PageRank Calculation using Hadoop and Spark

Jungwon Seo

University of Stavanger, Stavanger, Rogaland 4036, Norway,
j.seo@stud.uis.no,
https://github.com/thejungwon/PagerankMRJob

Abstract. While showing great performance, PageRank requires computers to compute tons of operations. This paper introduces PageRank calculation in the Hadoop and Spark system that facilitate using a network of many computers to solve problems involving massive amounts of data and computation. This project demonstrates the step-by-step implementations while solving the challenges that occur each step. PageRank is calculated by the preprocessing step using MRJob and the algorithm step using both MRJob and Spark.

Keywords: PageRank, Hadoop, MapReduce, MRJob, Spark, Hive

1 Introduction

There are many areas including computer science, biology, economy, and society have a system that can be represented as a graph. Not like the tree structure or the list structure, the graph is often to be complicated to analyze it because of the many-to-many relationship. Especially, when the data size is huge, there are big obstacles to analyze the graph in terms of the storage space and the computation time. This situation makes it for us unavoidable to use the cluster-computing frameworks like Hadoop and Spark.

When analyzing the graph, the PageRank algorithm is often to be useful to find the meaningful node among the complicatedly related vertices. PageRank is the algorithm that is used by the Google Search Engine to measure the importance of the web-pages. According to Google: "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]"

In this paper we make the following key contributions:

- We use MRJob to preprocess the plain Wikipedia HTML files.
- We present the way to overcome the file split caused by the HDFS block storage system.
- We demonstrate the implementation of PageRank algorithm in two different way using MRJob implementation and Spark implementation.
- We compare the calculational performance for PageRank score between MRJob and Spark.

2 Background and Motivation

2.1 PageRank

PageRank is an algorithm that can analyze the link-based data with assigning a weight value to each linked target so that it can measure the relative importance of each data. This algorithm can be applied to any areas that contain the set of data with references. In the web area, PageRank is calculated based on the graph of web pages. Each of the nodes and edges is created by each of the pages and the hyperlinks respectively. The rank score shows the relative importance of a particular page. A hyperlink to a page is considered as a vote of support. Moreover, not like the algorithm

$$PR_i = 1 - d + d \sum_{j \in \{1,\ldots,n\}} \frac{PR_i}{c_j}$$

Fig. 1. PageRank Equation

simply counting the number of incoming links, a page that is linked to by many pages with high PageRank receives a high rank.

There are several ways to calculate the PageRank score. In our case, we decide to use the non-normalized version of the equation as shown in Fig.1. PageRank has the theory that a surfer who is randomly clicking on links will stop clicking at some point. The probability, at any step, that the person will continue is a damping factor d. We decided to set the damping factor to 0.85 based on various studies [2].

2.2 Use Case

As we mentioned above, the PageRank algorithm can be used for not only the search engine but also many different areas.

One example is, this score can be used for the airline companies when they have to decide the new route to extend their traveling area. For instance, there are some popular places only for local people. Even though this place has high potential to be a globally popular place, this place does not have an international airline. Therefore, many tourists have to go to this country's international airport first to transfer to this local place. If we use the PageRank algorithm, we can help the airline company to discover this hidden place and to decide the new route using relative importance [4].

Another example is, PageRank can be used to measure the value of the research papers. To measure the quality of the research paper, one of the important features that we can consider is the number of being cited by other papers. If many other research papers reference a certain paper, this paper can be considered important research result from this area. If we use the PageRank algorithm for this case, we can measure not only the simple number but also the relative importance [5].

2.3 Wikipedia Dataset

To test the PageRank, we decide to use Wikipedia dataset for two main reasons. The first reason is that the original data should be unorganized. Our primary purpose of this project is using the Hadoop and MapReduce as freely as possible. This project aims to handle big data to compute the complex algorithm. Therefore, we decide to use pure HTML dataset so that we can demonstrate the importance of preprocessing. The second reason is that we should know the page that will have the highest PageRank score. In this dataset, the highest score page is always 'index.html' because all pages contain the link to this page. If we do not know the answer to this algorithm first, we can not argue that our implementation works properly; hence, we did not exclude this index page on purpose.

