UNIVERSITÀ DI PISA

Master Degree in Artificial Intelligence and Data Engineering

Cloud Computing

Hadoop and Spark PageRank

Tommaso AMARANTE
Pietro CALABRESE
Francesco MARABOTTO
Edoardo MORUCCI
Enrico NELLO

Chapter 1

Hadoop Implementation

1.1 Main function

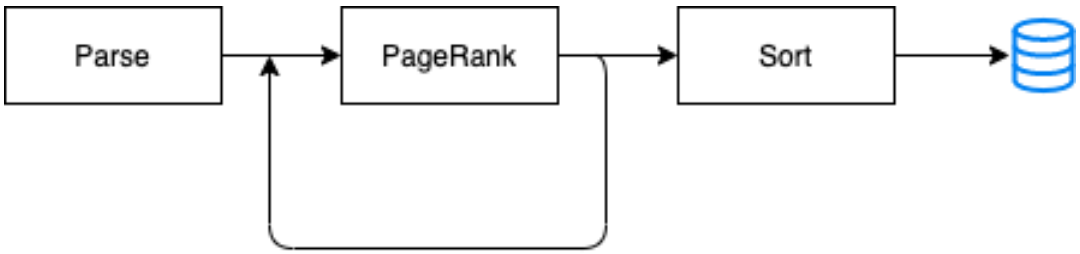

Figure 1.1: Stages execution.

The PageRank main function has to launch different jobs:

- **Parsing stage**: get the input *wiki-micro.txt* file and parse it
- **PageRank stage**: this is iterated several time, the number of iteration is chosen by the user at the launch of the application. This stage calculate the page rank.
- **Sort stage**: this stage order the output of PageRank stage.

1.2 Parsing Stage

Algorithm 1 Graph Building Mapper

```
1: procedure MAP(Key k, Page p)
2:     titlePage ← getTitle(p)
3:     outLinks ← getOutLinks(p)
4:     if outLinks == 0 then
5:         EMIT(title, "sinknode")
6:     else
7:         for all link in outLinks do
8:             EMIT(title, link)
```

Algorithm 2 Graph Building Reducer

```
1: procedure REDUCE(Title t, Links [l_1, l_2, ...])
2:     listOutlinks ← null
3:     for all l in Links do
4:         listOutlinks.add( l)
5:     EMIT(t, listOutlinks)
```

We processed every row of the input file *wiki-micro.txt* in order to obtain the following structure:
titlePage >> PageRank → *outlink*1 → *outlink*2 → ... → *outlinkN*.

In case there are no outlinks, then we have a *sink node*. We decided to add a fake outlink called **"sinknode"** in order to recognise this special nodes in the PageRank Stage.

For the number of **total nodes** we decided to calculate it as the number of rows processed by the mapper in this stage. Because of that the initial pagerank can only be calculated in the first PageRank Stage.

1.3 Page Rank Stage

Algorithm 3 Page Rank Mapper

```
1: procedure SETUP
2:     N ← totalNumberNodes
3:     H ← new AssociativeArray

4: procedure MAP(Key k, Node n)
5:     if n.outLinks.size == 0 then
6:         return
7:     if n.pageRank == 0 then
8:         n.pageRank ← 1/N
9:     EMIT (n.titlePage, {n.pageRank, n.outLinks})          ▷ Pass Graph Structure
10:    if n.outLinks(0) == "sinknode" then
11:        return
12:    fatherContribution ← n.pageRank / n.outlink.size
13:    for all outLink in n.outLink do
14:        H{outLink}.add(fatherContribution)

15: procedure CLEAN UP
16:    for all listNode ln in H do
17:        if ln.size == 1 then then
18:            EMIT (titlePage,ln)
19:        else
20:            sumPR ← 0
21:            for all node in ln do
22:                sumPR ← sumPR + node.pageRank
23:        EMIT (titlePage, sumPR)
```

Algorithm 4 Reducer PageRank

```
1: procedure SETUP
2:     N ← totalNumberNodes
3:     alpha ← getAlpha

4: procedure REDUCE(Title t, Nodes [n_1, n_2, ...])
5:     sumPR ← 0
6:     sendNode ← null
7:     for all node in Nodes do
8:         if node.outLinks.size > 0 then
9:             sendNode = node
10:        else
11:            sumPR ← sumPR + node.pageRank
12:    sendNode.pageRank ← alpha/N + (1-alpha) * sumPR
13:    EMIT (t, sendNode)
```

Differently for the standard PageRank algorithm we adopted some differences. First of all we implemented a custom **NodeWritable** class that implements *Writable*, we use this class to send values as [**PageRank, List<outlinks>**] between mappers and reducers.

