Wikipedia time analysis using the Apache Spark framework

Gabriele Gemmi
University of Trento, Italy
gabriele.gemmi@studenti.unitn.it

Diego Lobba
University of Trento, Italy
diego.lobba@studenti.unitn.it

ABSTRACT

In this report it is described the analysis of the dumps from English *Wikipedia* using the *Spark* framework.
Firstly the *JSON* representation of a single Wikipage has been obtained from the XML dump. Then it has been used to derive the differences among two consecutive versions of the Wikipage. Finally, all the links between every Wikipage revision have been extracted and the duplicated links for each page have been counted.

Keywords

Wikipedia; Spark; JSON

1. INTRODUCTION

Wikipedia is the free encyclopedia of the web housing millions of articles interlinked between each other. This results in an enormous amount of information freely accessible and highly referenced. At the root of the system lies its ideas of freedom and accessibility. Articles are user contributed, anyone is therefore able to modify a page entry by adding its knowledge on the topic, so that it can spread what they knows with other readers.

Changes to an article are peer reviewed by trusted members of the Wikipedia community either to prevent a so-called act of vandalism as to guarantee an high quality of the article, making it easier to read and to structure. Thus it is of crucial importance to make it easier to detect those parts of the article that have changed across versions. Differences can also be useful in order to keep track of the expansion of subtopics within an article, that could reflect the historical chain of events, as it can happen for pages on people or cities, whose content is shaped on a regular basis by related events happening.

Wikipedia pages (referred to *entities*) are written following a text format which allows to define structured elements like paragraphs, tables and images with an easily understandable style. The text, when an entity is requested by a user is parsed and rendered by *Mediawiki*, which is the Wikipedia software module responsible to render articles from the written text format entered by users to HTML elements that makes up the online article.

This component is what actually defines tokens expressed in the text, interpreting them and creating information that are stored and later used to link pages and resources that effectively are the knowledge base of the encyclopedia. Links are expressed with a particular syntax, so are also other complex structures like *infoboxes*, small amount of information that describe the main facts of the article that are rendered usually as a small box in the first part of page (upper-right corner for left-oriented languages). In addition to infoboxes, the software is highly modular making easier for users to add a new structure whose rendering is customized by a specification written by users, in a similar way to XML tags with CSS rules associated.

For a new structure to be effective the only thing required is a wide adoption by contributors. In the case the newly created element is acknowledged and used from everybody in new articles then the effect it provides is an higher coherency within articles sharing a common type of topic like cities, regions, actors... If instead the structure is not made known there are several issues that lead to a very poor and cumbersome result on final pages. For example it can happen that the same information could be expressed by two infoboxes created by different users who didn't know about the existence of an already existing structure describing the information.

This problem is addressed and solved automatically by the community, which will state the correct structure to use thus eliminating inconsistencies and correcting incoherencies introduced by its adoption. The only issue with this is on the amount of time required before this happens. By the time group members recognize the problem and address it, several edits could have been made to an article and in particular the wrong structure (which will be later deprecated) could spread to other pages, making it difficult to track it and replace it.

The complexity of Mediawiki and the highly freedom involved in taking decision within the Wikipedia community are the main sources of problems when a structured approach is used for analyzing entities. Assumptions need to be made and trades off on the final result are required. As an example the name used to define infoboxes is not uniform. In the case of the English Wikipedia, which is the one we will consider, the vast majority of entities define infoboxes prefixing the word *Infobox* to the structure name. But we cannot exclude that this holds for all infoboxes, especially when other versions of the encyclopedia are considered.

Also, although the syntax used to define the *wikitex* (the format in which articles are written in) is fairly trivial, the defined grammar is ambiguous, thus is not possible for a parser to recognize the grammar in an efficient and reliable way, making the final rendering of the same Wikipage application-dependant.

Finally, user defined structures like infoboxes are handled using the same Wikipedia knowledge base. Structures are

parsed and the corresponding action is stored withing the Wikipedia database. Thus by just having the original infoboxes or any other interpreted element, is not possible to infer the final result as it will be rendered and displayed on the online page of the article. As a trivial example let us consider two links, referring to the same entity: *Trento*. One could refer to the entity either by a link with label the same name of the city (namely `[[Trento]]`) or by clearly stating the Wikipedia version on which the entity should be searched, like `[[en:Trento]]` in the case of the English encyclopedia. If one would consider all possible cases she or he should use the same Mediawiki to parse and interpret the content of the article, but this is not possible since it will involve a huge amount of calls to the service that would disrupt the normal requests traffic made by users.

So a separated and dedicated lexer is required, accepting, as previously mentioned, trades off and assumptions due to the lack of the knowledge base which is instead available to the official software module.

2. TECHNOLOGIES

The processes involved in the analysis performed are done on a single Wikipedia page at the time. Note that the term page is referred to the set of revisions the page had from its first appearance to its last modification (limited by the time the encyclopedia dump containing the page was taken). This means that page existing in the 2004 could potentially contain more than 1000 revisions. Since is considered only pure text (images are not enclosed in the page itself but by links to the referenced image), the size of the single page is well under 1 MB size, [5] but the high number of revisions within a single page can increase the total size of a single page to several megabytes.

The type of operations performed on data can be described (and are effectively implemented) by means of *map-reduce* functions. Map-reduce[1] is a programming approach first developed at Google. It consists on defining a function which takes a set of key-value pairs and produce an intermediary set of key-value pairs by applying a function to each pair (*mapping*). In a similar way a reduce function is defined, which takes pairs with the same key and apply an operation to merge pairs values. This approach allows to perform the same operation on a huge chunk of data, which can be loaded from text files or directly from a distributed storage.

Although the amount of memory the process has to handled is fairly small, the operation it has to perform are quite intensive. To obtain the set of differences within two adjacent versions an algorithm of pattern matching against all the words within the revisions is performed.

This technology has allowed to exploit the computational power of a cluster instead of a single processor. This was possible because the type of computation performed was not by it's nature parallel.

