UNIVERSITY OF PISA

Cloud Computing

Year 2020/2021

PageRank implementation in Hadoop and Spark

Project specification

MORTEZA AREZOUMANDAN

FRANCESCO DEL TURCO

AHMED SALAH TAWFIK IBRAHIM

Link GitHub: https://github.com/FranDT/CloudComputingProjects/tree/master/PageRank

1.0 Hadoop

In this section, we will show how we implemented our version of the PageRank algorithm exploiting Hadoop, in particular using Java language to develop the algorithm. For our project, we exploited Maven to import the dependencies to handle Hadoop's libraries, in particular we imported the two following dependencies, that are necessary for the implementation (the version was the latest at the time of the development):

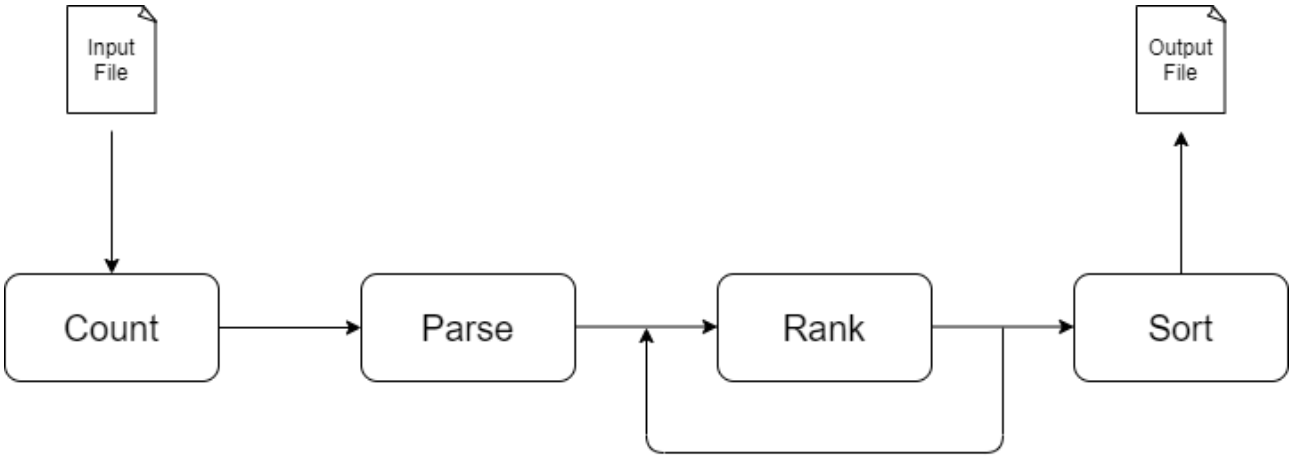
```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>3.3.1</version>
</dependency>
```

