UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering

Cloud Computing

PageRank

Project Documentation

TEAM MEMBERS:
Federica Baldi
Daniele Cioffo
Edoardo Fazzari
Mirco Ramo

Academic Year: 2020/2021

Contents

1 — PageRank

1.1 Pseudocode

The pseudocode presented in this chapter should be considered as a *generic* implementation of the **PageRank Algorithm**, a detailed and framework-based version of the procedures is shown in the next chapter.

Algorithm 1 PageRank

```
1: procedure Counting Nodes(dataset d)
2:     N ← d.countNodes()
3:     return N

4: procedure Graph Construction - Parsing(dataset d)
5:     nodesList new AssociativeArray
6:     for all page in d do
7:         title ← page.getTitle()
8:         outgoingEdges ← page.getOutgoingEdges()
9:         nodesList.add({title, outgoingEdges})
10:    return nodesList

11: procedure Compute PageRank(nodesList NL, numberOfNodes N, nOfIterations NI)
12:    NL.addInitPageRankToNodes(1/N)
13:    for i in range(NI) do
14:        NL.Map()
15:        NL.Reduce()
16:    NL.sortByPagerank()

17: procedure Map(title t, {outgoingEdges oe, pagerank p})
18:    for all e in oe do
19:        EMIT(e, p/oe.length)
20: procedure Reduce(title t, pagerankContributions [p1, p2, ...])
21:    damping ← 0.8
22:    sum ← 0
23:    N ← numberOfNodes
24:    for all p do in pagerankContributions
25:        sum ← sum + p
26:    pagerank = (1−damping)/N + damping * sum
27:    EMIT(t, pagerank)
```

2 — Spark Implementation

2.1 Introduction and Main Steps

The goal of this work is to compute the page rank value for a set of nodes (pages) stored in an input file. For each node, the page rank is determined by the number of **ingoing** links of the considered node, but we can get from the inputs only the list of **outgoing** links for each node. Thus, the steps to compute the page ranks are the following:

1. Decode the input file to get the title of the page and the set of outgoing links
2. Assign to each node the initial page rank value of 1/N (N is number of considered nodes)
3. Compute the contribution that the considered node gives to the pages addressed by its outgoing links
4. Discard the contributions destined to pages that are out of the scope (linked nodes that are not in the original set of considered nodes)
5. Sum the contributions received by each node, apply the page rank formula
6. Go to 3 for a certain number of iterations, then:
7. Sort results by page rank and output them in a text file

2.2 Pseudocode and DAG

Starting from these steps, we can write the Spark pseudocode considering that:

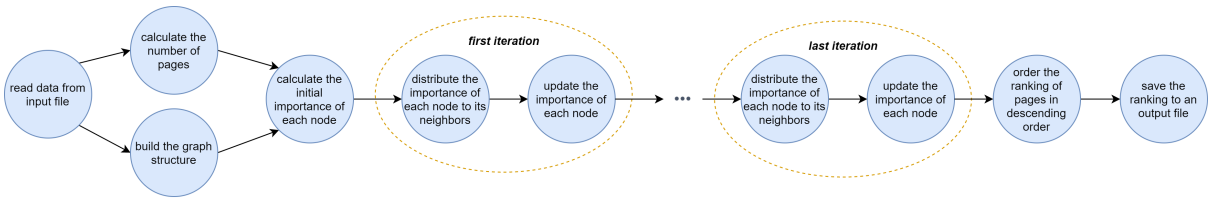
- We decided to save the network structure and the computed page ranks into 2 different RDDs, because the first one will be static for the whole application lifespan, so it is better to modify frequently a small RDD instead of a big one the same number of times
- Every node contributes to itself with a 0 factor, this record is required not to lose the nodes that have no incoming links
- To avoid spurious links, for which concerns the wiki-micro.txt dataset we consider only the links in the text section