In order to reduce the amount of data that have to be transmitted we decided to implement an *inMapper Combiner* in the *clean up* method, in the combiner we calculate the partial pagerank as the sum of all the contribution that appears in the same node.

Moreover we handled *SinkNodes* and *DanglingNode* in a special manner. For the SinkNodes we only emit the title page in order to maintain the graph structure, for the DanglingNode we decided to ignore them because their pagerank will be calculated automatically, since we want them in the final output it is not possible to completely omit them.

1.4 Sorting Stage

Algorithm 5 Sort Mapper

```
1: procedure MAP(Key k, Node n)
2:     EMIT (n.pageRank, n.title)
```

Algorithm 6 Sort Reducer

```
1: procedure REDUCE(PageRank pr, Titles [t_1, t_2, ...])
2:     for all all title in Titles do
3:         EMIT (title, pr)
```

In this stage we decided to implement a custom **DoubleComparator** class with a redefined *Compare* method that was previously defined in the **DoubleWritable** class. In the mapper we swapped keys with values in order to have the pagerank as a key, in this way with our redefined method we have the ordered output thanks to the *shuffle and sort* mechanism between mapper and reducer.

1.5 Performance

Normally we decided to run our application with 3 reducers in the ranking stage since this is the most computational expensive part. In order to collect some data we changed the number of reducers we used, we tried with a single reducer and then with 6 reducers, we collected the following data.

Number of reducers	Time
1 reducer	115955 ms
3 reducers	119975 ms
6 reducers	128449 ms

As we can see the process time increases with the number or reducers, this strange behaviour may be caused by the small size of the input file. In particular, the increase in execution time is due to the small size of the output from the mapper function, this brings a significant overhead in IO operations in order to split the output and to assign the chunks to the different reducers.

Thanks to the *inMapper combiner* in the PageRank Stage we managed to reduce the key-value pairs that have to be transmitted from the map to the reduce function. The initial pairs number was **47220**, with the inMapper combiner the total number is **36712**. This consist in a 23% traffic reduction. Implementing the combiner in the PageRank Stage, that is iterated several times, we obtained this traffic reduction for each iteration.

Chapter 2

Spark Implementation

2.1 RDD Persistance

We decided to store using the *cache()* function only for two RDDs:

- **input_data**: this RDD is obtained after the *textFile()* function since we use this RDD to count the total node and to compute the total *graph structure*.
- **rows**: this RDD contains the graph structure, we use this RDD multiple times.

2.2 Use of MapValues

This function operates only on the values and not on the entire record, this results in a performace boost. We use this function to set the initial page rank and to calculate the page rank with the total formula, in both cases we do not need to access the key but only the values.

2.3 DAG

In the following image we represents the DAG for two iteration of page rank.

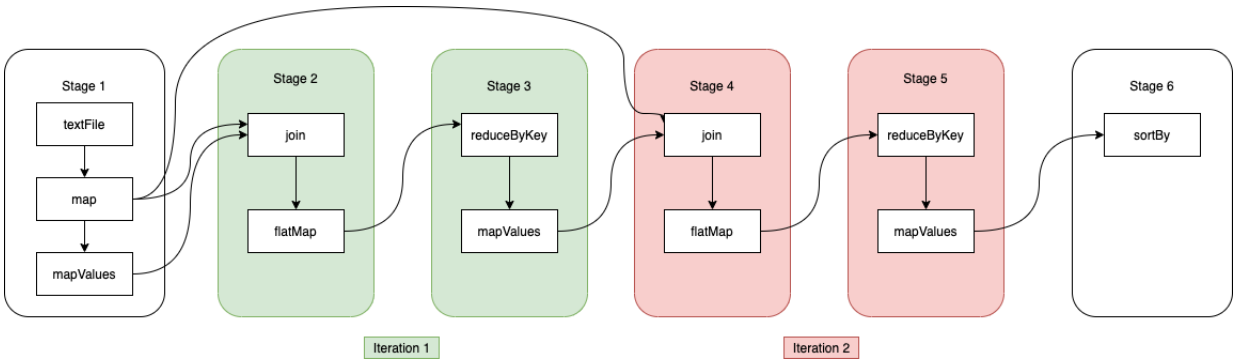

Figure 2.1: Spark DAG