Evaluating the evolution of Wikipedia's navigability

(COM-416) Project in communication systems II

Concetto Emanuele Bugliarello[1]

[1] *Communication Systems, Master semester 4*

Prof. Dr. Robert West
Data Science Lab (dlab)
École polytechnique fédérale de Lausanne (EPFL)

June 9, 2017

People are regularly faced with tasks that can be seen as navigating information spaces in which they only get access to local and neighboring information, while lacking a global view of the network. The aim of this project is to study which properties of a network have the largest impact on navigability. We do not achieve this goal during the semester because most of the time has been spent to properly transform the original data to provide a final dataset that can then be used in different scenarios. So, we meticulously describe here all the processing phases we apply in order to produce such final dataset. We also show some early results regarding the evolution of the properties underlying the Wikipedia network over time.

1 Introduction

People navigate information networks in their everyday life in order to acquire the knowledge they seek. In fact, information is usually spread over a different number of interconnected sources and it is then fundamental to being able to efficiently navigate these networks. Example networks are citation networks to find the work done in a given field, or just browsing the Web.

This type of navigation can be mapped to a search in a graph, where nodes represent pieces of knowledge and edges indicate existing connections between two different nodes. However, we do not usually have a full view of the underlying networks. Hence, if we want to find the shortest path connecting two different resources, we cannot rely on well-known results such as Dijkstra's algorithm. Furthermore, people often do not know which nodes exist in the network and they could easily get lost.

It is then of primary importance to understand how people navigate across the contents available online and provide insights into how the task of wayfinding in information networks can be made easier. The aim of this project is indeed to evaluate how different decentralized search algorithms behave on different snapshots of a giant network and then infer a model to help people in exploratory searches, as we usually gather information from different sources which are commonly not known in advance.

A distinct information network is the one underlying Wikipedia [15], the largest free online encyclopedia, which not only does it store human knowledge, but also provides connections among single pieces of information through hyperlinks.

We choose Wikipedia as our validation domain not only because it contains a rich knowledge database, but also because of the existence of agents trained on humans performing a game, $Wikispeedia$, in which the goal is to find the shortest path from a given start to a given target article by clicking only on the available links in each page.

We have available the entire history of revisions of Wikipedia and so we can analyze how the properties of the network evolved over time and investigate what types of modifications have the largest impact on navigability. For instance, while having more links could reduce the average shortest path as more shortcuts would be available, this could also increase the risk of getting lost during the search as the user is flooded with a large number of possible pages she could visit next.

We do not reach the navigability study in this project as we spent all the time to properly processing the data in order to have a final dataset that can be used in future projects as well. Nevertheless, we introduce here some early results regarding the evolution of the properties of the Wikipedia network over the years.

The remainder of the report is organized as follows.

We first introduce the closest studies to our work, and we give an overview of the machines that we use to crunch the data. We then move to the core part of the project: processing the massive Wikipedia revisions dataset. After that, we report on some early results that we have obtained and discuss some of the challenges related to this work. Finally, we present directions for future work and conclude this report. In addition, we report on some extremely useful knowledge that we have acquired during the semester in the appendices.

2 Related Work

The work related to our project can be separated in three parts: Data management systems that enable users to store large volumes of historical graph data, graph properties evolution over time and decentralized search in networks.

Several temporal data indexing techniques have been proposed for relational databases. Salzberg and Tsotras [8] compared different indexing techniques for supporting efficient access to temporal data. Here, they examined two extreme approaches to supporting snapshot retrieval queries, called the "copy" and the "log" approaches. The "copy" approach stores a copy of the entire database for each time at least one change occurs. These copies are indexed by time. Clearly, the major disadvantage of the "copy" approach is with the space requirements. To reduce the space requirements, the "log" approach stores only the changes that occur in the database, thus ensuring requesting only the minimal space. However, this approach incurs large reconstruction costs. Combinations of these two straightforward approaches are possible and are referred as "copy+log" approaches.

Many of the properties of interests in studies on graph structures are based on two parameters: the nodes' degrees and the distances between pairs of nodes (as measured by shortest-path length). Networks evolve over time by the addition and the deletion of nodes and edges, and most models of network evolution capture two patterns in the growth pattern: constant average degree and slowly growing diameter. However, Leskovec, Kleinberg and Faloutsos [6] showed that different real networks from several domains experience an increasing average degree and a decreasing effective diameter as the networks grow.

Decentralized search in networks considers a scenario where a starting node s is trying to send a message to a given target node t by forwarding the message to one of its neighbors, and this process continues until t is eventually reached. These studies can be traced back to Milgram's small-world experiment [7] and the algorithmic problem of decentralized searches in networks [5]. West and Leskovec [10] performed a large-scale study of user navigation behavior. The authors analyzed a collection of clickstreams of users who were playing a navigation game (Wikispeedia [11]) in a network of links between the concepts of Wikipedia. In their work, they found that human navigation behavior, while mostly very efficient, differs from shortest paths. For example, users typically navigate through high-degree hubs in the early phase and then apply content similarity as a criteria for finding the destination node.

In subsequent work [9], the authors analyzed a number of decentralized search algorithms using various distance functions and benchmarked them against their human click corpus. The authors also found that even simple search strategies, such as utilizing node degrees, outperform human information seeking. They introduce different agents, both heuristic-based and machine learning-based. In particular, our first step towards navigability studies over time would have started with a heuristic, similarity-based agent using a similarity measure not investigated in this paper.

3 System Overview

Before diving into the core of the project, we briefly introduce the three architectures we worked on during the semester and that will be referenced in the next section. More information and useful pointers are then provided in Appendix A.

We refer to them as `laptop`, `dlab-server` (or simply `server`) and `cluster`, and their specifications are reported in Table 1.

The laptop is only used to develop the algorithms and test them on small samples of the datasets. The server is mainly used to run tasks that are not very computationally expensive and, in particular, to generate data structures that are used during the main processing steps. Once the data is processed, the analysis of the resulting graphs is carried out on the server. Finally, the cluster is where the big data processing happens. It consists of seven nodes for a total of 266 VCores and $1.63\ TB$ of RAM. Table 2 shows, instead, the storage that is available on the remote systems.

In particular, on the server, there is no backup for data in the home directory and on `/scratch/`. Moreover, on the cluster, there is a replica 3 for the data on the HDFS user directory `/user/gaspar/` and so only $\approx 1.6\ TB$ are actually available.

Machine	Directory	Storage
dlab-server	/scratch/	3.6 TB
dlab-server	/dlabdata1/ (NFS)	1 TB
cluster	/home/gaspar/	31 GB
cluster	/user/gaspar/ (HDFS)	5 TB

Table 2: *Available storage on the remote machines.*

Name	Address	CPUs	RAM	Spark
laptop	localhost	4 processors (Intel Core i5-5200U)	8 GB	1.6.3
dlab-server	iccluster111.iccluster.epfl.ch	48 processors (Intel Xeon E5-2680 v3)	252 GB	1.6.3
cluster	hadoop.iccluster.epfl.ch	7 nodes, dual CPUs (Intel Xeon E5-2680 v3)	1.63 TB	1.6.0

Table 1: *Specifications of the machines employed throughout the project.*

4 Data, Data & Data

This project is on big data analytics: so, let's start by talking about the data that will be used for our analysis. The code used for this section can be found in the GitHub repository [1] under the `data_transformations` directory.

