UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering / Computer Engineering

Cloud Computing

PageRank

Project Documentation

Leo Maltese
Giulio Federico
Clarissa Polidori
Leonardo Poggiani

Academic Year: 2020/2021

Table of Contents

1 Introduction

1.1 Description of PageRank

PageRank is a measure of the quality of a web page and is used to propose the results of a search to the user in descending order of priority.

> PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites. [1]

In this project, we implemented one version of the PageRank algorithm through the Hadoop framework and another version using the Spark framework, in both Python and Java.
We also ran tests to verify the execution and convergence times of the two solutions, so that we could compare them.
The code of the various implementations together with useful comments to understand it can be found at `https://github.com/CloudComputing-PosamanGroup/TonellottoProject`.

1.1.1 Theoretical formula and algorithm

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

Where:

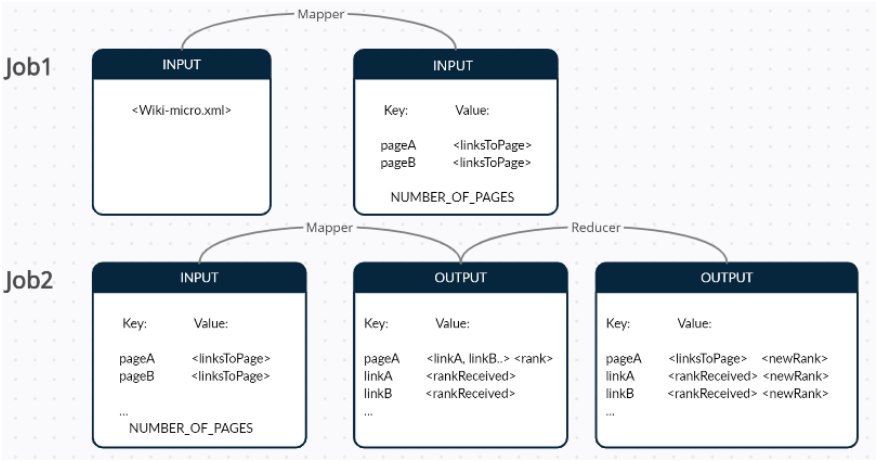
- **P(n)** is the new value PageRank.
- **P(m)** is the old value PageRank.
- α is the *random jump factor*, usually equals to 0.15.
- **N** is the total number of pages.
- **1-α** is the *damping factor* that is the reciprocal of the *random jump factor*, therefore usually set to 0.85.
- **L(n)** set of pages linked to n.
- **C(m)** is the set of pages with which m is connected.

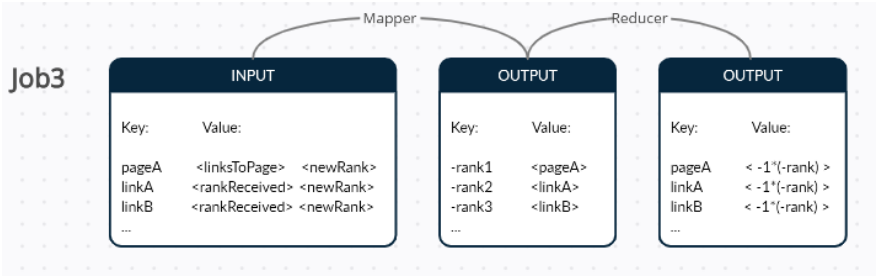
[1] "Facts about Google and Competition".

2 | Page Rank in Hadoop

We decided to structure the algorithm in 3 different jobs: parsing of the XML to page with links, computations for the new page ranks and sorting.

- In the first job we parse each page and get the list of its outlinks, in addition we count the number of pages that we will need in the next job
- In the second job we calculate the rank of each page by applying the formula described above
- In the third job, we sort the pages by descending rank.





