COEN 242 Final Project Report (Team 15) "PageRank"

Vyoma Shah(*Author*)
Student, Dept. of Computer Science and Engineering
Santa Clara University
Santa Clara, US
vshah2@scu.edu - W1607985

Drashty Majmudar(*Author*)
Student, Dept. of Computer Science and Engineering
Santa Clara University
Santa Clara, US
dmajmudar@scu.edu - W1621128

Mitali Sahoo(*Author*)
Student, Dept. of Computer Science and Engineering
Santa Clara University
Santa Clara, US
msahoo@scu.edu - W1632271

Poonam Kanani(*Author*)
Student, Dept. of Computer Science and Engineering
Santa Clara University
Santa Clara, US
pkanani@scu.edu - W1620239

Abstract - This report is subject to our final project for Big Data (COEN 242) course at Santa Clara University, Spring 2022. As our final project, we implemented a PageRank algorithm over a Berkely-Stanford web graph dataset. We implemented using spark framework over the university linux cluster system.

Keywords — BigData, PageRank, pySpark, Naive, Power Iteration, Dataset

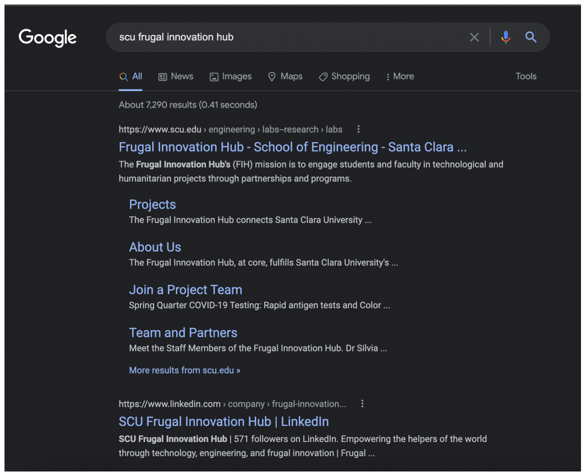
I. Introduction

It is an eminent fact that data transmission over the internet via computers has led to the third revolution of human civilization. It is an era of emerging technological revolution alongside industrial and agricultural revolutions. With the help of the internet, accessing information from all around the globe has become easier and is becoming faster progressively. If a person wishes to gather information on a specific topic, he can simply search on a search engine in the browser. Now-a-days, numerous search engines have emerged but it goes without saying that *Google* is colloquial with the term *search engine.* Moreso, it has been observed that Google has become the predominant search engine by achieving 92.47% market share as of June 2021. This sky-high achievement by Google includes various factors but there is one significant attribute that played a remarkable role: the PageRank algorithm invented by Larry Page, one of the founders of Google.

The very first search engines – JumpStation, World Wide Web Worm and other initial search engines used automatic programs called robots and spiders, to request web pages and return those requested web pages to users when found in a database. Page Rank algorithm helps us to retrieve information by ranking pages in such an order where pages with most accurate information will be ranked first hence appearing on the top. Google uses PageRank algorithm to rank the web pages and the main advantage of this algorithm is that it saves time of users as they do not have to scroll through various links to fetch the information. It is an iterative process and it works by counting the number of results appearing on search engines. Before page rank algorithm invention, it so happened that the user and the primary content of the web page were at odds with each other. This resulted in network traffic and disappointment among users.

II. Motivation

In this technological era, search engines play a vital role in surfing the web. Oftentime, search engines refer to Google, as it is now a quintessential part of innovation and our lives. Within a fraction of a second, anyone from anywhere sitting in one corner of the world can gain knowledge and can access information with the help of Google. Moreover, being curious about how these algorithms work behind this fancy front-end. We wanted to know about page rank in detail over high dimensions of a dataset where millions of rows and columns combine together. This final project lended a wonderful opportunity for us to implement and learn many new aspects of big data based platforms. To initialize this project and to understand page rank better we simply search "scu frugal innovation hub" on Google and look at what we observed! We thought of implementing the knowledge we gained on spark during our classes to requisite the program and were able to make visible evidence through our final project. These factors motivated and influenced our team to go ahead and make a contribution in this project. Big data - a term which I would state as the large data which might be structured and/or unstructured that is used to manage a dataset on a daily basis. Traditionally, big data was articulated by three V's: *Volume, Velocity and Variety*. In this tech emerging era, two more metrics have been added in traditional three V's and they are: *Variability and Veracity*.