All the datasets introduced in this section are stored as Apache Parquet [3] files, compressed with Snappy [4]. Apache Parquet is an open-source column-oriented data store of the Apache Hadoop ecosystem. It provides efficient data compression and encoding schemes, where compression is performed column by column (hence enabling different encoding schemes to be used for text and integer data).
Snappy is a fast data compression and decompression library written by Google based on LZ77. It does not aim for maximum compression but rather for very high speeds and reasonable compression.

4.1 Original data

We start by describing the Wikipedia dataset available at EPFL comprehending almost 16 years of article revisions (from January 2001 to October 2016). More precisely, the timestamp of the earliest record is `1/16/2001 21:08:33`, while the one of the latest is `10/1/2016 21:39:16`.
The dataset thus contains all the revisions in Wikipedia until October 1, 2016 since Wikipedia was launched on January 15, 2001.

The dataset is available on the cluster's Hadoop Distributed File System (HDFS) under `/datasets/wikipedia/en-oct-2016/`. It is distributed in $100,000$ parts, from `part-r-00000` to `part-r-99999` amd it has $685,387,342$ entries, for a total of $5.9\ TB$.

It comes as a Spark DataFrame (defined in Spark SQL: Spark's interface for working with structured and semistructured data). The corresponding schema is shown in Table 3 along with an example.
We now provide a brief description of most of the fields in this schema (the ones used in this project).

Schema	Example
`redirect`: *UTF-8*	null
`ns`: *UTF-8*	4
`title`: *UTF-8*	Wikipedia:Bot req...
`id`: *UTF-8*	912023
`sha1`: *UTF-8*	4qpzrtsom7ls28291...
`revision_id`: *UTF-8*	336934282
`parentid`: *UTF-8*	336930575
`model`: *UTF-8*	wikitext
`text`: *UTF-8*	{{Wikipedia progr...
`text_xmlspace`: *UTF-8*	preserve
`ip`: *UTF-8*	null
`timestamp`: *int96*	2010-01-10 05:06:...

Table 3: *Schema and example of the original DataFrame.*

- `redirect`: the article this article redirects to. It can either be `null`[1] (if an article is not redirecting to another one) or a string including the name of the article it redirects to, such as `{@title=Tautology}`;
- `ns`: the Wikipedia namespace of this article. A Wikipedia namespace is a set of Wikipedia pages whose names begin with a particular reserved word recognized by the MediaWiki software (followed by a colon). The list of the 34 current namespaces can be found at [17];
- `title`: the title of this article;
- `id`: the id of this article;
- `revision_id`: the id of this revision (unique);
- `parentid`: the revision id of the version modified by this entry;
- `text`: string containing the text of this article as wiki markup [13]. You can see the format of this text by clicking on the "Edit source" tab in any Wikipedia article (at the left of the search box);
- `ip`: IP address of the person who created this revision;
- `timestamp`: date and time of this revision. Spark stores Timestamp in Parquet files as INT96 to avoid precision lost of the nanoseconds field. In Python, it is an instance of a `datetime.datetime` object.

[1]Equivalent to `None` in Python.

4.2 Extracting hyperlinks from text

Recall that we want to study the evolution of Wikipedia's underlying graph. A graph consists of vertices and edges. In this case, articles constitute the vertices of the graph and hyperlinks in the text of a given article define the outgoing edges from this vertex to the corresponding articles' vertices.
Hence, in our study we only need hyperlinks rather

than the entire content of an article. This has a main effect: tremendously reducing the size of the data.

In Wikipedia's markup language, hyperlinks to other Wikipedia articles in the text of an article are created by putting double square brackets around the title of the Wikipedia article to link to. Thus, for example, `[[Texas]]` is a link to the Wikipedia page with title Texas, i.e. a link to `https://en.wikipedia.org/wiki/Texas`. Optionally, it is possible to use a vertical bar (`|`) to customize how the link is rendered. For example, `[[Texas|Lone Star State]]` would produce a link that is displayed as "Lone Star State" but in fact links to Texas. For our aim, we just extract the text before the vertical bar.

Apart from these standard links, hatnotes [16] have been introduced, presumably, in September 2005 [2]. Hatnotes are short sentences placed at the top of an article or a section of an article with the aim of helping readers locate a different article they might be seeking. Since we want to extract a view of Wikipedia that resembles as close as possible the experience a reader is offered, we extract links contained into several possible hatnotes. Namely, we extract links contained in the following templates:

- "*Main article: . . .*";
- "*For more details on . . . , see . . .*";
- "*See also . . .*";
- "*Further information: . . .*";
- "*This page is about . . . For other uses . . .*";
- "*This page is about . . . It is not to be confused with . . .*";
- "*For . . . , see . . .*";
- "*For other uses, see . . .*";
- "*. . . redirects here. For other uses, see . . .*";
- "*Not to be confused with . . .*";
- "*. . . Not to be confused with . . .*".

Each of these templates is different and so we wrote a regular expression (regex) and further parsing to extract the hyperlinks in each of them. You can have a look at the results of applying these regular expressions at [2], where we merged all the regular expressions into one by means of vertical lines (alternation operator) to show all the regexs in just one web page.

The parsing applied to these regular expressions depends on which hatnote we find. Some of them are easier to parse, for instance by just splitting the retrieved content whenever a verical line (`|`) is found, while others require more effort. For instance, let's consider the case of "about" hatnotes:
"This page is about ... For other uses ...".

- The main template is: `{{About|USE1}}`, which produces the following hatnote:
  This page is about USE1. For other uses, see PAGETITLE (disambiguation),
  where $PAGETITLE$ is the title of the article the hatnote is into, and the text in blue indicates a hyperlink to the page with the blue string as title.
- There are many other possibilities, but a general pattern can be identified as the following: `{{About|USE1|USE2|PAGE2{{!}}PAGE2TITLE|and|PAGE3#SUBSECTION|other uses}}`, producing:
  This page is about USE1. For USE2, see PAGE2TITLE and PAGE3. For other uses, see PAGETITLE (disambiguation),
  where the `{{!}}` magic word is used to give the link a different title, and *PAGE3* is a hyperlink to PAGE3's *SUBSECTION* section.

So, we see that if only one "use" is present or "other uses" is specified at the end of the template, we need to generate a link to a page having as title the article's title followed by " *(disambiguation)*". Finally, we only need to extract strings referring to other articles (`PAGE*`) and not strings explaining the meaning of the corresponding pages (`USE*`).

Our first attempt to extract the hyperlinks from the text used all the regular expressions combined into a single one by concatenating all of them into a single string, separating each others with a vertical line. The idea behind this approach was to have each revision scanned only once and hence improve the performance (considering the large corpus of articles that we have). However, after the parsing, some articles went missing; forcing us to run each regex separately. Surprisingly, the speed achieved on the cluster by running each regex separately is comparable with the one achieved when combining them. Actually, for a batch of reviews we were using for debugging, running each regex separately was faster than when regexs were concatenated into one (30 vs. 37 minutes). On the other hand, running the combined regex was 3 minutes faster than running each one separately on the laptop using a 1-GB sample (7 vs. 10 minutes).

Before extracting the hyperlinks from an article, its text is pre-processed as follows:

1. \n are replaced by a single space;
2. all the image maps [12] are removed from the text: we are not interested in images and the links shown in these pictures are encoded with the same format as links in the text.