2.1 Page rank with Map Reduce

2.1.1 Job 1

Mapper

Algorithm 1 PageRank MapperJob1

```
1: procedure MAP(key, pageLine)
2:     N ← 0
3:     pageTitle ← Parse(pageLine)                    ▷ Build node
4:     N.outlinks ← Parse(pageLine)
5:     N.rankReceived ← -1
6:     N.pageRank ← -1
7:     Emit(pageTitle, N)
8:     NumberOfPages.increment()
```

Implementation choices

We decided not to implement the reducer for this phase, jobs of this type are called *Hadoop Map-Only Job*.
Sort and shuffle are responsible for sorting the keys in ascending order and then grouping values based on same keys. This phase is very expensive and if reduce phase is not required we should avoid it, as avoiding reduce phase would eliminate sort and shuffle phase as well. This also saves network congestion as in shuffling, an output of mapper travels to reducer and when data size is huge, large data needs to travel to the reducer.
The output of mapper is written to local disk before sending to reducer but in map only job, this output is directly written to HDFS. This further saves time and reduces cost as well. Also, there is no need of partitioner and combiner in *Hadoop Map-Only job* that makes the process fast.
In Hadoop, **Map-Only job** is the process in which mapper does all task, no task is done by the reducer and mapper's output is the final output.
The list of outlinks for each page is on the same line and counting the number of lines can be done directly in the mapper but to do this we had to take advantage of a special tool in the Hadoop framework: **Counters**.
Counters in Hadoop are a useful channel for gathering statistics about the MapReduce job like for quality control or for application-level and are also useful for problem diagnosis.
Each counter in MapReduce is named by an "Enum" and it has a long for the value. In addition to built-in counters, Hadoop MapReduce permits user code to define a set of counters then it increment them as desired in the mapper or reducer. Like in Java to define counters it uses **enum** .
A job may define an arbitrary number of 'enums' each with an arbitrary number of fields. The name of the enum is the group name and the enum's fields are the counter names. Hadoop counters are maintained by each task attempt and periodically sent to the application master so they can be globally aggregated.

The custom counters we have defined are the following:

- NUMBER_OF_PAGES
- CONVERGENCE

2.1.2 Job 2

Mapper

Algorithm 2 PageRank MapperJob2

```
1: procedure MAP(pageTitle, N)
2:     rank ← N.PageRank
3:     outlinks ← N.outlinks
4:     if rank == -1 then                                   ▷ First iteration
5:         rank ← 1/numberPages
6:     Emit(pageTitle, N)                                   ▷ Emit parent node
7:     M ← 0
8:     if outlinks ! = 0 then                               ▷ Build children nodes
9:         internal_outlinks ← Parse(outlinks)
10:        countOutlinks ← Count(internal_outlinks)
11:        for each link ∈ internal_outlinks do
12:            M.rankReceived ← rank/countOutlinks
13:            Emit(link, M)                                ▷ Emit child node
```

Reducer

Algorithm 3 PageRank ReducerJob2

```
1: procedure REDUCE(pageTitle,[N1,N2..Nn])
2:     sum ← 0
3:     N ← 0
4:     for each M ∈ [N1,N2..Nn] do
5:         if M.rankReceived == -1 then                     ▷ Is parent node
6:             N ← M
7:         else
8:             sum+ = M.rankReceived
9:     newPageRank ← α · 1/numberOfPages + (1 − α) · sum
10:    N.pageRank ← newPageRank
11:    Emit(pageTitle, N)
```

Implementation choices

The second job takes care of implementing the actual algorithm on parsing the pages obtained at the output of the previous job. To avoid an intermediate job that initializes the ranks to the starting value $\frac{1}{Numberofpages}$, we put the value -1 in the *rankPage* field of the page node, so that we can go directly to the mapper of job 2 to discriminate whether we are on the first iteration or not. We can use the value of the number of pages that was calculated at the previous job and retrieved and saved in the configuration from the main.

