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:         nodeLists.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 — Hadoop Implementation

2.1 Introduction

In this section a description of the MapReduce implementation of *Page Rank* is given. The algorithm is carried out in **three distinct steps**:

1. *Graph Construction* phase
2. *Page Rank Computation* phase
3. *Sorting* phase

We have decided to represent the structure of the graph through adjacency lists, so each node (that represents a page) will keep the list of outgoing edges as status information, as well as its ranking.

2.2 First Phase: Graph Construction

In this phase we parse the information in the input file taking the information that interests us, i.e. the title of the page and the outgoing edges. In addition to this, we take advantage of this phase to also perform the count of the graph nodes.

Algorithm 2 Graph Construction Mapper

```
1: procedure MAP(PageId id, Page p)
2:     title ← getTitle(p)
3:     outgoingEdges ← getOutgoingEdges(p)
4:     EMIT(title, outgoingEdges)
```

Algorithm 3 Graph Construction Reducer

```
1: procedure INITREDUCE(Configuration c)
2:     N ← c.numberOfNodes
3: procedure REDUCE(Title t, ListOfListOfEdges [e_1, e_2, ...])
4:     initialPageRank ← 1/N
5:     edges ← e_1
6:     EMIT(title, {initialPageRank, edges})
```

In *Algorithm 4* only e_1 is considered because the title is a unique identifier of the pages, so the list of input values will always consist of a single element. Only one mapper will manage one page.

2.3 Second Phase: PageRank Estimation

In this section, the pagerank iteration is presented. The number of iteration is fixed at the start of the execution. We do not converge to a (or a more or less) consistent state, because the presence of dangling nodes will cause importance (i.e., pagerank mass) to leak out.

Note: *outgoingEdge* is a title itself.

Algorithm 4 PageRank Computation Mapper

```
1: procedure MAP(Key k, FormattedPage p)
2:     EMIT(p.title, {0, p.outgoingEdges})
3:     for all outgoingEdge in p.outgoingEdges do
4:         EMIT(outgoingEdge, {p.pagerank / p.outgoingEdges.length, NULL})
```

Algorithm 5 PageRank Computation Reducer

```
1: procedure INITREDUCE(Configuration c)
2:     N ← c.numberOfNodes
3:     D ← Damping
4: procedure REDUCE(Title t, Nodes [n_1, n_2, ...])
5:     n new Node
6:     for all node in nodes do
7:         if node.hasOutgoingEdges() then
8:             n.outgoingEdges = node.outgoingEdges
9:         else
10:            s = s + node.pagerank
11:    n.pagerank = (1−D)/N + D * s
12:    EMIT(t, n)
```

2.4 Third Phase: Sorting

The final step is sorting the webpages by decreasing rank, this is done making advantage of the sorting mechanism of MapReduce.

Algorithm 6 Sorting Mapper

```
1: procedure MAP(Key k, FormattedPage p)
2:     title ← p.title
3:     pagerank ← p.pagerankk
4:     EMIT(pagerank, title)
```

Algorithm 7 Sorting Reducer

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

2.5 Implementations Details

In this following sections is briefly described how we have implemented in Hadoop the pseudocode presented in the previous chapter. After that, the performance evaluation of the computation is reported.

2.5.1 First Phase: Graph construction

Parsing is a job that doesn't need to create a conglomerate value, but it creates records which, each of them, is treated as a standalone piece of information. We decide to use three reducers since we have three working nodes.

Initially we had thought of inserting an initial step dedicated exclusively to the node count, but later we came up with a better solution, complicating the parsing phase. In the section on performance the differences between the two approaches will be shown, showing the improvement obtained.
First we consider the Mapper, in which each record is parsed, looking for the title and all the outgoing links. So for each record we are going to emit a pair **(title, outgoing edges)**. In addition to this we must also count the number of nodes, to do this we use an **In-Mapper Combiner**, a global counter inside the Mapper. It will be initialized to 0 in setup, and incremented at each execution of the map function; during the cleanup we will transmit a single cumulative value for that Mapper, transmitting the pair **("", N)**, with N the value of the counter. As a key we use the empty string "", which cannot belong to any title and which is always placed first in lexicographic order. Since we send two different information between Mapper and Reducer, we have decided to implement our own **partitioner**, to always send partial counters to Reducer 0 and therefore be sure to have this value in the file part-r-00000. All other keys will be split into the other reducers, using an hash function, like the default behavior. As for the reducer, it takes care of adding the partial values of the counters to obtain N, the number of nodes; in addition it will get the title and outgoing links values of each record. The initial page rank value is set to -1, and will be calculated at the first iteration of the next step. All this information will be saved in the filesystem, to be processed in the next step. The next step will take in input all the output file containing the Node information. All this process has been schematized in 1