After extracting the links, we post-process them as follows:

1. discard links to images or files, e.g., `[[File:Example.jpg|thumbnail]]`;

[2]The first revision of [16] is on September 14, 2005.

Schema	**Example**
`id`: *UTF-8*	912023
`title`: *UTF-8*	Wikipedia:Bot requests
`timestamp`: *int96*	2007-01-26 16:54:18
`standard_outlinks`: *array(array(string))*	[[User:seans potato business, 1], [Skynet, 1], ...]
`hatnotes_outlinks`: *array(array(string))*	[]
`length`: *int*	29155
`redirect`: *UTF-8*	null
`revision_id`: *UTF-8*	103395195
`ip`: *UTF-8*	null
`parentid`: *UTF-8*	103389117
`ns`: *UTF-8*	4

Table 4: *Schema and example of the Links DataFrame.*

2. discard links pointing to a section inside the same article (self-loops), i.e., links starting with a pound sign (#);
3. remove the section from an article's title if the link is pointing to a specific section of the article. That is, if we have `[[Target page#Target section]]`, then we only keep *Target page*;

4.3 General Transformations

We firstly describe the processing steps applied to the entire dataset. These transformations are independent of the scope of this project and can be used in other scenarios involving this dataset.

4.3.1 Links DataFrame

The first step in our pipeline basically consists of extracting the links from the text of each revision (as described in Section 4.2). While doing so:

- in order to take into account how many times a hyperlink to a certain article is present in the text of an article, we keep a counter of its frequency;
- we store titles and frequencies for the standard hyperlinks in an article (between double square brackets) and titles and frequencies for hatnotes separately. By having them split, we can then study whether the latter actually help navigating Wikipedia;
- we extract the titles of the articles in the `redirect` column: for instance, `{@title=Tautology}` simply becomes `Tautology`;
- we add a new column, denoted `length`, with the number of characters in the text of each revision.

The resulting schema along with an example is shown in Table 4.

4.3.2 Resolved Links DataFrame

Not all the pages in Wikipedia are articles. Among them, $redirects$ play an important role. A redirect is a page which automatically sends visitors to another page, usually an article or section of an article [18]. For instance, if you type "UK" in the search box or in the address bar, you will be taken to the article "United Kingdom", and there will be a special note at the top of the page saying "(Redirected from UK)". A redirect page contains special wikitext which defines it as a redirect page and specifies the target page (or even a section within the target article).

Even though we have the wikitext of each revision in our Links DataFrame, we already have the redirect information readily available. In fact, after the previous step, the `redirect` column either contains `null` or the title of the article a given revision is pointing to. Hence, the only thing we need to do is substituting each occurrence of links to redirect pages with the titles of the articles those pages divert to. However, a page might change target over time and this makes the replacing task intensive: For each revision in the dataframe, we need to determine what is its view of Wikipedia in terms of redirects.

Our approach processes the entire dataset in semester-long chunks, each starting on January 1 or July 1, for each year between 2001 and 2016. For each of these chunks, we retrieve all the entries in the dataframe which point to other articles (whose `redirect` column is not `null`). We then group these entries by title and create a dictionary in which, for each title, we have a list of `(target page, first timestamp)` pairs, where `first timestamp` refers to the first time a redirect page diverts to a given `target page` until the target is replaced by another one (therefore avoiding a new entry in the list for each revision if the target article is unchanged).

Before resolving all the links, we proceed with a normalization of all the titles as follows.

By looking at some of the extracted links, we noticed that some of them start with a slash (`/`). These are links to subpages [19]. The pages which these links point to are considered "subordinate" to the respective host pages. Moreover, the subpage feature has been disabled in the main namespace (0) of the English Wikipedia. Our dataset contains some of these subpages under revisions with title "`host page title/subpage title`". Thus, in our normalization step, we prepend the title of the host page to every link starting with a slash.

While going through the lists of links to find subpages, we also perform another normalization transformation in the titles (revision title, titles in the revision's links and title in the redirect column): We replace white spaces by underscores (_) and capitalize the first letter in the string.

After that, for each revision in the semester, we simply do a lookup in this dictionary for each article in its list of extracted hyperlinks and pick the target page whose first timestamp is the greatest among the ones that are earlier than the revision's one. Note that after substituting the redirecting articles with their target articles, a new multiplicity count is needed as, for instance, more pages could redirect to the same one.

The resulting schema is the same as in Table 4.

With this approach, we basically map a redirect page to the page specified in its `redirect` column.
This might still results into substituting a redirect page with another redirect if the target page is a redirect page too. To circumvent this potential problem, we tried to recursively lookup into the mapping dictionary until the target page was not among its keys. However, we ended up entering into an infinite loop already during the first semester of data. Hence, for the sake of simplicity, we only apply a single lookup for each link to a redirect page and keep in mind that some links point to redirect pages.
Other approaches could be devised in the future, such as iterating up to a fixed maximum number of times or making sure that the starting redirect page is not reached in successive lookups.

With the described approach, we compute a dictionary of redirects from scratch for each semester while later semesters simply extend the entries in the dictionaries of previous semesters. Hence, we could have just created a dictionary with all the entries for the entire history once and used it in all the batches. However, the main drawback of this approach is that each process has to store in memory this large dictionary even for the earliest periods, thus always requiring large amount of memory reserved on the cluster. In addition, loading the entire dictionary and distributing it to the nodes in the cluster takes as much time as creating it. With our approach, instead, we managed to tune the memory allocation for the different chunks so as to process the data with the minimal required memory leading to no failure. See Appendix B for these details.
Another approach could have been to store the dictionary at the end of each semester batch so as to make it available to the next one. With this procedure, for each semester, we would have had to load a dictionary at the driver, add entries in the new semester and at the end store it. However, loading such dictionaries is at least as slow as creating them anew.
Finally, a different method that we initially thought of was to create a DataFrame of redirects. That is, a DataFrame only containing entries whose value in the `redirect` column is not `null` and it is different from the redirecting value of the same page at the time of its previous revision (to avoid redundant information). Even though the resulting DataFrame would have been useful in general, we did not take this approach mostly for time constraints. Nevertheless, the code to generate it is available in the `deltas_utils` module.

4.3.3 Normalized Links DataFrame

This is the last step that we apply for the general transformations of the dataset.

While playing with the data, we never noticed values in the `id`, `revision_id`, `parentid` and `ns` columns not consisting of integers. So, we firstly perform an inspection to assess whether all these columns actually contain integer values only. We noticed that when converting a string column that does not consist of integer values (such as the `title` column), the resulting transformed entries are `null`. Hence, our inspection simply consists of comparing the number of null values in each of these columns before and after applying the integer cast. Since the number of `null` values is the same before and after the conversion for each of these columns, it is then possible to proceed representing them as integers.

After normalizing the columns in the dataset, we conclude our normalization phase with a final step: splitting the links in the standard and hatnotes lists in "reachable" and "unreachable".
To apply this transformation, we make use of a DataFrame of "first revisions" containing, for each normalized title in the history, the timestamp of its first revision. This DataFrame is useful in the splitting step in two ways: (i) it allows us to discard links pointing to non-existing pages (pages not among the titles in this data structure) due to, for example, typos; and (ii) it allows us to discard links to pages not existing at the moment the revision was created (they are colored in red in Wikipedia). These links might have been created afterwards but a reader could not navigate to them at the revision's time.
Hence, the links in the standard and hatnotes lists are split into two lists each: a list with all the reachable links and a list with all the unreachable links (for any of the two previous reasons).

