# TP PageRank

Marc Jeanmougin

May 23, 2019

## 1 Introduction to Hadoop

The generic term *Hadoop* refers to a collection of many tools in the ecosystem of distributed computing, mostly developed by the *Apache Software Foundation*[1].

The *Hadoop* framework per se denotes several modules like Yarn (a scheduling tool for distributed applications), HDFS (a distributed filesystem), and MapReduce, an implementation of a programming model that was published and made popular by Google[1]

The opensource implementation of Hadoop MapReduce is primarily a Java tool made for easy development in Java. However, we'll use here the "streaming" API, which allows to run arbitrary scripts or executables as mapper or reducers.

The process is as follows:

1. The *mapper* reads the *input file(s)* and outputs *key/value pair* lines. The first tab in a line separates the key from the value.
2. The framework aggregates values with the same key (it sorts the output on keys).
3. The *reducer* is invoked on the output of the mapper and outputs key/value pairs.

Common mistakes and precisions:

- A mapper can usually only read a chunk of the input data, and does not have access to all of it
- In the streaming environment (that we use here), reducers can have access to several keys, but for a given key they have access to all mapper outputs associated with that key. Their input is sorted (all reducers inputs for one key, then all reducer inputs for another key, etc).
- For the simple case where we have one mapper and one reducer, mapreduce usage can be emulated as `cat input | ./mapper | sort | ./reducer` which will allow you to test your code without hadoop.

***Please submit your answers on Moodle. It should contain files named "Task$nMapper.py" and Task$nReducer.py when applicable, or Task$n.txt for text answers (text answers will be graded separately later).***
***You should test your code locally on examples and on real data (evaluation will use a whole wikipedia dump for a small language, you are encouraged to download one and run your code on it to see how fast it runs), or test it on the execution servers. When you are done with the TP, and only then, you can submit your code for automatic grading. YOU WILL LOSE 1 POINT PER ATTEMPT.***

[1] Such as Hadoop, Zookeeper, Spark, Storm, Kafka, HBase, CouchDB, Flink, Cassandra, Hive...

**Task 1 (2 Points)** *Install (unzip) the latest hadoop version:* `http://www.apache.org/dyn/closer.cgi/hadoop/common/hadoop-3.2.0/hadoop-3.2.0.tar.gz` *and test the "mapreduce" utility* [2]*. Test the following example (from the hadoop documentation[3]) and explain what it does. Submit the file "Task1.txt".*

```
mapred streaming \
-input myInputDirs \
-output myOutputDir \
-mapper /bin/cat \
-reducer /usr/bin/wc
```

# 2 Hadoop and Python

A "WordCount" program is the most classic example in distributed algorithm, and just counts words in a text, to show what programs in a framework look like.

As such, here is an example implementation of WordCount in python, that you can use as a template[3]. The mapper just extracts every word from the input and counts them "once", and the reducer adds them.

Mapper.py

```
#!/usr/bin/python3
import sys

def main(argv):
    for line in sys.stdin:
        wordlist = line.strip().split()
        for word in wordlist:
            print(word+"\t"+"1")
if __name__ == "__main__":
     main(sys.argv)
```

[2] If you have a problem, make sure you have the environment variable JAVA_HOME set

[3] This code follows some good practices like __main__ that could be simplified out.

Reducer.py

```
#!/usr/bin/python3
import sys

def main(argv):
    current_word = None
    wordcount = 0
    for line in sys.stdin:
        word, n = line.strip().split("\t",1)
        n = int(n)
        if current_word == word:
            wordcount += n
        else:
            if current_word:
                print(current_word+"\t"+str(wordcount))
            wordcount = n
            current_word = word
    print(current_word+"\t"+str(wordcount))
if __name__ == "__main__":
     main(sys.argv)
```

# 3 PageRank

## 3.1 Linking Graph

The pageRank algorithm is the basic block of modern search engines. It is a simple algorithm that operates on the graph of *links* in the World Wide Web, first detecting which page links which other page. The first thing we will have to do is thus to build such graph.

***When the input format of mapper files is not specified, it is XML code similar to the example code given with the project (which is the format of the wikipedia content dumps[2]). You can use/adapt the code in the Job2Mapper.py which takes care of parsing it.***

**Task 2 (3 Points)** *Given a mapper (see files given with the project) that takes a webpage and outputs "Page1\tPage2" for every link found in Page1 to Page2, write a reducer that gives for every page the ordered, unique, list of pages it links to. Submit the file "Task2Reducer.py"*

| Example reducer input | Example reducer output |
|---|---|
| Page1 Page2<br>Page1 Page3<br>Page2 Page1<br>Page1 Page2<br>Page3 Page1<br>Page3 Page4 | Page1 Page2,Page3<br>Page2 Page1<br>Page3 Page1,Page4 |

However, this mapper is not satisfying: indeed, "sinks" (pages that have

no outgoing links) cannot be differentiated from dead links (pages that do not exist)!

