Cloud Computing
A.A 2020/2021

PageRank

Bruno Casu Federico Fregosi Giuseppe Martino
Laura Lemmi Nicola Barsanti

July 2021

Contents

1 Overview

This project focused on designing a MapReduce implementation (both in Hadoop and Spark) of the PageRank algorithm used by Google Search to rank web pages in their search engine results. Initially, a pseudocode implementation and the design assumption are presented. Then, the Hadoop and Spark actual implementation are shown and explained and finally, the validation results obtained by running the applications with different configurations are provided.

2 Introduction

2.1 PageRank

PageRank is a measure of web page quality based on the structure of the hyperlink graph. Although it is only one of thousands of features that is taken into account in Google's search algorithm, it is perhaps one of the best known and most studied. According to Google: *"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"*.

A vivid way to illustrate PageRank is to imagine a random web surfer: the surfer visits a page, randomly clicks a link on that page, and repeats ad infinitum. PageRank is a measure of how frequently a page would be encountered by our tireless web surfer. More precisely, PageRank is a probability distribution over nodes in the graph representing the likelihood that a random walk over the link structure will arrive at a particular node. Nodes that have high in-degrees tend to have high PageRank values, as well as nodes that are linked to by other nodes with high PageRank values. This behavior makes intuitive sense: if PageRank is a measure of page quality, we would expect high-quality pages to contain "endorsements" from many other pages in the form of hyperlinks. Similarly, if a high-quality page links to another page, then the second page is likely to be high quality also.

Formally, the PageRank PR of a page $\mathrm{p_i}$ is defined as follows:

$$PR(p_i) = \alpha\frac{1}{N} + (1-\alpha)\sum_{p_j \in L(p_i)} \frac{PR(p_j)}{C(p_j)}.$$

Given:

- a page p_i among the total N nodes (pages) in the graph;
- the set of pages $L(p_i)$ that link to p_i;
- and the out-degree $C(p_j)$ of node p_j;
- the random jump factor α;

PageRank can be computed iteratively. In this way, at t=0, the initial probability distribution is assumed

$$PR(p_i, 0) = \frac{1}{N}$$

For the next time steps, the computation become

$$PR(p_i, t+1) = \alpha \frac{1}{N} + (1-\alpha) \sum_{p_j \in L(p_i)} \frac{PR(p_j, t)}{C(p_j)}.$$

2.2 Assumptions

In order to test the program, we have computed the "importance" of various Wikipedia pages/articles as determined by the PageRank metric, and present them in decreasing order of importance. This is done using the given `wiki-micro.txt` real world dataset, a collection of pre-processed simple Wikipedia corpus in which the pages are stored in an XML format. Each page of Wikipedia is represented in XML as follows

```
<title>web page name</title>
...
<revision optionalVal="xxx">
    ...
    <text optionalVal="yyy">page content</text>
    ...
</revision>
```

The pages have been "flattened" to be represented on a single line. The body text of the page also has all newlines converted to spaces to ensure it stays on one line in this representation. This makes it easy to use the default InputFormat, which performs one map() call per line of each file it reads. Links to other Wikipedia articles are of the form [[page name]]. Starting from this input file, first the hyperlink graph is costructed and then the PageRank is computed for each node. The random jump factor α (sometimes called the "teleportation" factor) and is usually set to 0.15, estimated from the frequency that an average surfer uses his or her browser's bookmark feature. Finally, the proposed implementation does not rely upon the iteration convergence measured as the difference between consecutive PageRank values as stopping criteria: for our scope, a fixed number of iterations, chosen by command line, are performed.

3 Hadoop implementation

3.1 Pseudocode

Our solution is made of multiple MapReduce stages, illustrated in the following

3.1.1 Stage 1: Node counter

In this first phase, the information in the input file are parsed and only the meaningful information are taken into account i.e. the title of the page and the outgoing edges. Then, for each title we just emit a constant key (we are not interested in the title itself in this phase) and a value 1. The result of this phase, after the reduction, is the total number of nodes N in the dataset.