Algorithm 2 PageRank

```
1: procedure SPREAD RANK(node n, outgoingLinks ol, rank r)
2:     rankList ← [(n, 0)]
3:     if ol is not empty then
4:         for all link in ol do
5:             rankList.addContribution(link, r/ol.length)
6:     return rankList

7: procedure PAGERANK COMPUTATION(damping d, inputFile if, outputFile of, iter it)
8:     inputDataRDD ← getFromInput(if)
9:     numberOfNodes ← inputDataRDD.count()
10:    nodes ← decodeInput(inputDataRDD)
11:    pageranks ← (nodes[title], 1/numberOfNodes)
12:    for i in range(it) do
13:        contributionList ← join(nodes, pageranks).spreadRank(title, outgoingEdges, rank)
14:        consideredContributions ← contributionList.filter(receiver.title is in nodes
15:        pagerank ← consideredContributions.sumByKey().computeRanks((1−d/numberOfNodes)+
   d ∗ totalContributions)
16:    sortedPageRank ← pageranks.sortBy(pagerank)
```

Following the pseudocode above, we implemented the algorithm in both Python and Java (the code can be found on the GitHub repository).
Below is the DAG of the actions involved in the algorithm and the dependencies between them. It is important to note that each action (DAG node) can be subdivided into smaller actions. For instance, once we have the ranking of the previous iteration (or the initial one if we are at the first iteration), the calculation of the importance of each node is independent of all others; the same is true for the distribution of the importance of each node to its neighbors.



2.3 Performance Optimization

2.3.1 Partitioning

Since Spark uses 2 worker nodes in order to execute the applications, and due to the independence of each input record from the others, the *textFile* action is explicitly configured to use 2 parallel readers, so that to speed up this operation and not to exchange data across the network on the first partitioning (again we specify we want two partitions for the target RDD). Thanks to that and to the other optimizations reported in the following paragraphs, we can have **co-partitioned** join operations, which means no need for shuffling.

2.3.2 Caching

Once every worker has computed its own partition of the graph structure (nodes + outgoing links), it will be going to need this static data multiple times across the iterative computation. For this reason, workers are asked to **cache** in memory the **nodes** RDD (containing the graph

structure), so that to avoid the recomputing of it across the iterations. This choice lead to a sensible performance boosting and required time reduction (more than 3 times less in a run with 5 iterations).
With small datasets, partitions of this RDD can fit completely in memory. On the contrary, huge datasets could degrade the performance gain, but in this case the disk persistency of the RDD is still a better choice rather than the full re-computation from scratch.

2.3.3 Shuffle Minimization

Right after the flatMap operation applied to the joined RDD, we obtain a RDD of (page, contribution) records. Since we want to consider the initial pages only, we must consider the contributions destined to nodes inside the set of considered one. At this point we only have a (cached) nodes RDD and the contributions RDD, so we could:

- Join the 2 RDDs through an inner join, which will automatically keep only the contributions that match a page included in the nodes RDD
- Take the keys of the *nodes* (obtaining the keys list) and filter out contributions whose receiver is out of this *keys* list

The first approach is obviously easier to develop but requires a **many-to-many** shuffling of a **very big data structure** (the contributions RDD); on the contrary with the second approach we can simply apply a filter to local data, keeping only records matching the predicate (belonging to the keys list): in this way we need **no data exchange** between workers. The main drawback is the fact that Spark does not allow to reference a RDD inside a transformation, thus we need to **explicitly collect** the keys RDD into a keys list.
The performance evaluation of the 2 alternative approaches shows a very significant improvement with the collect+filter strategy (till 2.5 times less)

2.3.4 Use of MapValues

Every time is possible, the use of the mapValues is suggested since it does not modify the keys of the records, thus speeding up the computation and **preserving data partitioning**. In our case, we exploit this property twice:

1. Once we have the *nodes* RDD, made of (title, outgoingEdges) pairs, in order to compute the *pageranks* RDD, it is pointless to re-read the titles from the input file and associate to each of them the initial value. A much faster approach is to take the *nodes* RDD and remap the keys (titles) associating to them the initial value through a mapValues, so that to keep also the partitioning (every worker has a partition on nodes with the same pages that are in the corresponding partition of *pageranks*. The collateral benefit is that the first join in the iteration **does not need an actual shuffling**.
2. Right after the reduceByKey we obtain a list of (title, sum of contribution) pairs; from this data structure we need to compute the page ranks. Since this computation does not affect the key (does not depend on the title of the page), we simply use a mapValues

