CS 744 Assignment 1

Siddhant Garg *sgarg33@wisc.edu*
Varun Batra *vbatra@wisc.edu*
Rahul Jayan *jayan@wisc.edu*

October 2, 2018

1 Part-1: Setting Up Apache Hadoop and Spark

Using the instructions mentioned in the assignment problem statement, we tried setting up Apache Hadoop and Apache Spark on the cluster machines. Some issues we faced while setting up:

- After installing openjdk-8-jdk on all 4 machines, parallel ssh was not working directly as it was overriding the prompt asking for allowing to connect to the other machine. We resolved this by first using ssh from one machine to the other and then using parallel-ssh after the connection was once established.

- RSA keys to communicate between the leader and followers which were generated using ssh keygen on the leader were periodically getting deleted from the $/.ssh/authorized\_keys$ file and we resolved it by adding the ssh keys into cloudlab.

- During Apache Hadoop set up, we were facing an issue that only the namemode process was running on the leader while the datanode process was not running on the followers. We resolved this by adding the IP to hostname mapping of the leader in the /etc/hosts file.

- During experiments of pagerank, the datanode process for follower-1 was not starting and hence we rebooted follower-1 and re-stepped through the setup process. The datanode process then started running on follower-1

- During experiments of pagerank on the big data set, the process was running out of disk space on hdfs (as the dataset was occupying 29.7GB of 34.1GB space on HDFS disk)and giving an error. We resolved this by mounting the external 1 T disk on HDFS.

2 Part-2: Sorting

We used PySpark to write the Spark application for sorting the csv file. Our main code has an overview as follows:

```
my_rdd = sc.textFile(csv_file_path)
my_rdd = my_rdd.filter(lambda l:
not l.startswith("battery_level"))
sorted_rdd=my_rdd.sortBy(lambda line:
(line.split(",")[2],line.split(",")[-1])
```

- We read the csv file as a RDD and filter out the header of the file using *rdd.filter*
- Then we use *rdd.sortBy* on the RDD using keys of column 3 and the last column (namely **country code** and the **timestamp**)
- We then coalesce the sorted_rdd across partitions and write to a single file (part-00000) inside a folder (part2)

This code executes within a couple of seconds when run using spark-submit and hence we did not experiment on this part by changing the number of partitions, caching, etc.

3 Part-3: Pagerank

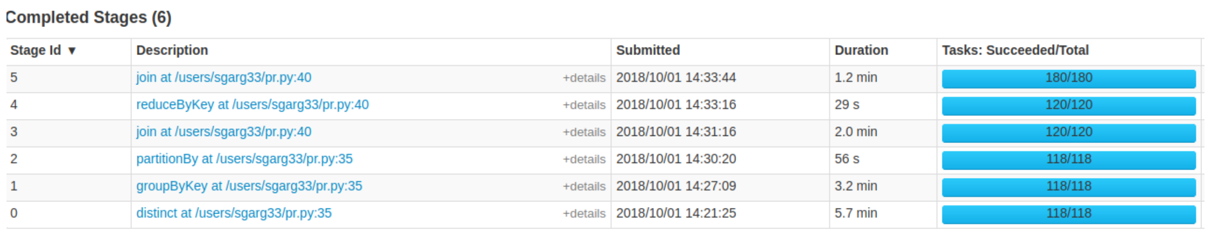
We used PySpark to write the Spark application for computing pagerank for a file having nodes and edges. Our main code has an overview as follows:

```
def calculate_pagerank(neighbors, current_rank):
    result=[]
    for n in neighbors:
        result.append((n, current_rank/len(neighbors)))
    return result

if __name__ == "__main__":
    sc = SparkSession.builder.appName("pagerank").getOrCreate()

    lines = sc.read.text(sys.argv[1]).rdd.map(lambda l: l[0])

    lines=lines.filter(lambda l: ("\t" in l) and ((":" not in l)
    or ("\tCategory:" in l)) and (not l.startswith("The files"))
    and (len(re.split(r"\t",l.strip()))>1))

    edges = lines.map(lambda node: (re.split(r"\t",node
    .strip())[0]
    .lower(), re.split(r"\t",node.strip())[1].lower() ))
    .distinct()
    .groupByKey().partitionBy(int(sys.argv[3]))
    .persist(pyspark.StorageLevel.MEMORY_AND_DISK)

    pagerank = edges.map(lambda edge: (edge[0], 1.0))

    for i in range(10):
        pagerank = edges.join(pagerank).flatMap(lambda item:
        calculate_pagerank(item[1][0], item[1][1]) )
        .reduceByKey(lambda a,b :
        a+b).mapValues(lambda r: 0.15 + 0.85*r)

    pagerank.coalesce(1).saveAsTextFile(sys.argv[2])
```