Node counter

```
class Mapper
    method Map(rowid $a$, row $r$)
        if "<title>...</title>" in row $r$ then
            EMIT(const $key$, count 1)
        end

class Reducer
    method Reduce(key $k$, counts[ $c_1, c_2, ...$ ])
        $sum \leftarrow 0$
        for all count $c$ in counts[ $c_1, c_2, ...$ ] do
            $sum \leftarrow sum + c$
        EMIT(key $k$, $sum$)
```

3.1.2 Stage 2: Graph construction

Here, the hyperlink graph is built: for each title, we are interested in the list of link reachable from it (outlinks). In order to have those information, we just check for pages contained in double square bracket inside the text field of a line, and we collect them inside an adjacency list. Note that we emit something only if there is an actual outlink in the page: this means that if a page has no outlinks, it is discarded from the graph because otherwise his rank would constantly grow. Then, before the reducer phase, we used a Setup method in order to configure the initial PageRank ($1/N$) of each page. Finally, we emit as output the adjacency list and the initial PageRank for each title.

Stage 2: Graph construction

```
class MAPPER
    method MAP(rowid $a$, row $r$)
        if "<title>page_title </title>" in row $r$ then
            $title \leftarrow$ page_title
            if "<title>content </title>" in row $r$ then
                for all "[[pagelink]]" in content do
                    $pagelink \leftarrow$ "pagelink"
                    EMIT($title$, $pagelink$)
            end
        end

class REDUCER
    $pagerank \leftarrow 0.0$
    method SETUP()
        $N \leftarrow$ conf.getInt("N")
        $pagerank \leftarrow 1/N$
    method REDUCE(term $t$, adjacency_list[ $l_1, l_2, \ldots$ ])
        $outputvalue \leftarrow pagerank + adjacency\_list$
        EMIT(term $t$, $outputvalue$)
```

3.1.3 Stage 3: Pagerank algorithm

In this stage, the calculation of the PageRank for each page is actually computed and sent in output. An iterative call of this phase must converge to the correct PageRank. In addition to the new PageRank values that we compute, we must forward to the next iteration also the graph structure in order to not lose it. To overcome this problem, we implemented a new class `ReducerReceivedData` composed only 2 value: a boolean that is used to distinguish between PageRank records (true) and Graph records (false), and the actual value referred to the key.

```
class ReducerReceivedData
    Boolean type
    String value

class Mapper
    method Map(nodeid $n$, node $Node$)
        $weight \leftarrow Node.pagerank$ / $Node.adjacency\_list$
        $adjacency\_list \leftarrow Node.adjacency\_list$
        $rrd.value \leftarrow adjacency\_list$
        $rrd.type \leftarrow$ 'false'
        EMIT(nodeid $n$, ReducedReceiverData $rrd$)
        for all nodeid $m$ in Node.adjacency_list do
            $rrd.type \leftarrow$ 'true'
            $rrd.value \leftarrow weight$
            EMIT($m$, $rrd$)

class Reducer
    method Setup()
        $alpha \leftarrow$ conf.getInt("alpha")
        $a \leftarrow alpha$
    method Reduce(nodeid $n$, ReducerReceivedData [$rrd_1$, $rrd_2$, ...])
        sum $\leftarrow 0$
        for all values $c$ in ReducerReceivedData[ $rrd_1$, $rrd_2$, ... ] do
            if ($c.type$ == 'false')
                $adjacency\_list \leftarrow values.value$
            end
            else ($c.type$ == 'true')
                $sum \leftarrow sum + c.value$
            end
            if ($adjacency\_list$ != null)
                $pagerank \leftarrow a/N + (1-a)*sum$
                $outputvalue \leftarrow pagerank + adjacency\_list$
                EMIT(nodeid $n$, $outputvalue$)
            end
```

3.1.4 Stage 4: Sorting algorithm

Finally, in this stage, all the nodes are sorted in relation to their PageRank in descending order. This is done thanks to a custom WritableComparable class, called after the reducer phase, that allows us to reverse the natural order (ascending) in which the reducer collect values.

```
class PageRankComparable
    compare()

class Mapper
    method Map(Text title, Text value)
        pagerank ← value.split(" ")[0]
        EMIT(pagerank, title)

class Reducer
    method Reduce(DoubleWritable key, Text titles[ t_1, t_2, ... ])
        for all value v in titles [ t_1, t_2, ... ] do
            EMIT(v, key)
```