Schema	Example
`id`: *int*	24657
`title`: *UTF-8*	Standard_Chinese
`timestamp`: *int96*	2016-09-17 02:58:55
`standard_outlinks`: *array(array(UTF-8))*	[[Compound_(linguistics), 1], ...]
`hatnotes_outlinks`: *array(array(UTF-8))*	[[Standard_Chinese_(disambiguation), 1]]
`standard_outlinks_failed`: *array(array(string))*	[[Wikt:mingzi, 1], [Wikt:goodbye, 2], ...]
`hatnotes_outlinks_failed`: *array(array(string))*	[]
`length`: *int*	49416
`redirect`: *UTF-8*	null
`revision_id`: *int*	739788881
`ip`: *UTF-8*	null
`parentid`: *int*	739788073
`ns`: *int*	0

Table 5: *Schema and example of the Normalized Links DataFrame.*

The schema of the so-created normalized DataFrame is shown in Table 5 along with an example.

4.4 Testing

Once the general processing is finished, we perform some tests to make sure that everything went well. With this goal in mind, we looked for and eventually found an article containing many corner cases.

In particular, while looking for such an article, we made sure it still has red links as of the date we performed the test. If this is not the case, then it would take much longer to manually find if any of the links in the text were red. In fact, if we browsed an old revision of a page on Wikipedia and the entries corresponding to its red links have been added to Wikipedia when the test is performed, then those links would show up blue. Having red links allows us to assert whether they have been placed in any of the `_failed` lists.

Moreover, we were also looking for an article containing non-ASCII characters to make sure they have been preserved along the pipeline and that we could print them without any errors.

The article matching these criteria is "Standard Chinese" (id 24657). Specifically, we take the revision 739788881, edited on September 17, 2016. The results of the test are shown in the `TestNormalizedRow` IPython notebook.

1. We firstly made sure that no link in the DataFrame had white spaces between the words composing the target pages.
2. Secondly, we checked that the red link was in one of the lists of failed links.
3. Afterwards, we manually clicked, on Wikipedia, many of the hyperlinks in the text of the article, and we collected 13 links redirecting to other pages. By inspecting these links, we could make sure that the resolving phase was successful. During our inspection, all of the found redirects were correctly resolved.
4. Finally, we printed links containing non-ASCII characters (Chinese characters) to make sure Unicode characters had been correctly preserved.

During the last phase, we also added some functions in the `deltas_utils` module to write and print entries of the DataFrame as in the plain text on Wikipedia (no Unicode-escaped characters). You can have a look at how they work on the `Print&StoreDataframeInUnicode` IPython notebook.

We also performed a general sanity check: we asserted that the total number of entries in the Normalized Links DataFrame is the same as the total number of entries in the original DataFrame.

4.5 Project-related Transformations

In our first plan, we decide to analyze the evolution of Wikipedia on a monthly basis.
Hence, after the general-purpose processing phases described above, we now present the processing steps applied to the Normalized Links DataFrame to build monthly snapshots of Wikipedia.

4.5.1 Monthly Links DataFrame

We start by creating a filtered DataFrame where there is at most one entry per month for each article. In particular, for each month, we keep the revision with the latest timestamp for any of the articles modified in that month. Moreover, we have decided to focus our study only on articles in the main namespace (0) at the beginning. So, as a first step, we only consider articles whose namespace is 0. Note that they still contain links to pages in other namespaces. We remove them in the next phase.

4.5.2 Monthly Edge Lists

The first step to generate a graph from the Monthly Links DataFrame consists of removing any information not related to a graph and storing the graph in an appropriate format. There exist many libraries to process graphs and one of the widely accepted representations to read a graph from a file is an edge list. An edge list file contains an edge per line as a source-target pair. In particular, we create a TSV file and we repeat an edge as many times as its frequency in the original article.

In order to ensure a wide usability, each edge is stored as a pair of integer values. In our case, we use the `id` of each article as its integer representation. Hence, we first create two dictionaries mapping ids to titles and vice versa.

Collisions & other issues.
Before continuing describing the edge lists creation, we open a small parenthesis to introduce an issue that we have faced while creating the dictionaries that map an id to its corresponding title and vice versa.
In creating these dictionaries, we firstly create a dictionary having ids as keys and titles as values. After that, we invert keys and values and create a dictionary mapping titles to ids. However, after performing this step we notice that the number of keys in the latter dictionary is smaller than the number of values in the former one. That is, multiple ids map to the same title. There are 123 titles that have each two ids associated. The vast majority consists of single-letter Unicode characters. When we capitalize each title, the lower case version of each of these letters becomes the same as the title of the page containing the originally capitalized letter. Even though most of the collisions are of this type, this is not a big issue. In fact, most of them redirect to their "normal" representation, both in their lowercase page and in the uppercase page. Nevertheless, we create a new TSV file mapping each id to its correct title: to do so, we use Wikipedia's query API [**wiki:query**]. We do so in order to ensure that the title to id dictionary has the correct mapping (and not a random one determined by the last processed title in each pair).
The remaining 18 collision pairs did not collide due to the normalization process. Instead, they collided because the title in the `title` column of the original DataFrame is wrong. For instance, id 5702430 is said to correspond to $Akalgarh,_I ndia$, while it actually refers to $Akalgarh,_L udhiana$. This is what happens in most cases. We checked some of these wrong mappings, and, actually, none of them had ever a revision; in other words, these ids have always been associated with the wrong title in the original DataFrame. As a result, any link pointing to any of these pages would be incorrectly marked as failed.
A few colliding titles have different sources of error. For instance, the two ids we have associated with `Sam Brooks` are: `13144660` and `50220117`. The former does not exist on Wikipedia: this could be the id of a duplicate article later removed, while the latter is the id for the page entitled `Sam Brooks (dramatist)`. What is interesting, though, is that there actually exists a page called `Sam Brooks` and it has a different id associated to it (`51959236`) which, however, is not among the ids in the original DataFrame. To maintain an internal consistency, we then map id `50220117` to `Sam Brooks` (instead of its true title `Sam Brooks (dramatist)`). On the other hand, titles like `Occitante` and `Ray Carver` each have one of their ids not in Wikipedia anymore and so we just map each of them to the right one. These examples show the types of collisions that we observe in mapping titles to ids. To solve this issues, two paths can be taken: (i) substitute wrong titles with the correct ones in all the DataFrames or (ii) ignore missing titles but ensure that each title is mapped to the correct id when there are collisions. We take the second approach for the sake of simplicity due to the small number of entries. We then create an updated version of the dictionary mapping titles to ids that ensures that colliding titles point to the correct ids. We do this only for namespace 0 since it is the one we use in the edge lists creation.

Going back to creating edge lists, we proceed as follows. We process each month of the Monthly Links Dataframe independently. Each month has at most one entry per article and for each article, we create two edge lists: one for the standard links and one for the hatnotes. These lists are only retrieved from the non-failing links. To map titles to ids, we use the updated dictionary described in the previous paragraph. By using a dictionary with entries in namespace 0 only, we can throw away links to articles not in the main namespace by just checking whether they have an entry in the dictionary or not (`O(1)`).
We then obtain edge lists that only contain vertices of articles in namespace 0. The result of this operation, for each month, is two folders (standard and hatnotes links) containing various text files constituting different parts of these lists (as usual with Spark). Once this process is finished, we aggregate the different parts inside each folder into a single file, and then append the content of these two files into a third one that thus contains all the edges of all the articles that have been modified in a month.

