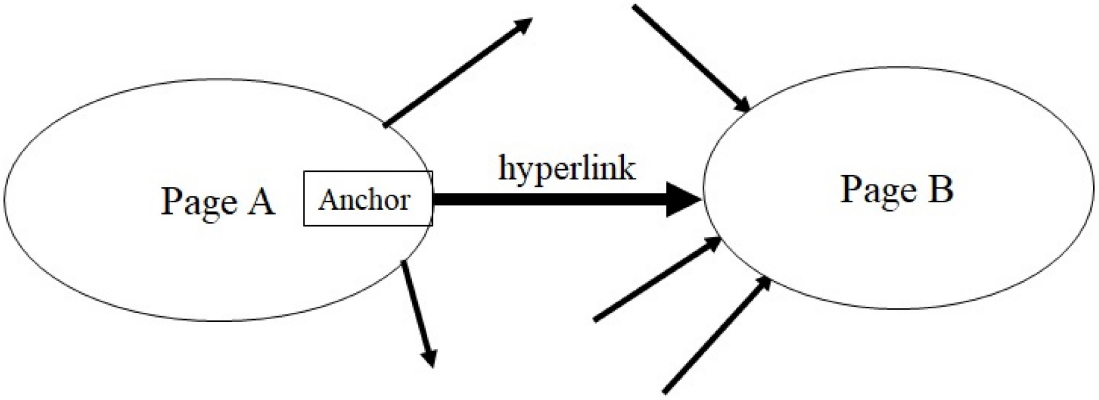
A Hadoop application: PageRank

FRANCESCO CASTIGLIONE

1 INTRODUCTION

The purpose of this application is to develop a map-reduce application that perfom the PageRank computation. The Web is a Directed Graph:



- A hyperlink between pages denotes author perceived relevance
- The text in the anchor of the hyperlink describes the target page

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.

1.1 How we can use PageRank?

The Web is a graph, a set of nodes corresponds to the various pages and the hyperlinks are the arcs. We start from a random page then:

- Randomly pick an outogoing link with probability λ
- Randomly jump into a page with probability 1-λ

The PageRank of page x $Pr(x)$ is the probability of being on page x at a random moment in time.

1.2 Algorithm

1.2.1 Simplified PageRank algorithm. The algorithm stars by initializing a uniform distribution of PageRank across all the nodes:

$$PR(x) = \frac{1}{N} \quad \forall x \, Where \, N \, is \, the \, number \, of \, nodes \, in \, the \, graph \tag{1}$$

Than for each step, we compute the PageRank as:

$$PR(x) = \frac{1-\lambda}{N} + \lambda \sum_{y_i \to x} \frac{PR(y)}{out(y)} \quad \forall x \tag{2}$$

$PR(x)$ corresponds to the probability to be ant node x at the current moment. I can get there in two ways, on the previous step I flip a coin and:

- if I get tail (1-λ), I perform a random hop and go to x with $Pr = \frac{1}{N}$
- if I get head (λ), the only way to go to x is to be in a node y which points to x and select that arc that connects y to x with $Pr = \frac{1}{number\ of\ outgoing\ links}$. I have to sum up this second contribution over all the nodes y_i that points to x.

We procede in a new iteration if $|PR(t+1) - PR(t)| < \epsilon$

At the beginning of each iteration, the PageRank values of all nodes sum to one. PageRank mass is preserved, so we have a valid probability distribution at the end of each iteration.

1.2.2 Dangling nodes. Dangling nodes are nodes in the graph with no outgoing edges. If we run the simplified PageRank algorithm on a graph with this nodes the total PageRank mass will be not conserved. So we have to redistribute the "lost" mass on dangling nodes across all the nodes in the graph. The redistribution process is done updating the current PageRank of each nodes according to the formula:

$$PR_{new}(x) = (1-\lambda)\frac{1}{N} + \lambda(\frac{m}{N} + PR_{current}(x)) \quad \forall x \quad \textit{Where m is the missing PageRank mass} \tag{3}$$

2 INPUT FILE

Before starting to calculate PageRank, we need to acquire the graph. Considering that the entire Web Graph is too big for my local machine, i've decided to considering a smaller one as the Wikipedia Graph. Wikimedia Foundation releases data dumps of Wikipedia and all Wikimedia Foundation projects on a regular basis. Is possible to download the latest dumps (for the last year) here. I have used two sql files, wikidatawiki-latest-page.sql.gz and wikidatawiki-latest-pagelinks.sql.gz. The latter contains page-to-page link lists the former, instead contains info about each page.