In order to develop a map-reduce approach on a clustered environment the *Apache Spark*[6] framework has been used. Spark has been preferred over Apache Hadoop for its ability to keep in memory the result of a computation, without the need to write it to a shared storage across map-reduce operations as opposed to the Apache Hadoop framework (and the original MapReduce implementation) All processing is done using the *Resilient Distributed Datasets* (RDD) provided by Spark, constructs provided by the framework to offer fault tolerance and distributed processing of data across *workers* within the cluster.

Apache Spark therefore takes care of providing the abstraction mechanism that allows the developer to focus on the task he wants to implement, without concerning on the underlying implementation handled by spark.

3. JSON FORMATTING

This section describes the process used to obtain JSON objects given a set of Wikipedia pages in the XML format.

The first issue addressed is obtaining dumps of articles containing the complete history of each page. Wikipedia dumps are available in XML format. Different type of dumps are made available. Based on the content provided are available dumps containing only the last revision of each page or dumps containing their full history, where the full article at each revision is given. In this analysis the full history dumps have been chose.

This implementation gives the user freedom on how to retrieve dumps. They can either define a desired dump version and the software automatically downloads dumps from a mirror and stores them locally, or they can load dumps that were downlaoded previously from the hard disk.

Once dumps are retrieved the first task begins. It consists in transforming dumps from the original *wikitext* format to a JSON representation for later use with the spark framework. The structure of the final JSON object is the one showed in listing 1.

Listing 1: JSON structure

```
<revision> ::=
    <title> <timestamp> <user>
        <structured-part> <unstructured-part> |
    <title> <timestamp> null
        <structured-part> <unstructured-part>

<structured-part> ::= <name-value-pair>*
<name-value-pair> ::=
    <name> <visible-value> <entity-link> |
    <name> <visible-value> null
<unstructured-part> ::= <token-location-pair>*

<text-location-pair> ::=
            <token> <entity-link> <position> |
            <token> null <position>

<token> ::= any word or link
```

The structured part is given by the name-value pairs defined in the main infobox for a given entity. Also, note that the number of detected token was used as position metric. The choice to differentiate the presence of a link with an additional value is both given by convenience and correctness. It's more correct since what appears on a Wikipage can be different from the effective name of the entity linked, for example a phrase could be used as a link to an entity. It's convenient since by doing this we can process links easily, automatically distinguishing them from other tokens. If this wasn't done it would not be possible to distinguish a normal token by a link without a prior knowledge of which tokens represent entities and also, motivated by the prior example, it would not be possible to detect an entity described by a token which is different from the entity name.

To parse Wikipedia pages a third-party lexer[2] has been used. Its functionality were used for recognizing links, text

regions and infoboxes. Any other more complex structure has been ignored (like tables, Wikipedia specific XML tags). The structured part is made of elements identified by the first infobox detected by the lexer, assuming it actually represents the main properties for the page. Infoboxes are syntactically equivalent to templates, to be more precise: infoboxes are a particular type of template. A template is a structure used to describe an action the Wikipedia parser has to perform while rendering the page element. Templates are used within pages to describe conditional actions or to call functions (to retrieve the time of the day for example). It is therefore required to make an assumption in order to differentiate infobox templates from other type of structures. In the English Wikipedia the vast majority of pages identify infoboxes with the prefix `"Infobox:"`. In this way it was possible to identify infoboxes within the template structure. Tokens detected by the lexer are then encapsulated within software objects which allow to reflect the definition related to the structured-part and unstructured-part defined in listing 1. Other revision elements are directly taken by the XML dump, since the contributor, timestamp and page fields are defined there. These values are stored in the same object structure previously mentioned, which serializes the Wikipage in the JSON format by means of a method.

Once the processing required to transform a Wikipage from its original wikitext format to a JSON object has been defined the Spark framework was used to effectively perfom the conversion on dumps. A set of pages (each page containing all its revisions) was extracted from the dump, called *chunk*. Pages revisions are then considered for the processing leading to the application of a simple map function to obtain the JSON format out of Wikipedia pages as in algorithm 1.

Algorithm 1 Convert wikitext to JSON.

```
1: function JSONIFY(revisionContent)
2:     revision ← ParseRevision(revisionContent)
3:     return revision.toJson()
4: procedure WIKITEXTTOJSON
5:     //chunk contains entries structured
6:     //       <title, timestamp, contributor, wikitext>
7:     for chunk in getNextChunk() do
8:         chunkRDD ← Parallelize(chunk)
9:         chunkRDD.map(Jsonify)
```

The result of algorithm 1 is a set of pure JSON string representing Wikipage information. The results were stored in JSON files, where each file contains all revisions within a single page, one revision (so one JSON object) per line. This was accomplished by collecting the set of existing pages, separating them by their page title and revision and then by saving revisions related to the same page in a common file. Then it has been implemented a mapper which parses the actual JSON representation of the page and returns the sequence *<title, timestamp, json-page>*.

4. FINDING DIFFERENCES BETWEEN REVISIONS

This section describes the analysis of tokens added or deleted among two adjacent revisions. In particular the processing starts from the set of JSON strings of the previous task and aims to obtain a JSON file containing the information described in listing 2. The final file contains a description format which describes, given the page and the two adjacent revisions (expressed by their timestamp), the set of actions that happened in the next revision with regards to the current.

Algorithm 2 Mapper used to obtain revision information.

```
function COLLECTREVINFORMATION(revisionString)
    revisionJson ← LoadJSON(revisionString)
    title ← GetPageTitle(revisionJson)
    timestamp ← GetTimestamp(revisionJson)
    return <title, timestamp, revisionString>
```

Listing 2: Difference information

```
<diff-revision> ::=
    <title>
    <ts-current-rev> <ts-next-rev>
    <diff-set>

<diff-set>   ::= <diff-entry>*
<diff-entry> ::= <action> <position> <token>+
<action>     ::= "INS" | "DEL"
```

The loading of the JSON strings previously stored can be easily achieved by using the *Dataframe* structure, provided by Spark, as intermediary to the RDD. Once the strings are loaded the page title, the revision timestamp and the text with its associated position is extracted. Since the analysis performed involves the position of a token, only the *unstructured-part* of the page will be considered, as its content is the only one providing this information.