4.5.3 Monthly Snapshots Edge Lists

After the previous step, we obtained edge lists only consisting of edges whose sources have been modified in a given month. Instead, as a final result, we want each file for a given month, to represent the edge list for all the articles in Wikipedia; i.e., a snapshot of Wikipedia for that month. We achieve this with the transformation presented here.

We start from the first month in the dataset and incrementally build a dictionary where for each article id, we have the latest list of outgoing links (in terms of article ids). Then, for each month, we substitute the entries of the modified sources with their new list of edges. Moreover, we also purge sources whose last revision in the original DataFrame is at least one year older than the current month. In fact, if we did not do that, we would end up accumulating articles that do not exist anymore. This is just an approach we devised to handle the missing information of removed articles. We will be able to remove articles in the correct manner when the deletion history will be made easily available by the Wikimedia Foundation. By applying this heuristic approach, we notice that the number of articles in namespace 0 in the last revision is actually close to the real number of articles in Wikipedia ($5,273,880$ vs. $5,420,384$). If we kept all the articles, instead, the number of nodes in the graph in October 2016 would be much larger (more than 13 millions).

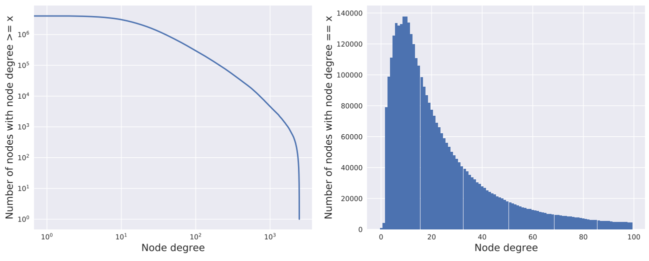

Figure 1: *Outdegree distribution in the last snapshot of Wikipedia.*

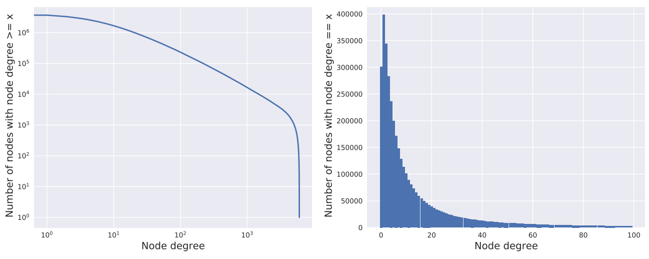

Figure 2: *Indegree distribution in the last snapshot of Wikipedia.*

5 Early Results

After the previous step, all the data processing is finished and we can start analyzing how properties of the Wikipedia's network evolve over time. Note, however, that the results shown in this section are partially wrong. This is because we noticed that some redirections failed when we obtained these instances of the graphs. An updated version of the dataset, where these issues have been fixed, is under processing at the moment of this writing and its results will be shown during the project presentation.

Last snapshot.
As a first step, we have a look at the graph corresponding to the last snapshot we generate (relative to October 2016). Here, we see that the number of articles in namespace 0 that are not redirecting pages is equal to $5,273,880$, while the total number of articles declared by Wikipedia is equal to $5,420,384$ as of 06/08/2017 [14].
We first notice that there exist $148,220$ self loops in the network despite we removed all the links pointing to a section in the same article. Their presence can be attributed to the redirect resolution step. Hence, we firstly proceed by removing all the self loops. Afterwards, we notice that there is a huge number of nodes with 0 out-degree ($1,231,733$). After noticing that many of them were indeed redirects pages that had not been correctly resolved, we have check all of them with the current version of Wikipedia. We obtain that 86% of them were indeed redirects. There are also $25,905$ articles with out-degree 1. Also in this case, we queried the current version of Wikipedia, discovering that 84% of them were redirects. Hence, we decide to drop all the articles having out-degree 0 or 1.
While we still assume that removing self loops will be necessary for the final graph, we expect the number of 0 and 1 out-degree nodes to be much smaller and there will be no need to drop them.

Figures 1 and 2 show the distribution of the outdegree and indegree, respectively, for the last snapshot. Here, we can see that only few nodes have a very large degree, being it outdegree or indegree. Moreover, the curves resemble the shape of a power law, but we will need to run proper tests to assert that this intuition is true. In the case it is, we could then evaluate how the parameters of a power law distribution evolved over time in Wikipedia. We now introduce early results on the evolution of the properties of Wikipedia's network.

As we expected from an encyclopedia, in Figure 3 the number of nodes increases over time as more articles are added. We notice that the number of nodes decreases in the last year but this could be due to the pruning step that we apply.