About 7,290 results (0.41 seconds)

https://www.scu.edu › engineering › labs--research › labs

Frugal Innovation Hub - School of Engineering - Santa Clara ...

The **Frugal Innovation Hub's** (FIH) mission is to engage students and faculty in technological and humanitarian projects through partnerships and programs.

Projects

The Frugal Innovation Hub connects Santa Clara University ...

About Us

The Frugal Innovation Hub, at core, fulfills Santa Clara University's ...

Join a Project Team

Spring Quarter COVID-19 Testing: Rapid antigen tests and Color ...

Team and Partners

Meet the Staff Members of the Frugal Innovation Hub. Dr Silvia ...

More results from scu.edu »

https://www.linkedin.com › company › frugal-innovation...

SCU Frugal Innovation Hub | LinkedIn

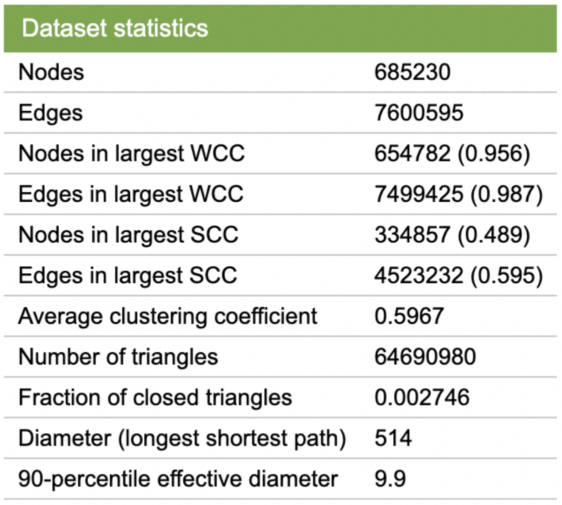
SCU Frugal Innovation Hub | 571 followers on LinkedIn. Empowering the helpers of the world through technology, engineering, and frugal innovation | Frugal ...

Figure 1: An example of the Google search result to show the number and quality of page rank algorithms. Around 7k important results were found in less than a second!!!

III. Dataset

The dataset which we used for implementing in our project was Berkeley-Stanford web data graph. It is a popular dataset and was also recommended by our professor on camino. It consists of numerous nodes and edges over which we can do calculations. The dataset link is as below: https://snap.stanford.edu/data/web-BerkStan.html

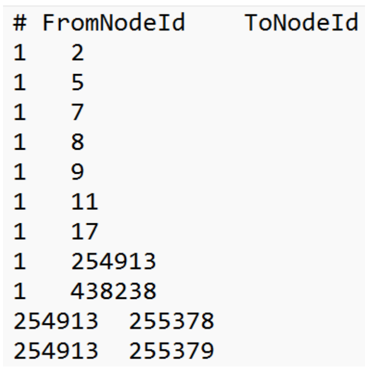
Below attached is the image of the dataset details consisting of nodes, edges, number of triangles, and other parameters.



Dataset statistics	
Nodes	685230
Edges	7600595
Nodes in largest WCC	654782 (0.956)
Edges in largest WCC	7499425 (0.987)
Nodes in largest SCC	334857 (0.489)
Edges in largest SCC	4523232 (0.595)
Average clustering coefficient	0.5967
Number of triangles	64690980
Fraction of closed triangles	0.002746
Diameter (longest shortest path)	514
90-percentile effective diameter	9.9

Figure 2: Metrics of Berkeley-Stanford web data graph

The dataset consists of two tab-separated columns namely “#FromNodeId” and “#ToNodeId”. The image below shows top few rows of the dataset:



```
# FromNodeId    ToNodeId
1   2
1   5
1   7
1   8
1   9
1   11
1   17
1   254913
1   438238
254913  255378
254913  255379
```

We performed some analysis on this dataset and table below shows some numbers:

Total nodes	685230
Total edges	7600595
Number of nodes with outgoing link	680486
Number of nodes with incoming link	617094
Nodes with max outgoing links	292640, 514788
Max outgoing links	249
Nodes with max incoming links	438238
Max incoming links	84208
Number of nodes with no outgoing link	4744
Number of nodes with no incoming link	68136

From the above table we can see that the number of nodes having at least 1 outgoing link is higher than that of nodes having at least one incoming link. Maximum number of outgoing links from a single node is 249, which is a really small fraction of the total nodes. Maximum number of incoming links are 84208, which is around 12% of all nodes in the dataset.