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-core</artifactId>
    <version>1.2.1</version>
    <scope>provided</scope>
</dependency>
```

For this implementation, we considered the list of pages with their outlinks as a connected graph: in particular, the nodes, represented through the Node.java class, are the pages, while the arches are the outlinks. In the Node.java class, that implements Writable, we can find three fields: a double corresponding to the rank of the page; a List of String corresponding to the list of outlinks; a boolean field, that is used during the ranking phase to check if the value we're emitting from the map phase is actually a Node object or is the contribution from a node to the receiving outlink.

1.1 Design of the algorithm – Phases

The following picture represents the sequence of phases that the input file has to go through when processed with our algorithm:



- **Count phase:** during the count phase, we retrieve the number of pages, for which we don't just consider the lines of the input document but we also consider all the possible outlinks, that we must retrieve line by line, for a total of 34.483 different pages: for the outlinks, we consider that they will be placed between "[[" and "]]". In

the map phase we emit the key-value pairs (pageName, 1); in the reducer, we increment a static counter for each different page name and the value of the counter is emitted at the end of the reducer's processing phase, thanks to the cleanup method. In this way, the number of pages will be equal to the number of different pages we can find in the original document, with the removal of duplicates being performed automatically by the reduce phase. The number of pages is then saved in order for it to be used in the following phases;

- **Parse phase:** in this phase, we perform during the map phase something similar to what we did in the map phase of the Count, but now we actually emit the page name and the list of outlinks of that page (if a page does not have outlinks, we emit the page name and an empty list). In the reducer, we first run the setup method in order to retrieve the number of pages, then we initialize through the reduce method all the Nodes, with an initial rank equal to 1 divided by our computed number of pages. This phase can be run on multiple reducers, as well as the next phase;
- **Rank phase:** this phase is the hearth of the algorithm and is a phase that is repeated multiple times, working in each iteration on the output of the previous one. The aim of this phase is to take, for each node, the rank calculated in the previous phase (in the first iteration we take the nodes with the default value equal to 1 over the number of pages) and to divide this rank by the number of outlinks, so that we can represent the contribution of the page with respect to each outlink. In particular, during the map phase we have two possible emits: the first type is a (pageName, Node) pair, that is emitted to propagate the structure of the graph; the second type is a (pageName, mass) pair, where mass is a double equal to the contribution made by a certain page to the receiving one. In the parse phase, after we obtained the number of the pages and the value of alpha through the setup method, we check for each page all the incoming values, adding the sum of all the contribution and using it to finally calculate the new rank using the formula:

$$PR = \frac{\alpha}{N} + (1 - \alpha) * \sum_{m \in L(n)} \frac{PR(m)}{C(m)}$$

  In this formula, N is the number of pages as we calculated it, L(n) is the set of outlinks that link to n, C(m) is the number of links on page m and alpha is the random jump factor. Finally, we emit the page name and the Node corresponding to that page, to be used in the following iterations or by the final stage;
- **Sort phase:** during this last phase, we exploit a WritableComparable object called Page, that contains a String corresponding to the name of a page and a double corresponding to its final rank, in order to perform the sorting. In particular, during the map phase we retrieve the Nodes and we translate them into a Page object, we

emit the page as key to perform the sorting over it (along with a NullWritable object as value, since we're not interested in the value); in the reduce phase, we just retrieve the title and the rank for each page and we write them in the final output file, with the sort being performed automatically by Hadoop's sort and shuffle phase.

Each phase is triggered by a program called Driver, which takes care of checking eventual errors and to pass the right parameters to all the phases.

1.2 Pseudocodes

In this section, we will present the pseudocodes of the phases we saw previously, divided in map and reduce phases.

1.2.1 Count pseudocode

```
class Mapper
    method Map(lineid l, text T)
        title <- T.getTitle()
        outlinks <- T.getOutlinks()
        if(title != NULL)
            Emit(title, 1)
            if(outlinks.size() > 0)
                for all link in outlinks do
                    Emit(link, 1)
```

```
class Reducer
    N <- 0
    method Reduce(text P, counts[c1, c2, ...])
        N <- N + 1
    method Cleanup()
        Emit(text A, N)
```

1.2.2 Parse pseudocode

```
class Mapper
    method Map(lineid l, text T)
        title <- T.getTitle()
        outlinks <- T.getOutlinks()
        if(title != NULL)
            if(outlinks.size() > 0)
                for all link in outlinks do
                    Emit(title, link)
                    Emit(link, NULL)
            else
                Emit(title, NULL)
```

```
class Reducer
    method Setup()
        pageNumber <- getPageNumber()
    method Reduce(text P, links[l1, l2, ...])
        outlinks <- new List
        for all link in links[l1, l2, ...] do
            if(link != NULL)
                outlinks.add(link)
        rank = 1.0/pageNumber
        N <- new Node(rank, outlinks, True)
        Emit(P, N)
```

1.2.3 Rank pseudocode

```
class Mapper
    method Map(text P, node N)
        Emit(P, N)
        mass <- N.Rank / |N.Outlinks|
        for all link in N.Outlinks do
            Emit(link, mass)
```

```
class Reducer
    method Setup()
        alpha <- getAlphaValue()
        pageNumber <- getPageNumber()
    method Reduce(text P, nodes[n1, n2, ...])
        M <- 0
        s <- 0
        for all n in nodes[n1, n2, ...] do
            if isNotNode(n) do
                s <- s + n
            else
                M <- n
        M.Rank <- (alpha/pageNumber) + (1 - alpha) * s
        Emit(P, M)
```

Note that all the items inside nodes[n1, n2, ...] in the reduce phase can either be a mass or an actual node: the mass is propagated as a contribution for the node which is propagated towards, while the node is propagated to propagate the structure of the graph; without the propagation of the node, it would be impossible to understand which are the outlinks of each node, therefore it is a mandatory operation in our implementation. The formula to obtain the new rank is the same showed in paragraph 1.1.

1.2.4 Sort pseudocode