Page Table. The page table can be considered the "core of the wiki". Each page has an entry here which identifies it by title and contains some essential metadata.

Field	Type	Null	Key	Default	Extra
page id	int(10) unsigned	NO	PRI	NULL	auto increment
page namespace	int(11)	NO	MUL	NULL	
page title	varbinary(255)	NO		NULL	
page restrictions	tinyblob	NO		NULL	
page is redirect	tinyint(3) unsigned	NO	MUL	0	
page is new	tinyint(3) unsigned	NO		0	
page random	double unsigned	NO	MUL	NULL	
page touched	binary(14)	NO			
page links updated	varbinary(14)	YES		NULL	

Table 1 continued from previous page

Field	Type	Null	Key	Default	Extra
page latest	int(10) unsigned	NO		NULL	
page len	int(10) unsigned	NO	MUL	NULL	
page content model	varbinary(32)	YES		NULL	
page lang	varbinary(35)	YES		NULL	

Table 1. Page Table schema summary

PageLinks Table. Tracks all internal links in the Wiki. Each entry contains the source page's ID and namespace (number), and the article name (in text) and namespace (number) that is being linked to from within that source page. There may be many instances of the source page's ID, as many as the internal links within it, but there can be only one entry per internal link for any page ID

Field	Type		Null	Key	Default	Extra
pl from	int(10)	unsigned	NO	PRI	0	
pl from namespace	int(11)		NO	MUL	0	
pl namespace	int(11)		NO	PRI	0	
pl title	varbina	ry(255)	NO	PRI		

Table 2. PageLink Table schema summary

2.1 Parsing

First of all I have used this Python script to "convert" the two .sql file to a more readable .csv format. Than I have implemented a Hadoop MapReduce application to parse the .csv file.
Page Parsing: is a MapReuce job in which taken as input *enwiki-latest-page.csv* and extract the two relevat fields *page id* and *page title*.
Starting from:

```
10,0,AccessibleComputing,,1,0,0.33167112649574004,20190312173137,20190105021557,854851586,94,wikitext,NULL
12,0,Anarchism,,0,0,0.786172332974311,20190320094317,20190320094317,888620758,100274,wikitext,NULL
14,0,AfghanistanGeography,,1,0,0.952234464653055,20190228203758,20190228204407,783865160,92,wikitext,NULL
```

The result will be:

```
10 AccessibleComputing
14 AfghanistanGeography
12 Anarchism
```

The Mapper then starts splitting the key using ',' as delimiter and check if the page come form the "main namespace" where the "Real" content articles are contained. Than set as result key the *page id* and the correspondent result value is the *page title*.

```
  public static class PageParser extends Mapper<Text, Text, IntWritable, Text>
{
  public void map(Text key, Text value, Context context) throws IOException,
   InterruptedException
  {
    IntWritable map_key = new IntWritable();
    Text map_value = new Text();
    String[] entry=key.toString().split("\\,");
    if(entry[1].equals("0")){
      map_key.set(Integer.parseInt(entry[0]));
      map_value.set(entry[2]);
      context.write(map_key,map_value);
    }
  }
}
```

Listing 1. Page Parser Map code

Pagelink Parsing: is a bit more complex with respect to the Page Parsing. This is due to the fact that each Wikipedia Pagelink table contains the *page id* of the linked page as *integer*and the *page title* of the target page as *text* so I have to perform an additional "mapping" from the *page title* to the corresponding *page id*.
Starting from:

```
4748,0,AccessibleComputing,0
9773,0,AccessibleComputing,0
15154,0,AccessibleComputing,0
25213,0,AccessibleComputing,0
```

The result will be:

```
10      4748 9773 15154 25213
14      4355567  1028188 1340746
12      1436942 2875276 4355567
```

First of all I've defined a custom writale called PairWritable that contains, the PageRank value, the Adjacency list and a boolean value used to distinguish if the current entry represent a node or a PageRank value.

```
public class PairWritable implements Writable{
private DoubleWritable pagerank;
private Text adj_list;
private BooleanWritable isNode;
```

Listing 2. Pagelink Parser Map code

I have created two Mapper, the first one recive in input the results of the Page Parser(the list of all the *page id* and the correspondent *page title*), the second, take as input *enwiki-latest-pagelinks.csv* the file containing all the links between the pages.