2.4 Performance Evaluation

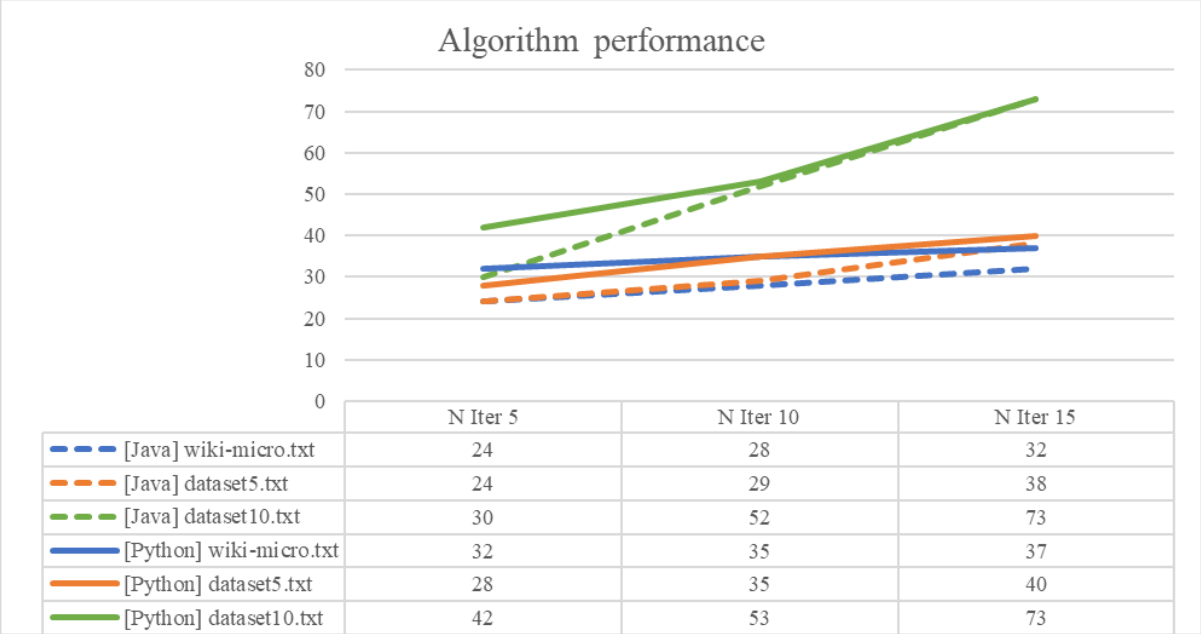
To evaluate the performance of our algorithm, we considered three input files, with different numbers of pages. Precisely, we used the following files:

1. **wiki-micro.txt**, which contains 2427 pages. This file is a pre-processed version of the Simple Wikipedia corpus in which the pages are stored in an XML format.

2. **dataset5.txt**, which contains 5000 pages. This file is a synthetic dataset we created using the same structure as wiki-micro.txt. Each node has a random value of outgoing edges between 0 and 10.

3. **dataset10.txt**, which contains 10000 pages. This file is a synthetic dataset we created using the same structure as wiki-micro.txt. Each node has a random value of outgoing edges between 0 and 10.

In the graph below are the results obtained with 5, 10 and 15 iterations of the algorithm written in Java and the one written in Python for each of the three files.
As can be seen, the execution time grows fairly linearly with the number of iterations and the number of pages within the file. Also, we could not see a significant difference between the two implementations; we can actually say that they have essentially the same performance



	N Iter 5	N Iter 10	N Iter 15
[Java] wiki-micro.txt	24	28	32
[Java] dataset5.txt	24	29	38
[Java] dataset10.txt	30	52	73
[Python] wiki-micro.txt	32	35	37
[Python] dataset5.txt	28	35	40
[Python] dataset10.txt	42	53	73