```
class Mapper
    method Map(text P, node N)
        p.Title <- P
        p.Rank <- N.Rank
        Emit(p, NULL)
class Reducer
    method Reduce(Page p, NULL)
        title <- p.Title
        rank <- p.Rank
        Emit(title, rank)
```

In this case, the value p always represents a Page. In the Reducer, since we exploit Hadoop's Sort, we just need to take, for each page, the title and the rank and emit them.

1.3 Optimization

In this section, we show how we addressed different efficiency issues.

1.3.1 Reducers

We tested our application with more than 1 reducers for the Parse and Rank phases, since they're the only ones that actually can exploit multiple reducers: in fact, we can't use multiple reducers in the Count phase, since we exploit in its reduce phase a static variable, that won't be shared between multiple reducers since they may run on different virtual machines (and even on different nodes); at the same time, it's not worth to use multiple reducers in the Sort phase, since the reducers will write their results in different files, so they won't return a global sorting but only a sorting on the keys they obtain; these problems are not met in the Parse and Rank phases, so we can exploit multiple reducers for them.

We tried therefore to run the algorithm using 1, 3 and 5 reducers, obtaining the following results:

Reducers: 1

	Map time	Reduce time	Map MB/ms	Reduce MB/ms
Count	3,022	2,561	1,547,264	1,311,232
Parse	4,435	5,209	2,270,720	2,667,008
Rank1	5,036	3,988	2,578,432	2,041,856
Rank2	4,491	4,099	2,299,392	2,098,688

Reducers: 3

	Map time	Reduce time	Map MB/ms	Reduce MB/ms
Count	3,854	3,844	1,973,248	1,968,128
Parse	3,131	11,281	1,603,072	5,775,872
Rank1	16,343	11,607	8,367,616	5,942,784
Rank2	24,513	11,412	12,550,656	5,842,944

Reducers: 5

	Map time	Reduce time	Map MB/ms	Reduce MB/ms
Count	2,748	2,612	1,406,976	1,337,344
Parse	3,041	29,132	1,556,992	14,915,584
Rank1	39,152	32,461	20,045,824	16,620,032
Rank2	43,129	34,965	22,082,048	17,902,080

In the tables, we reported, for both Parse and Rank, the completion time for the map tasks (reported in milliseconds), the completion time for the reduce tasks (reported in milliseconds), the total amount of megabytes by millisecond taken by the map tasks and the total amount of megabytes by millisecond taken by the reduce tasks. We also showed the same parameters for the Count phase, that can be used as reference.

1.3.2 Customized Hadoop objects

As already mentioned previously, we used in our implementation two customized Hadoop objects:

- **Node:** this object implements the Writable interface and is used as value in the reducer phase of the Parse and in the map and reduce phase of the Rank. This object is created for to ease data serialization/deserialization, in order to decrease the data size overhead to improve the performances of the algorithm;
- **Page:** this object implements the WritableComparable interface and is used as in the Sort and Shuffle phase of the Sort, emitted as a key from the mapper. Thanks to this object, we can exploit Hadoop's Sort mechanism to easily sort our Page objects on the rank (and eventually, on the page name if the rank is the same).

1.3.3 Setup and cleanup methods

In our implementation, we exploited both cleanup and setup methods in different occasions. The cleanup method has been used inside the reduce phase of Count, in order to emit the

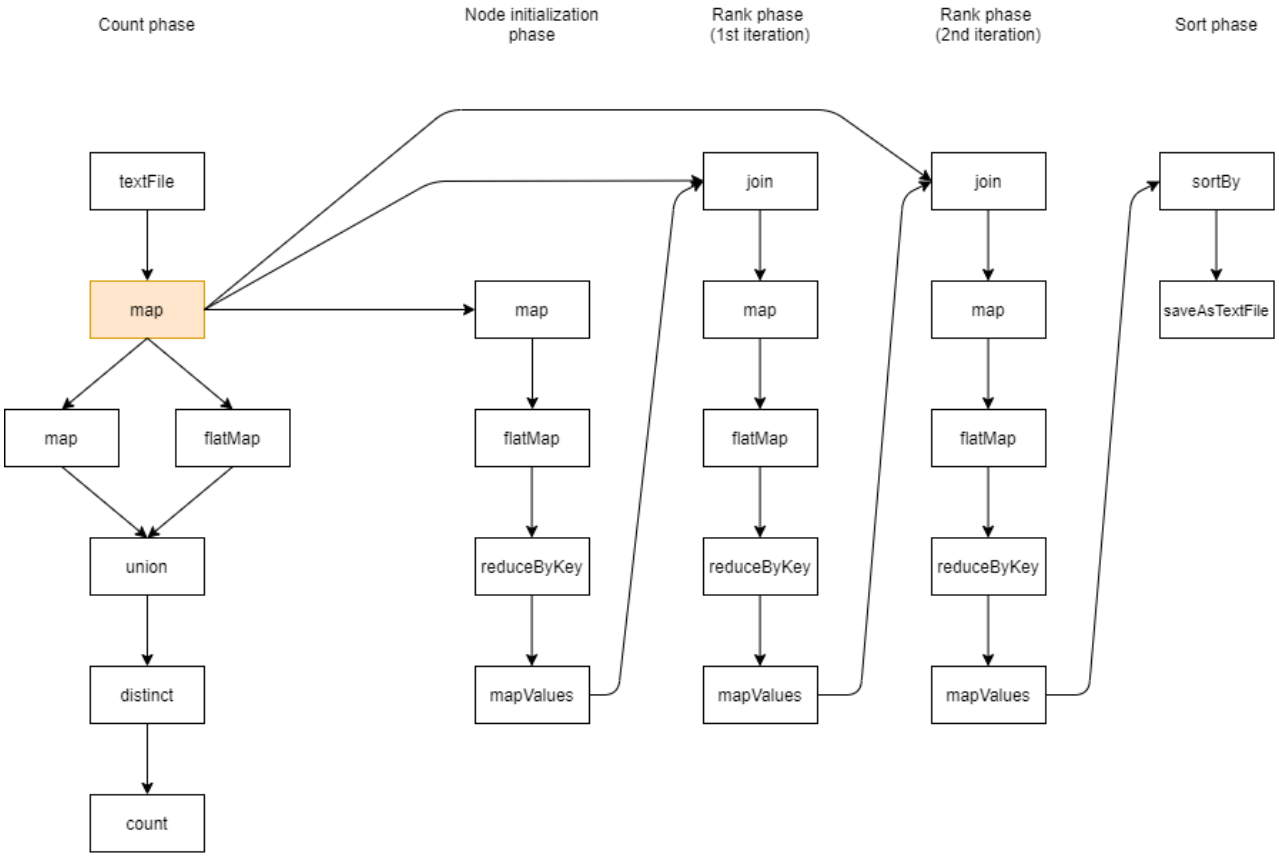
number of total pages calculated by the reducer: we exploit this method since it is performed at the end of the processing phase of the reducer, so that we are sure that the value we're emitting is the correct one.
The setup method is instead used in two occasions, but with the same purpose: in fact, it is used in the reduce phase of the Parse and the Rank to pass to the reducer some parameters, in particular the number of pages in both cases and the value of alpha in the Rank.