The Wikipedia dataset can be found in https://dumps.wikimedia.org/ and we can also choose the language. For this project, we decide to use mainly German version dataset. The size is around 40GB and the number of pages is 2,681,947. The file structure is shown in Fig.2.

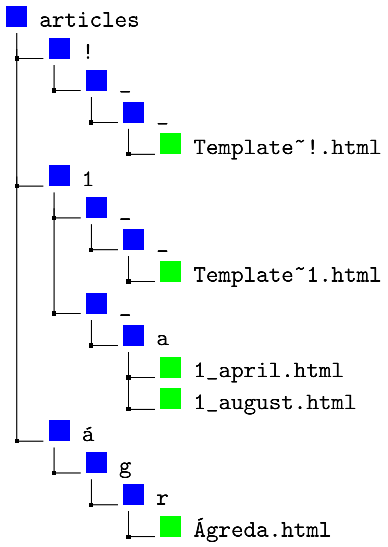


Fig. 2. File structure

3 Preprocessing

This section presents the preprocessing with a step-by-step approach, starting from the link-extraction using MRJob. From this link-extraction, we progressively address the challenges of extracting the link, overcoming the split page, and cleaning the useless links.

3.1 Preparation and Dataset Structure Analysis

```
7za e -so wikipedia.tar.7z | hadoop fs -put - /wiki/wikipedia.tar
```

Fig. 3. Direct Decompression to HDFS Using Pipe

The dataset of Wikipedia is initially compressed with tar and 7z, therefore, to read the data line-by-line we need to decompress it with 7z first. However the decompressed file can be bigger than one machine's storage space; therefore, we need to extract the original file directly to HDFS using a pipe as shown in Fig.3.

After the first decompression, the 'tar' compressed Wikipedia file is in the HDFS. We have to decide whether to decompress the tar compressed file or not. However, we do not need to decompress the tar version file for several reasons. First of all, we can directly read the tar file like a regular text file. Secondly, this original file is the set of HTML files, that means, if we decompress the file, it will be many small files, which can cause severe performance degradation in HDFS[3].

If we read the tar file, we can get the output as shown in Fig.4. When the single HTML file starts, it contains file name with the original path in the directory. Moreover, if we see line 4 and line 5, we can differentiate each file's starting point and endpoint with closing HTML tag and the next filename.

```
ie/index.html0000664000076500007640000006462111026364070013726 0ustar
    tstarlingwikidev<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="ie" lang="ie" dir="ltr">
...
</html>
ie/articles/!/_/_/Template~!.html0000664000076500007640000001305511026362620 0ustar
    tstarlingwikidev<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
...
```

Fig. 4. File structure of 'tar' compressed HTML files

3.2 Extracting Link from the HTML Files

The primary requirement when we extract the links is grouping the file name and the links that the current file contains. Therefore, we can use the multi-line input algorithm.

In our case, the starting line will be the line that contains the file name and *<!DOCTYPE html>*, and the ending line is closing HTML tag(</html>).

```
class MRLinkExtractor(MRJob):
  def mapper_init(self):
     self.from_page = ""
     self.in_links = False
     self.outgoing_pages = []
  def mapper(self, _, line):
      line = line.strip()
      if starting_condition(line) and not self.in_links:
          self.in_links = True
          self.from_page = line[line.find("articles"):line.find(".html")+5].split("/")[-1]
          self.outgoing_pages = []
      if ending_condition(line):
          if self.in_links:
              page_name = self.from_page
              yield page_name+".html", ' '.join(self.outgoing_pages)
          self.from_page = ""
          self.in_links = False
          self.outgoing_pages = []
      if link_condition(line) :
          link =  line[line.find("href=")+6:line.find(".html")+5].split("/")[-1]
          self.outgoing_pages.append(link)
```