IV. Naive Implementation

The naive page rank algorithm is akin to an election where there are following rules:

- Each page has a score called PageRank score
- Each page contributes votes to the pages that it points to, thereby resulting in the score of each page equal to that page’s

current PageRank score divided by the number of pages it points to.

- The PageRank score of every page is re-calculated based on the total of all the votes it received.
- This process is repeated until a given condition is met (convergence condition).

According to the original paper on the PageRank algorithm by Sergey Brin and Lawrence Page, the PageRank for page A can be given as:

$$PR(A) = (1 - d)/N + d\,(PR(T1)/C(T1) + \ldots + PR(Tn)/C(Tn))$$

Here, it is assumed that pages T1...Tn which point to page A. d is a damping factor which is the probability, at any step, that any random surfer who is clicking on the links will continue to click on any random link. Its value is usually set to 0.85 and is between 0 and 1. C(A) is defined as the number of links going out of page A. N is the total number of pages and the probability of all pages sum to 1.

The pseudo-code of this algorithm is as follows:

```
Step-1: Read input file and
Step-2: initialize rank of each page as 1/N
Step-3: For i=1 to number of iterations:
1) Calculate the contribution of each node as: $\sum_{i}^{j} (r_i / d_i)$
2) Set rank of each page as:
$rank = (1 - d)/N + d * contribution$
Step-4: Repeat till results converge, which is achieved when until no PageRank value changes by more than $\epsilon = 0.001$ between the current rank values and the new rank values.
```

A. Configuration

In order for the program to run till completion, it is necessary to set the memoryOverhead parameters as follows:

i)`.config("spark.driver.memoryOverhead","2048")`

This is the amount of non-heap memory to be allocated per driver process in cluster mode.

ii)`.config("spark.executor.memoryOverhead","2048")`

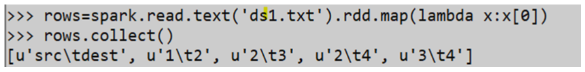
This is the amount of additional memory to be allocated per executor process.

iii)To set the default number of partitions in RDDs returned by the join, groupByKey, reduceByKey transformations, the below configuration is used:

`.config("spark.default.parallelism","<no_of_partitions>")`

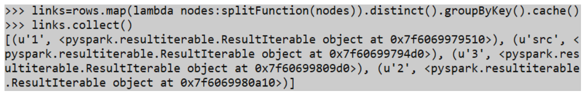
B. Code implementation:

i) Extracting each row into “rows” RDD:



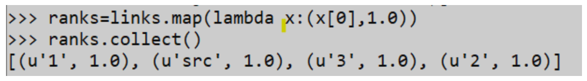
```
>>> rows=spark.read.text('ds1.txt').rdd.map(lambda x:x[0])
>>> rows.collect()
[u'src\tdest', u'1\t2', u'2\t3', u'2\t4', u'3\t4']
```

ii) “links” RDD is a collection of key-value pairs, where key is a node and values are the outgoing links for the key:



```
>>> links=rows.map(lambda nodes:splitFunction(nodes)).distinct().groupByKey().cache()
>>> links.collect()
[(u'1', <pyspark.resultiterable.ResultIterable object at 0x7f6069979510>), (u'src', <
pyspark.resultiterable.ResultIterable object at 0x7f60699794d0>), (u'3', <pyspark.res
ultiterable.ResultIterable object at 0x7f60699809d0>), (u'2', <pyspark.resultiterable
.ResultIterable object at 0x7f6069980a10>)]
```

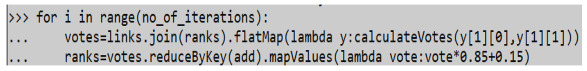
iii) Setting ranks of each node represented as key in “links” RDD to 1/N:



```
>>> ranks=links.map(lambda x:(x[0],1.0))
>>> ranks.collect()
[(u'1', 1.0), (u'src', 1.0), (u'3', 1.0), (u'2', 1.0)]
```

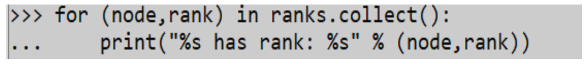
iv) Update the ranks for i=1 to i=number of iterations:

- Calculate votes to the rank of other nodes.
- Recalculate ranks based on neighbors' votes.



```
>>> for i in range(no_of_iterations):
...     votes=links.join(ranks).flatMap(lambda y:calculateVotes(y[1][0],y[1][1]))
...     ranks=votes.reduceByKey(add).mapValues(lambda vote:vote*0.85+0.15)
```

v) Print rank of each page.