We discussed the possibility of leveraging the reducer as a combiner to perform a local computation before sending the data over the network to the reducer. Because of how we decided to structure our algorithm the adoption of a combiner does not make sense. The combiner can only be used in the case where the reduce function is both commutative and associative. This is because the values are combined locally before remixing in arbitrary order, and this is not our case. Since we don't have any prior knowledge of the amount of values for each key, thinking

about the adoption of a partitioner in this case is completely inappropriate. This job is executed iteratively until a given level of convergence is achieved. We also exploited in this job a custom counter that was mentioned in the previous paragraph. We use the *CONVERGENCE* counter, which is incremented in the reducer if the new rank calculated for a page deviates by a certain threshold from the previous rank. The counter is read by the driver process, and if its value is different from 0, then we continue to iterate, otherwise we stop. In any case after 10 iterations the algorithm stops.

2.1.3 Job 3

Mapper

Algorithm 4 PageRank MapperJob3

```
1: procedure MAP(pageTitle, N)
2:     Emit(−N.PageRank, pageTitle)
```

Reducer

Algorithm 5 PageRank ReducerJob3

```
1: procedure REDUCE(rank, [title1, title2.. titleN])
2:     for each title ∈ [title1, title2..titleN] do
3:         Emit(title, −rank)
```

The third job takes advantage of the data sorting done by Hadoop in the *shuffling and sorting* phase before the data arrives at the reducer. This sorting is done by Hadoop in ascending order on the key, while we want a descending order on the rank that is, in our case, the value. In the mapper we are going to put the value of the rank, changed of sign as the key, and the title as the value, while in the reducer we exchange them again to have the output in the correct format that can be given in input to the mapper for the next iteration of the job.
Using multiple reducers we will get a local ranking on multiple files. A simple merge or append wouldn't be enough to retrieve the overall ranking. We therefore use only one reducer to have a global ranking, also because any additional reducer wouldn't do complex computation but would limit itself to writing only to files.

2.2 Java implementation

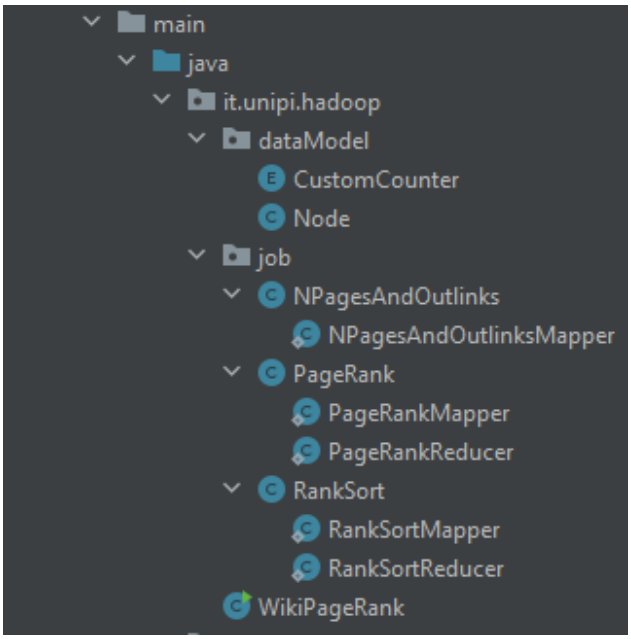
Below is shown the command with its parameters to start the PageRank algorithm:

```
# hadoop jar PageRankHadoop-1.0-SNAPSHOT.jar it.unipi.hadoop.WikiPageRank
randomJumpingFactor inputFile outputFile
```

Descrizione dei parametri:

- *randomJumpingFactor*: The factor that is used in the formula to mimic the "random" behavior of a user navigating between pages (recommended 0.15)
- *inputFile*: file to be analyzed by the Page Rank algorithm
- *outputFile*: path where the result of the Page Rank algorithm is saved

2.2.1 Project structure



- ***dataModel* package**: Contains some utility functions
  - ***CustomCounter* class**: Contains the custom counters that we have defined to count the pages to analyze and for the convergence.
  - ***Node* class**: Represents the element page with all its possible information. In particular it can contain the list of outlinks, actual page rank, and a field with the part of the page rank received from parent node. The latter is also useful to discriminate if the node is what we usually call *parent node* or *child node*[1]. Parent Node will have the value of this field equal to -1 for all the time. Viceversa, the child node will always have the outlinks list at -1.
- ***job* package**: This package contains the three map reduce jobs described in the previous paragraph.

