

## **CSC326 Final Project report**

Lizhou Wang  
998327733

Qiuyi Guan  
998373175

## Part 1 Design highlights

In this part, we will discuss features we developed to enhance the search engine:

### *Front end:*

We designed our website mobile/tablet friendly by implementing Bootstrap. In frontend, we implemented an auto-correction system (mimic Google's "Did you mean: xxx")

### *Back end:*

We used Mapreduce to calculate the PageRank (on Hadoop).

We implemented a multithread crawler.

We used mongoDB as our data warehouse.

### *Deployment:*

We load balanced traffic to different cores, and got better max number of concurrent connection. Used async server (tornado) instead of default multithread in bottle.

Our code is available in lab4 submission file as well as Github:

[https://github.com/wlz1028/pl\\_works](https://github.com/wlz1028/pl_works)

## 1.1 Calculating PageRank using Mapreduce (Hadoop)

### *Motivation*

Google introduced Mapreduce to calculate PageRank score on billions of webpages more than 100 times everyday. It's impossible to fit all the data on a single machine, because of limited compute resources. Google simply distribute work to thousands of cheap computers. Google file system is a distributed file system which responsible to store all the pages. 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 (references)*

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 data 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**

$$M = \begin{bmatrix} 0 & 1/3 & 0 & 1/2 \\ 1/2 & 0 & 0 & 0 \\ 0 & 1/3 & 0 & 0 \\ 1/2 & 1/3 & 1 & 1/2 \end{bmatrix}$$

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  $P(\text{page2})=1/4*1/2$ , and  $P(\text{page1})=1/4*1/2$ .

### *Reducer*

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

$$pr = \text{beta} * (p1 + p2 + \dots) + (1 - \text{beta}) * 1/n$$

Where  $\text{beta}$  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 task until PageRank 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:

```
lizwang-Air:pg lizwang$ cat l.txt
1      0.25  2
2      0.25  1      3      4
3      0.25  4
4      0.25  4      1
```

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

Then the mapper will output following data:

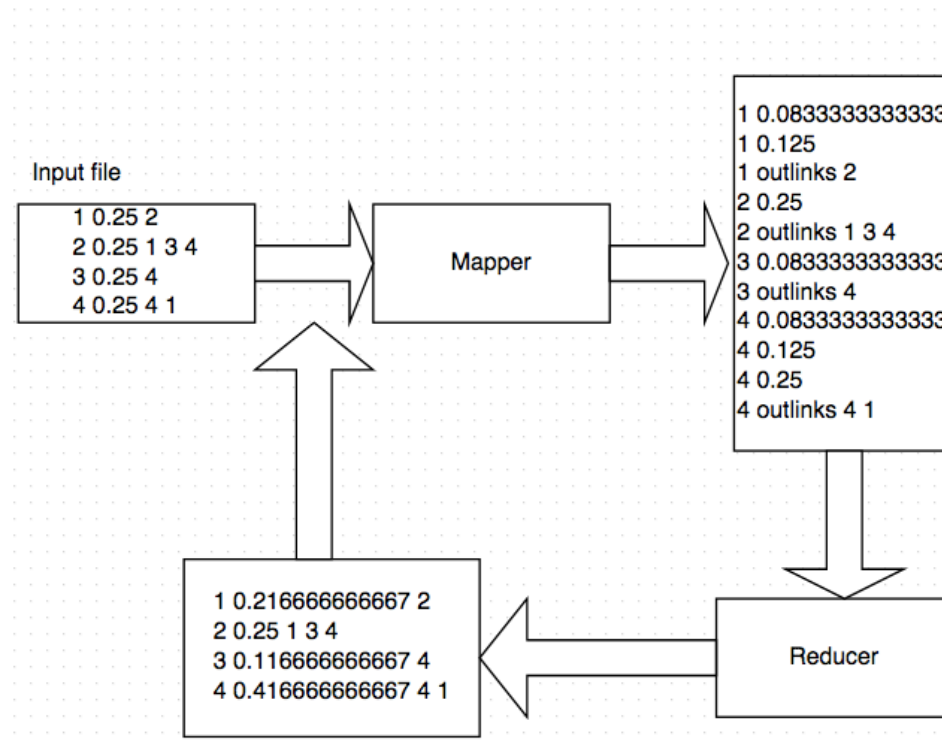
```
lizwang-Air:pg lizwang$ cat l.txt | python pg_mapper.py | sort
1      0.0833333333333
1      0.125
1      outlinks      2
2      0.25
2      outlinks      1      3      4
3      0.0833333333333
3      outlinks      4
4      0.0833333333333
4      0.125
4      0.25
4      outlinks      4      1
```

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

The 3<sup>rd</sup> 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 workflow:



### 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. However, generator doesn't work with linux pipe('|') command which will be used to test mapper and reducer.