Fig. 5. Preprocessing Naive MRJob Code

First, we implemented the naive multi-line input MRJob code. As shown in Fig.5, in the *mapper_init* phase, we can set some variables that this mapper will use until it finishes the work and in the *mapper* phase we can collect the links from each HTML file. After the mapper starts to work if it reaches the starting condition that was mentioned above, it stores the name of the current

HTML file. Then, the mapper keeps appending the link that the mapper finds while iterating the file line-by-line until it meets the ending condition that was also mentioned above.

However, this naive multi-line input algorithm has a critical limitation. When we consider the block storage architecture in HDFS, if the file size is bigger than the block size, HDFS will automatically split the original file based on its default block size. It means that some pages may not meet the ending condition when the mapper reaches the end of the block. Moreover, some mapper will skip the first few lines until it meets the first starting condition. For some cases, this a few missing data can be ignorable, however, in the PageRank algorithm, which is finding the most valuable page, this can cause a low reliable result. Moreover, this mapper cannot decide whether the link is meaningful or not because the mapper can see only one line. However, this problem will be solved partially in Section 4.

3.3 Split Page Merging

To solve the split block issue, we need to advance the naive multi-line input algorithm.

Before advancing the algorithm, if we analyze the data set again, we can find out that the HTML files in this tar file are sorted in alphabetical order. As shown in Fig.2, files start with the special character and ends with a special alphabet that is not in the regular English characters range. Therefore, this split page has a hint that can be used when we merge them later.

```python
# inside the starting condition in mapper
if not starting_condition(self.first_line) and \
not self.incomplete_sent:
    self.incomplete_sent = True
    page_name = self.from_page
    page_name = replace_useless_word(page_name)
    page_name = four_digit_escape(page_name.split(".")[0])
    if self.from_page :
        self.incomplete_pages = set(self.incomplete_pages)
        self.incomplete_pages = list(self.incomplete_pages)
        yield page_name+".htmk", ' '.join(self.incomplete_pages)
    self.incomplete_pages = []
```

Fig. 6. Yielding the lower half incomplete links

```python
def mapper_final(self):
    if self.in_links:
        page_name = self.from_page
        page_name=replace_useless_word(page_name)
        page_name = four_digit_escape(page_name.split(".")[0])
        self.outgoing_pages = set(self.outgoing_pages)
        self.outgoing_pages = list(self.outgoing_pages)
        yield page_name+".htmm", ' '.join(self.outgoing_pages)
```

Fig. 7. Yielding the upper half incomplete links

As shown in Fig.6-7, the advanced multi-line input algorithm has two additional part. Firstly, when the mapper meets the end of the block which is not the ending condition, this mapper will yield the incomplete page with the suffix 'htmm'. Secondly, if the beginning of the block is not the starting condition of HTML, then the mapper will keep storing the link until it reaches the new starting point. When the mapper meets the new starting point, it will yield all the links. However, this links has no page name that it should be grouped, therefore, we can temporarily use the new starting line's page name with suffix 'htmk'. There are two reasons that we use the suffix 'htmm' and 'htmk'. First of all, they are the mark that we can find later when we have to merge incomplete links or pages. Secondly, this can keep the order of HTML files. Especially, we are using the next starting point's name temporally, to avoid the mixing ordering we need to find the smaller or lower version of this page name. Therefore, this 'htmk' file will be always above or before the temporarily borrowed name.

```python
def four_digit_escape(string):
    string = ''.join(u'u%04x'%ord(char) for char in string)
    return string
```

