS. In order to load preprocessed data into the graph database, a small script for NodeJS has been written. There are two version of this script, depending on whether the data are stored on the HDFS or on the local filesystem. However, the former can not be used because Neo4j does not currently support importing files from HDFS, but only from local files or through *http* protocol. Therefore, at the moment data must be transfered from the HDFS to the local system where Neo4j resides or on a web server that will provide the access to them.

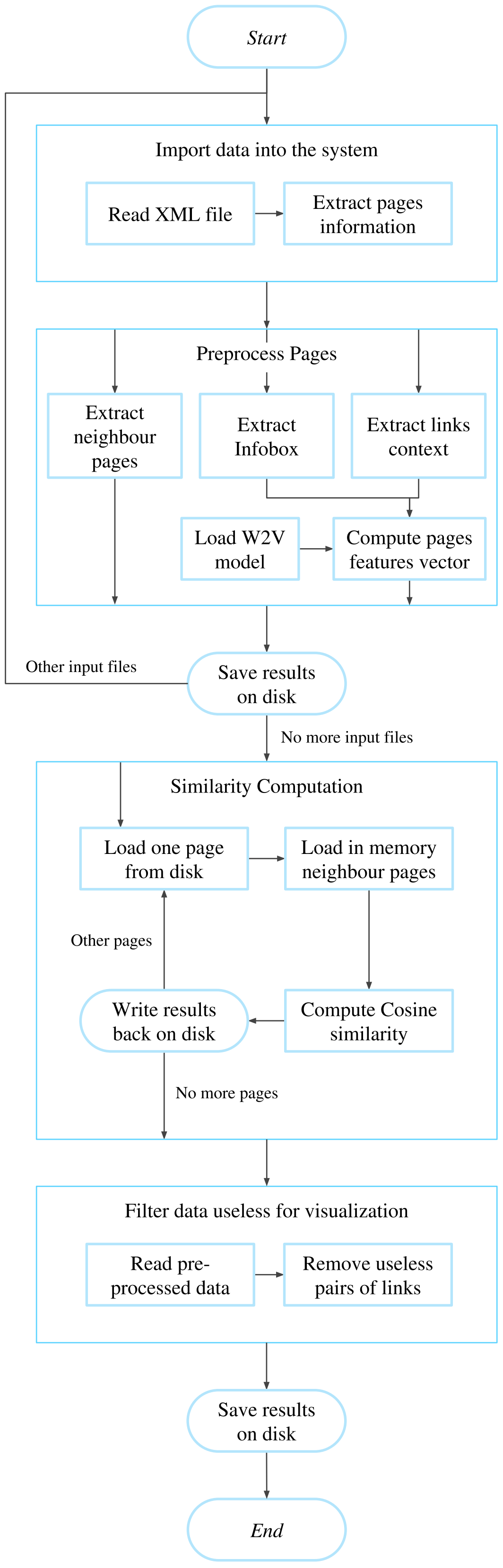

Figure 3: Preprocessing Pipeline

The scripts operates by first retrieving the list of all the csv files that contains the relevant information and then loading the data into the graph database using the `load` method of Neo4j. The below schema represents how data are saved within the graph database. It has two main components and it was designed to both easily represent Wikipedia pages and links and write the data retrieval queries:

- **Page**(*title*: String)
- **Is_Linked_To**(*ts_from*: Timestamp, *ts_to*: Timestamp, *similarity*: Double)

5 WIKIPEDIA GRAPH VISUALIZATION

This part of the project is where the data visualization tool has been developed. It is a single page web application that relies on a NodeJS back-end to extract and process the data from the graph database. The back-end is composed by three components:

- the application initializer: it is in charge to start the application and provide the endpoints for accessing it
- the application controller: it is responsible for managing requests that arrive from the endpoints and the communication between the client and the data manager
- the application *data manager*: it is in charge of retrieving requested data from the graph database and format them so that the client application can employ them to visualize the graph

Regarding the front-end side, a simple interface has been provided to users to navigate across the graph visualization. Indeed a slider has been placed to control which moment in the time should the application show. Moving the slider toward left or right will update the displayed data towards the past or the future respectively. With this simple method is then possible to analyze how the Wikipedia graph has evolved across the time, such as the number of nodes, edges and the weight on them that represent the similarity between the two pages.

