UNIVERSITÀ DI PISA

University of Pisa

Laurea Magistrale (MSc) in Artificial Intelligence and Data Engineering

Project

Cloud Computing

PageRank

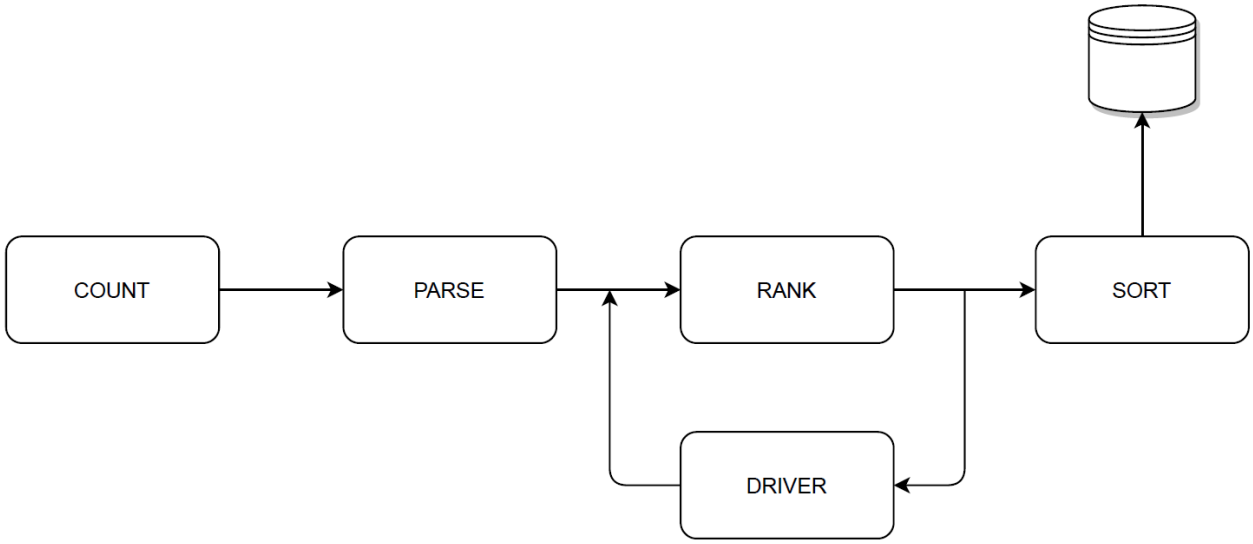
Academic year 2020-2021

Alessio Serra, Fabio Malloggi, Farzaneh Moghani, Marco Simoni, Valerio Giannini

Github: https://github.com/ValeGian/CC_PageRank

Algorithm Design

To perform the different parts required to implement the PageRank algorithm and present the final results, we pipelined 4 MapReduce stages:



The **Driver** simply runs the **Rank** stage for a fixed amount of iterations (specified by command line).

Pseudocodes

Algorithm: PageCount

```
1:  class Mapper
2:      method Map(lid l, text L)
3:          Lo ← "Page Count"
4:          emit(Lo, count 1)

1: class Reducer
2:     method Reduce(text Lo, counts [c1, c2, ….])
3:         sum ← 0
4:         for all count c in counts [c1, c2, ….] do
5:             sum ← sum + c
6:         emit(line Lo,  count sum)
```

Algorithm: Parsing Phase

```
1:  class Mapper
2:      method Map(lid l, text L)
3:          title ← L.getTitle()
4:          outlinks ← L.getOutLinks()
5:          if (title != NULL)
6:                  if (outlink.size() > 0)
7:                      for all link in outlinks do
8:                          Emit(title, link)
9:                  else
10:                         Emit(title, NULL)
```

```
1: class Reducer
2:     N ← 0
3:     method Setup()
4:         pageCount ← readPageCount()
5:     method Reduce(title t, links[link1, link2, ….])
6:         for all link l in links[link1, link2, ….] do
7:             if l != NULL
8:                 N.AdjacencyList ← N.AdjacencyList + l
9:         N.PageRank ← 1.0 / pageCount
10:        Emit(t, node N)
```

Algorithm: Ranking Phase

```
1:  class Mapper
2:      method Map(nid n, node N)
3:          p ← N.PageRank / |N.AdjacencyList|
4:          Emit(nid n, N)
5:              for all nodeid m in N.AdjacencyList do
6:                      Emit(m, p)
```

```
1: class Reducer
2:     method Setup()
3:         alpha ← readAlphaParameter()
4:         pageCount ← readPageCount()
5:     method Reduce(nid m, [p1, p2, ….])
6:         M ← 0
7:         for all p in [p1, p2, ….] do
8:             if isNode(p) then
9:                 M ← p
10:            else
11:                s ← s + p
12:        M.PageRank ← (alpha / PageCount) + (1 - alpha) * s
13:        Emit(nid m, node M)
```