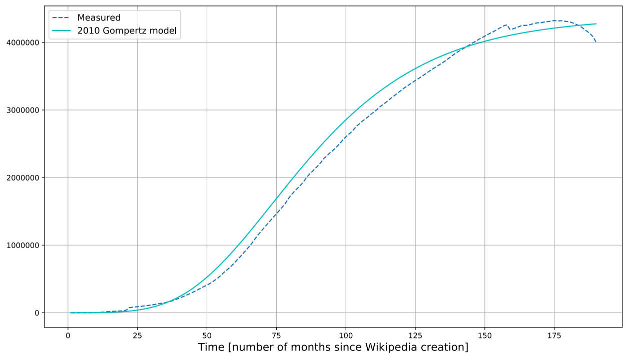

Figure 3: *Evolution of the number of nodes in Wikipedia's network.*

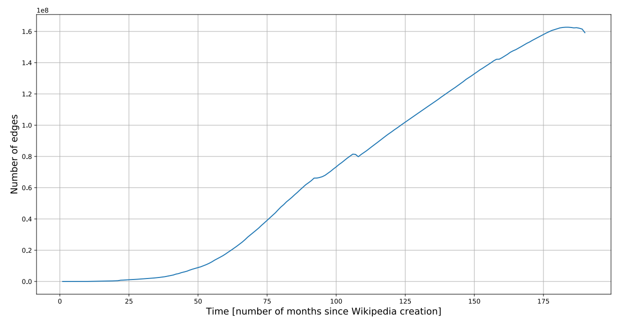

Figure 4: *Evolution of the number of edges in Wikipedia's network.*

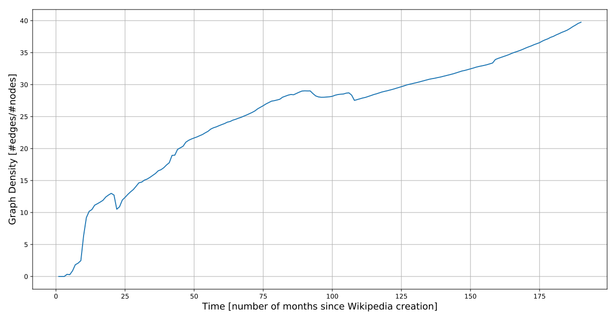

Figure 5: *Evolution of the graph density in Wikipedia's network.*

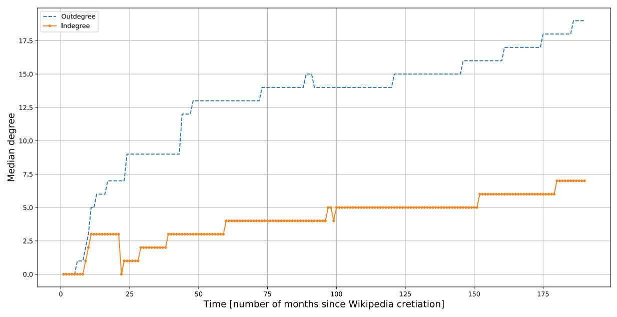

Figure 6: *Evolution of the median degrees in Wikipedia's network.*

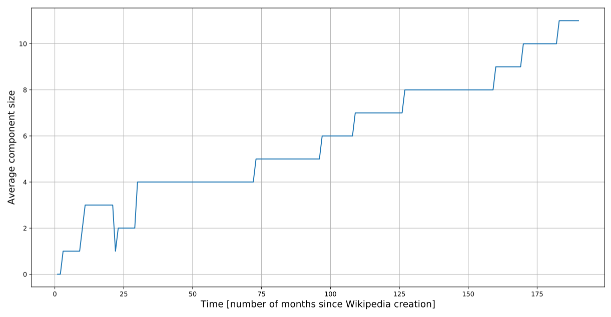

Figure 7: *Evolution of the average size of the strongly connected components.*

In this graph, we also plot the Gompertz growth model that was created in June 2010 [14]. We see that the number of nodes approximately follows this function, though more accurate parameters could be obtained by fitting the function on the data we now have available.
Figure 4 shows that also the number of edges increases over time and, in particular, as can be seen in Figure 5, their growth rate is higher than for the number of nodes, resulting in a densification of the graph.

Figure 6 plots how the median outdegree and indegree evolved over time. We can clearly see an increasing trend for both quantities but the median number of outdegree increases much more rapidly. This is what we expected as more articles are available and hence more interconnections can be established. This is also another way to see the densification over the years.

Finally, we have a look at the connected components. Figure 7 displays the average size of the strongly connected components over time. Here, we can see that components expand as more articles add pieces of missing information for different articles on related topics. Moreover, the giant strongly connected component enlarges over time, as shown in the plot at the left of Figure 8. It is also interesting to look at the graph in the right inside the same Figure. It plots the ratio between the size of the giant component in a given month and the number of nodes in that month. We can see that it has an increasing trend, meaning that the addition of new nodes does not impact badly the connectivity of the network, which instead keeps improving through the years.
Overall, the latter two figures suggest a reduction of the diameter in Wikipedia's network over time.

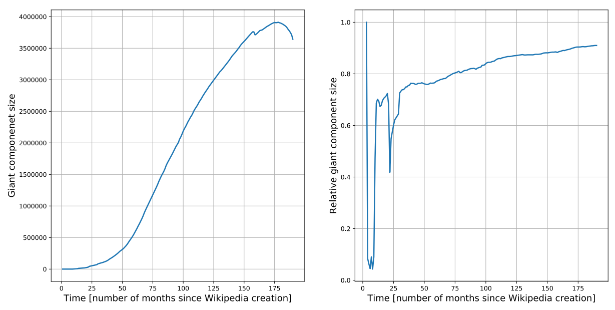

Figure 8: *Evolution of the size of the largest strongly connected component in absolute values and as a percentage of the total number of nodes in each month.*

6 Conclusion and Future Work

Processing big datasets not only is a challenging task, but also a time-consuming one. Throughout this project, we successfully extracted the outgoing links in the text of each revision of all the articles in Wikipedia since its inception. Trivial as it might sound, this task required a lot of effort and much more time than it is usually assumed to be spent on a semester project.

We provide a parser that not only does it extract standard hyperlinks from the wiki-text of an article, but also the majority of hatnotes. We believe such parser could be useful to many researchers who are interested in working with a network of links that closely resemble the true experience a reader is offered.

By the end of the project we have available two major datasets: A DataFrame containing the outgoing hyperlinks in its text, split between reachable and unreachable, and for which all the redirects have been resolved; and monthly snapshots of Wikipedia's graph in the format of edge lists.

The code is written in Python to work on Spark. We provide three useful libraries: the aforementioned Wikipedia parser to extract the links, a module that provides all the functions needed to process the dataset and another one to interact with Snap.py to compute relevant properties of the network. All the code has been extensively commented as well as clear README files have been produced, with the aim of making our results easily reproducible. Additionally, IPython notebooks show how some functions were built, containing the values taken by intermediate results.

A major part of the time and economical resources have been spent on tuning Spark so as to require the least amount of RAM allocated in order to successfully process the data. We provide all the Bash scripts tuned with these different parameters so that each processing phase can be simply run without incurring in any failures.

The processing pipeline has not always been as described in Section 4 but it evolved during the semester as different results appeared. The libraries that we provide contain working functions even for those approaches than we discarded. Among them, the functions to create a DataFrame of deltas (on the same line as the log approach presented in the related work) and to create a Redirect DataFrame (though a dictionary is already available).

We have also given some early results on the evolution of Wikipedia's network, showing how the graph densifies over time and that a shrinking diameter is expected to show up, similarly to the findings in [6].

There is still much work to do in this project and there is plenty of interesting research questions to be asked.

Regarding the data processing part, we would like to add the position of each link in the text. Moreover, it is to be investigated whether it is easy to read Spark DataFrames in other platforms, especially in Hadoop.

Regarding the evolution of the properties of the network, we still need to evaluate quantities like the evolution of the diameter and of the density of hyperlinks per article. Other interesting studies would involve understanding how much time is needed by an article to saturate its number of incoming links.

Finally, all the studies on navigation need to be explored.

Now that the processed dataset is available, a lot of fun awaits their users!

A Working Environments

In this section, we give some tips and useful links to run your code on the server and on the cluster.

A.1 Server

The dlab-server is a machine shared among all the people at dlab. Table 2 in Section 3 gives the specification of the machine. We only stress that `/dlabdata1/` is to be used only to store important data that will otherwise require intensive, long computations in order to obtain it. Moreover, being a NFS, it is slower to access data stored there. Instead, `/scratch/` should be used whenever possible as it is faster. However, data is not backed up and you may incur the risk of losing it for any potential failure.

If you want to use IPython notebooks either to run standard Python code or to open a Pyspark shell as a notebook, have a look at the `jupyter_and_spark_config.txt` file in the repository. It also contains the instructions to download and install Spark 1.6.3. The notebook will start at the server's address (`iccluster111.iccluster.epfl.ch:PORT`), where PORT is the port number that you have chosen while setting your Jupyter configurations. Make sure that PORT is not a value in the reserved range (0, 1024). Hence, anyone connecting to that address could access your notebooks. To avoid any inconvenience, we suggest adding a password, as described in the configuration file mentioned above.