While iterating over elements of the unstructured-part a differentiation is made: if the token is not a link it will be selected as it is, otherwise if it is a link the entity will be returned pre-pending an `"[E]"`. In this way the knowledg of what originally it was is not lost. In algorithm 3 is described the mapper function used to collect tokens. The function must be applied using the *flatMap* function provided by the Spark framework. The result is a set of entries *<title, timestamp, token, position>*.

Algorithm 3 Mapper used to obtain tokens.

```
function COLLECTTOKENS(revisionString)
    revisionJson ← LoadJSON(revisionString)
    title ← GetPageTitle(revisionJson)
    timestamp ← GetTimestamp(revisionJson)
    unstruct-part ← GetUnstructPart(revisionJson)

    tokens ← empty list
    for element in unstruct-part do
        position ← GetPosition(element)
        if element is a link then
            token ← GetVisibleText(element)
            prefix token with [E]
        else
            token ← GetEntity(element)
        tokens.insert(<title, timestamp,
                       token, position>)
    return tokens
```

4.1 Indexing the timestamps

Revisions are identified by timestamps which are strings in a specific format that indicates the exact time in which changes were applied to the page. The format allows to compare timestamps in order to have a meaningful information on the precedence of a revision over the other, but this does not give us a very manageable way to determine whether there have been other revisions between two given timestamps. This makes hard to understand whether two revisions are adjacent in time. In order to achieve this goal a series of map operations have been performed to obtain an integer representation out of a timestamp.

Starting from the quadruplet *<title, timestamp, token, position>* of the previous operation entries are mapped in order to consider only their title and their timestamp, then double entries are removed. The result of this phase is a set of *<title, timestamp>* entries, where title can be seen as the key for the entry. Then values of entries having the same title are grouped together. Then a mapper function is used to sort the timestamps of each page and to assign an index to each value. A rough idea of the algorithm used for the mapper is given in algorithm 4. The mapper is applied only to values of entries and the result is flatted, so that the result is in the form: *<title, timestamp-string, timestamp-index>*.

Algorithm 4 Indexer mapper.

```
function INDEXER(timestampList)
    orderedTsList ← Sort(timestampList)
    indexSet ← empty list
    index ← 0
    for timestamp in orderedTsList do
        indexSet.insert(<timestamp, index>)
        index ← index + 1
    return indexSet
```

The next step is the replacement of the timestamp with the index obtained in the previous operation.

This is done by joining the two sets according to a common key. Since both sets have the title and timestamp-string fields in common it is possible to join them using these field as common composite key. Since join operations can be performed only on sequences of pairs, elements of each entries in both sets must be grouped. Once the sets are correctly joined entries belonging to the two sets sharing the same composite key are merged together, allowing the replacement of the timestamp-string with the timestamp-index field. fig. 1 depicts the set of operations described so far.

4.2 Pairing adjacent revisions

Having associated an indexed timestamp to each revision makes the operation of pairing adjacent revisions trivial. Given a revision whose index is i is necessary to pair it with revision $i + 1$.

To do this is performed the collection of tokens belonging to a specific revision within a single entry leading to entries of form
<<title, ts>, [<token$_1$, pos$_1$>, ..., <token$_n$, pos$_n$>]>.
Note that due to RDD properties of preserving the order of information (changed only by particular operations) it is possible to assume that tokens (and therefore locations) respect the original order of appearance in the wikitext.

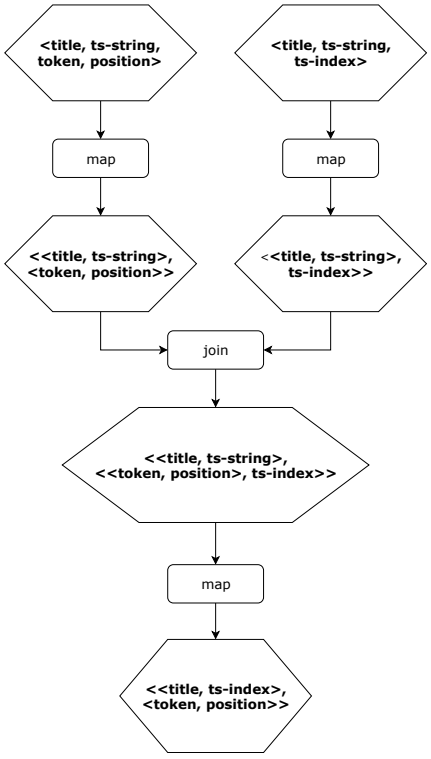

Figure 1: Timestamp indexing.

Then a field, which represents the next revision, is added to the entry. This can be simply achieved with a map operation which sets a new field before the current timestamp with value equal to the current timestamp index plus one. The newly created field replaces the current timestamp field in the composite key, so that the new entry will be *<<title, next-ts>, <current-ts, [<token$_1$, pos$_1$>, ..., <token$_n$, pos$_n$>]>>*.

In parallel with the creation of this extended set another operation is performed.

A replica of the set prior to the index changes is made, then this set is joined with the one returned by the previous operation. In this way the adjacent revision are aligned and can be joined using a common key.

The result of this process is a set of entries of the type:

<<title, next-ts>, <<next-ts, token-list>
<current-ts, token-list>>

Since we created an "artificial" mapping associating a revision with the next one, we have also associated the last revision to a non-existing entry, so we need to perform a post processing map operation to cleanup this issue.

4.3 Obtaining the difference set

It is now possible to effectively compute the difference between elements of two adjacent revisions by using a mapper function that analyze revisions tokens and compare them.

The first solution implemented in order to compare revisions was to simply check that tokens related to the same position were equal. This would have simplified a lot the processing without requiring all the complex operations described so far. Yet this solution would have been extremely

imprecise, since all tokens following the change would have been recognized as edited too. In order to have more precise diff a comparison in the style of the popular Unix-like utility *diff* have been chose. Since the diff problem is well known it was possible to use the implementation of the algorithm provided in the *difflib* module [3].