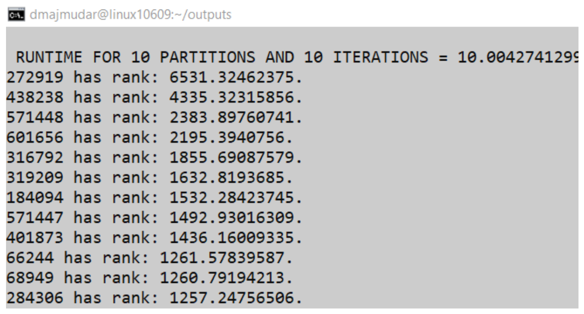


```
>>> for (node,rank) in ranks.collect():
...     print("%s has rank: %s" % (node,rank))
```

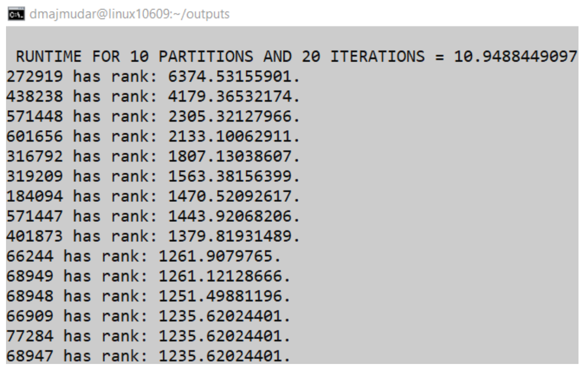
vi) Repeat the process till convergence.

C. Results:

Naive output:

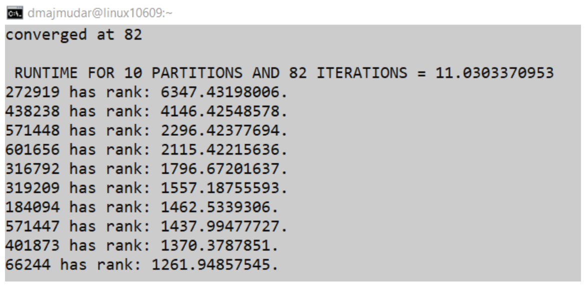


```
dmajmudar@linux10609:~/outputs

 RUNTIME FOR 10 PARTITIONS AND 10 ITERATIONS = 10.004274129
272919 has rank: 6531.32462375.
438238 has rank: 4335.32315856.
571448 has rank: 2383.89760741.
601656 has rank: 2195.3940756.
316792 has rank: 1855.69087579.
319209 has rank: 1632.8193685.
184094 has rank: 1532.28423745.
571447 has rank: 1492.93016309.
401873 has rank: 1436.16009335.
66244 has rank: 1261.57839587.
68949 has rank: 1260.79194213.
284306 has rank: 1257.24756506.
```



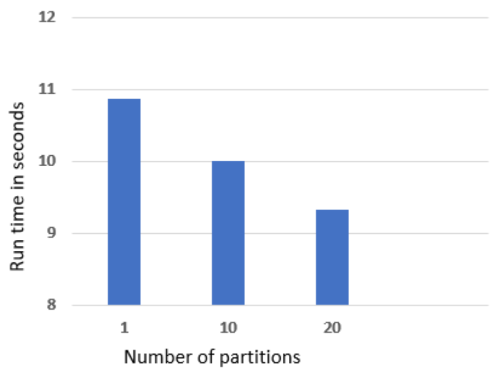
```
dmajmudar@linux10609:~/outputs

 RUNTIME FOR 10 PARTITIONS AND 20 ITERATIONS = 10.9488449097
272919 has rank: 6374.53155901.
438238 has rank: 4179.36532174.
571448 has rank: 2305.32127966.
601656 has rank: 2133.10062911.
316792 has rank: 1807.13038607.
319209 has rank: 1563.38156399.
184094 has rank: 1470.52092617.
571447 has rank: 1443.92068206.
401873 has rank: 1379.81931489.
66244 has rank: 1261.9079765.
68949 has rank: 1261.12128666.
68948 has rank: 1251.49881196.
66909 has rank: 1235.62024401.
77284 has rank: 1235.62024401.
68947 has rank: 1235.62024401.
```

- Convergence at 82nd iteration with stop criterion $\epsilon = 0.01$ and l_1 norm.