The underlying framework which power the visualization is the JS Cytoscape library [2, 5]. It provides a simple interface that easily let create nodes and edges of the graph from JSON data. The final result can be seen in figure 4 and 5 where they show the application displaying a portion of the Wikipedia graph in two different period of time. In addition to the simplicity, the Cytoscape library enables users to interact with the graph, so that it is possible to zoom in or out, select nodes and move them. For example, clicking on an edge is possible to see the value of similarity between two linked pages. This value ranges between 0 and 1 and the greater is the value, the similar are the pages. In order to immediately visualize the similarity between two pages it has been decided to draw the edges with the two following characteristics:

- the edge width is proportional to the similarity between the two pages
- the edge color ranges from a dark shade for low values of similarity to a bright shade of light blue for really similar pages.

The main reasons for choosing this library rather then other alternatives are:

- it is focused on drawing graphs only
- it is highly customizable
- it has built-in interaction functions
- it works with the HTML `canvas` element rather than creating as `svg` elements.

The greatest advantage of the last point is the low consumption of the memory. Indeed, while canvas elements requires just a raster image that is redrawn when users interact with it, svg-based graphs need to create a svg element for each node and edge of the draw, which can require a non-negligible amount of memory in case of drawing big graphs, such a portion of Wikipedia graph. As a result of employing the former approach it is possible to load hundred of nodes and edges without crashing the page for a lack of memory. Nonetheless, it has been set a custimizable limit on the number of displayed nodes for the clearness of the visualization.

6 TESTS

Finally, after having developed all the previous modules, some tests have been carried out to check the processing pipeline's performances. For testing the process, it has been used both a single computer and a small cluster composed by eight machines (2 CPUs - 6 GB effective memory per machine). Unfortunately, at the moment, are available only the results for the single machine.

The specifications of the single machine set up comprise a computer with 8 CPUs and 8 GB of main memory. Moreover, it runs with other two cheap machine a HDFS cluster for storing processed

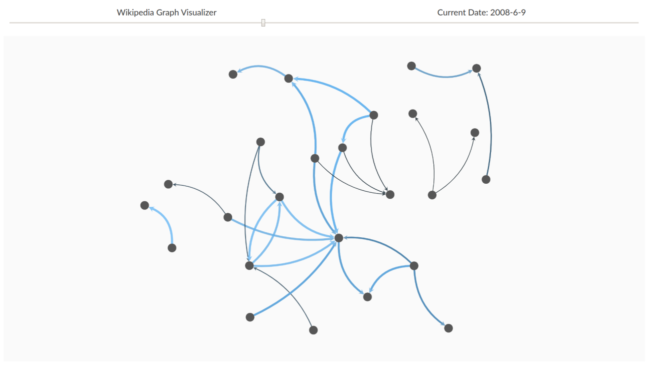

Figure 4: Example of displayed graph on a first date

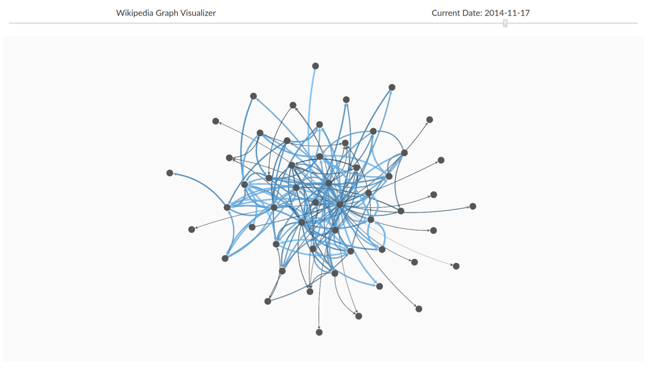

Figure 5: Example of displayed graph on a second date

Table 1: Testing results of the processing pipeline on single machine

Title	#Input_files	Input Size	#Pages	#Links	#Final_links	Time
ngc	1	2.6 MB	5	18274	127	165s (0h 02m 45s)
time	1	354.5 MB	26	30661741	70463	1632s (0h 27m 12s)
music	1	1.0 GB	67	93752787	117033	4783s (1h 19m 43s)
m_files	13	629.4 MB	13	17192146	19060	3746s (1h 02m 26s)
m_topics	9	2.4 GB	138	406907272	657346	6h+

Note: *m_topics* has been computed on a different machine with more main memory (4 CPUs - 12 GB of RAM).

data. The computers are connected through a fast-ethernet link (100 Mbps) via a switch.