Once the diff function is defined it is possible to write the mapper function which applies it, as described in algorithm 5, note that here the *Diff* function is a custom function defined by the developer which performs the diff-like comparison between token lists and returns the set of actions required to convert the current revision into the next one. The function is applied to values of the entries, without affecting keys. The overall process of pairing revisions and the processing to obtain difference sets is depicted in fig. 2

Algorithm 5 Difference-set mapper.

```
function DifferenceSet(ts1, tokens1, ts2, tokens2)
    Determine which is the current revision
            and which is the next one
    diffSet ← Diff(currentTokens, nextTokens)
    return <ts-current, ts-next, diffSet>
```

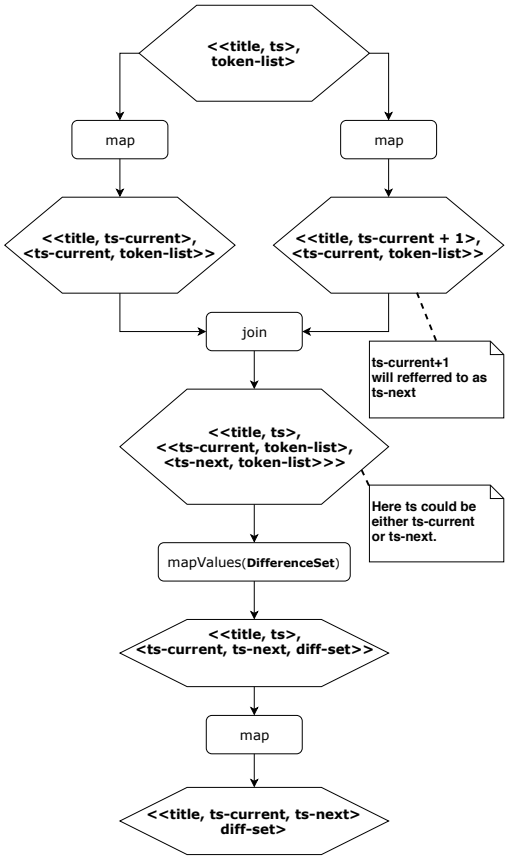

Figure 2: Difference processing flow.

4.4 Reversing the index

The process is not finished yet, since timestamps are represented as indexes. The last step is to put back the string format associated to the index.

The process uses the indexer previously created and apply the same mapper function described in algorithm 4 The only difference is that the results are not flattened. The result of the operation has the following type:

$$<title,\ [<ts\text{-}string_1,\ ts\text{-}index_1> \ldots, <ts\text{-}string_n,\ ts\text{-}index_n>]>$$

Then a new map function is defined and perform the operation of returning string-index pairs between couples of adjacent revisions, as described in algorithm 6. The result is a set of entries of the form

$$<<title,\ ts\text{-}index\text{-}current,\ ts\text{-}index\text{-}next> <ts\text{-}string\text{-}current,\ ts\text{-}string\text{-}next>>$$

which can be joined with the set of entries obtained from the diff process. Fields are then moved in order to obtain the desired effect, that is to replace the the index representation of timestamps with their string representation. The process is briefly summed up in fig. 3

Algorithm 6 Adjacent revision indexer.

```
function GenPairIndexer(associationList)
    strings ← empty list
    indexes ← empty list
    for association in associationList do
        ts-string ← GetTsString(association)
        ts-index ← GetTsIndex(association)
        strings.insert(ts-string)
        indexes.insert(ts-index)
        //timestamp representations for a given
        //   revision share the same index
        //   among string and indexes lists.
    pairs ← empty list
    for i ← 0 to Length(indexes) - 2 do
        pairs.insert(<indexes[i], indexes[i+1],
                     strings[i], strings[i+1]>)
    return pairs
```

This task is concluded by saving results using a JSON structure. Firstly a schema for the JSON needs to be defined . Then is possible to convert the RDD into a Dataframe and save directly to a JSON file the result.

5. COUNTING ENTITIES

This section describes the processing of the entity-links performed in the task 3.

The aim of this operation is to produce data set containing all the links between all the pages of Wikpedia. The input of this operation is the JSON representation of the Wikipage performed in the task 1 and the output is a sets of JSON containing the number of links between every revision of a Wikipage.

After having loaded the JSON Wikipage in an RDD the function *collectLinks* is applied to the RDD. This function checks every token in the Wikipage and returns a set of all the entity-link in the page. Using a map function the key of the entity-link is set to the tuple $<P1,\ P2,\ t>$, in this way a reduce function can be applied and all the links connecting

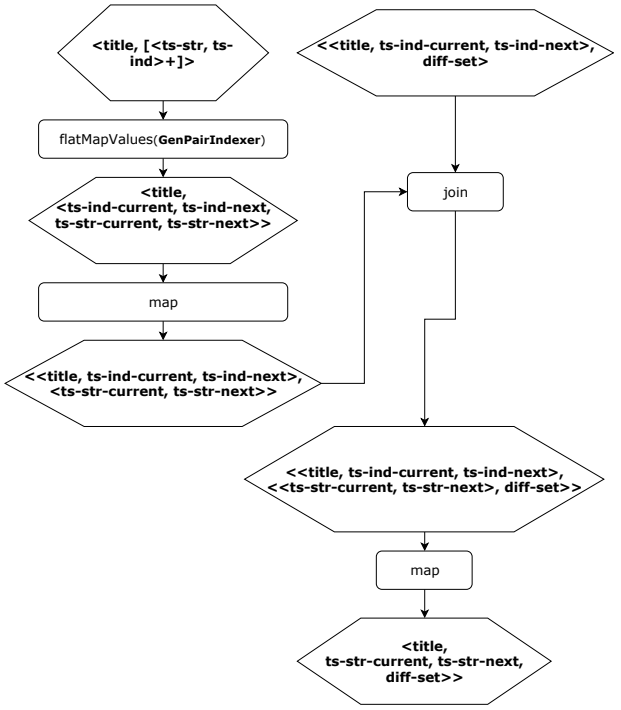

Figure 3: Flow chart for the reverse indexing process.