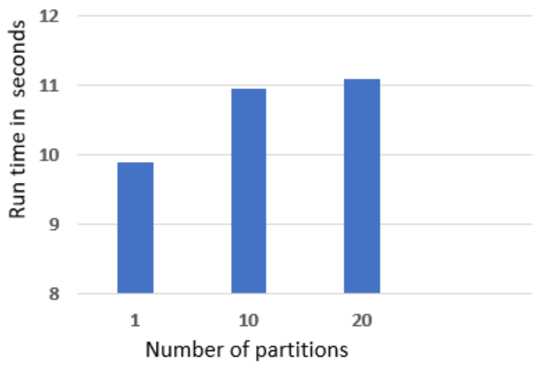


```
dmajmudar@linux10609:~
converged at 82

 RUNTIME FOR 10 PARTITIONS AND 82 ITERATIONS = 11.0303370953
272919 has rank: 6347.43198006.
438238 has rank: 4146.42548578.
571448 has rank: 2296.42377694.
601656 has rank: 2115.42215636.
316792 has rank: 1796.67201637.
319209 has rank: 1557.18755593.
184094 has rank: 1462.5339306.
571447 has rank: 1437.99477727.
401873 has rank: 1370.3787851.
66244 has rank: 1261.94857545.
```

- Graphs indicating runtimes for different number of partitions and iterations



Runtimes for 10 iterations



Runtimes for 20 iterations

V. Power Iteration Implementation

Power iteration is another method for computing the eigenvalue of a matrix. A very useful feature of the power method is that it not only produces the dominant eigenvalue, but it also produces a corresponding eigenvector. Power iteration is a well-known method used by Google to compute the PageRank for the web in practice.

The iterative method to calculate the Page Rank is as follows:

- Pick an initial guess v_0 (v_0 is a vector)
- $v_1=Mv_0$
- $v_2=Mv_1=M_2v_0$
- $v_3=Mv_2=M_3v_0$
- … …
- Compute v_n until it converges (e.g., $|v_n-v_{n-1}|<\varepsilon$ where ε is a small value)

$$PR(p_i) = \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

PR(Pi) is the page rank value of a node, which takes into account, three major attributes in calculating the page rank score, that is quality (how important the incoming page) node P_j is with $PR(P_j)$ term, quantity (how many pages are linking to the page P_i), and how exclusively the neighboring pages P_j are linking to the page P_i.

The pseudo code of the algorithm is given as below:

Step-1: Read the file and load the edge with source and destination node values into RDD.

Step-2: Initialize ranks of nodes to 1/N.

Step-3: Using map and reduce function, iterate through the page rank of each node until convergence.

Step-4: With every iteration, there is a new page rank value assigned to each node. This is computed as the previous page rank score value divided by the number of outgoing links.

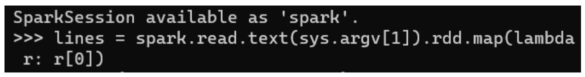
Step-5: Repeat the iterations until the any two consequent page rank score values has no more than $\epsilon = 0.001$ difference.

A. Configuration

Follow the same configuration as given for naive implementation in IV. A.

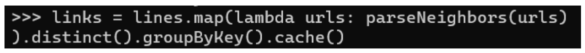
B. Code implementation:

i) Read each line from the file and map to rdd.



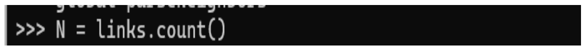
```
SparkSession available as 'spark'.
>>> lines = spark.read.text(sys.argv[1]).rdd.map(lambda
 r: r[0])
```

ii) pairs of node values are created which are then grouped in links value

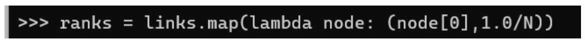


```
>>> links = lines.map(lambda urls: parseNeighbors(urls)
).distinct().groupByKey().cache()
```

iii)Counts the number of nodes and assign (1/total no of nodes) rank to each node



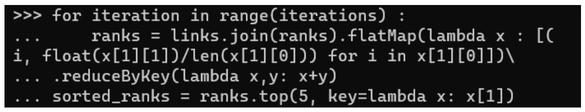
```
>>> N = links.count()
```



```
>>> ranks = links.map(lambda node: (node[0],1.0/N))
```

iv) Update the ranks for i=1 to i=number of iterations:

- Calculate the rank of each node by dividing the previous iteration's page rank value by number of outgoing links.
- Sort the rank values of nodes based on the page rank score in descending order.