Algorithm: Sorting Phase

```
1:  class Mapper
2:      method Map(nid n, node N)
3:          Page p ← n.Title
4:          Page p ← n.PageRank
5:          M0 ← "KeyPages"
5:          Emit(M0, Page p)
```

```
1: class Reducer
2:     method Reduce(Text M0, Pages[p1, p2, ….])
3:         Sort(Pages[p1, p2...])
4:         for all page in Pages[p1, p2, ...] do
5:             Emit(page.Title, page.PageRank)
```

Efficiency Issues

For the Hadoop implementation, we use two **customized objects**:

- **Node**: a *Writable* implementation used as *value* by *Parse Reducer, Rank Mapper* and *Rank Reducer*
- **Page**: a *WritableComparable* implementation used as *key* by the *Sort Mapper* and *Sort Reducer*.

Through the use of the **Page** *WritableComparable* we are able to sort our results directly exploiting the *Shuffle & Sort* phase of the framework.

To reduce the quantity of *intermediate data* produced, we implemented some **Combiners** and tested them on the *wiki-micro* dataset:

- A sort of *In-Mapper Combining* in the *Count stage*(from 2427 to 1 intermediate pair)
- A *Combiner* in the *Parse stage*(from 92300 to 78407, a 15% reduction)
- A *Combiner* in the *Rank stage*(from 80592 to 68966, a 14.43% reduction)

They allow us to reduce the amount of data shuffled across the network. The cost to be paid is

- For the *Count In-Mapper Combining*, we have to store an Integer for each Count Mapper instantiated
- For the other two *Combiners*, we increase the computation to be done on the machine running the *Mapper*, since the number of key-value pairs emitted by the mappers doesn't change and the combiners will have to go through all of them

For the *Sort* stage a *Combiner* is useless, since we want to emit the entirety off our result dataset in order to let the *Shuffle & Sort* phase handle its sorting.

We exploit **setup** methods in order to read configuration values.

We tested our application with **more than 1 reducer**. The number of reducers to be used is read from command line and passed to the *Parse* and *Rank* stages. The *Count* and *Sort* phase always use a single reducer since they do not need more than that.

The following measures have been collected in order to briefly show different performances varying the number of reducers, specified at compile-time. We analyzed the cases in which the application leverages 1, 3 and 5 reducers for the visualization of performances with an increasing level of parallelism. Moreover we have also retrieved statistics for 10 reducers application to estimate performances for an improved load balancing behaviour.

We have also decided to show the number of failed map tasks, which could surely represent a significant performance penalty. In particular the tables report the underlying statistics:

-Total time spent by all map tasks (ms)

-Total time spent by all reduce tasks (ms)

-Total megabyte-milliseconds taken by all map tasks

-Total megabyte-milliseconds taken by all reduce tasks

As we can clearly see, for what concerns both ranking 0 and ranking 1 stages, incrementing the number of reducers, the obtained results testify increasing execution time values. Probably this behavior is due to the small dimension of the original dataset that will result in the creation of multiple small files whenever the number of reducers increase and this will decrease application performances in terms of milliseconds.

-m-m = megabyte-milliseconds - m.t = map task

REDUCERS 1

	failed m.t	maps time(ms)	reduce times(ms)	map m-m	reduce m-m
parsing	0	4604	3831	1178624	980736
ranking 0	0	3725	3674	953600	940544
ranking 1	1	7550	6273	1932800	1605888
sorting	1	8887	2510	2275072	642560

REDUCERS 3

	failed m.t	maps time(ms)	reduce times(ms)	map m-m	reduce m-m
parsing	0	2846	9904	728576	2535424
ranking 0	3	33926	56403	8685056	14439168
ranking 1	4	38025	51195	9734400	13105920
sorting	0	9203	2428	2355968	621568

REDUCERS 5

	failed m.t	maps time(ms)	reduce times(ms)	map m-m	reduce m-m
parsing	1	2813	16221	720128	4152576
ranking 0	2	52512	53634	13443072	13730304
ranking 1	6	60748	67336	15551488	17238016
sorting	3	47005	7992	12033280	2045952

