# Part 2 Design highlights

In this part, we will discuss features we developed to enhance the search engine. In back end, we used Mapreduce to calculate the PageRank (on Hadoop).

Our code is available in lab4 submission file as well as github: https://github.com/wlz1028/pl\_works

## Calculating PageRank using Mapreduce (Hadoop)

### Motivation

Google introduced Mapreduce to calculate Mapreduce on billions of webpages more than 100 times everyday. It’s impossible to fit all the data on single machine, because of limited compute resource. Google simply distribute work to thousands of cheap computers. The Google file system is a distributed file system which is faults tolerant. In this lab, we used a popular Mapreduce Hadoop to implement a simple version of PageRank, and we actually ran the application on Hadoop.

On the other hand, Hadoop is extremely hot in data mining area. We hear it every day but never worked on Hadoop. It’s a great chance to get hands on experience.

### Resources (reference)

This article introduced how does Google calculate PageRank using Mapreduce

**[1]** http://www.cs.utah.edu/~jeffp/teaching/cs5955/L24-MR+PR.pdf

This article shows data structure and Mapreduce pseudo code(simple version)

**[2]** <https://code.google.com/p/joycrawler/downloads/detail?name=Readme-0.20.0.pdf&can=2&q>=

This article depicts how to write Hadoop application in python. The mapper and reducer take advantage of Hadoop streaming feature. All the input/output dat are stdin/stdout, and handled by Hadoop streaming automatically. The article also gave a simple way to test mapper/reducer function on command line

**[3]**<http://www.glennklockwood.com/di/hadoop-streaming.php#wordcount:shuffle>

This article introduced how to install Hadoop on Ubuntu.

**[4]** <http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/>

### Data structure

We will describe our own implementation of data structure based on article [1] and [2]. We will walk through a simple 4x4 example to illustrate our design.

For example we have the following graph:

**Matrix 1**

M = [ 0 1/3 0 1/2]

[ 1/2 0 0 0 ]

[ 0 1/3 0 0 ]

[ 1/2 1/3 1 1/2 ]

We can save it as a sparse matrix:

**Matrix 2**

1 2 4 # Page id= 1 has two links that point to page 2 and page 4

2 1 3 4

3 4

4 4 1

The first column represents each page id and followed by out link id (see above comment).

This data structure works well with our crawler, because when it crawls on a page, crawler simply output page id and followed by out links id (or saved to database directly).

### Mapper

Each page has a PageRank score (denote as pr), and each page has k numbers of out links. So for each out links on a specific page, it will has a pr \* 1/k probability to be clicked. For example, initially pr=1/4(where 4 is total page number) for matrix 2, so P(page2)=1/4\*1/2, and P(page1)=1/4\*1/2.

### Reducer

Mapper out put multiple P with the same id. The reducer aggregates all the P associate to the same page id, and then calculate

pr = beta\*(p1+p2…)+(1-beta)\*1/n

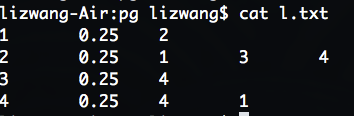
Where beta is the damping factor=0.85, and n is total number of page ids

### Iteration

Each Mapreduce task yields a set of PageRank score. We can simply iterate the Mapreduce procedure until PageRank score converge (article [2] mentioned that 15 iteration is good enough for most cases; for Google maybe 30+ iterations).

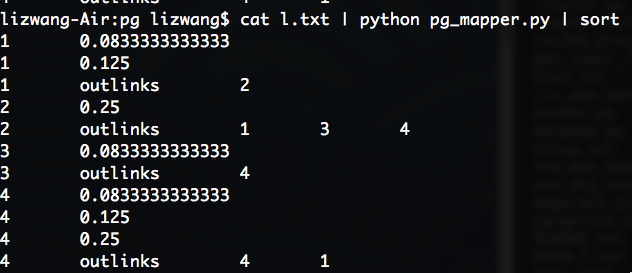
### Implementation

Firstly, we need to improve the data structure for coding convenience purposes. We improve matrix 2 to the following:



Where the second column is the PageRank score for the current page. As we mention before, initially we set PageRank = 1/n where n is the total number of page ids.

Then the mapper will output following data:

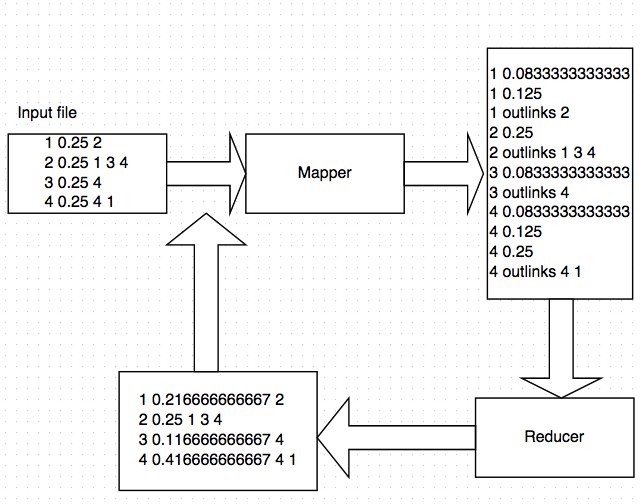


Where first column is page id, and second column is the probability (see mapper step above)

The 3rd row represent page id and out links relationship, so the reducer can output data, which can be fed to mapper in the next iteration.