3.2 Performance Evaluation

As previously mentioned, we did not deal with measuring the convergence of the algorithm which is notoriously achieved with a low number of iterations (consistently with what found in the original PageRank paper: "*on a 322 million link database, convergence to a reasonable tolerance is obtained in roughly 52 iterations* "). Instead, we focused on analyzing the performance variation, in terms of execution time, using a different number of reducers for the various stages. In particular, we tested our implementations with 5, 10 and 15 iterations of the third stage, using as input the wiki-micro.txt file with the following configurations:

1. First configuration:
    - **stage 1**: 1 reducer;
    - **stage 2**: 1 reducer;
    - **stage 3**: 1 reducer;
    - **stage 4**: 1 reducer.
2. Second configuration:
    - **stage 1**: 1 reducer;
    - **stage 2**: 3 reducer;
    - **stage 3**: 3 reducer;
    - **stage 4**: 1 reducer.

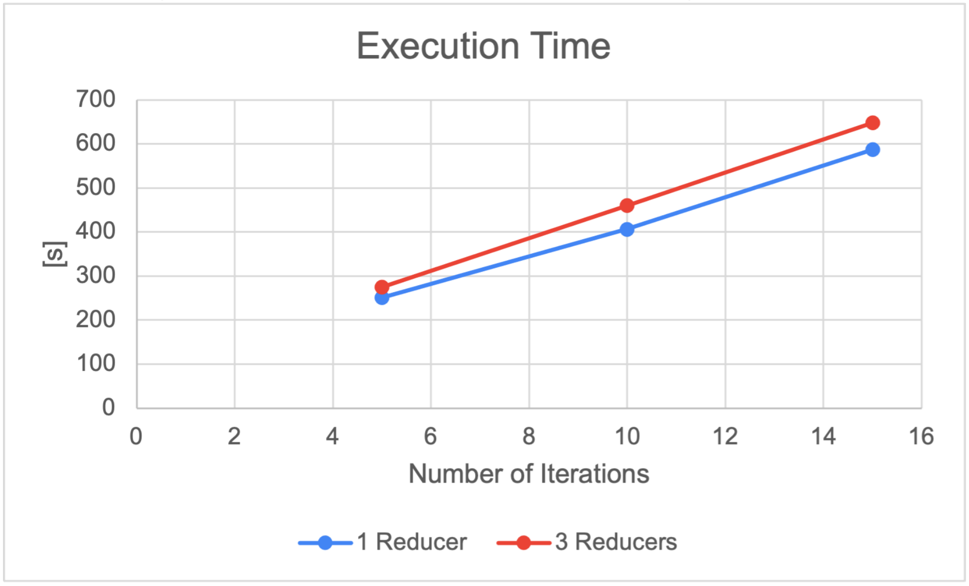
Note that for the first and fourth stages, the number of reducer is static set to 1: this because in the first stage we have only 1 key, so it would be useless to use more then a single reducer, while in the fourth stage the use of more then 1 reducer would result in a sort limited to the result of each reducer, independently from the others. The results of the tests are shown in the following table:

As we can see also from the following picture (Figure 1), the difference between the use of 1 or 3 reducers for the phases 2 and 3, is almost nothing.