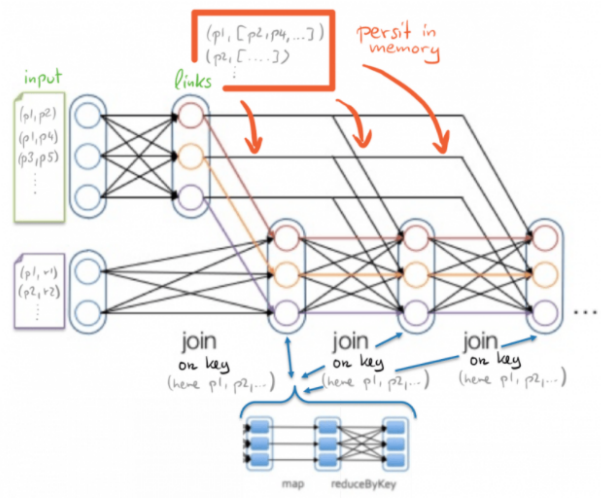
[1]With parent node we mean a line of XML file, that is a page with all its outlinks associeted, child node is a link

  - ***NPagesAndOutLinks* class**: Contains the class we use to extend the class mapper of job 1
  - ***PageRank* class**: Contains the classes we use to extend the mapper and the reducer classes of job 2
  - ***RankSort* class**: Contains the classes we use to extend the mapper and the reducer classes of job 3

- ***WikiPageRank* class**: Contains the main method with all the configurations.

3 | Page rank in Spark

The same algorithm just seen was implemented via the Spark framework using both the Python and Java languages.



Spark Algorithm

Algorithm 6 PageRank Spark

```
1: inputRDD ← sc.inputFile
2: N ← inputRDD.count()
3: pageAndOutlinks ← inputRDD.map(parseInputLine())
4: pagesRank ← pageAndOutlinks.map(assignInitialRank())
5: for each Iteration do
6:     pagesContribution ← pageAndOutlinks.join(pagesRank).flatMap(computeContributions())
7:     pagesRank ← pagesContribution.reduceByKey(addOperator).mapValues(computeRank())
8: pagesRankOrdered ← pagesRank.sort()
9: pagesRankOrdered.saveAsFile()
```

As you can see from the pseudo-code just above, the logic is very similar to that used in the solution in Hadoop. One of the few differences is in the use of *"join"* and *"flatMap"*. The Join allows us to get a tuple:
('title', ([outlinks], pageRank))
So it allows us to build the same *"Node"* structure used during the algorithm in Hadoop, associating to each page its own outlinks and rank. The flatMap, a slightly different transformation than the map, allowed us to remove the *"innermost brackets"* of the previous tuple in order to calculate the contributions of the various pages and obtain a list of tuples of this type:

('title', contribution)

Come già visto già in Hadoop, per ogni pagina questi "contribution" verranno sommati (*reduceByKey()*) e utilizzati per calcolare il pageRank di ogni pagina (*mapValues()*).

Like the implementation in hadoop, we are going to iterate the algorithm 10 times. In this solution with Spark we have not implemented the stop criterion of the algorithm, however it can be obtained in a similar way to the one used by hadoop, exploiting the distributed shared variables that the Spark framework offers: **Accumulators**.

3.1 Implementation with Python

Below is shown the command with the relative parameters to start the python PageRank algorithm:

```
# spark-submit SparkPageRank.py iterations inputFile outputFile
```

Parameters Description::

- *iterations*: Number of iterations PageRank must perform.
- *inputFile*: file to be analyzed by the Page Rank algorithm.
- *outputFile*: folder that should contain the results of the algorithm.