- We read the text file as a dataframe and convert it to a RDD using map and then filtering out important records using RDD.filter.
- We then transform a line of the form (node1 \t node2) using RDD.map to (node1 : node2) and then use RDD.groupByKey() to combine all neighbours of a node together in the form (node1 : [neighbours of node1]) in the RDD named edges
- We initialize pageranks of all nodes having outgoing edges with 1
- For every iteration, we combine a node's current pagerank with its neighbours using RDD.join and then use RDD.flatMap to return pagerank contributions of this node to all its neighbours. We then use RDD.reduceByKey and add the contributions of each node and finally apply the pagerank formula to iteratively compute the pagerank.

- We then coalesce the computed pagerank across partitions and write to a single file (part-00000) inside a folder (part3)

One observation we made which corroborates well with Spark paper is the fact that initial stages of execution take larger amounts of time than later stages [Since later on in-memory RDDs are exploited and Disk I/O overhead can be reduced]. This observation is made clear in the below image which shows execution times for starting stages of the pagerank compututation.



Completed Stages (6)

Stage Id ▼	Description		Submitted	Duration	Tasks: Succeeded/Total
5	join at /users/sgarg33/pr.py:40	+details	2018/10/01 14:33:44	1.2 min	180/180
4	reduceByKey at /users/sgarg33/pr.py:40	+details	2018/10/01 14:33:16	29 s	120/120
3	join at /users/sgarg33/pr.py:40	+details	2018/10/01 14:31:16	2.0 min	120/120
2	partitionBy at /users/sgarg33/pr.py:35	+details	2018/10/01 14:30:20	56 s	118/118
1	groupByKey at /users/sgarg33/pr.py:35	+details	2018/10/01 14:27:09	3.2 min	118/118
0	distinct at /users/sgarg33/pr.py:35	+details	2018/10/01 14:21:25	5.7 min	118/118

3.1 RDD Partitioning

A partition is a logical chunk of a large distributed data set. Spark manages data using partitions that helps parallelise distributed data processing with minimal network traffic for sending data between executors.

Here we use hash partitioning and partition the *edges* RDD from our code and compare the execution times, stages and tasks executed by the workers as we vary the number of partitions parameter value [repartition() is used for specifying the number of partitions considering the number of cores and the amount of data you have while partitionBy() is used for making shuffling functions efficient and is beneficial in cases where a RDD is used for multiple times].

Partitions	Tasks	Time (min)	Stages
3	745	82	24
6	1135	55	24
9	1525	43	24
12	1915	38	24
30	4255	35	24
60	8155	33	24
90	12055	37	24

Table 1: Changing number of partitions and observing effect on the execution time of the pagerank algorithm and number of tasks executed by Spark workers

Increasing the number of partitions leads to a decrease in the execution time initially as more processes can be parallelized and many more CPU cores can be utilized. However on increasing the number of partitions, the shuffle time(process of redistributing data across partitions) starts increasing. This shuffle time starts dominating over the computation time for very large values of partitions thereby negating the impact of partitioning on reducing execution time.

We also tried partitioning over just the *pagerank* RDD and not the edges RDD. Since pagerank is being updated every iteration, partitioning it did not provide improvement in execution time [On the contrary it increased the number of tasks by 10 more (one per iteration) as well as the number of tasks to be executed].

RDD Partitioned	Partitions	Tasks	Time (min)	Stages
edges	30	4255	35	24
pagerank	30	4795	48	34

Table 2: Changing the RDD being partitioned and observing effect on the execution time of the pagerank algorithm and number of tasks executed by Spark workers.