Fig. 8. Converting to Unicode

For using this approach, we still have a critical issue to resolve. There are some file names that contain the character will be converted to Unicode. This partially converted Unicode will break the original order of files. In order to solve this problem, we convert all the characters to Unicode with the code shown in Fig.8, so that we can keep the total order of the original content.

```python
from pyhive import hive
cursor = hive.connect('localhost').cursor()
cursor.execute("SELECT * FROM `wiki` WHERE `id` LIKE\
'%.htmm%' OR `id` LIKE '%.htmk%'  ORDER BY `id` ASC")
rows = cursor.fetchall()
```

Fig. 9. HiveQL in vanilla Python code

Now we need to merge the incomplete pages that we have generated above. We can also use the MRJob for this process, however, we decide to use Hive when we merge the pages that end with 'htmm' or 'htmk'. There are several reasons that we choose to use Hive. Firstly, Hive gives a SQL-like interface to query data stored in various databases and file systems that integrate with Hadoop. It means as shown in Fig.9, with this simple query we can filter out the incomplete pages. Secondly, We can use Hive like the way to use a database in Python; therefore, it is easy to implement the merging process with the vanilla Python code. Lastly, HiveQL is also performed as Map-Reduce; hence, HiveQL will perform as much as or better than our best MRJob code because it is already optimized for the query to MapReduce conversion [10].

After merging the incomplete pages, we can initialize the data to be an input for the actual PageRank calculation. The simple MRJob code can process this task by counting the number of outgoing links, setting the initial score, and encoding to the JSON format.

```
"page1.html"
{"length": 9, "rank": 1.0,
"links": ["page2.html", "page3.html", "page4.html", "page5.html", ...]}
"page2.html"
{"length": 8, "rank": 1.0,
"links": ["page2.html","page3.html",...]}
...
```

Fig. 10. Output format after the pre-processing

Finally, we can finish the pre-processing with the output format shown in Fig.10. After the preprocessing, the size of data that we will use for the algorithm is reduced from 40GB to 7GB.

4 Algorithm and Implementation

Computing PageRank algorithm using Map-Reduce is very intuitive. If we analyze the equation shown in Fig.1, we can easily separate the equation into Mapper and Reducer respectively. We need to yield the data with two types to the reducer from the mapper. Firstly, we yield the entire line, which means the current page and its information includes the links. Secondly, we iterate the whole links in this page and yield each link with weight value to the reducer.

From the reducer part, we can collect all the link that has the same key. In this case, the key will be the page name that the outgoing edges are pointing out. Therefore, we can calculate the new PageRank score, and this reducer will yield the new data including the page name and the links to the mapper again. We can repeat this iteration until we achieve enough convergence using MRStep. The flow diagram of this calculation is shown in Fig.11.

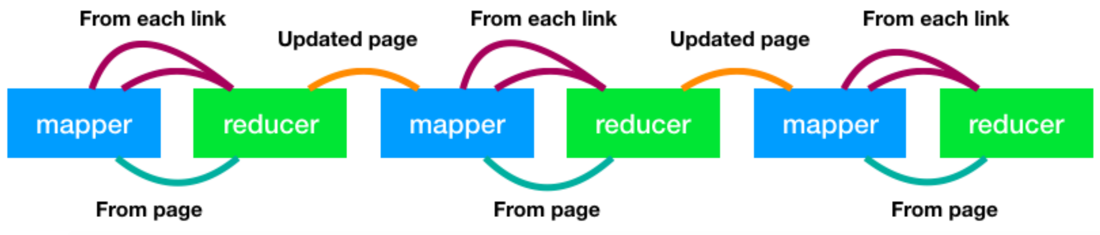


Fig. 11. PageRank calculation flow in Map-Reduce

4.1 Page Rank Implementation

As shown in Fig.12, in the mapper phase, it yields two types of data. The important point in this phase, we have to specify the type of data for the reducer phase; therefore, we have the 'page' type and 'rank' type. If we use the 'tuple' data structure in Python, we can send the complex type of data to the mapper or reducer step(line 3 to 8).