The reducer consumes mappers output, and then out a set of PageRank score which can be fed to next Mapreduce iteration

The Mapreduce task works as following:

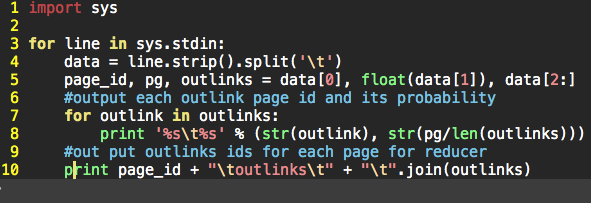


### Mapper/Reducer in python

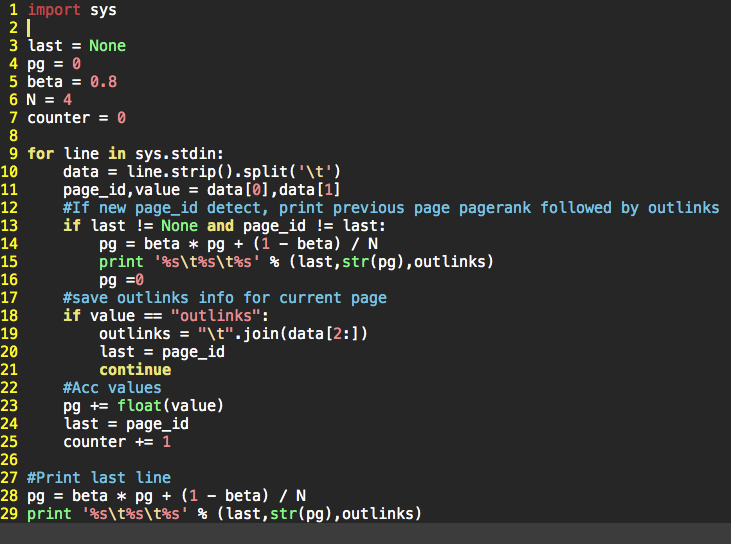
In this demo, we use Hadoop streaming feature to run python mapper and reducer. Basically, Hadoop streaming uses stdin/stdout to handles input/output data. So mapper/reducer consumes stdin and then yield stdout.

In later section, we will improve mapper/reducer by introducing generator.

backEnd/pagerank\_mapreduce/**pg\_mapper.py**



backEnd/pagerank\_mapreduce/**pg\_reducer.py**



### Testing on Linux command line:

Article [3] showed an elegant way to test mapper and reducer in classic word count example on command line. We test our code by using this method. We will use the 4x4 matrix we described above as the input.

**Input matrix**

1 0.25 2

2 0.25 1 3 4

3 0.25 4

4 0.25 4 1

First Mapreduce iteration:

$ cat pagerank.txt |python pg\_mapper.py | **sort -k1n** | python pg\_reducer.py

1 0.216666666667 2

2 0.25 1 3 4

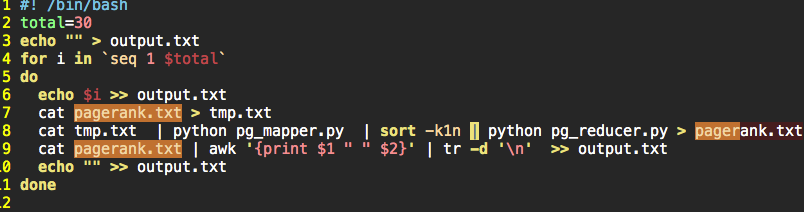
3 0.116666666667 4

4 0.416666666667 4 1

**Note**: We sorted the mapper’s output by the first column(“sort -k1n”). In Hadoop, mapper will sort output by key (default key is the first column in Hadoop streaming) automatically

Next, we iterate Mapreduce task 30 times. We accomplish this in a simple bash script:

backEnd/pagerank\_mapreduce/MR\_iteration.sh



We can notice that after 20 iterations, PageRank converges (first 5 decimals)