Also, Anaconda3 is available on the server, but you can also install Anaconda2 on your home directory. We did so because the version of Python in Spark with which the cluster operates is Python2.7 and hence we found useful to have such installation. Moreover, also Snap.py works on Python2.7. To install it, just follow the instructions at [**snap.py**] but make sure that you use the correct Python executable. For instance, if you want to use the Python executable that is available on the Anaconda2 distribution that you have installed in `path` in the server, run a command similar to the following one in the last step of your installation:

```
sudo /path/anaconda2/bin/python setup.py install
```

A.2 Cluster

The following URLs are useful while running Spark on the cluster:

- `iccluster075.iccluster.epfl.ch:8088/cluster/scheduler`: This gives the webpage containing the information about resources usage. You can find other interesting information by following the links in the left box. In particular, Nodes will give more information about the capacity of each node in the cluster; while Applications contains the list of all the applications that ran on the cluster. This proves to be helpful when you find out that your application has failed as you might not be able to track the original error back from the stack trace (especially if you run your applications with tmux).
- `10.90.38.26:4040`: This is the URL where Spark's UI interface is launched. Each running job runs on a different port number, starting from port `4040`.
- `install.iccluster.epfl.ch/bigdataservices/`: Using the cluster has its costs. You can see the updated prices being applied at: `https://icitdocs.epfl.ch/display/clusterdocs/Big+Data+Platform` This is the web page that allows you to check how much you are spending. Once you access that page, click on the Login button and then choose your dlab account when you can see the drop-down menu. You might still not see any costs. If that is the case, just open a ticket by sending an email to `support-icit@epfl.ch`. Remember to logout when you have finished checking and to select dlab again while doing so. Otherwise, costs might not be updated if you check later.

As usual, to access a remote machine, just use `ssh gaspar@address`.
Additionally, it is possible to transfer data from the HDFS in the cluster to `/dlabdata1/` by connecting to a specific machine in the cluster, issuing the following commands:

```
ssh gaspar@iccluster050.iccluster.epfl.ch
hadoop fs -get HDFS/data/path /dlabdata1/path
```

To move data the other way around, instead:

```
ssh gaspar@iccluster050.iccluster.epfl.ch
hadoop fs -put /dlabdata1/path HDFS/data/path
```

B Datasets Processing

All the datasets mentioned in Section 4 can be found both in the cluster and in the dlab-server. The reference directory in the cluster is `hdfs:///user/bugliare/data/`, while `/scratch/bugliare/` is the one in the server. Moreover, a sample of roughly 50 revisions per dataset is available in the `data/` folder in the repository.

In this section, we report the amount of resources used and the time needed along with the configurations we set for each of the processing phases that run on the cluster. Figure 9 summarizes the following results.

Starting from the original dataframe, we firstly generate the so-called Links DataFrame. It is stored

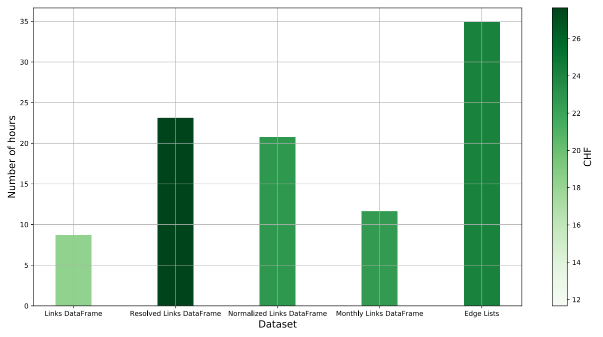

Figure 9: *Time and cost of each processing phase.*

Step	Execution time	RAM	VCores
generate 0	28 min	322 GB	211
generate 1	55 min	322 GB	211
generate 2	44 min	322 GB	211
generate 3	56 min	322 GB	211
generate 4	36 min	322 GB	211
generate 5	1 hour	322 GB	211
generate 6	1 hour	322 GB	211
generate 7	1.8 hours	322 GB	211
generate 8	30 min	322 GB	211
generate 9	31 min	322 GB	211
combine	14 min	154 GB	211

Table 6: *Resources requested to generate the Links DataFrame.*

as "wikipedia-links" under the directories mentioned above and its size is $251.7\ GB$. Table 6 shows the resource breakdown to build it. The total time is of $8.7\ hours$ and the cost is of $CHF\ 2.43$ for the RAM and $CHF\ 9.23$ for the cores, for a total of $CHF\ 11.66$.

The next step is transforming this dataset in the Resolved Links DataFrame. This is stored as "wikipedia-resolved-links" and its size is of $295.5\ GB$. Table 7 shows the resource breakdown to build it. The total time is of $23.11\ hours$ and the cost is of $CHF\ 15.32$ for the RAM and $CHF\ 12.32$ for the cores, for a total of $CHF\ 27.64$.

The final step in the general transformations is the Normalized Links DataFrame. This is stored as "wikipedia-normalized-links" and its size is of $254.3\ GB$. Table 8 shows the resource breakdown to build it. The total time is of $20.72\ hours$ and the cost is of $CHF\ 12.14$ for the RAM and $CHF\ 7.16$ for the cores, for a total of $CHF\ 19.3$.

The first project-related transformation is building the Monthly Links DataFrame. This is stored as "wikipedia-monthly-links" and its size is of $96.5\ GB$. Table 9 shows the resource breakdown to build it. The total time is of $11.61\ hours$ and the cost is of $CHF\ 2.45$ for the RAM and $CHF\ 16.49$ for the cores, for a total of $CHF\ 18.94$.

Step	Execution time	RAM	VCores
generate 1	6.7 min	807 GB	151
generate 2	7.2 min	807 GB	151
generate 3	7.5 min	807 GB	151
generate 4	7.7 min	807 GB	151
generate 5	7.4 min	807 GB	151
generate 6	7.3 min	807 GB	151
generate 7	13 min	807 GB	151
generate 8	15 min	807 GB	151
generate 9	14 min	807 GB	151
generate 10	20 min	807 GB	151
generate 11	23 min	807 GB	151
generate 12	34 min	807 GB	151
generate 13	36 min	1.24 TB	106
generate 14	32 min	807 GB	151
generate 15	33 min	807 GB	151
generate 16	33 min	1.03 TB	151
generate 17	40 min	1.03 TB	151
generate 18	43 min	1.37 TB	151
generate 19	1.2 hours	1.37 TB	151
generate 20	1 hour	1.37 TB	151
generate 21	55 min	1.37 TB	151
generate 22	44 min	1.37 TB	151
generate 23	55 min	1.37 TB	151
generate 24	1.7 hours	1.24 TB	106
generate 25	2.2 hours	1.24 TB	106
generate 26	1.3 hours	1.24 TB	106
generate 27	1.3 hours	1.24 TB	106
generate 28	1.1 hours	1.24 TB	106
generate 29	1.5 hours	1.24 TB	106
generate 30	1.5 hours	1.24 TB	106
generate 31	1.4 hours	1.54 TB	106
generate 32	1 hour	1.54 TB	106
combine	19 min	385 GB	106

Table 7: *Resources requested to generate the Resolved Links DataFrame.*

Finally, the last transformation that we run on the cluster consists of creating the edge lists. They are all stored under "monthly-edges-lists" and their total size is $89.4\ GB$. Table 10 shows the resource breakdown to build it. The total time is of $34.9\ hours$ and the cost is of $CHF\ 2.54$ for the RAM and $CHF\ 19.09$ for the cores, for a total of $CHF\ 21.63$.