3.2 Implementation with Java

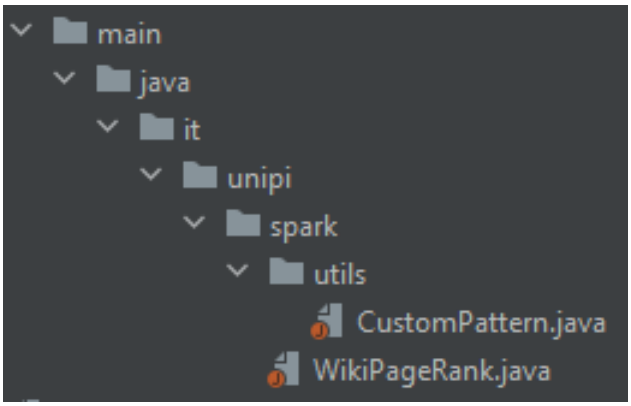
Below is shown the command with the relative parameters to start the java PageRank algorithm:

```
# spark-submit -class it.unipi.spark.WikiPageRank SparkJavaPageRank-1.0-SNAPSHOT.jar
iterations inputFile outputFile
```

Descrizione dei parametri:

- *iterations*: Number of iterations PageRank must perform.
- *inputFile*: file to be analyzed by the Page Rank algorithm.
- *outputFile*: folder that should contain the results of the algorithm.

3.2.1 Project structure



- ***CustomPattern* class**: Implements all the methods for parsing the lines of the input XML file to get the title and outlinks of the page
- ***WikiPageRank* class**: it is the main class that implements all the methods useful for the page rank algorithm and the main method

3.3 Optimizations

In order to optimize the execution of the PageRank algorithm, we decided to use a native Spark construct that allows us to persistently maintain an RDD so that we don't have to retrieve it every time from different nodes, the *.cache()* method.

```
# build an RDD from input file
lines = sc.textFile(address + str(sys.argv[1]))

# map transformation on each line
# example of tuple⟶('title', ['outlink1', 'outlink2', ....., 'outlinkN'])
titles = lines.map(lambda page: parsePages(page)).cache()
```

Figure 3.1: Use of the .cache() method

We decided to cache the RDD containing the titles because it is reused many times, even within the for that performs the PageRank operations and it was therefore computationally very expensive to have to retrieve it every time.

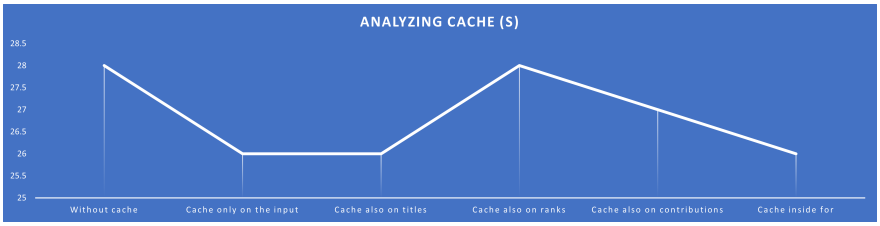
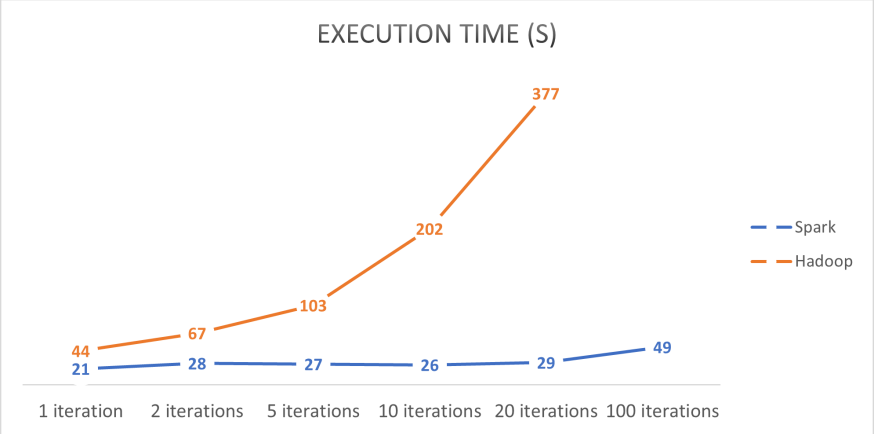