cat output.txt

1

1 0.216666666667 2 0.25 3 0.116666666667 4 0.416666666667

2

1 0.283333333333 2 0.223333333334 3 0.116666666667 4 0.376666666667

3

1 0.260222222222 2 0.276666666666 3 0.109555555556 4 0.353555555556

4

1 0.2652 2 0.258177777778 3 0.123777777778 4 0.352844444445

5

1 0.259985185186 2 0.26216 3 0.118847407407 4 0.359007407408

6

1 0.263512296297 2 0.257988148149 3 0.119909333333 4 0.358590222222

7

1 0.262232928395 2 0.260809837038 3 0.118796839506 4 0.358160395062

8

1 0.262813447902 2 0.259786342716 3 0.119549289877 4 0.357850919506

9

1 0.26241672586 2 0.260250758322 3 0.119276358058 4 0.358056157762

10

1 0.262622665324 2 0.259933380688 3 0.119400202219 4 0.35804375177

11

1 0.262533068891 2 0.260098132259 3 0.119315568183 4 0.358053230667

12

1 0.262580794203 2 0.260026455113 3 0.119359501936 4 0.358033248749

13

1 0.262553687529 2 0.260064635362 3 0.11934038803 4 0.358041289078

14

1 0.262567085061 2 0.260042950023 3 0.11935056943 4 0.358039395485

15

1 0.262560544867 2 0.260053668049 3 0.119344786673 4 0.358041000411

16

1 0.262564044978 2 0.260048435894 3 0.119347644813 4 0.358039874316

17

1 0.262562199298 2 0.260051235982 3 0.119346249572 4 0.358040315149

18

1 0.262563122321 2 0.260049759438 3 0.119346996262 4 0.358040121979

19

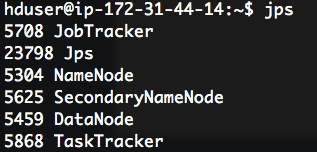
**1 0.262562651308 2 0.260050497857 3 0.119346602517 4 0.358040248318**

**20**

**1 0.262562898756 2 0.260050121046 3 0.119346799429 4 0.358040180769**

### Run Mapreduce on real Hadoop

Due to time limitation, we setup Hadoop on a single node to demonstrate out design. We followed article [4] to setup single a node on AWS. This node runs the following components including HDFS.



### Copy to HDFS

First we copy our simple 4x4 matrix to HDFS:

hadoop dfs -copyFromLocal ./pagerank.txt pagerank/pagerank.txt

*In this demo, we use Hadoop streaming feature to run python mapper and reducer. Basically, Hadoop streaming uses stdin/stdout to handles input/output data. So mapper/reducer consumes stdin and then yield stdout.*

### Run Mapreduce

> hadoop jar /usr/local/hadoop/contrib/streaming/**hadoop-streaming-1.0.3.jar**

-**mapper** "python/home/hduser/pl\_works/backEnd/pagerank\_mapreduce/**pg\_mapper.py**"

-**reducer** "python/home/hduser/pl\_works/backEnd/pagerank\_mapreduce/**pg\_reducer.py**"

-**input** "pagerank/pagerank.txt"

-**output** "pagerank/pagerank\_1.result"

Copy result file from HDFS to local

hadoop dfs -copyToLocal pagerank/pagerank\_1.result .

cat pagerank\_1.result/part-00000 | sed 's/\t/ /g'

1 0.216666666667 2

2 0.25 1 3 4

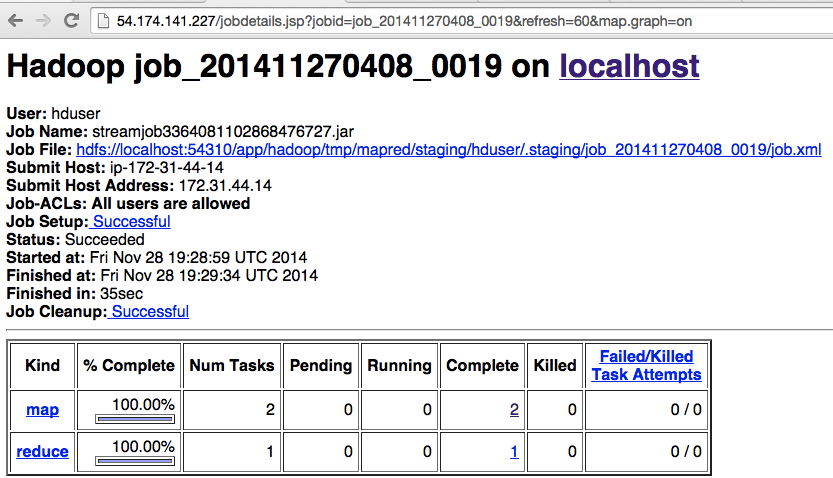
3 0.116666666667 4

4 0.416666666667 4 1

Solution matches command line testing which demonstrated in previous section.

### Mapreduce job detail

Below screenshot displays the job details. Although the input file has only 4 lines(4x4 matrix), the job assigned two mappers, and only reducer. The runtime is 35 sec on Hadoop which implies the overhead is significant in Hadoop.



Improved Mapper/Reducer:

Inspired by Prof. Zhu’s lecture, instead of iterate stdin directly, we wrapped stdin into a generator. Generator saves memory and computation resource. Imagine input stdin is huge, without generator, the program saves all the stdin into memory and then iterate. However, with generator, each iterable element is generated on the fly.

#TODO screen shot of improved mapper reducer

### Conclusion:

We didn’t get any improvement by using Hadoop because 1) Hadoop should be used to analyze “big data”(terabytes of data) on many nodes, and our 2) our dataset is too small and Hadoop overhead is huge. However, theoretically, if we have billions of webpages, we can take advantage of HDFS that stores data across multiple nodes. Furthermore, HDFS is faults tolerant, so if one node is down, replicated data will be available on another node. Moreover, the Mapreduce can distribute workload to multiple machines. Therefore, Hadoop is a better choice if data set is huge.