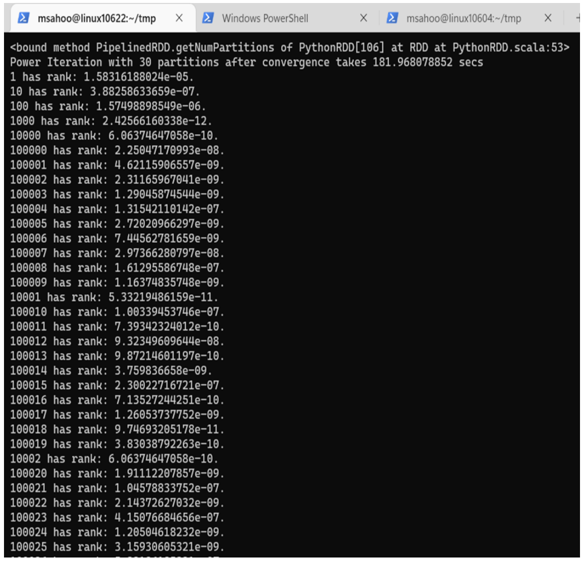


```
>>> for iteration in range(iterations) :
...     ranks = links.join(ranks).flatMap(lambda x : [(
i, float(x[1][1])/len(x[1][0])) for i in x[1][0]])\
... .reduceByKey(lambda x,y: x+y)
... sorted_ranks = ranks.top(5, key=lambda x: x[1])
```

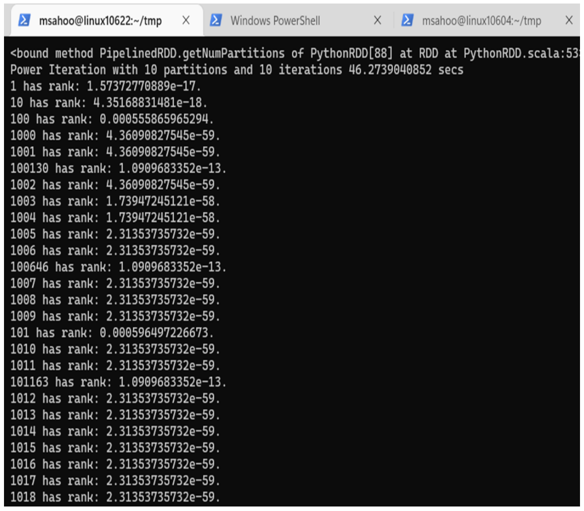
v) Repeat the above iterations until convergence.

C. Results

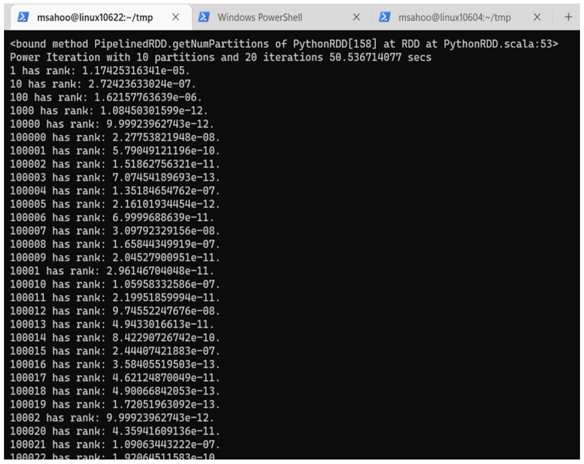
- Power Iteration output:



```
<bound method PipelinedRDD.getNumPartitions of PythonRDD[106] at RDD at PythonRDD.scala:53>
Power Iteration with 30 partitions after convergence takes 181.968078852 secs
1 has rank: 1.58316188024e-05.
10 has rank: 3.88258633659e-07.
100 has rank: 1.57498898549e-06.
1000 has rank: 2.42566160338e-12.
10000 has rank: 6.06374647058e-10.
100000 has rank: 2.25047170993e-08.
100001 has rank: 4.62115906557e-09.
100002 has rank: 2.31165967041e-09.
100003 has rank: 1.29045874544e-09.
100004 has rank: 1.31542110142e-07.
100005 has rank: 2.72020966297e-09.
100006 has rank: 7.44562781659e-09.
100007 has rank: 2.97366280797e-08.
100008 has rank: 1.61295586748e-07.
100009 has rank: 1.16374835748e-09.
10001 has rank: 5.33219486159e-11.
100010 has rank: 1.00339453746e-07.
100011 has rank: 7.39342324012e-10.
100012 has rank: 9.32349609644e-08.
100013 has rank: 9.87214601197e-10.
100014 has rank: 3.759836658e-09.
100015 has rank: 2.30022716721e-07.
100016 has rank: 7.13527244251e-10.
100017 has rank: 1.26053737752e-09.
100018 has rank: 9.74693205178e-11.
100019 has rank: 3.83038792263e-10.
10002 has rank: 6.06374647058e-10.
100020 has rank: 1.91112207857e-09.
100021 has rank: 1.04578833752e-07.
100022 has rank: 2.14372627032e-09.
100023 has rank: 4.15076684656e-07.
100024 has rank: 1.20504618232e-09.
100025 has rank: 3.15930605321e-09.
```