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

```
1 import sys
2
3 for line in sys.stdin:
4     data = line.strip().split('\t')
5     page_id, pg, outlinks = data[0], float(data[1]), data[2:]
6     #output each outlink page id and its probability
7     for outlink in outlinks:
8         print '%s\t%s' % (str(outlink), str(pg/len(outlinks)))
9     #out put outlinks ids for each page for reducer
10    print page_id + "\toutlinks\t" + "\t".join(outlinks)
```

backEnd/pagerank\_mapreduce/pg\_reducer.py

```
1 import sys
2 |
3 last = None
4 pg = 0
5 beta = 0.8
6 N = 4
7 counter = 0
8
9 for line in sys.stdin:
10     data = line.strip().split('\t')
11     page_id,value = data[0],data[1]
12     #If new page_id detect, print previous page pagerank followed by outlinks
13     if last != None and page_id != last:
14         pg = beta * pg + (1 - beta) / N
15         print '%s\t%s\t%s' % (last,str(pg),outlinks)
16         pg =0
17     #save outlinks info for current page
18     if value == "outlinks":
19         outlinks = "\t".join(data[2:])
20         last = page_id
21         continue
22     #Acc values
23     pg += float(value)
24     last = page_id
25     counter += 1
26
27 #Print last line
28 pg = beta * pg + (1 - beta) / N
29 print '%s\t%s\t%s' % (last,str(pg),outlinks)
```

#### 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 mapper/reducer 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

```
1 #!/bin/bash
2 total=30
3 echo "" > output.txt
4 for i in `seq 1 $total`
5 do
6   echo $i >> output.txt
7   cat pagerank.txt > tmp.txt
8   cat tmp.txt | python pg_mapper.py | sort -k1n | python pg_reducer.py > pagerank.txt
9   cat pagerank.txt | awk '{print $1 " " $2}' | tr -d '\n' >> output.txt
10  echo "" >> output.txt
11 done
12
```

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 our design. We followed article [4] to setup single a node on AWS. This node runs the following components including HDFS.

```
hduser@ip-172-31-44-14:~$ jps
5708 JobTracker
23798 Jps
5304 NameNode
5625 SecondaryNameNode
5459 DataNode
5868 TaskTracker
```

### 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 0.2166666666667 2
2 0.25 1 3 4
3 0.1166666666667 4
4 0.4166666666667 4 1
```

Solution matches command line testing which demonstrated in previous section.

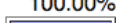
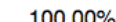
### Run Mapreduce on Hadoop

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.



## Hadoop job\_201411270408\_0019 on localhost

User: hduser  
Job Name: streamjob3364081102868476727.jar  
Job File: [https://localhost:54310/app/hadoop/tmp/mapred/staging/hduser/.staging/job\\_201411270408\\_0019/job.xml](https://localhost:54310/app/hadoop/tmp/mapred/staging/hduser/.staging/job_201411270408_0019/job.xml)  
Submit Host: ip-172-31-44-14  
Submit Host Address: 172.31.44.14  
Job-ACLs: All users are allowed  
Job Setup: [Successful](#)  
Status: Succeeded  
Started at: Fri Nov 28 19:28:59 UTC 2014  
Finished at: Fri Nov 28 19:29:34 UTC 2014  
Finished in: 35sec  
Job Cleanup: [Successful](#)

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	<a href="#">Failed/Killed Task Attempts</a>
<a href="#">map</a>	100.00% 	2	0	0	<a href="#">2</a>	0	0 / 0
<a href="#">reduce</a>	100.00% 	1	0	0	<a href="#">1</a>	0	0 / 0

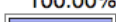
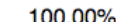
Next, we crawl on cbc.ca with depth=2, and generate a 16746 rows input file(which implies 16746 numbers of pages). The runtime is 37secs. Comparing with 4x4 matrix, the runtime increase 2 sec. We conclude that for Hadoop has huge overhead, so it's not good for small data set.

### Input file:

backEnd/pagerank\_mapreduce/cbc\_pr.txt

### Result screen shot

User: hduser  
Job Name: streamjob8233267745120620231.jar  
Job File: [https://localhost:54310/app/hadoop/tmp/mapred/staging/hduser/.staging/job\\_201411270408\\_0022/job.xml](https://localhost:54310/app/hadoop/tmp/mapred/staging/hduser/.staging/job_201411270408_0022/job.xml)  
Submit Host: ip-172-31-44-14  
Submit Host Address: 172.31.44.14  
Job-ACLs: All users are allowed  
Job Setup: [Successful](#)  
Status: Succeeded  
Started at: Sat Nov 29 18:21:04 UTC 2014  
Finished at: Sat Nov 29 18:21:42 UTC 2014  
Finished in: 37sec  
Job Cleanup: [Successful](#)

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	<a href="#">Failed/Killed Task Attempts</a>
<a href="#">map</a>	100.00% 	2	0	0	<a href="#">2</a>	0	0 / 0
<a href="#">reduce</a>	100.00% 	1	0	0	<a href="#">1</a>	0	0 / 0

### 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. However, like we mentioned before, we won't be able to test mapper/reducer by using pipe on command line.

Please see:

backEnd/pagerank\_mapreduce/pg\_mapper\_generator.py  
backEnd/pagerank\_mapreduce/pg\_reducer\_generator.py

### 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, Mapreduce can distribute workload to multiple machines. Therefore, Hadoop is a better choice if data set is huge.

### 1.2 Multithread crawler

The single thread crawler is slower because the crawler is blocked by HTTP read which is slow. Our multithread crawler opens a new thread to handle HTTP read, and doesn't block the crawler. Thread lock is used to keep common resources safe, such as url\_queue. Please see comment in crawler\_multi\_Thread.py for details

We also tested our multithread crawler:

#### \* **Robustness testing:**

\* Command:

```
- Run the following command 10 time
  > python crawler_multi_Thread.py
      # Default Setting
      #   URL = http://www.eecg.toronto.edu
      #   depth =2
```

\* Result:

- 10 runs without crash
- CPU 68% usage

#### \* **Correctness testing:**

\* Command:

```
python unit_test.py
```

\* Setting:

compare number of word\_id and url\_id again single thread

\* Result:

Some test runs failed due to different timeout in single thread and multithread. Otherwise, result is the same.

### 1.3 MongoDB

Instead of using 4 tables demonstrated in the handout, we created 2 collections (tables) in MongoDB. The advantage of MongoDB is that we don't need to define schema. For example, in lab4, we need to store webpage title and description into the database. In RDBMS (such as MySQL), we need to rewrite table schema, and update SQL statements. In MongoDB, we simply change each document (json) data structure.

### 1.4 Frontend mobile/tablet friendly

Mobile and tablet become extremely popular in recent years, so it becomes more important to design webpage which is mobile/tablet friendly.

There are several options to achieve this goal.

#### *Option 1:*

On the server side, the server can detect what kind of device is used by each user. The solution is that we can write multiple templates for each page that fit different resolutions. The server will handle which template to send.

The big disadvantage of this solution is that the website becomes hard to manage because more templates have to be managed. It also burdens the server, and causes poor performance.

#### *Option 2:*

A popular way to solve this is by using Bootstrap. Bootstrap contains HTML and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions. More importantly, Bootstrap adopted a mobile first design philosophy, emphasizing responsive design by default.

Bootstrap's Grid system can adjust webpage based on different screen resolution. This article explains how to use Bootstrap:

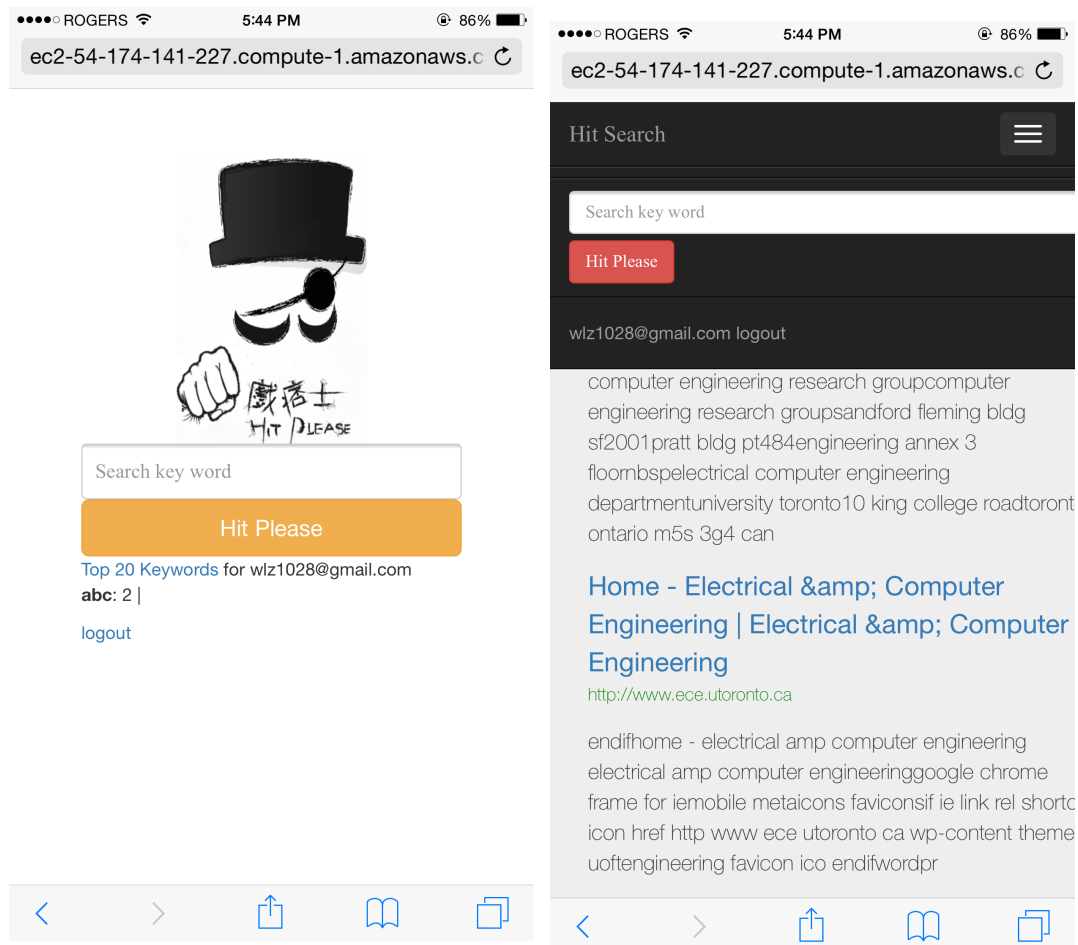
<http://getbootstrap.com/examples/grid/>

Since Bootstrap is a HTML and CSS based template, it doesn't burden the server. Browsers generate right webpage for different screen resolution. It's also easy for developer to maintain variable webpages.

#### *Design decision*

We pick option 2 to design a mobile/tablet friendly website because it's easier to maintain and performance better than option 1. Furthermore, there are tons of beautiful templates available.

Sample mobile display out put:



#### 1.4 Auto-correction

We take into account the possibility that a typo is encountered during the searching process. When user input a word that was not in our dictionary, the result page will have an option stating: "Did you mean: xxx". In which 'xxx' will be a word's correction based on our word\_correction algorithm.