C On Spark & Yarn

A very important remark is that you cannot work with the entire dataset at once. In fact, even a `count()` operation fails. The first idea behind the failure was that the data was corrupted. However, when we loaded data in batches of around $10,000$ parts each (for example, `part-r-0*`), we could actually count the number of entries in each batch. Moreover, the counting process was successful even when concatenating several

Step	Execution time	RAM	VCores
generate 1	47 min	1015 GB	64
generate 2	36 min	1015 GB	64
generate 3	35 min	1015 GB	64
generate 4	34 min	1015 GB	64
generate 5	34 min	1015 GB	64
generate 6	35 min	1015 GB	64
generate 7	32 min	1.36 TB	64
generate 8	31 min	1.36 TB	64
generate 9	32 min	1.36 TB	64
generate 10	34 min	1.36 TB	64
generate 11	37 min	1.36 TB	64
generate 12	41 min	1.36 TB	64
generate 13	39 min	1.36 TB	64
generate 14	42 min	1.36 TB	64
generate 15	41 min	1.36 TB	64
generate 16	41 min	1.36 TB	64
generate 17	41 min	1.36 TB	64
generate 18	42 min	1.36 TB	64
generate 19	41 min	1.36 TB	64
generate 20	39 min	1.36 TB	64
generate 21	40 min	1.36 TB	64
generate 22	39 min	1.36 TB	64
generate 23	41 min	1.36 TB	64
generate 24	40 min	1.36 TB	64
generate 25	42 min	1.36 TB	64
generate 26	38 min	1.36 TB	64
generate 27	41 min	1.36 TB	64
generate 28	42 min	1.36 TB	64
generate 29	41 min	1.36 TB	64
generate 30	41 min	1.36 TB	64
generate 31	44 min	1.36 TB	64
generate 32	40 min	1.36 TB	64
combine	18 min	385 GB	106

Table 8: *Resources requested to generate the Normalized Links DataFrame.*

batches: starting from `part-r-1*`, we recursively concatenated and tried counting `part-r-i*`, for $2 \leq i \leq 9$. The count operation failed when we concatenated `part-r-6*`. So, roughly, only half of the dataset could be used at once. The problem was then to be searched in the Yarn configuration of the cluster. It could either be a limitation in the (i) number of parts or in the (ii) total size that can be used at once. We cannot modify the setup of the cluster and the people managing the IC cluster could not fix it either. Hence, we need to process the data in batches.
Processing the data in batches is how we have proceeded throughout our development and, we think, it is at the core of effectively processing big data.

Step	Execution time	RAM	VCores
generate 1	41 min	217 GB	211
generate 2	41 min	217 GB	211
generate 3	41 min	217 GB	211
generate 4	40 min	217 GB	211
generate 5	42 min	217 GB	211
generate 6	45 min	217 GB	211
generate 7	46 min	217 GB	211
generate 8	47 min	217 GB	211
generate 9	47 min	217 GB	211
generate 10	45 min	217 GB	211
generate 11	44 min	217 GB	211
generate 12	44 min	217 GB	211
generate 13	45 min	217 GB	211
generate 14	43 min	217 GB	211
generate 15	45 min	217 GB	211
generate 16	37 min	217 GB	211
combine	3.9 min	385 GB	106

Table 9: *Resources requested to generate the Monthly Links DataFrame.*

Step	Execution time	RAM	VCores
generate	34.9 hours	367 GB	91

Table 10: *Resources requested to generate the Edge Lists.*

References

[1] Emanuele Bugliarello. *emanuele-wikipEdits*. [Online; accessed 9-June-2017]. 2017. URL: `https://github.com/epfl-dlab/emanuele-wikipEdits`.

[2] Emanuele Bugliarello. *Hatnotes regular expressions*. [Online; accessed 28-April-2017]. 2017. URL: `https://regex101.com/r/RFAHhi/1/`.

[3] Apache Software Foundation. *Parquet*. [Online; accessed 26-April-2017]. 2013. URL: `https://parquet.apache.org/`.

[4] Google. *Snappy*. [Online; accessed 26-April-2017]. 2011. URL: `https://github.com/google/snappy`.

[5] Jon Kleinberg. “The Small-world Phenomenon: An Algorithmic Perspective”. In: *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*. STOC ’00. Portland, Oregon, USA: ACM, 2000, pp. 163–170. ISBN: 1-58113-184-4. DOI: `10.1145/335305.335325`. URL: `http://doi.acm.org/10.1145/335305.335325`.

[6] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. “Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations”. In: *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge*

Discovery in Data Mining. KDD '05. Chicago, Illinois, USA: ACM, 2005, pp. 177–187. ISBN: 1-59593-135-X. DOI: 10.1145/1081870.1081893. URL: http://doi.acm.org/10.1145/1081870.1081893.

[7] Stanley Milgram. *The Small-World Problem*. 1967. URL: http://snap.stanford.edu/class/cs224w-readings/milgram67smallworld.pdf.

[8] B. Salzberg and V. Tsotras. "Comparison of access methods for time-evolving data". In: *ACM Computing Surveys* 31.2 (June 1999), pp. 158–221. URL: http://dl.acm.org/citation.cfm?id=319816.

[9] Robert West and Jure Leskovec. "Automatic Versus Human Navigation in Information Networks". In: *Proceedings of the 6th International AAAI Conference on Weblogs and Social Media*. ICWSM'12. 2012, pp. 362–369. URL: https://dlab.epfl.ch/people/west/pub/West-Leskovec_ICWSM-12.pdf.

[10] Robert West and Jure Leskovec. "Human Wayfinding in Information Networks". In: *Proceedings of the 21st International Conference on World Wide Web*. WWW '12. Lyon, France: ACM, 2012, pp. 619–628. ISBN: 978-1-4503-1229-5. DOI: 10.1145/2187836.2187920. URL: http://doi.acm.org/10.1145/2187836.2187920.

[11] Robert West, Joelle Pineau, and Doina Precup. "Wikispeedia: An Online Game for Inferring Semantic Distances Between Concepts". In: *Proceedings of the 21st International Jont Conference on Artifical Intelligence*. IJCAI'09. Pasadena, California, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 1598–1603. URL: http://dl.acm.org/citation.cfm?id=1661445.1661702.

[12] Wikipedia. *Extension:ImageMap*. [Online; accessed 28-April-2017]. 2017. URL: https://en.wikipedia.org/wiki/Extension:ImageMap.

[13] Wikipedia. *Help:Wiki markup*. [Online; accessed 28-April-2017]. 2017. URL: https://en.wikipedia.org/wiki/Help:Wiki_markup.

[14] Wikipedia. *Size of Wikipedia*. [Online; accessed 9-June-2017]. 2017. URL: https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia.

[15] Wikipedia. *Wikipedia*. [Online; accessed 26-April-2017]. 2017. URL: https://en.wikipedia.org/wiki/Wikipedia.

[16] Wikipedia. *Wikipedia:Hatnote*. [Online; accessed 28-April-2017]. 2017. URL: https://en.wikipedia.org/wiki/Wikipedia:Hatnote.

[17] Wikipedia. *Wikipedia:Namespace*. [Online; accessed 28-April-2017]. 2017. URL: https://en.wikipedia.org/wiki/Wikipedia:Namespace.

[18] Wikipedia. *Wikipedia:Redirect*. [Online; accessed 21-May-2017]. 2017. URL: https://en.wikipedia.org/wiki/Wikipedia:Redirect.

[19] Wikipedia. *Wikipedia:Subpages*. [Online; accessed 21-May-2017]. 2017. URL: https://en.wikipedia.org/wiki/Wikipedia:Subpages.