Then, different inputs have been provided to the processing pipeline to test its capabilities, varying the the size of the input, the number of pages and the number of files on which the input was split. The results can be observed in table 1.

One noteworthy point that it has been found while testing the processing module is that the bigger were the input file, the more parallelism can be achieved. However, due to Spark mode of operation (in-memory first approach) and hardware limitations, it was not possible to load huge files since there were not enough main memory. For this reason current implementation is able to read multiple relative small files to achieve a final larger input size, even though some adjustment have to be made. Another minor issue that should be updated, is how to output the final processing outcome, since the application currently generate a large number of tiny files to be able to cope with all the previous produced data. This behaviour makes HDFS inefficient, since its best performances are achieved with files of a large size. One manner to improve the final results could be to group them with a further level of processing, after all the data filtering has been performed.

7 CONCLUSIONS

In conclusion this project provides an application, that works according to the MapReduce paradigm, able to convert raw Wikipedia pages' text into a graph representation composed by node as pages and edges as links between pages. On top of that, the similarity between the two connected pages has been computed and exposed through edges' label. In addition, a graph database has been exploited to offer a tool to visualize the processed portion of Wikipedia graph and analyze its evolution.

In the end, with this project I have started learning some of the knowledge that a data scientist should own, such as how to deal with Big Data, how to write a working Spark application and how to set up and manage a simple HDFS cluster with Spark on top of it. Besides, I have also learnt the basics of graph databases and how to exploit them to manage data that can be represented as a graph. Therefore I consider this project as a valuable experience for my future career.

REFERENCES

[1] Apache Spark [n. d.]. A unified analytics engine for large-scale data processing. Retrieved July, 2018 from http://spark.apache.org/
[2] Cytoscape JS library [n. d.]. Retrieved July, 2018 from http://js.cytoscape.org/
[3] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492
[4] Amira Abd El-atey, Sherif El-etriby, and Arabi S. kishk. 2012. Article: Semantic Data Extraction from Infobox Wikipedia Template. *International Journal of Computer Applications* 40, 17 (February 2012), 18–23. Full text available.
[5] Max Franz, Christian T. Lopes, Gerardo Huck, Yue Dong, Onur Sumer, and Gary D. Bader. 2016. Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics* 32, 2 (2016), 309–311. https://doi.org/10.1093/bioinformatics/btv557
[6] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. 2018. Learning Word Vectors for 157 Languages. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018).*
[7] Hadoop Framework [n. d.]. Retrieved July, 2018 from http://hadoop.apache.org/
[8] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013). arXiv:1301.3781 http://arxiv.org/abs/1301.3781
[9] Neo4j Graph Database [n. d.]. Retrieved July, 2018 from https://neo4j.com/
[10] Parquet File Documentation [n. d.]. Retrieved July, 2018 from https://parquet.apache.org/documentation/latest/
[11] Scala Programming Language [n. d.]. Retrieved July, 2018 from https://www.scala-lang.org/
[12] Wikipedia Export Tool [n. d.]. Retrieved July, 2018 from https://en.wikipedia.org/wiki/Special:Export
[13] Wikipedia Infobox description [n. d.]. Retrieved July, 2018 from https://en.wikipedia.org/wiki/Help:Infobox
[14] Wikipedia Infobox Template Documentation [n. d.]. Retrieved July, 2018 from https://en.wikipedia.org/wiki/Template:Infobox
[15] Wikipedia Page XML Schema [n. d.]. Retrieved July, 2018 from https://www.mediawiki.org/xml/export-0.10.xsd
[16] XML Data Source for Apache Spark [n. d.]. Retrieved July, 2018 from https://github.com/databricks/spark-xml