2.0 Spark

In this section, we want to show the implementation of the PageRank algorithm through Spark. In particular, we've used Python and Java for our implementation: it's important to underline that the general assumptions made for Hadoop about the implementation of the algorithm are valid for this implementation as well, therefore we consider the number of pages equal to the sum of the outlinks and the pages corresponding to the lines, removing duplicates. Moreover, we use the same formula as before to calculate the rank at each iteration.

2.1 Design of the algorithm in Python – DAG

In the following picture, we can see the DAG for the Python implementation of PageRank through Spark, in which we show three iterations of PageRank (one is performed in the node initialization phase):



- **Count phase:** the Count phase of this implementation does not correspond to the count phase of the Hadoop implementation, since it completes the count but it also partially overlaps Hadoop's Parse phase, since at the start we create from the input file the list of outlinks for each node. This is done by the map transformation that comes after reading from the textFile: the resulting RDD is also cached, since we have to access it at all iterations, since it contains all the titles of the pages and their

corresponding outlinks. On this RDD, we perform another map transformation to retrieve the page titles, then we perform a flatMap transformation to retrieve the outlinks, in order to retrieve a unique list of outlinks instead of a list of list of strings. We then merge the two RDD obtained through a union transformation and we perform on the resulting RDD a distinct transformation, to make sure that there are no duplicates. We finally count the elements in the resulting RDD, obtaining our number of pages N;

- **Node initialization phase:** this phase partially overlaps with the remaining part of Hadoop's Parse phase, but also implements the first iteration of the ranking phase. Starting from the RDD containing the page titles and their outlinks, we append the initial rank equal to 1/N during the map phase to each page; after that, we calculate the contribution of each page through the flatMap transformation, that returns an RDD with a single list of pairs (pageName, contribution). In the reduceByKey transformation, we take the previous RDD and we obtain the sum of the contribution for each page, that is finally used in the mapValues to obtain, for each page, the new rank: note that we need to use mapValues since we're interested in summing the contribution, that in the previous RDD were saved as value as in the pair shown above;
- **Rank phase:** in the rank phase, we take the result of the mapValues of the previous iteration (therefore, the page title and its newly calculated rank) and we join it with the list of outlinks over the page name. We then perform a map transformation to obtain the ranks to be used in this iteration; the rest is the same as in the previous phase;
- **Sort phase:** in this phase, we take the final result obtained from the iterative rank phase and we use the sortBy transformation to obtain an RDD that is sorted by value, that corresponds to the rank. This RDD is our result, so we just have to save it through the saveAsTextFile action.

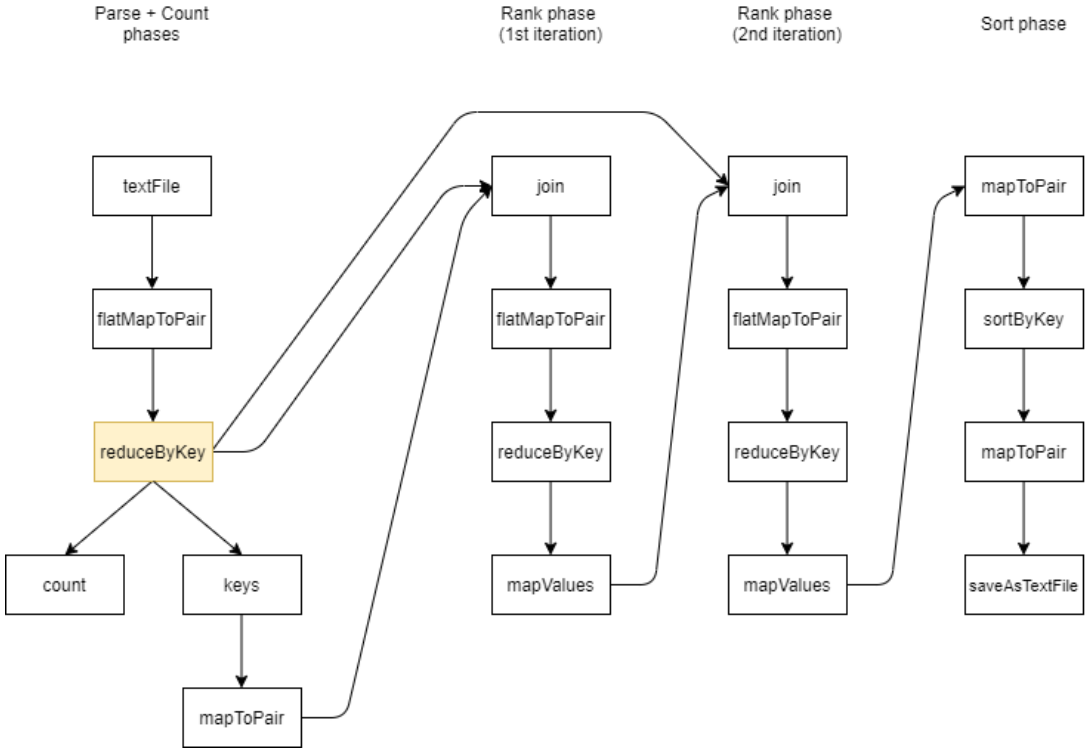
2.2 Design of the algorithm in Java – DAG

We have implemented the PageRank in Spark using Java, taking the same general assumption and trying to imitate as much as possible the Python approach: in some cases, we had to make some mandatory changes due to the different implementation of the functionalities provided by the library.

First of all, we have used the following dependency to work on Spark with Java:

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.12</artifactId>
    <version>3.1.2</version>
</dependency>
```