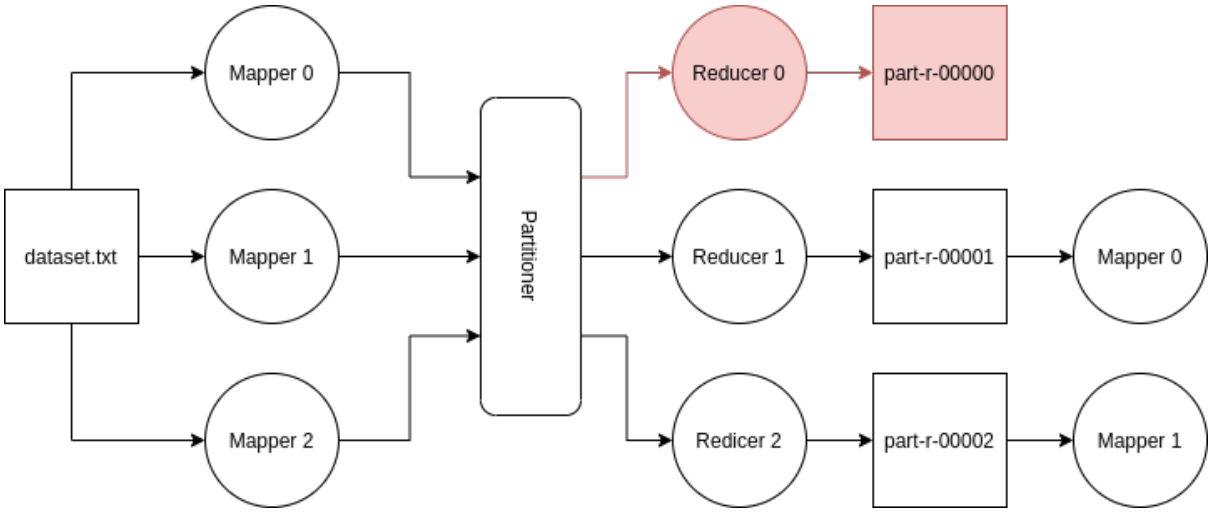

Figure 1: Parsing phase

2.5.2 Second Phase: PageRank Estimation

Since the previous phase constructs multiple outputs, this phase has to take multiple inputs As in the *Graph construction* phase multiple reducers are used for accelerating the computation. Thus, from the first iteration to the last one this MapReduce computation takes multiple inputs and produces multiple outputs.

2.5.3 Third Phase: Sorting

Since the previous phase constructs multiple outputs, also this phase has to take multiple inputs. Here a single reducer is used in order to take advantage of the automatic sorting done over the keys by Hadoop. Keys are in this case the pagerank value of each page, and are passed by the mapper to the reducer as **DoubleWritable** objects. We have implemented a **WritableComparator** for getting the descending order.

2.6 Performance

For measuring the performance we considered three files with a different number of nodes:

- **wiki-micro.txt**: number of nodes 2427. Most of the nodes points to sites that are not present in the initial set.
- **dataset5.txt**: number of nodes 5000. This dataset is synthetic dataset created using the same structure of *wiki-micro.txt*. Each node has a random value of outgoing edges between 0 and 10.
- **dataset10.txt**: number of nodes 10000. This dataset is synthetic dataset created using the same structure of *wiki-micro.txt*. Each node has a random value of outgoing edges between 0 and 10.

and we test them over different numbers of iteration: 5, 10, 15.

In order to prove the performance gain in *In-Mapper Combiner* instead of counting the nodes in a MapReduce, we provide the result of both the approches. Those results are computed as the mean of 5 iterations.

Table 1: Performance with MapReduce Node Counter

File	N Iter 5	N Iter 10	N Iter 15
wiki-micro.txt	184417	314290	436651
dataset5.txt	186320	310445	467191
dataset10.txt	206945	339101	455945

Table 2: Performance with counting in Parsing

File	N Iter 5	N Iter 10	N Iter 15
wiki-micro.txt	164519	295216	419528
dataset5.txt	168450	331793	430656
dataset10.txt	172402	294239	437840

The values in the **tables** are expressed in milliseconds.