- In the first Mapper each enty represents an active page so the Mapper emits as key *page title* and as value a Pairwritable in which the Pagerank is Double.NaN and the Adjacency list contain the correspondent *page id* for that *page title*.

  AccessibleComputing 10 ⟶ AccessibleComputing NaN 10

- In the second Mapper each enrty represents a links between two pages, so the Mapper emits as key *pl title* and as value a PairWritable in which the Pagerank is 0 and the Adjacency list is the linked page.

  4748,0,AccessibleComputing,0 ⟶ AccessibleComputing 0 4748

```
public static class PagelinkParser_Map_1 extends Mapper<Text, Text, Text, PairWritable>
  {
    public void map(Text key, Text value, Context context) throws IOException,
     InterruptedException
    {
      PairWritable map_value = new PairWritable();
      map_value.set(Double.NaN,key.toString());
      context.write(value,map_value);
    }
  }
public static class PagelinkParser_Map_2 extends Mapper<Text, Text, Text, PairWritable>
  {
    public void map(Text key, Text value, Context context) throws IOException,
     InterruptedException
    {
      Text map_key = new Text();
      PairWritable map_value = new PairWritable();
      String[] entry=key.toString().split("\\,");
        if(entry[1].equals("0")){
          map_key.set(entry[2]);
          map_value.set(0.0,entry[0]);
          context.write(map_key,map_value);
        }
    }
  }
```

Listing 3. Pagelink Parser Map code

The Reducer iterates over all the values:
If the *val.getPagerank().get()* is different from Double.NaN, this means that the val represent a link between two pages so I append val . getAdj_list () . toString () to the adjacency list String. In the other case, val represent the *page id* so it is my result key. I also set the *boolean* variable *isActive* to *true*. This has to be done because the target page may or may not exist, and due to renames and deletions may refer to different page records. Only the pages that comes from the Page Table file can be considered as active and at the end, only if a node has been set as active will be written into the contex. The Reducer emits for each page as key the *page id* and as value a Pairwritable containing the initial Pagerank $\frac{1}{N}$ and the Adjacency list of that page. It also set the boolean variable (of the Pairwritable) isNode to true because this entry represent a node.

```
public static class PagelinkParser_Red extends Reducer<Text, PairWritable, Text,
   PairWritable>
{
  private int nodes_number;

  public void reduce(Text key, Iterable<PairWritable> values, Context context) throws
   IOException, InterruptedException
  {
    Text red_key = new Text();
    PairWritable red_value = new PairWritable();
    String[] entry;
    boolean isActive=false;
    double pagerank=0;
    StringBuilder sbuilder = new StringBuilder(" ");
    for (PairWritable val:values){
      if(Double.isNaN(val.getPagerank().get())){
        red_key.set(val.getAdj_list().toString());
        isActive=true;
      }else{
        sbuilder.append(val.getAdj_list().toString()).append(" ");
      }
    }
    if(isActive){
      red_value.set(1.0/nodes_number,sbuilder.toString().trim());
      red_value.setNode(true);
      context.write(red_key,red_value);
    }
  }
```

Listing 4. Pagelink Parser Reduce code

3 PAGERANK CALCULATION

My PageRank implementation divide the entire work in tree phases: checks for dangling nodes, calculating and sorting.

3.1 Check for Dangling Nodes

This first phase take as input the file produced by PageLink_parser and simply checks if the adjacency list of each node is empty, and if it is true, it keeps track of the node's PageRank value using a Counter SINK. At the end of this first Map SINK will contain how much PageRank was lost at the dangling nodes.

```
public static class Dangling extends Mapper<Text, PairWritable, Text, PairWritable>
  {
    private int scale_factor;
    static enum Counter{
      SINK
    }
    public void map(Text key, PairWritable value, Context context) throws IOException,
     InterruptedException
    {
      if(value.getAdj_list().toString().isEmpty())
      {
        context.getCounter(Counter.SINK).increment((int)(value.getPagerank().get()*
     scale_factor));
      }
    }
```