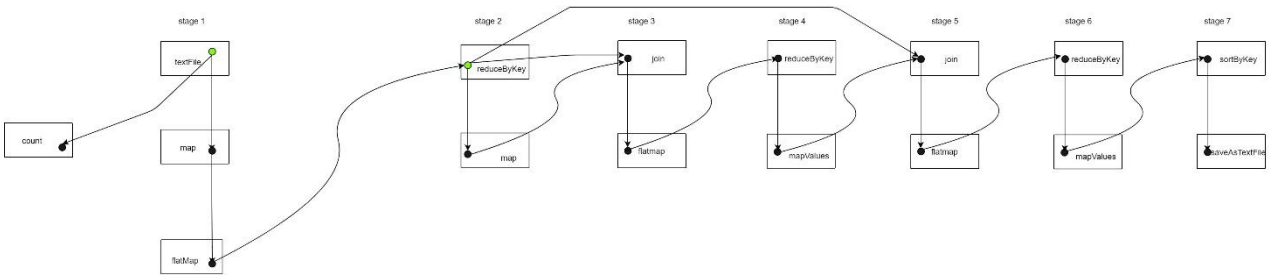
REDUCERS 10

	failed m.t	maps time(ms)	reduce times(ms)	map m-m	reduce m-m
parsing	1	4206	51315	1076736	13136640
ranking 0	1	115050	138978	29452800	35578368
ranking 1	1	91695	69551	23473920	17805056
sorting	5	122362	15738	31324672	4028928

Spark Implementation

We implemented Page Rank in Spark **both** with the **Python** and **Java API**.

This is how the DAG looks like, considering two iterations of the Page Rank algorithm.



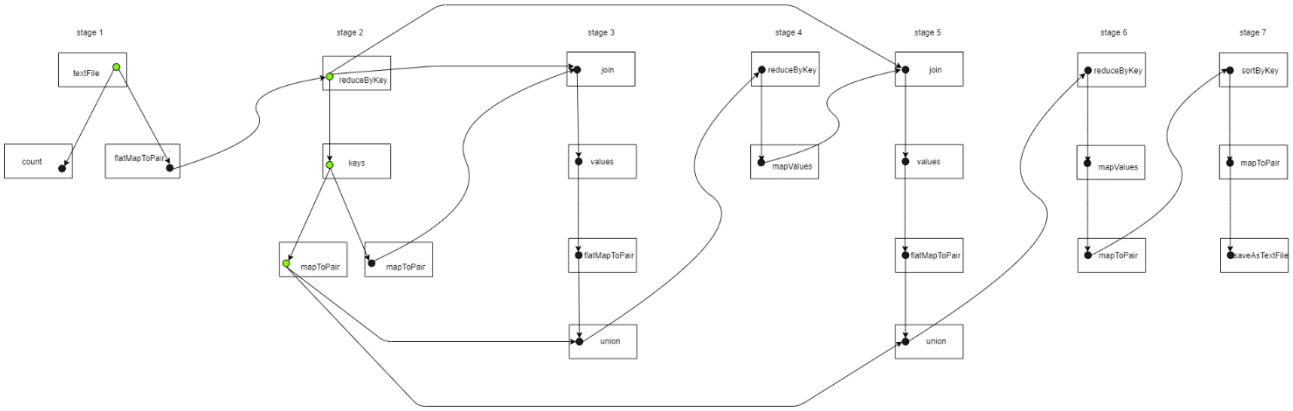
RDD PERSISTENCE

We store in the main memory two RDD using the cache() function:

- The RDD that stores the input file after the textFile(), since it is used both to count the number of pages in the file and to map node and outlinks.
- The RDD that stores for each node in the graph the list of outlinks, since it is used at each iteration of the algorithm.

JAVA

This is the DAG for Java implementation



It presents some differences with respect to the Python DAG:

- **Stage 1 [Parsing phase]**: Instead of executing a map() function followed by a flatMap() function, we perform only one flatMapToPair() function, that creates a JavaPairRDD mapping all possible nodes' name to keys and their corresponding outlinks lists to values
- **Stage 2 [mass Initialization]**: Instead of executing a map() function, first we perform a keys() function to obtain only the keys (all nodes' name), then we perform mapToPair()
- **Stage 2**: Over the RDD produced by the keys(), we perform another MapToPair() function, associating value "0" to all nodes. It is cached since we will use this RDD in every iteration of the Ranking phase
- **Stage 3 [Ranking phase]**: After executing the join() function, instead of performing a single flatMap() function to generate an RDD with both all distinct nodes along with the "0" mass value and each outlink along with its weighted mass, we split the computation of those two components in two different RDD, merging than the results with the union() function. The nodes with "0" mass are already performed once and for all in stage 2, while we perform the mass fan-out of each node along its outlinks through the values() and flatMapToPair()
- **Last stage [Sorting phase]**: since we are bounded to a JavaPairRDD class where keys and values are mapped to nodes' name and ranks' values respectively, before and after performing the sortByKey() function we need to swap keys and values through a mapToPair() function in order to perform the sorting over the ranks' values.