In the following picture, we've reported the DAG of the Java implementation, in which we show just two iterations of the PageRank algorithm:



- **Parse + Count phase:** in this phase, we perform both the parse and the count at the same time since they are performed on the same RDD. First of all, we read the text file; then, we perform a flatMapToPair transformation, equivalent to the flatMap in Python, that is used to parse and create the initial list of nodes and outlinks, creating and RDD containing either (title, emptyList) or (title, outlinks). This RDD is then reduced through reduceByKey in order to obtain an RDD containing the unique pages with their corresponding outlinks: this RDD is cached since it is used in all the iterations of the ranking phase. From this RDD, we can easily obtain the number of pages, performing on it the count action, but we also extract the keys, that are used to create another RDD through the mapToPair transformation, where the entries of the RDD are the title of the page and its initial rank equal to 1/N;
- **Rank phase:** in the rank phase, we start by joining the RDD with all the nodes and outlinks cached in the first phase with the RDD of titles and corresponding rank computed in the previous phase, either the previous iteration of the rank phase or the execution of the Parse + Count phase. On this RDD, we perform a flatMapToPair transformation to obtain all the outlinks with the corresponding contribution. This RDD is reduced in the following phase in order to obtain, for each outlink, the total contribution, that is finally used in the mapValues to calculate, for each page, its new rank, that is then used in the following iteration and passed as (title, rank);