Table 1: Execution times for the 2 configuration

		Number of iterations		
		5	10	15
Configuration	1	251,2	406,2	587,2
	2	274,2	460,2	648,2

Figure 1: Performance difference for the 2 configurations



4 Spark implementation

4.1 Pseudocode

```
procedure COMPUTE_RANK(rank $r$)
    return $a/N$ + (1 - $a$) * $r$

procedure COMPUTE_CONTRIBUTIONS(title $t$, outlinks $o$, rank $r$)
    contributions_list←[$t$,0]
    for all link in o do
        contributions_list.add((link,$\frac{r}{lenght(o)}$))
    return contributions_list

procedure PAGERANK(InputFile $f$, alpha $a$, nIterations $ni$)
    inputRDD ← createRDD($f$)
    $N$ ← inputRDD.count()
    graph ← createGraph(inputRDD).filter(lenght(graph[outlinks]) $\geq$ 1)
    pagetitle_keys←graph.keys()
    pageranks ← (graph[title],$\frac{1}{N}$)
    for all i in ni do
        complete_graph ← join(graph,pageranks)
        contributions ← compute_contributions(complete_graph[title],complete_graph[outlinks],
            complete_graph[rank]).filter(contributions[title] is in
            pagetitle_keys))
        pageranks ← contributions.reduceandSumvalues().compute_rank(contributions[rank])

    sorted_pageranks ← pageranks.sortBy(pageranks[rank])
```

An aspect to be taken into account is that we have only considered pages included in the tag `<title></title>`. However, when contributions are calculated, pages are also generated that are not part of those in the initial graph, so they must be removed. Then from the RDD of contributions, composed by pairs in the form (title, contribution), pages that are not part of the initial list of pages considered, i.e. those contained in the RDD ***graph***, have been deleted.

4.2 Main Optimizations

4.2.1 Caching

A first improvement in performances comes from the persistence of the RDD ***graph*** in memory. As it is evident from the pseudocode, at each iteration, there is a need to retrieve the graph structure (which does not change after the first calculation), so in order not to reload the RDD graph at each iteration, once the graph has been calculated, via the **cache()** method the nodes that store the RDD graph partition, save it in memory to reuse it at subsequent iterations

4.2.2 Broadcast Variables

At each iteration it is necessary to use a static variables: the variable ***pagetitle_keys*** (list of pages whose title is contained in the `<title></title>` tags) in order to eliminate those pages that are not part of this list. Being a read-only variables, we decided to keep it cached on each machine rather than shipping a copy of it with tasks at each iteration.

4.2.3 Performance evaluation

In the following graph we can see the performance gain obtained by using the optimizations of the previous paragraphs. To stabilise the results obtained, the values considered are an average of 10 simulations.

		Number of iterations		
		5	10	15
Optimizations	yes	51,52	66,52	96,91
	no	55,56	71,85	110,67

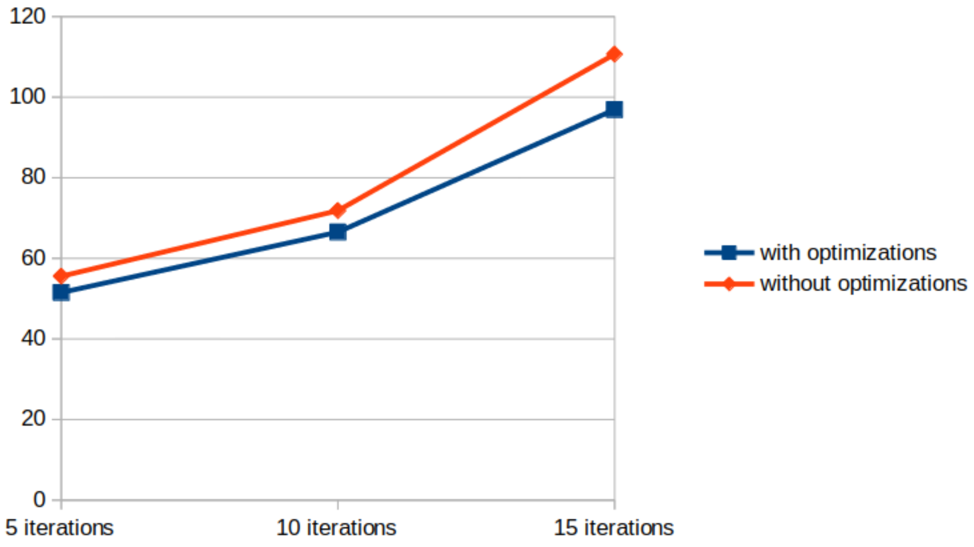


Figure 2: Performance difference using or not caching and broadcast variables