In the reducer phase, now we can collect all the links towards the one page. However, there are two types of data as we mentioned above. Therefore, we have to iterate the values while checking the type of each value. If the type is the 'page' type, we store the values into the page dictionary, and if we find the 'rank' type value, we can summarize that score to the total score value so that we can calculate the new PageRank for this current page (line 10 to 20).

Moreover, the PageRank calculation should be repeated to get a more reliable score. Which means, the page should be yield to mapper from the reducer again with updated PageRank score. For this task, we can use 'MRStep' so that we can specify the number of iteration (line 22 to 23).

4.2 Dealing with Sink Node

However, there is one critical issue for this calculation. In theory, if we initialize all the page with 1, no matter how many times we iterate the calculation, the sum of the rank of all the pages should be the same as the number of pages. In the first time, the sum of the PageRank score kept decreasing as many as we iterate the algorithm. This problem is caused by the dangling node.

To deal with this sink node problem, there are three well-known solutions [6]. First, eliminating such pages from the graph. Second, considering such pages to link back to the pages that link to them. Third, considering such pages to link to all web pages.

```
class MRPageRank(MRJob):
    INPUT_PROTOCOL = JSONProtocol
    def mapper(self, page_name, page):
        L = page.get("length")
        if L :
            yield page_name, ('page', page)
            for to_page in page.get('links'):
                yield to_page, ('rank', page['rank'] / L)
    def reducer(self, page_name, values):
        page = {}
        total_score = 0
        for which_type, value in values:
            if which_type == 'page':
                page = value
            elif which_type == 'rank':
                total_score += value
        d = 0.85
        page['rank'] = 1 - d + d * total_score
        yield page_name, page
    def steps(self):
        return ([MRStep(mapper=self.mapper, reducer=self.reducer)] * 4)
```

Fig. 12. PageRank using MRJob

Among these three approaches, we decide to use the first solution. The reason is, as we mentioned in Section 3, preprocessing with Mapper is not effective in terms of reliability. Due to the limitation of mapper that can only read the one line at that moment, it is difficult to eliminate the meaningless links. Therefore, we can organize the link with this additional cleaning process and solve the dangling node problem above all.

```
1  "page1.html"  "page2.html page3.html page4.html"
2  "page3.html"  "page5.html page6.html page7.html"
3  "page7.html"  ""
```

Fig. 13. Dangling Nodes Example

Eliminating the link should be considered in two cases. As shown in Fig.13, line 1 has a dangling node in the link section. line 3 has the page which does not contain any link. Therefore, if we eliminate the 'page7.html', this should be eliminated from line 2 as well.

Following code in Fig.14 demonstrate the three steps. First of all, the mapper yields the page with type 'page' and yield each link with type 'link' containing the page name as a value. Second of all, the reducer will filter out the link that does not have the page type, therefore, only the link which has the page will be yield. Finally, we use one more reducer to recover the data to the original format.

Now we can successfully calculate the PageRank score with following the total steps shown in Fig.15.

```python
class MRLinkCleaning(MRJob):
    def mapper(self, _, line):
        line=line.replace('"','')
        page, links = line.split("\t")
        page = page.split("/")[0]
        if len(links.split()):
            yield page, ('page','')
            for link in links.split():
                yield link, ('link',page)
    def reducer(self, page, values):
        page_exist = False
        from_pages = []
        for type, value in values:
            if type == "page":
                page_exist=True
            else :
                from_pages.append(value)
        if page_exist :
            for link in from_pages:
                yield link, page
    def reverse(self, page, values):
        yield page, ' '.join(values)
    def steps(self):
        return ([MRStep(mapper=self.mapper,
        ↪  reducer=self.reducer),MRStep(reducer=self.reverse)])
```

Fig. 14. Link Cleaning MRJob

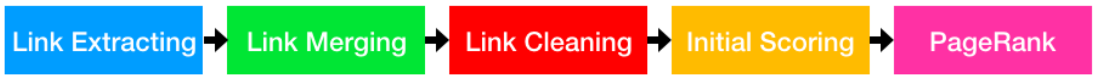


Fig. 15. Final Flow Diagram

However, there is a still unclear parameter that we have to decide. There is no fixed number of iteration to get the stabilized PageRank score [7]. Which means, the only way to determine the final PageRank score with the implementations that we have done is that iterating as many as possible or guessing the specific number.