- **Sort phase:** in the last phase, we can't directly perform a sort since there isn't a sortBy function in Java as there is in Python, so we first perform, on the RDD containing the titles and the corresponding final ranks, a mapToPair transformation that swaps keys and values of each entry of the RDD, we then perform the sortByKey on the new RDD (that has therefore the rank field as key) and we finally perform another mapToPair transformation to switch once again the keys and the values. Finally, we save the result through the saveAsTextFile action.

3.0 Execution

In the following section, we will explain how to execute the two implementations on the cluster we've been assigned with.

3.1 Hadoop execution

To execute the Hadoop implementation of the PageRank algorithm, it is necessary to create a jar file to be run on the cluster: to do this, we first need to clone the GitHub repository, then we need to run the following command inside the /HadoopPageRank folder:

```
mvn clean package
```

After this, the jar file will be created with the name "HadoopPageRank-1.0-SNAPSHOT.jar" and will be placed inside /HadoopPageRank/target. This operation can be performed either on a local machine or directly inside the namenode: in case it is performed on a local machine, it is necessary to move the jar file inside the namenode and we can do this with the command:

```
scp HadoopPageRank/target/ HadoopPageRank-1.0-SNAPSHOT.jar hadoop@172.16.3.207:
```

Now that we have our jar file on the namenode, we can access to the it at IP hadoop@172.16.3.207. To test the algorithm, we need to run the following command:

```
hadoop jar <jarPath> it.unipi.hadoop.Main <inputFile> <outputPath> <numIteration>
<alpha> <numReducers>
```

The jarPath will be equal to "/target/ HadoopPageRank-1.0-SNAPSHOT.jar" if the jar file was created directly inside the cluster, otherwise it will be equal to the name of the jar file; the inputFile can be moved to the cluster using the scp command as seen before; note that alpha must be a double between 0 and 1.
To check the result, from the namenode we need to run the following command:

```
hadoop fs -cat <outputPath>/sort/part-r-00000
```

It is suggested to use "| head" at the end of the command to see the pages with highest rank. It is also possible to check the results of all the intermediate phases by substituting "sort" with either "count" (it will show a single line with "Total pages - N"), "parse" (it will show the list of initialized node with the starting value) and "rank-i", where i is the output of the iteration we want to check (useful to check the evolution of ranks). Note that, since the parse phase and the rank phase can be executed with multiple reducers, we will have multiple files

inside the corresponding folders, all named part-r-0000x, where x goes from 0 to the number of iterators minus 1.