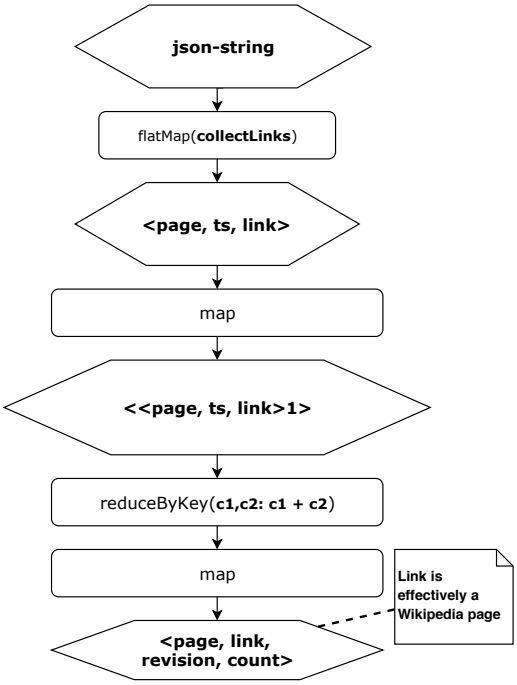

Figure 4: Counting entity-links processing flow

Figure 5: The Wikipedia page related to Giacomo Leopardi.

two specific entities can be merged together. During the reduce function the counts of the links is incremented by one for each merge. The resulting output is written to a JSON file using the following schema $<P1, P2, t, n>$ where $P1$ and $P2$ are the two pages connected by n links at the revision t.

6. EXPERIMENTS

In this section the results obtained from the processing described in previous sections are exposed. The first two tasks describes a set of statistics derived from results of the task considered, showing how values were computed and some considerations that can be inferred on data presented.

Given the huge amount of data made available by Wikipedia, considered only a small part of it was considered. This analysis was done considering a file belonging to the set of dumps produced in early July 2018, in particular the dump contains pages whose ids range from page 44931 to page 47733. The dump came compressed with a size about 2.6 GB.

Given the high performance of the compressing algorithm that yelded an uncompressed file of over 40 GB it has been chose to use a small subset with size around 4.5 GB. This was splitted into smaller files of approximately 500 MB, which were later compressed for a total size of 180 MB.

Pages that had a total amount of characters (considering all revisions) higher than 200000 characters were selected, trying to exclude pages related to Wikipedia talks or users.

6.1 Experiment1: JSON formatting

Here is showed a very brief example of the results obtained from the first task. The example regard the page of *Giacomo Leopardi* at the 21:55 of the 28^{th} of June 2018.

The result is as showed listing 3, the JSON object contains a single revision of a given Wikipage. The file produced contains a JSON object per line. Information such as the title of the page, the timestamp in which the revision has been taken and the contributor are first-level properties of the object. In this case the contributor wasn't detailed in the dump, so the field has value *null*.

Then is shown the structured and the unstructured-part. The structured-part contains name-value pairs belonging to the first infobox found within the page. It's possible to note that entities contained as a value in the infobox are reported in the third field of the array. It can be seen that some information is not properly captured by the lexer, for example the date of birth and the date of death are not correctly identified. This happens due to the fact that the information is expressed as a template, thus the lexer has not the required knowledge to parse correctly the information. In this case the lexer simply returns null. In listing 4 it's showed in details the unstructured-part. Each word is considered as a

token which is counted in order to assign to each token the position assumed within the text. In the case of a token representing an entity the visible text (as it can be read from the article) is stored in the first field, while the second field contains the actual name of entity which can be made of several words (as the token *"Enlightenment"* showed in the last line of listing 4.

Listing 3: Result of task1

```
{"title": "Giacomo Leopardi",
 "timestamp": "2018-06-28T21:55:17Z",
 "contributor": null,
 "structured_part": [
     ["region", "Western philosophy", null],
     ["era", "19th century", null],
     ...
     ["name", "Giacomo Leopardi", null],
     ["birth_date", "", null],
     ["birth_place", "Recanati Papal States",
        "Recanati"],
     ["death_date", "", null],
     ["death_place",
        "Naples Kingdom of the Two Sicilies",
        "Kingdom of the Two Sicilies"],
     ["death_cause",
        "Pulmonary edema or cholera",
        "cholera"],
     ["school_tradition",
        "Classicism later enlightenment
         Romanticism",
        "Age of Enlightenment"],
     ["nationality",
        "Italian",
        "Italian people"],
     ...
     ["notable_ideas",
        "Pessimism",
        "Pessimism"]],
 "unstructured_part": [
     ["Giacomo", null, 1],
     ["Taldegardo", null, 2],
     ["Francesco", null, 3],
     ...]
}
```

Listing 4: Details on the unstructured part

```
"unstructured_part": [
  ["Giacomo", null, 1], ["Taldegardo",null,2],
  ["Francesco",null,3], ["di", null, 4],
  ["Sales", null, 5], ["Saverio", null, 6],
  ["Pietro", null, 7], ["Leopardi", null, 8],
  ["29", null, 9], ["June", null, 10],
  ["1798", null, 11], ["14", null, 12],
  ["June", null, 13], ["1837", null, 14],
  ["was", null, 15], ["an", null, 16],
  ["Italian", null, 17],
  ["philosopher", null, 18],
  ["poet", null, 19], ["essayist", null, 20],
  ["and", null, 21],
  ["philologist", "philologist", 22],
  ...,
  ["Although", null, 57], ["he", null, 58],
  ["lived", null, 59], ["in", null, 60],
  ["a", null, 61], ["secluded", null, 62],
  ["town", null, 63], ["in", null, 64],
  ["the", null, 65], ["conservative", null, 66],
  ["Papal States", "Papal States", 67],
  ["he", null, 68],
  ["came", null, 69], ["in", null, 70],
  ["touch", null, 71],["with", null, 72],
  ["the", null, 73], ["main", null, 74],
  ["ideas", null, 75], ["of", null, 76],
  ["the", null, 77],
  ["Enlightenment", "Age of Enlightenment", 78],
 ...]
```

6.2 Experiment1: statistics

From the JSON objects obtained in the first task some statistics were collected. These allows to have a general understanding on the set of articles contained within the dataset considered. Statistics are extracted as explained in appendix A. In table 1 it's possible to see the amount of data processed in this analysis. It should be pointed out that a revision is actually a stand-alone Wikipedia page.