**Task 3 (2 Points)** *Modify the mapper and the reducer such that the reducer outputs "Page1\t" for a sink page, but nothing for a non-existing page. Submit two files Task3Mapper.py and Task3Reducer.py*

| Example reducer input | Example reducer output |
|---|---|
| `Page1 exists` | |
| `Page1 Page2` | |
| `Page4 exists` | |
| `Page1 Page3` | `Page1 Page2,Page3` |
| `Page2 Page1` | `Page2 Page1` |
| `Page1 Page2` | `Page3 Page1,Page4` |
| `Page3 exists` | `Page4 //<-tab here (print only the tab)` |
| `Page3 Page1` | |
| `Page3 Page4` | |
| `Page2 exists` | |

## 3.2 The PageRank Algorithm(s)

### 3.2.1 Global pagerank

*From here on, instead of a fixed format, we will use a serializer to pass values across from the mapper to the reducer. In particular, we will use JSON[4], which integrates well with python, and will allow us to simply encode and parse multiple values without ad-hoc joins and splits.*

**Quick reference:**

```
>>> import json

>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}]) # python object to json
'["foo", {"bar": ["baz", null, 1.0, 2]}]'


>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]') # json to python object
['foo', {'bar': ['baz', None, 1.0, 2]}]
```

Computing the pagerank is an iterative process: From one step to the next,

$$PageRank(Page) = 1 - d + d \sum_{p \in \text{ pages linking to } page} \frac{PageRank(p)}{\#outlinks(p)}$$

where N is the total number of pages and d is a damping factor (0.85).

**Task 4 (2 Points)** *Adapt the reducer from the previous task to initialize the PageRank to 1.0 per page. Submit the file Task4Reducer.py*

Example reducer input

```
Page1   exists
Page1   Page2
Page4   exists
Page1   Page3
Page2   Page1
Page1   Page2
Page3   exists
Page3   Page1
Page3   Page4
Page2   exists
```

Example reducer output

```
Page1   {"score":1.0, "links":["Page2", "Page3"]}
Page2   {"score":1.0, "links":["Page1"]}
Page3   {"score":1.0, "links":["Page1", "Page4"]}
Page4   {"score":1.0, "links":[]}
```

**Task 5 (6 Points)** *Write a mapper that outputs, for every page (used as key), its inbound pages, the pagerank of the origin (linking) page and the number of links of that page. Write a reducer that computes the new PageRank and writes it back to file, in a format that the mapper can understand as input.*

*Test your code by executing it on the given example data several times in a row.*

*What do you observe for each pageRank? Do you see flaws in this algorithm, or way(s) that this algorithm could be improved? Submit files Task5Mapper.py Task5Reducer.py and Task5.txt.*

Example reducer input

```
Page4   {"existenceSignal":true, "links":[]}
Page1   {"from":"Page2", "score":1.0, "nLinks":1, "existenceSignal":false}
Page1   {"existenceSignal":true, "links":["Page2", "Page3"]}
Page1   {"from":"Page3", "score":1.0, "nLinks":2, "existenceSignal":false}
Page2   {"from":"Page1", "score":1.0, "nLinks":2, "existenceSignal":false}
Page3   {"existenceSignal":true, "links":["Page1", "Page4"]}
Page3   {"from":"Page1", "score":1.0, "nLinks":2, "existenceSignal":false}
Page4   {"from":"Page3", "score":1.0, "nLinks":2, "existenceSignal":false}
Page2   {"existenceSignal":true, "links":["Page1"]}
```

Example reducer output

```
Page1   {"score":1.425, "links":["Page2", "Page3"]}
Page2   {"score":0.575, "links":["Page1"]}
Page3   {"score":0.575, "links":["Page1", "Page4"]}
Page4   {"score":0.575, "links":[]}
```

*Mapper input and reducer outputs follow the same format, which is also the output format of the reducer of the previous task.*

*Since the mapper output and the reducer input are only an intermediary format, you do not have to follow the format given here as an example (but it is recommended, as it conveys enough information to compute the pageRank (and more, the "from" is actually useless)). In this example, I re-encoded the "exists" signal from the previous task in a python dictionary so that we can quickly tell*

*if a message is a signal of existence or a regular link message with a construct like "if data['existenceSignal']:"*

### 3.2.2 The random surfer

The above algorithm needs to get a constantly up to date global state of the graph. In real life, we can't have that (crawlers are slow.) so we will simulate a *random surfer* who selects a page at random, then clicks on a random link, then there on another random link, and with some probability $1-d$ just go to a random page of the web (Think of it as "browsing wikipedia when you're really bored" : you go to a random article, read some stuff, click on a link, and after some time just click again on the random article button). We consider that clicking on a link "transfers" some pageRank amount to the new visited page $(\frac{current_{pagerank}}{\#links})$

**Task 6 (5 Points)** *Write a program that simulates the behavior of a random surfer, run it for $10^6$ iterations, and computes the number of times it goes to which page. Compare the result with the iterated program from the previous task, and discuss the results (in particular in terms of results, convergence speed of the algorithms and memory footprint). Your mappers will simulate the surfer behavior and pass all the pages visited to the reducer who will aggregate them and normalize the result, such that the sum of pageRanks should be the total number of pages. Submit files Task6Mapper.py Task6Reducer.py Task6.txt (for discussing results).*

Example reducer output

```
Page1   2.1
Page2   1.1
Page3   0.2
Page4   0.6
```

# References


[1] Dean, Jeffrey and Ghemawat, Sanjay *MapReduce: simplified data processing on large clusters*, **Communications of the ACM, 2008**

[2] `https://dumps.wikimedia.org/`

[3] `https://hadoop.apache.org/docs/r3.0.0/hadoop-streaming/HadoopStreaming.html`

[4] `https://docs.python.org/3.6/library/json.html`