The word\_correction algorithm was implemented based on Peter Norvig's article, "How to Write a Spelling Corrector" which can be viewed on line. Since each word can have multiple corrections, a probability theory was explained in the article to calculate the probability of each corrected word, where we used as the 'distance' from the corrected word to the original word. The estimated distance was saved in a model called WORDS\_COUNT.

Our algorithm first divides the string into multiple words and checks the correctness of each word separately. The algorithm computes all possibilities of the word by introducing 4 procedures: delete, transpose, replace and insert.

delete: go through each character and delete one at a time  
transpose: switch two adjacent characters  
replace: replace each character with 'a' to 'z'  
insert: insert one character at a time to different positions in the word

From above procedures, we save all possible words (words that is in our dictionary) into a set called candidate. Then, choose the element with the shortest distance, as estimated by the WORDS\_COUNT model.

## 1.5 Deployment

### Server choice:

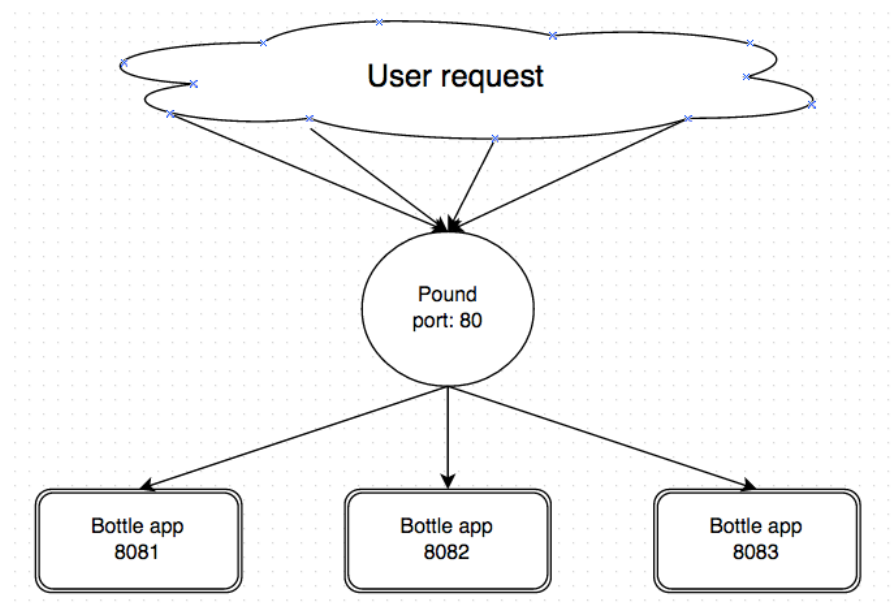
Instead of default multithread server in bottle, we switched to Tornado (Facebook use it). Tornado only uses one thread to handle concurrent connection. It's I/O non-block and uses async concept. For example, Tornado returns a state instead of waiting I/O request.

### Load Balancing:

We opened multiple bottle application on different ports (e.g. 8081,8082,etc.). Each application is a single thread because we picked Tornado as our server. We used a popular Reverse Proxy server called Pound to load balance income traffic to different port randomly. See below diagram.

*see deploy.sh for pound configuration*

*see frontEnd/master.sh for launching two bottle apps on two ports*



As result, we got 3000+ concurrent connections on query page(using mongoDB), and 4000+ concurrent connections on page without mongoDB. See [rpts/connection\\_test.txt](#) for test results.

## Part 2 Testing

### *Backend*

To design backend, we wrote unit test cases before coding. In order to identify corner cases, we wrote unit tests on something we can control instead of run it on real cases directly.

Crawler testing:

Instead of crawling on real websites, we wrote a local testing HTML file to validate the crawler. This is because real websites are not predicable. When we run the crawler on the same website multiple time, we got inconsistent results due to different timeouts, and updates on the websites. Therefore, we never know what is the correct result.

To test multithreading crawler, we run it on our local testing HTML. Once this test passes, we run both single and multi thread crawler on real websites, and compares their result. However, due to different timeout and network condition, the results are not identical in some cases. Therefore, we set a tolerance value to the tester.

To test persistent storage, we crawl real website, and compare in memory data with persistent data.