Table 1: Information related the amount of data processed during the analysis.

Total number of pages	224
Total number of revisions	154771
Average number of tokens	1583
Average number of links	337
Average number of revisions	690

6.3 Experiment2: Differences among adjacent revisions

In the second task the aim is to detect differences among two consecutive revisions of a given page. The result is a file containing a set of JSON objects (one per row) describing the title of the page considered, the timestamp of the two revisions and a set of entries describing the action (insertion or deletion), along with a list of tokens, that occurred on a given position. For instance, in listing 5 are reported the differences found between two of the most recent versions of the page related to Giacomo Leopardi. It is noticeable, by checking with the Wikipage at [4] that the tokens identified have been effectively inserted between the two revisions described and belong to the *"external links"* paragraph within the page.

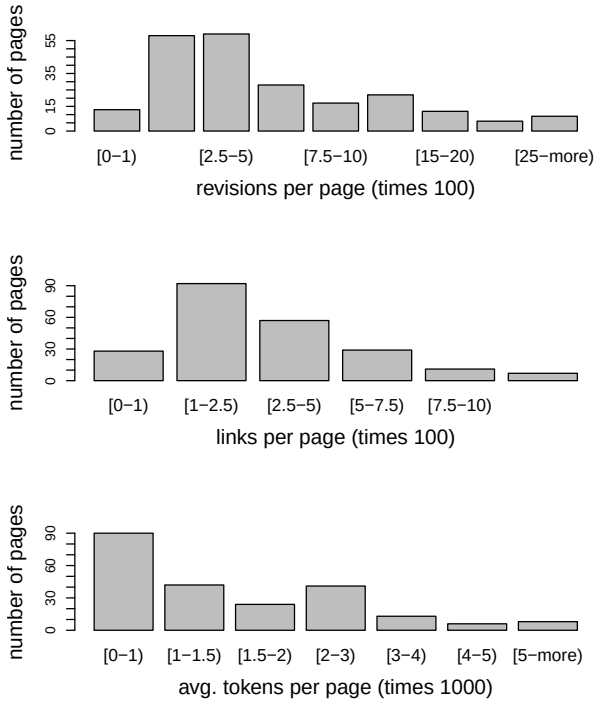

Figure 6: Statistics derived from results of task1

Listing 5: Result of task2: difference-set found among two adjacent revisions belonging to the page of Giacomo Leopardi.

```
{"page":"Giacomo Leopardi",
 "ver1":"2018-06-03T13:51:59Z",
 "ver2":"2018-06-28T21:55:17Z",
 "diff":[
   {"action":"INS","position":8515,
    "tokens":["The","calm","after",
       "the","storm",
      "and","other","poems","by","G",
      "Leopardi",
      "https://www.yeyebook.com/en/.../...",
      "in",
      "EN","IT","FR",
      "DE","ES","CH"]}
]}
```

6.4 Experiment2: statistics

From results of task2 it can be developed an approach to analyze the behavior of revisions related to a given page. From each page the total number of insertion and deletion operations, and the total amount of tokens inserted and deleted, were counted. The aim of this operation is to highlight whether on average insertion and deletion operations affect a single portion of the article within a single revision. If this is the case then the management of the Wikipage is easily trackable and can be easily reversed. If instead a huge amount of sparse operations are done within a single revision then the operations required to revert them can be cumbersome, probably leading to the creation of a new revision in order to *merge* and solve colliding information. This is particularly the case when either not all the information added to an article are good nor all of them are completely wrong. So the reviewer has to filter bad contents from good ones.

To detect this behavior the ratio between the total number of revisions and the total number of insertions and deletions was computed. Ideally a ratio near to 1 means that the number of insertions (deletions) is equal to the number of revisions, meaning that on average each revision affected one part (or a very limited subset) of the article. A ratio of a half means that approximately the number of insertions (deletions) is double the number of revisions, so the page was changed at least in two different parts within a single revision. While considering these values one should also take into account that the actions considered are only insertion and deletion. Due to this fact a replacement iS seen as a deletion on a given position followed by an insertion to the same position. Finally the ratio between the number of tokens inserted and the number of tokens deleted was computed.

From fig. 7 it is noticeable that while the deletion-revision ratio is on average near to a half the insertion-revision ratio is smaller. Finally it's possible to conclude that on average both insertion and deletion operations affect more than one portion of article per revision and that the number of insertions is greater than the number of deletions. It's interesting to see that the two ratios are quite similar. Unfortunately with the information available is not possible to determine whether this is given by a series of replacement operations or by a long series of deletions. Additionally in fig. 8 it is shown the ratio between the number of tokens deleted and the number of tokens inserted. It is possible to notice that the vast majority of Wikipages has more insertion operations than deletions, which highlight an overall increase of the article. We can also notice a very limited amount of pages where the number of insertions is less than the number of deletions (the ratio is greater than one).

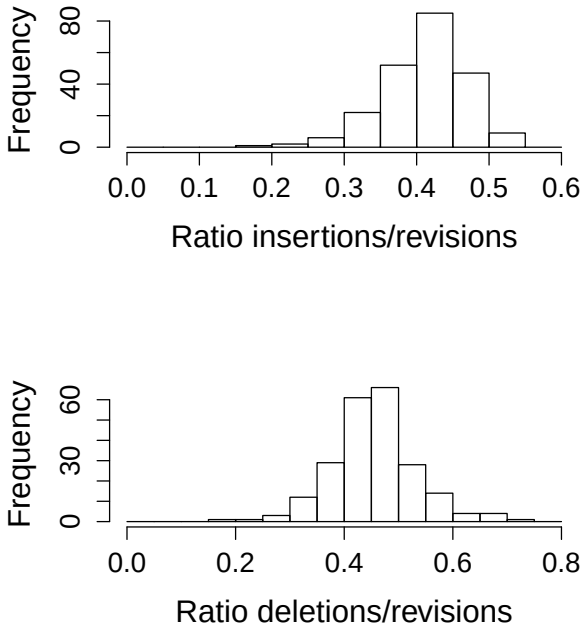

Figure 7: Ratio insertions-revisions and deletions-revisions.