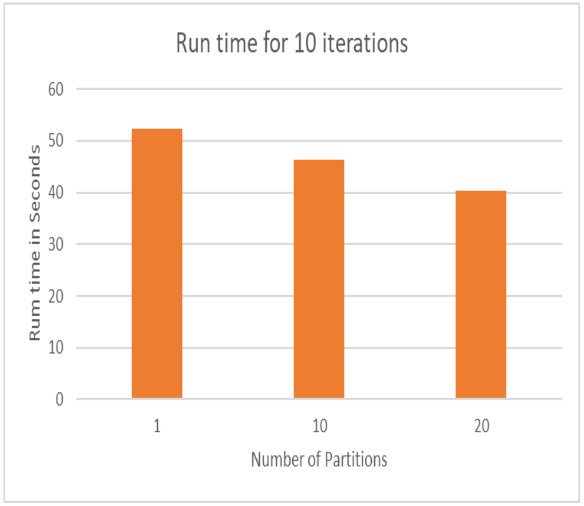


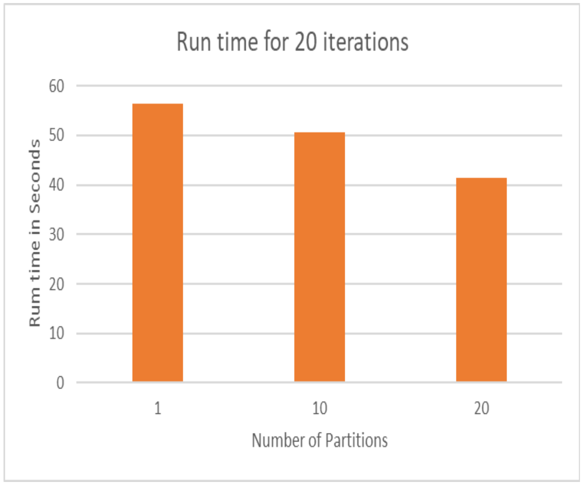
```
<bound method PipelinedRDD.getNumPartitions of PythonRDD[88] at RDD at PythonRDD.scala:53
Power Iteration with 10 partitions and 10 iterations 46.2739040852 secs
1 has rank: 1.57372770889e-17.
10 has rank: 4.35168831481e-18.
100 has rank: 0.000555865965294.
1000 has rank: 4.36090827545e-59.
1001 has rank: 4.36090827545e-59.
100130 has rank: 1.0909683352e-13.
1002 has rank: 4.36090827545e-59.
1003 has rank: 1.73947245121e-58.
1004 has rank: 1.73947245121e-58.
1005 has rank: 2.31353735732e-59.
1006 has rank: 2.31353735732e-59.
100646 has rank: 1.0909683352e-13.
1007 has rank: 2.31353735732e-59.
1008 has rank: 2.31353735732e-59.
1009 has rank: 2.31353735732e-59.
101 has rank: 0.000596497226673.
1010 has rank: 2.31353735732e-59.
1011 has rank: 2.31353735732e-59.
101163 has rank: 1.0909683352e-13.
1012 has rank: 2.31353735732e-59.
1013 has rank: 2.31353735732e-59.
1014 has rank: 2.31353735732e-59.
1015 has rank: 2.31353735732e-59.
1016 has rank: 2.31353735732e-59.
1017 has rank: 2.31353735732e-59.
1018 has rank: 2.31353735732e-59.
```



```
<bound method PipelinedRDD.getNumPartitions of PythonRDD[158] at RDD at PythonRDD.scala:53>
Power Iteration with 10 partitions and 20 iterations 50.536714077 secs
1 has rank: 1.17425316341e-05.
10 has rank: 2.72423633024e-07.
100 has rank: 1.62157763639e-06.
1000 has rank: 1.08450301599e-12.
10000 has rank: 9.99923962743e-12.
100000 has rank: 2.27753821948e-08.
100001 has rank: 5.79049121196e-10.
100002 has rank: 1.51862756321e-11.
100003 has rank: 7.07454189693e-13.
100004 has rank: 1.35184654762e-07.
100005 has rank: 2.16101934454e-12.
100006 has rank: 6.9999688639e-11.
100007 has rank: 3.09792329156e-08.
100008 has rank: 1.65844349919e-07.
100009 has rank: 2.04527900951e-11.
10001 has rank: 2.96146704048e-11.
100010 has rank: 1.05958332586e-07.
100011 has rank: 2.19951859994e-11.
100012 has rank: 9.74552247676e-08.
100013 has rank: 4.9433016613e-11.
100014 has rank: 8.42290726742e-10.
100015 has rank: 2.44407421883e-07.
100016 has rank: 3.58405519503e-13.
100017 has rank: 4.62124870049e-11.
100018 has rank: 4.90066842053e-13.
100019 has rank: 1.72051963092e-13.
10002 has rank: 9.99923962743e-12.
100020 has rank: 4.35941609136e-11.
100021 has rank: 1.09063443222e-07.
100022 has rank: 1.92064511583e-10
```