Figure 3.2: The execution refers to the Java implementation, but is equivalent in Python

4 Conclusions

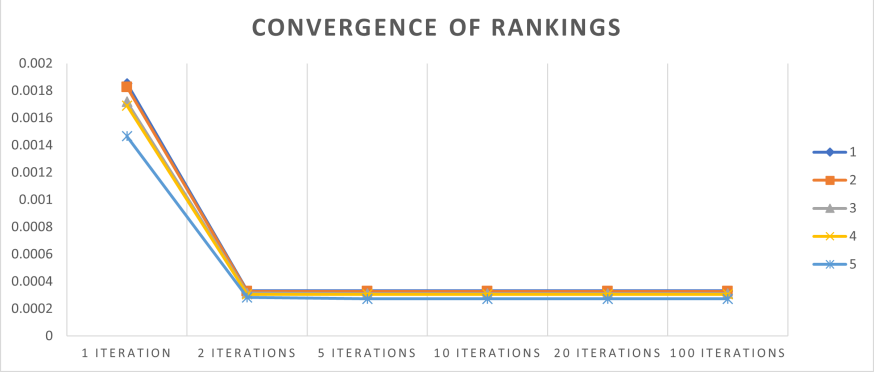
4.1 Experimental results

We analyzed runtime and page rank convergence on Hadoop and Spark and got very different results.



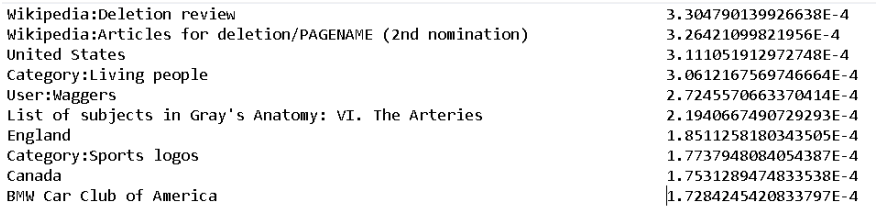
From the graph we can see that while Spark is able to tolerate up to 100 iterations without particular effort, the execution time of Hadoop explodes, until it becomes difficult to calculate for 100 iterations.

This however does not seem to be an excessive problem in the PageRank performed with the test dataset, as convergence is still obtained in a relatively low number of iterations.
However, if we think of a larger dataset on which more iterations have to be performed, this becomes a problem to be taken into account.



As we can see from the graph, convergence for Spark (at least for the first five positions) is achieved in a very low number of iterations (between two and five).

Some experiments conducted by calculating the difference in the rankings between one execution and the other for the entire ranking confirmed this result, setting the number of iterations at 4 after which convergence is reached in Hadoop.



Wikipedia:Deletion review	3.304790139926638E-4
Wikipedia:Articles for deletion/PAGENAME (2nd nomination)	3.26421099821956E-4
United States	3.111051912972748E-4
Category:Living people	3.0612167569746664E-4
User:Waggers	2.7245570663370414E-4
List of subjects in Gray's Anatomy: VI. The Arteries	2.1940667490729293E-4
England	1.8511258180343505E-4
Category:Sports logos	1.7737948084054387E-4
Canada	1.7531289474833538E-4
BMW Car Club of America	1.7284245420833797E-4

The image shows the output provided by the command:

```
# hadoop fs -head HadoopOutput/iterazioni_10/finalPageRank/part-r-00000.
```