### *Frontend*

Instead of writing unit test cases, we record how do we use Google everyday, and then repeat it on our own website to test different functionalities.

## Part 3 Lesson learned

We learned the importance of label on our code. The entire lab was developed over a long period, sometimes we forgot what did we write in the previous functions or our to do list. By properly labeling the code, it saves us a lot of time remembering TODOs and the functionality of each step.

To keep track of our work, we used Github as our version control tool. Each of us works separately and commits all our work to Git. Since it was our first time using Git, we encountered many problems such as: we did not push our files correctly causing the partner may have worked on wrong version of the file.

## Part 4 Alternative

For the frontend, we should design our website mobile/tablet friendly at beginning. We had to rewrite some templates in the final project.

We spent a big amount of time on Hadoop, because we had zero experience on it. If we would have more time, we could setup Hadoop on multiple machines, and test our design on “big data”. Stanford University provides some real data (<http://snap.stanford.edu/data/>). We need to write another Mapreduce which converts Stanford data to our own data structure.

## Part 5 Course materials

Lectures introduced many useful python technics. For example, we improved mapper/reducer jobs by using generator we learned in class.

## Part 6 Time out side lab session

Lab 1: 2hrs (one week)

Lab2: 5hrs (one week)

Lab3: 8hrs (two weeks)

Lab4 50+ hrs (one week) time distributed as following:

FrontEnd:

- \* (3hrs) result page Title and description(require update db)
- \* (5hrs) tablet, phone(bootstrap) friendly
- \* (2hrs) search key word auto correction

BackEnd:

- (20hrs)PageRank map reduce
- \* Research 5 hrs
- \* Mapper/reducer 5 hrs
- \* CLI testing 2 hrs
- \* hadoop 8 hrs

deployment:

- \* (4 hrs)load balancing (pound) and multiple bottle apps on multiple cores  
<http://bottlepy.org/docs/dev/deployment.html>
- \* (4 hrs) One-click deployment script:

Report:

5 hrs

Others(e.g. debugging): 5 hrs

## **Part 7 Useful parts**

We found deployment is useful especially on scalability part. We learned how to use Pound to load balancing on multiple cores, which the lab should spend more time on it.

## **Part 8 Useless components**

We didn't find anything useless in the lab.

## **Part 9 Feedbacks:**

It would be better if we can have more time on last lab.

This project should have version control requirements. Our group used git (github) to manage our code.

## **Part 10 Workload distributions and responsibilities**

The workload of the lab was divided 50/50.

Qiuyi Guan is mainly in charged of front end, implementing some back end algorithms.

Lizhou Wand is mainly in charged of back end development and enhancement.