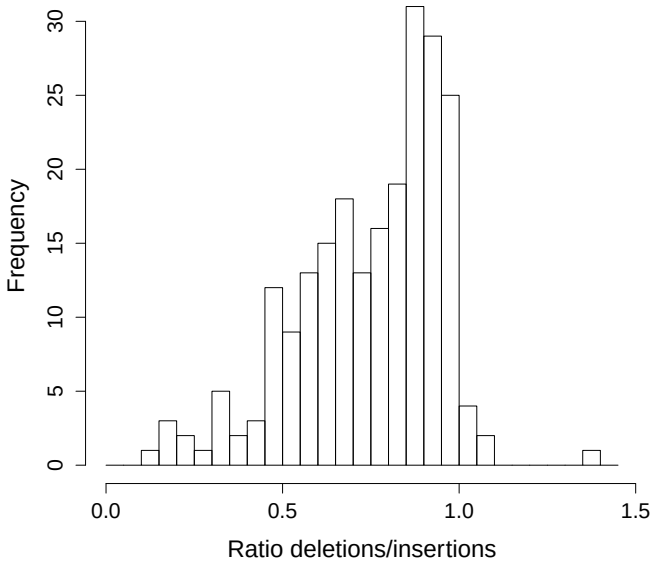

Figure 8: Ratio deletions/insertions.

7. CONCLUSIONS

In this report it was shown a way to use the Apache Spark framework to convert Wikipages from their original wikitext format to a JSON representation which has been found useful in developing a structured approach to retrieve and manipulate information contained within a page. In this first task it has been analyzed the structure of the wikitext and the difficulties involved in its processing. In particular the JSON representation of page revisions has been used to compute the difference-set of tokens between pairs of adjacent revisions, highlighting insertion and deletion operations. The same representation has been used to count the amount of time a given Wikipedia entity is contained within a given page to have a measure on the relationship that bounds two entities together. Finally a set of statistics has been presented showing information on the test set considered in the analysis and some conclusion that can be inferred from data obtained from each stage of the analysis process.

It is our feeling that much of this analysis has been shaped by the efficiency of the lexer adopted to identify article tokens, which highlights the quite fragile equilibrium between the grammar used to define Wikipages and the corresponding software module that must interpret them, resulting into an heterogeneous set of information whose ambiguities cannot be solved without the knowledge available in the same Meadiawiki module.

8. REFERENCES

[1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[2] Earwig. mwparserfromhell. https://github.com/earwig/mwparserfromhell.

[3] G. Rossum. Python library reference. Technical report, Amsterdam, The Netherlands, 1995.

[4] Wikipedia contributors. Giacomo leopardi — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Giacomo_Leopardi&oldid=847952251, 2018. [Online; accessed 15-July-2018].

[5] Wikipedia contributors. Wikipedia:article size — Wikipedia, the free encyclopedia, 2018. [Online; accessed 11-July-2018].

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, Berkeley, CA, USA, 2012. USENIX Association.

APPENDIX

A. COMPUTING STATISTICS FOR TASK 1

The statistics produced out of the data generated by the first task are the following, for every page we want to collect:

- the number of revisions associated,
- the total number of links for every revision,
- the average number of tokens computed among all revisions of a given page.

These statistics are helpful because they allow us to make some conclusions on the data. For example by considering the number of links within a given page we can say that a page with an high number of links is probably a better aggregator than a page with a smaller number of links. By considering the average number of tokens we have an estimation on the length of the article, while by considering the number of revisions we can deduce the status of the page. A page with a small number of revisions could mean that it is an article created recently, or it could be an article that is not read very frequently. Usually a page with a small number of revisions and with just one or two links is probably referencing a page that redirects to another article.

To generate the statistics we start from the output of the first task. We import JSON strings representing Wikipedia pages using the Dataframe structure and then we switch to the RDD, as already done for the second task. To compute the number of revisions within a page we retrieve the page and timestamp field from the JSON string using a map function. We then map entries in order to remove the revision field and add a new field with value one, in this way we have a number of entries *<title, 1>* for the amount of revisions of a given page. Finally we reduce entries based on their keys summing up counters for each page.

To collect links we start from JSON strings, collecting links from the structured and unstructured part of the page by applying the map function with a flatMap operation obtaining *<title, link>* pairs. We then remove doubles and map page of the entry to the value one, removing the field containing the link, since we are merely interested in counting the number of unique links within a page. So we merge, as done previously, values of entries sharing the same key with a reduce by key operation. The process of counting the average amount of tokens among revisions is slightly more complex. We collect tokens from the unstructured part of the JSON string by applying a flat-map operation using the mapper showed in algorithm 3 described in section 4. Similarly to what previously done in for other statistics we map entries keeping the page title field and replacing the value one to the token field. We then proceed with a reduce by key operation to sum entries belonging to the same page. Note that in this way we obtain the total amount of tokens considering *all revisions* of a page. To get the average we have just to divide this field when joining the set to the set containing the number of revisions of a given page.

Entry set are then joined together taking into account the case in which some value is missing, values are then saved into a comma-separated file. The three operations just described are depicted in figure fig. 9.