- Graphs indicating runtimes for different number of partitions and iterations





VI. Results

How does Pagerank help assess the importance of each webpage?

Earlier search engines determined the importance of a page by counting the number of incoming links for a particular webpage. However spammers can simply create many webpages and link them to unimportant webpages, increasing its ranking. Pagerank on the other hand doesn't consider all incoming links to be equally important. It decides the value of each incoming link based on the total number of outgoing links from the source webpage. This helps discredit pages that create spam outgoing links to increase the value of unimportant pages. This still creates the issue of the initial importance of the page. To solve this, pagerank uses an iterative method to determine weight given to each page.

Does parallelisation speed up the non-parallelized implementation?

During the execution of the algorithm it was noted that it took a significantly long time to reach the convergence condition when there was no parallelism. On the other hand, a high number of partitions increases the number of shuffle operations, resulting in high communication cost.

Why is Spark used to deal with this dataset and problem?

As mentioned in section III, the dataset consists of a large number of records. Also, the algorithm requires several transformations and actions to be performed on this data. Spark has the capability to process large amounts of data in-memory. Additionally, it makes parallelism easy and optimizes computation and execution through lazy evaluation and pipelining. Hence, Spark is the ideal choice to implement the PageRank algorithm on this dataset.

VII. Conclusion

We use search engines like Google everyday to retrieve information. By implementing Pagerank we learned how this information actually reaches us. Pagerank is a simple algorithm to understand but has many ways to implement it. In our project we have used big data concepts using the spark framework. We have implemented a naive and power iteration method and experimented with a different number of partitions. We found that performance improves with an increase in the number of partitions until a certain point and after that it starts degrading. So the parallelism factor needs to be optimal for best performance.

VIII. References

[1]Tiefengeist. "A Simplified Implementation of PageRank." Medium. Medium, July 9, 2019. https://medium.com/@sahirnambiar/a-simplified-implementation-of-pagerank-b8b5d282dc42.

[2]The pagerank computation. Accessed June 8, 2022. https://nlp.stanford.edu/IR-book/html/htmledition/the-pagerank-computation-1.html.

[3]Jen, George. “Page Rank with Apache Spark Graphx.” Medium. Medium, April 14, 2020. https://jentekllc8888.medium.com/page-rank-with-apache-spark-graphx-a51964467c56.

[4] Brin, Sergey, and Lawrence Page. “The Anatomy of a Large-Scale Hypertextual Web Search Engine.” Computer Networks and ISDN Systems. Elsevier, June 17, 1999. https://www.sciencedirect.com/science/article/abs/pii/S016975529800110X?via%3Dihub

[5] Su, Jessica. “Naive Formulation of Pagerank - Stanford University.” Accessed June 8, 2022. https://snap.stanford.edu/class/cs224w-2017/slides/handout-page_rank.pdf

[6] Wolohan, John T. “9.” Essay. In *Mastering Large Datasets with Python: Parallelize and Distribute Your Python Code*. Manning Publications Company, 2020.