Listing 5. PageRank Initializer Mapper code

3.2 Calculating PageRank

In this second phase the Mapper take as input the file produced by PageLink_parser and:

(1) Emits a pair key value in which the key is the *node id* and the value is the Pairwritable containing $PR(x)$ *and the list of adjacencies*. This will be used by the reducer to reconstruct the graph.

```
10    4748 9773 15154 ⟶ 10    7.14E-8    4748 9773 15154
```

(2) Emits for every items into the adjacency list a pair key value in which the key is the *node id* and the value is a Pairwritable in which the PageRank is the one of the starting node divided by the number of items into the adjacency list (which is the number of outgoing links from the starting node), and the adjacency list is empty. It also set the boolean variable (of the Pairwritable) isNode to false because this entry represent a PageRank information.

```
4748  ⟶ 1.78E-8
9773  ⟶ 1.78E-8
15154 ⟶ 1.78E-8
```

```
public static class Map extends Mapper<Text, PairWritable, Text, PairWritable>
{
  public void map(Text key, PairWritable value, Context context) throws IOException,
   InterruptedException
  {
    PairWritable map_value = new PairWritable();
    Text map_key = new Text();
    context.write(key,value);
    if(!value.getAdj_list().toString().isEmpty())
    {
      String[] adj_list = value.getAdj_list().toString().split("[ ]");
      for (String s: adj_list)
      {
        map_key.set(s);
        map_value.set(value.getPagerank().get()/adj_list.length,"");
        map_value.setNode(false);
        context.write(map_key,map_value);
      }
    }
  }
}
```

Listing 6. PageRank Mapper code

The reduce phase takes all the sorted pair key value and checks the boolean variable isNode:

- If it is true this means that the entry represents a node, so the reducer use the values of node id and adjacency list to reconstruct the graph.
- If it is false this means that the entry represents a Pagerank information, so the reducer sum the partial pagerank contained into val.getPagerank() to the variable sum for computing the final pagerank.

Subsequently the final pagerank is computed using the formula (3). The reducer emits a pair key value in which the key is the *node id* and the value is the Pairwritable composed by the final PageRank and the adjacency list for that *node id*. The next step is to calculate the convergence value and using a counter CONV, pass this value to the Driver that checks if a new interation has to be computed. We also use an other counter called SUM to check if the sum of all PageRanks goes to one.

```
public static class Red extends Reducer<Text, PairWritable, Text, PairWritable>
  {
    private int nodes_number;
    private double lambda;
    private int scale_factor;
    private int sink;
    private double dangling;
    public static enum Counter {
      CONV,
      SUM
    }
    public void reduce(Text key, Iterable<PairWritable> values, Context context) throws
     IOException, InterruptedException
    {
      Text red_key = new Text();
      PairWritable red_value = new PairWritable();
      double sum = 0;
      double pagerank = 0;
      String adj_list="";
      for (PairWritable val:values){
        if(!val.isNode().get()){
          sum+=val.getPagerank().get();
        }else{
          adj_list=val.getAdj_list().toString();
          pagerank=val.getPagerank().get();
        }
      }
      sum = ((1-lambda)/nodes_number)+((lambda)*((dangling/nodes_number)+((1-lambda)/
     nodes_number)+((lambda)*(sum))));
      context.getCounter(Counter.CONV).increment((int)(Math.abs(sum-pagerank)*scale_factor));
      context.getCounter(Counter.SUM).increment((int)(sum*scale_factor));
      red_value.set(sum,adj_list);
      red_value.setNode(true);
      context.write(key ,red_value);
    }
```

Listing 7. PageRank Reducer code

3.3 Sorting

The last step is the sorter, a map job that takes as input the output file from the PageRank Reducer and simply emits a pair key value in which the key is the *PageRank* and the value is the *node id*. Then I have defined a DescendingKeyComparator which is used to sort the key in descending order.