To find the proper number of iteration, we can still use the MRJob implementation. However, this task will produce more MRJob code, besides, adding more step or job while calculating the PageRank, can degrade the performance of the entire task [8]. To overcome this inconvenience and performance issue, we decide to use Spark.

5 Spark

The limitation that Map-Reduce has that it maintains the full data to HDFS after finishing each job. From the storage perspective, this is a very expensive work because there will be at least two types of I/O operation for the disk and the network. However, in the Spark, when the output of a task needs to be passed to another task, Spark send the data directly without using the persistent storage [8].

5.1 Spark Implementation

```
 1 def weight(page,urls, rank):
 2     yield (page, 0)
 3     for url in urls: yield (url, rank / len(urls))
 4 def parse_link(links):
 5     links=links.replace('"','').split("\t")
 6     return links[0].split("/")[0],links[1].split()
 7 if __name__ == "__main__":
 8     sc = SparkContext()
 9     links = sc.textFile(sys.argv[1]).map(lambda links: parse_link(links)).cache()
10     ranks = links.map(lambda (page, links): (page, 1.0))
11     ranks_previous= ranks
12     number_of_iter=0
13     while True:
14         alllinks = links.join(ranks).flatMap(
15             lambda (page,(links, rank)): weight(page,links,rank))
16         ranks = alllinks.reduceByKey(add).mapValues(lambda rank: rank*0.85+0.15)
17         diff= ranks_previous.join(ranks).map(lambda rank:
            ↪  abs(rank[1][0]-rank[1][1])).sum()
18         ranks_previous= ranks
19         number_of_iter+=1
20         if diff<1: break #skipping printing result part
```

Fig. 16. PageRank Implementation in PySpark

Using Spark can improve not only the calculation performance but also convenience of implementation.

When we use MRJob right before calculating the PageRank score, we also used additional MRJob code to initialize the score and produced JSON formatted data. In Spark, we can skip that initialization process. In the PySpark code in Fig.16, using the only line 10 can finish the initial scoring process. From line 9 to 10 we assign to list for the link itself and rank respectively. In line 14, we join the link and rank list by the key and calculate all weight of link using the function in line 1. This step can be considered a mapper task in the MRJob code. After that, we can reduce all the value based on its page name so that we can calculate the new rank. The entire MRJob code can be represented only with these two lines (line14 and 16). Moreover, to find the proper number of iteration, we can compute the total sum of score difference between current and previous iteration. If the difference is small enough (less than 1), we can stop the iteration and finish the PageRank computation.

5.2 Spark Result

To be sure that we solve the sink problem, we tested with a smaller dataset which is Ireland Wikipedia as shown in Fig.17. There are 2,213 valid pages; therefore, the sum of the PageRank should always be 2,213. It took 25 times iteration to reach the convergence. Based on [7], regardless of the data size, this convergence can be achieved in a small number of iteration.

```
Convergence : 234.321546905
...
Convergence : 22.7668937758
...
Convergence : 4.33816766951
...
Convergence : 2.53492433543
...
Convergence : 1.03525408674
Convergence : 0.865503180282
...
Categories 101d.html has rank: 106.968044202.
Julian Mendez e1cf.html has rank: 133.535885018.
index.html has rank: 307.756154618.
Total Rank should be the same= 2213.0
Total number of interation 25
```