3.2 Spark execution

To execute the Spark code, we need to have the /SparkPageRank/pagerank.py file on the namenode, that we can move from the local clone of the repository as seen before for the Hadoop jar file. Once the file is on the namenode and we performed the access to the namenode itself, we just need to run the following command:

```
spark-submit <inputFile> <outputPath> <alpha> <numIteration>
```

To check the result, from the namenode we need to run the following command:

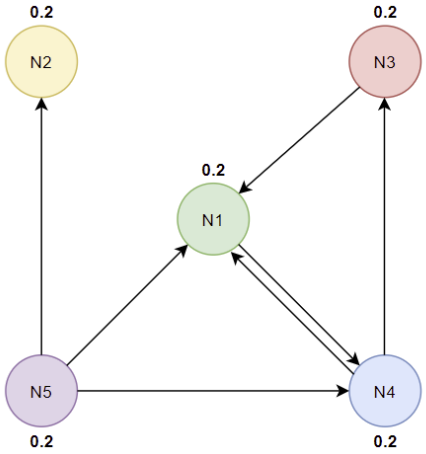
```
hadoop fs -cat <outputPath>/part-00000
```

4.0 Testing

To check the correct functioning of the algorithm, we tested it on a synthetic dataset composed as follows:

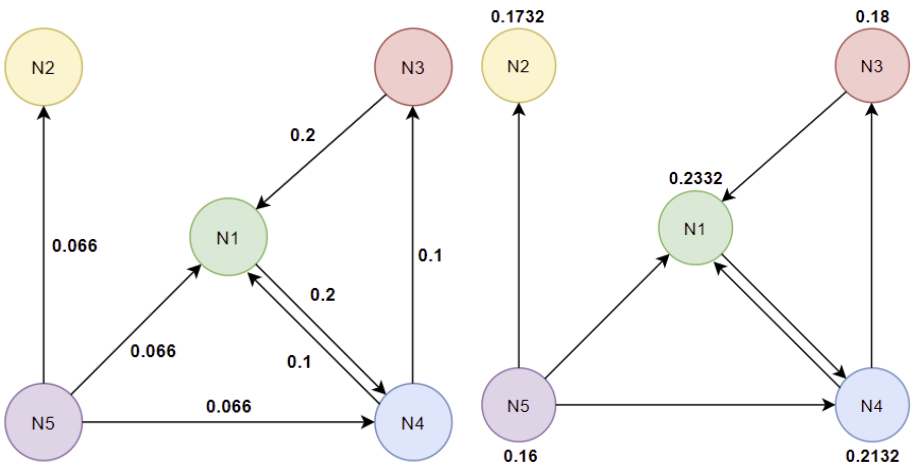
```
<title>N1</title><revision><text>content [[N4]]</text></revision>
<title>N2</title><revision><text>content</text></revision>
<title>N3</title><revision><text>[[N1]] content</text></revision>
<title>N4</title><revision><text>[[N1]] [[N3]] content</text></revision>
<title>N5</title><revision><text>[[N1]] [[N2]] [[N4]]</text></revision>
```

This file corresponds to the following graph:

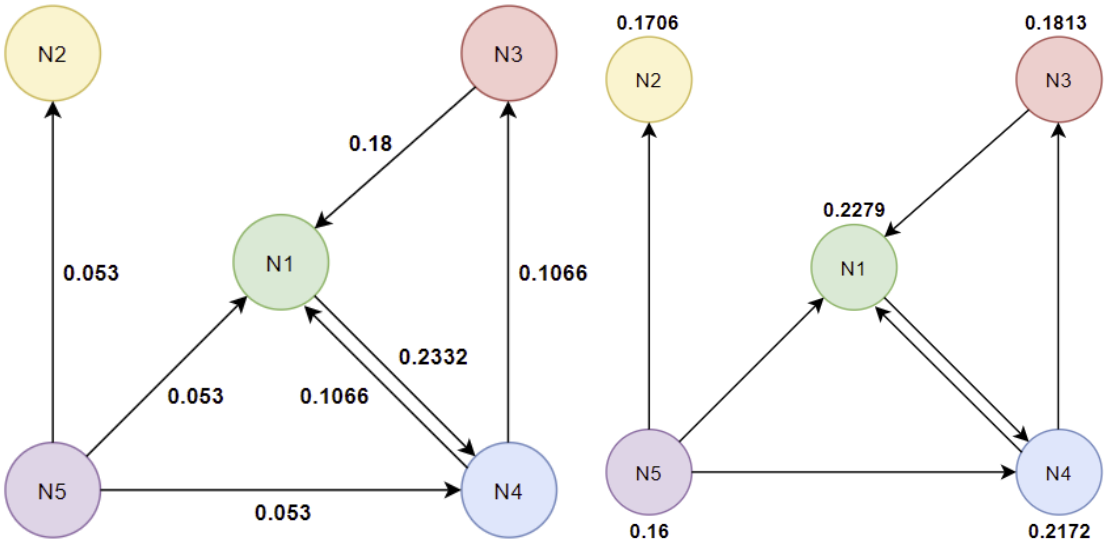


The values associated to each node are their initial ranks, equal to 1/N, where N=5 since it's equal to the total number of nodes. We want to show the result of the algorithm with two iterations and alpha equal to 0.8: we will compare this result with our output to check if the algorithm is working correctly.

Starting from the previous situation, we calculate the contributions to be propagated inside the structure, from which we will obtain the contributions as we can see them in the picture on the left. Summing all the contributions in each node and considering the formula to calculate the rank, we obtain the rank we can see on the right:



At this point, we can perform the second iteration on the graph above, obtaining the following result:



The result we obtain is therefore the following:

```
(N1    0.2279)
(N4    0.2172)
(N3    0.1813)
(N2    0.1706)
(N5    0.16)
```

Running the implementations of PageRank on this dataset, we obtain for Hadoop:

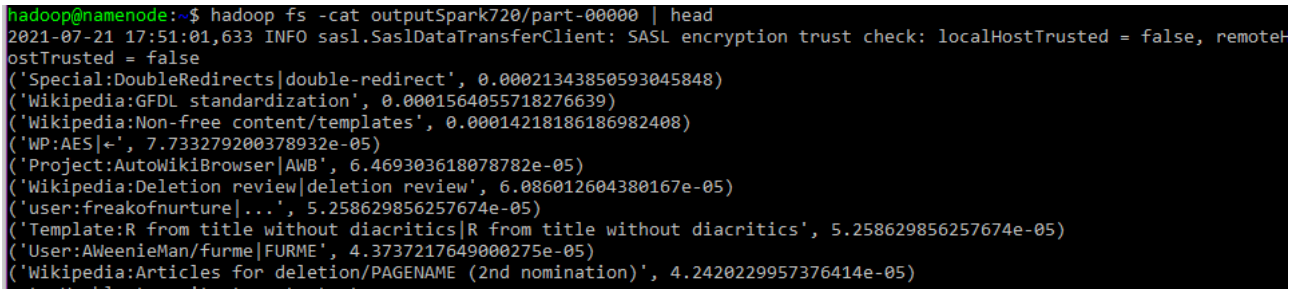
```
hadoop@namenode:~$ hadoop fs -cat output-hadoop750/sort/part-r-00000
2021-07-21 17:39:17,408 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteH
ostTrusted = false
N1      0.22799999999999998
N4      0.21733333333333332
N3      0.18133333333333332
N2      0.17066666666666666
N5      0.16
```

The output of Spark is the same, both for the Java and the Python implementation.

Since we've obtained the right result, although we tested it on a small dataset, we can assume the implementations are correct.

We can therefore show the result of our tests on the real-world dataset provided for the project, named wiki-micro.txt. In the following picture, we have respectively the result for Hadoop and for Spark, showing just the highest ranked pages:

```
hadoop@namenode:~$ hadoop fs -cat output-hadoop730/sort/part-r-00000 | head
2021-07-21 17:50:13,955 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteH
ostTrusted = false
Special:DoubleRedirects|double-redirect 2.134385059304588E-4
Wikipedia:GFDL standardization  1.5640557182766384E-4
Wikipedia:Non-free content/templates    1.4218186186982406E-4
WP:AES|←        7.733279200378932E-5
Project:AutoWikiBrowser|AWB     6.469303618078782E-5
Wikipedia:Deletion review|deletion review       6.0860126043801695E-5
Template:R from title without diacritics|R from title without diacritics        5.2586298562576735E-5
user:freakofnurture|... 5.2586298562576735E-5
User:AWeenieMan/furme|FURME     4.373721764900027E-5
Wikipedia:Articles for deletion/PAGENAME (2nd nomination)       4.2420229957376414E-5
```



As we can see, we obtain the same result for Spark and Hadoop.