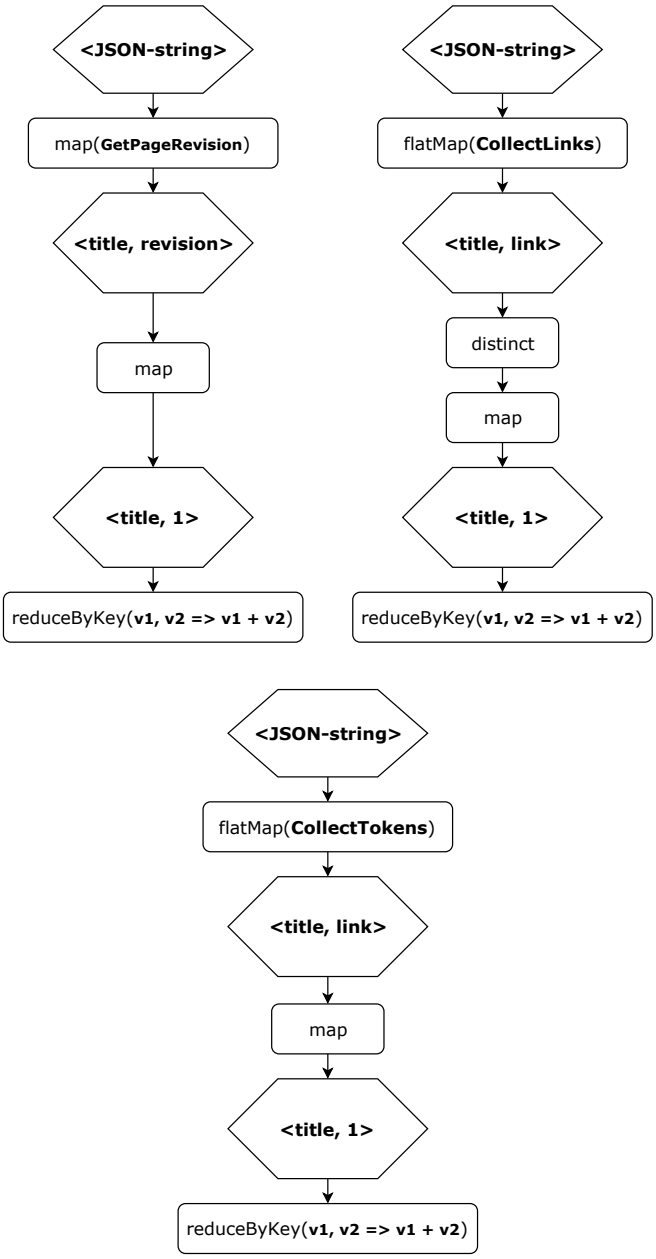

Figure 9: (from left to right and bottom) The flow of operations required to compute the three statistics: number of revisions, number of unique links and total number of tokens among all revisions (to be averaged when joining later), respectively.

B. COMPUTING STATISTICS FOR TASK 2

In this section we describe how to get a series of statistics that complemented with the ones of appendix A help in giving a deeper insight to a page, allowing to infer helpful information in the understanding page dynamics over the time. The information we want to have at the end of this step is, for each page:

- the number of insertions,
- the number of the deletions,
- the total number of tokens inserted,
- the total number of tokens deleted.

The number of insertions and deletions can, combined with the number of revisions of the page, provide a clue on the type of changes the page is subject to. If the ratio between insertions and revisions is near to 1 (or slightly above) it means that edits to the page targeted mainly a portion of the article per edit; while an high ratio insertions-revisions would mean that lots of edits on different locations were made on a single revision.

We start the computation from the output produced by the second task. We load difference sets contained within JSON files into a Dataframe, we then use its functionalities to select entries having a non empty list of differences (meaning that there was effectively some change between two adjacent revisions). We then switch to the RDD structure, mapping entries in order to create a composite key made of title, version1 and version2 fields while leaving the difference set as value. Note that we could have avoided to keep information about the two versions, since we are only interested in processing the difference set, but we leave as a starting point for a further more fine-grained analysis on differences between revisions. We then use a map function which iterates over the difference set of each entry, collecting the required information as described in algorithm 7. We then map entries in order to remove the two versions where the edit takes place, finally we merge per-page values using a reduce by key operation. Values are then saved into a comma separated file. The set of operations described so far is depicted in fig. 10.

Algorithm 7 Mapper used to collect difference-set information.

```
function CollectCountStatistics(differenceSet)
    //differenceSet contains entries of type:
    //   <action, pos, token-list>
    insertions ← 0
    deletions ← 0
    tokensAdded ← 0
    tokensRemoved ← 0
    for diff in differenceSet do
        if GetAction(diff) is "INS" then
            insertions ← insertions + 1
            tokensAdded ← Length(GetTokenList(diff))
        else
            deletions ← insertions + 1
            tokensRemoved ← Length(GetTokenList(diff))
    return <insertions tokensAdded,
                deletions, tokensRemoved>
```

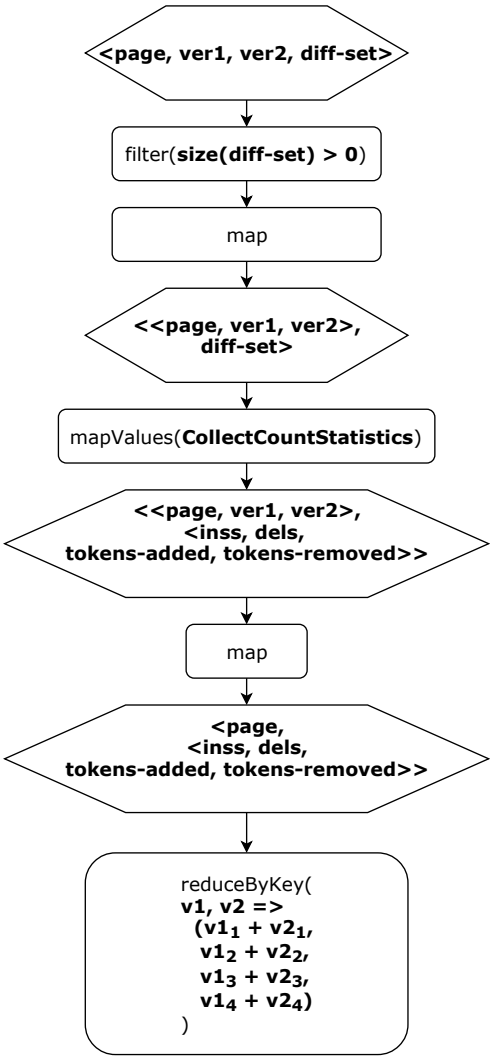

Figure 10: Operation flow to obtain statistics from results of task2.