Fig. 17. Top20 PageRank(Ireland Wikipedia) Result from Spark

6 Analysis and Performance Tunning

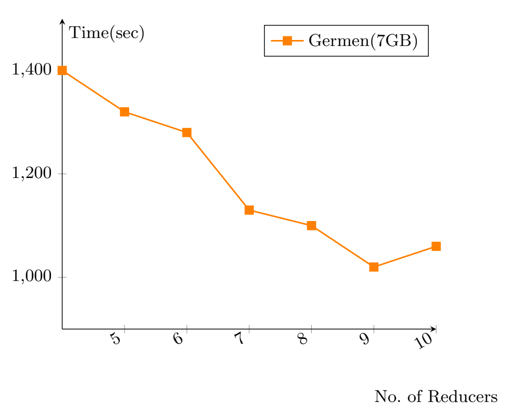
In this section, we mainly analyze MapReduce performance. To improve the performance of MapReduce, we should be aware of the factors that can affect. First of all, MapReduce programs read input data from disk, map a function across the data, reduce the results of the map, and store reduction results on disk. Secondly, there will be two types of I/O operation such as disk I/O and network I/O. Thirdly, the number of replicas can affect the amount of I/O operation. Lastly, from the memory perspective, using swap memory can affect the performance.

Type	Present Capacity(GB)	Memory(GB)	Number of Cores
Master	39	4	2
Slave	15.95	2	1
Slave Total	143.59	18	9

Fig. 18. Testing Environment Spec

6.1 Number of Reducers

We tested the performance difference based on the number of reducers. The default number of reducer was four in our case; however, this task did not work because of the disk space problem. We have noticed that every time the job finishes the task, it produces the intermediate file to deliver to the next job which can be reducer or mapper. If we use only four reducers that four machines cannot hold the entire dataset. Therefore we had to start the test with five reducers in German dataset case. As we expect from 5 to 9 the number of reducer shows the increased performance. However, the performance starts to decrease, when we use more than nine reducers, which is number more than the number of cores in our test environment. With different dataset, the total time is different. However, the graph shows the same symptom as shown in Fig.19.



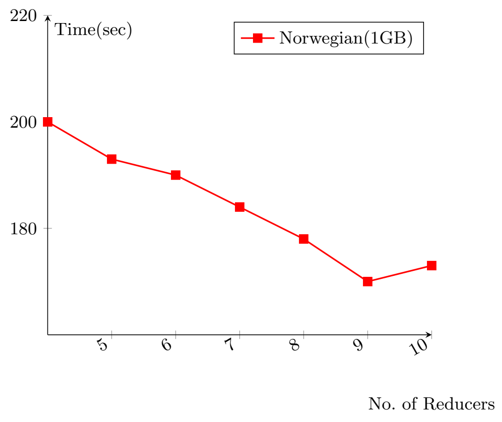


Fig. 19. Performance Comparison Between German and Norwegian Wikipedia

6.2 Number of Replicas

We also tested the calculation with the different number of the replica in HDFS. The original number of the replica was three; however, we could not finish the PageRank calculation, because even with 40GB data, it will be 120GB in HDFS; therefore, the task faced the disk space error. We could have successfully finished the job by setting the replica to two. Based on [11], if we reduce the number to one we assumed that we could have achieved better performance. However, if we use one replica, if one of the tasks fails, it can not retry the task again to the other replica. This situation makes it easy to fail the entire task. Therefore we decided to fix the number of replicas to two.

6.3 Using Combiner

```
def combiner(self, page_name, values):
    page = {}
    partial_score = 0.0
    pageExist = False
    linkExist = False
    for which_type, value in values:
        if which_type == 'page':
            page = value
            pageExist = True
        elif which_type == 'rank':
            partial_score += value
            linkExist= True
    if pageExist : yield page_name, ('page', page)
    if linkExist : yield page_name, ('rank', partial_score)
```

Fig. 20. Combiner MRJob Code

We also considered adding the combiner in the calculation[9]. Our assumption was if the combiner finishes the partial summation as shown in Fig.20. There will be less data that the reducer should iterate. However, the performance was worse than without using the combiner. Without combiner, it took an average 1,020 seconds; however, if we use the combiner, it took over 2,400 seconds. We assumed that the additional calculation in each data node was more burden than more iteration in the reducer.