4 EVALUATION AND RESULTS

4.1 First little test

The first little test consists in a very simple graph:

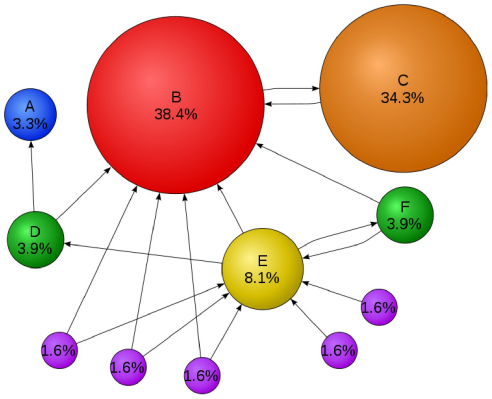


Fig. 1. First PageRank calculation graph

```
hadoop@debian: /media/sf_cane
File  Edit  View  Search  Terminal  Help
hadoop@debian:/media/sf_cane$ cat simplewiki-page.csv
1,0,uno,,0,0,0,0,0,0,0,wikitext,NULL
2,0,due,,0,0,0,0,0,0,0,wikitext,NULL
3,0,tre,,0,0,0,0,0,0,0,wikitext,NULL
4,0,quattro,,0,0,0,0,0,0,0,wikitext,NULL
5,0,cinque,,0,0,0,0,0,0,0,wikitext,NULL
6,0,sei,,0,0,0,0,0,0,0,wikitext,NULL
7,0,sette,,0,0,0,0,0,0,0,wikitext,NULL
8,0,otto,,0,0,0,0,0,0,0,wikitext,NULL
9,0,nove,,0,0,0,0,0,0,0,wikitext,NULL
10,0,dieci,,0,0,0,0,0,0,0,wikitext,NULL
11,0,undici,,0,0,0,0,0,0,0,wikitext,NULL
hadoop@debian:/media/sf_cane$ cat simplewiki-pagelinks.csv
3,0,due,0
2,0,tre,0
2,0,quattro,0
1,0,quattro,0
2,0,cinque,0
4,0,cinque,0
6,0,cinque,0
2,0,sei,0
5,0,sei,0
2,0,sette,0
2,0,otto,0
2,0,nove,0
5,0,nove,0
5,0,dieci,0
5,0,undici,0
```

Fig. 2. First PageRank calculation input files

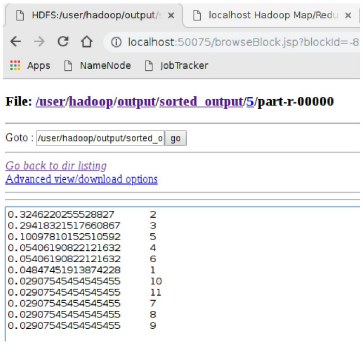
In order to test also the parser, I've created the two input files as they were the real Wikipedia files. Assuming that hadoop and maven are working correctly, we can build and compile the Pagerank.java application inside the Pagerank directory. First of all we have to add to the hdfs the two input files:

hadoop@debian:~/hadoop−1.2.1/Pagerank$ hadoop fs −put simplewiki−page.csv simplewiki−page.csv
hadoop@debian:~/hadoop−1.2.1/Pagerank$ hadoop fs −put simplewiki−pagelinks.csv simplewiki−pagelinks.csv

Than we can execute:

hadoop@debian:~/hadoop−1.2.1/Pagerank$ mvn clean package
hadoop@debian:~/hadoop−1.2.1/Pagerank$ hadoop jar target/Pagerank−1.0−SNAPSHOT.jar it.cnr.isti.pad.Pagerank simplewiki−page.csv simplewiki −pagelinks.csv output

After six iterations we obtain:




```
----------------------------------------
iteration:      0
sum:    1.011
convergence:    0.07239090909090909 / 0.01
----------------------------------------
----------------------------------------
iteration:      1
sum:    1.0161
convergence:    0.04454545454545455 / 0.01
----------------------------------------
----------------------------------------
iteration:      2
sum:    1.0178
convergence:    0.025045454545454544 / 0.01
----------------------------------------
----------------------------------------
iteration:      3
sum:    1.0202
convergence:    0.01748181818181818 / 0.01
----------------------------------------
----------------------------------------
iteration:      4
sum:    1.0207
convergence:    0.011754545454545455 / 0.01
----------------------------------------
----------------------------------------
iteration:      5
sum:    1.021
convergence:    0.008445454545454546 / 0.01
----------------------------------------
```

So checking the initial graph, the result is correct and also for each iteration the sum of all pageranks goes to one.

4.2 The Wikipedia test