6.4 Using Swap Space

When we run the job without any configuration for the first time, we could not finish the task. The reason that we found out was the out of memory issue. 2GB memory is not enough, because when it loads the job it already takes more than 1GB; therefore, it is easy to face the out of memory error. To solve this issue, we decide to put the swap space from the disk with the following command shown in Fig.21. We could successfully test the German dataset case with this additional swap space.

```
sudo fallocate -l 1024M /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
```

Fig. 21. Adding Swap Space for the Memory

6.5 Performance Comparison

We have tested the performance difference between MRJob and Spark under the condition that is three iterations of calculation and the 7GB cleaned German dataset. To achieve the best performance from MRJob, we set the number of reducer to 9 but no additional configuration for Spark. Even with the same condition, the test showed a different result in terms of time. Therefore we tested five times each and calculated the average time.

	MRJob	Spark
Time(sec)	1,020	231

Fig. 22. MRJob and Spark Performance Comparison

As shown in Fig.22, Spark shows more than four times better performance compared to MRJob.

7 Limitation

For this project, there are several limitations for both preprocessing and performance tuning.

First, using MRJob makes it challenging to extract a meaningful link. In other words, it is difficult to exclude the meaningless links located in sidebar or footer on the website.

Second, to tune the performance, there are many combinations that we have to consider at the same time. For example, Disk I/O, Network I/O, Memory usage, CPU overhead, Frequency of swap space usage and Network condition of the testing environment. In this project we could not fully consider each case; therefore it can show the different result with different environment or setting of configuration.

8 Conclusion

PageRank is a useful algorithm to find the relative importance of data in the graph data structure. To compute the massive amount of data that can not be calculated in one single machine is efficient in cluster-computing frameworks like Hadoop and Spark. Preprocessing task is the key phase of the entire project to make the algorithm work accurately. Using Spark can guarantee the benefit of both performance and implementation. We developed a PageRank calculation environment from the very beginning of the preprocessing to the two types of algorithm computation.

References

1. Facts about Google and Competition, Google Inc., 4 Nov. 2011, www.google.com/competition/howgooglesearchworks.html.

2. Brin, Sergey, and Lawrence Page. "The anatomy of a large-scale hypertextual web search engine." Computer networks and ISDN systems 30.1-7 (1998): 107-117.
3. Bende, Sachin, and Rajashree Shedge. "Dealing with small files problem in hadoop distributed file system." Procedia Computer Science 79 (2016): 1001-1012.
4. Xu, Kecheng, et al. "A Global Flight Networks Analysis Approach Using Markov Clustering and PageRank." 2017 IEEE International Conference on Big Knowledge (ICBK). IEEE, 2017.
5. Dellavalle, Robert P., et al. "Refining dermatology journal impact factors using PageRank." Journal of the American Academy of Dermatology 57.1 (2007): 116-119.
6. Ronny, et al. Introduction to Search Engine Technology. Yahoo Labs, Haifa, 2013.
7. Page, Lawrence, et al. The PageRank citation ranking: Bringing order to the web. Stanford InfoLab, 1999.
8. Zaharia, Matei, et al. "Spark: Cluster computing with working sets." HotCloud 10.10-10 (2010): 95.
9. Senger, Hermes, et al. "BSP cost and scalability analysis for MapReduce operations." Concurrency and Computation: Practice and Experience 28.8 (2016): 2503-2527.
10. Dokeroglu, Tansel, et al. "Improving the performance of Hadoop Hive by sharing scan and computation tasks." Journal of Cloud Computing 3.1 (2014): 12.
11. Berlińska, Joanna, and Maciej Drozdowski. "Scheduling divisible MapReduce computations." Journal of Parallel and Distributed Computing 71.3 (2011): 450-459.