Apart from the default hash partition, using custom partitioning can also provide benefits as far as reducing the execution times. Since we use a join operation on edges and pagerank in every iteration, a lot of time is wasted as shuffle time across the partitions. To avoid this, we tried to partition pagerank in the same manner as edges by changing the code from : *pagerank = edges.map(lambda edge: (edge[0], 1.0))* to *pagerank = edges.mapValues(lambda edge: 1.0).* This leads to join being done in parallel inside each partition without shuffling. To try using custom partition over edges other than the default, we examined the execution DAG and computing the shuffle time. However the shuffle read time for the case of 30 partitions was small and thus custom partitioning would not have improved the execution time drastically.

3.2 Persistent RDDs

Persistence helps saving interim partial results so as to reuse them in subsequent stages. These interim results as RDDs are thus kept in memory (default) or disk.

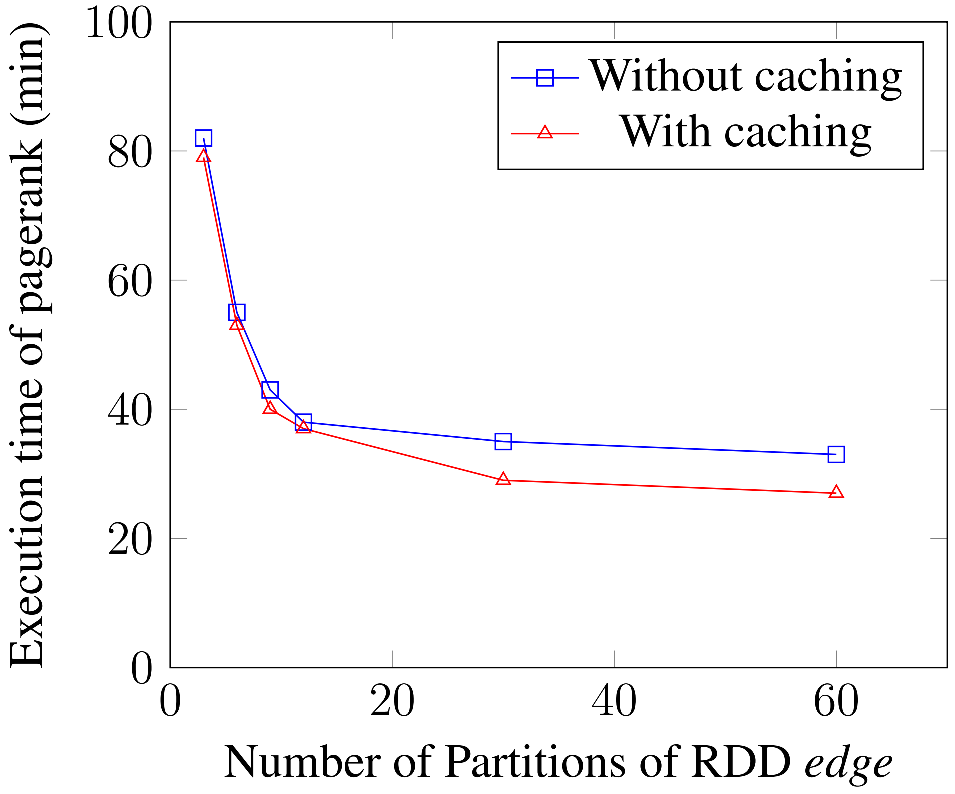
We performed persistence on 2 RDDs, namely, the pagerank and the edges RDD. The pagerank RDD is dynamic and keeps on updating, while the edges RDD [corresponding to the adjacency list between different URLs] stays static. As a result, persisting pagerank RDD did not decrease execution time for computing the pagerank when compared to without using persist. On the other hand, using persist with the edges RDD lead to a decrease in execution time in computing the pagerank when compared to without using persist. The results of this experiment are presented in the table below:
We observe a minor improvement in the execution time as now the RDD values are cached in memory and hence network overhead in transferring data across workers is marginally reduced.

Partitions	Tasks	Time(min)[After persisting]	Time(min)[Before persisting]	Stages
3	745	79	82	24
6	1135	53	55	24
9	1525	40	43	24
12	1915	37	38	24
30	4255	29	35	24
60	8155	27	33	24

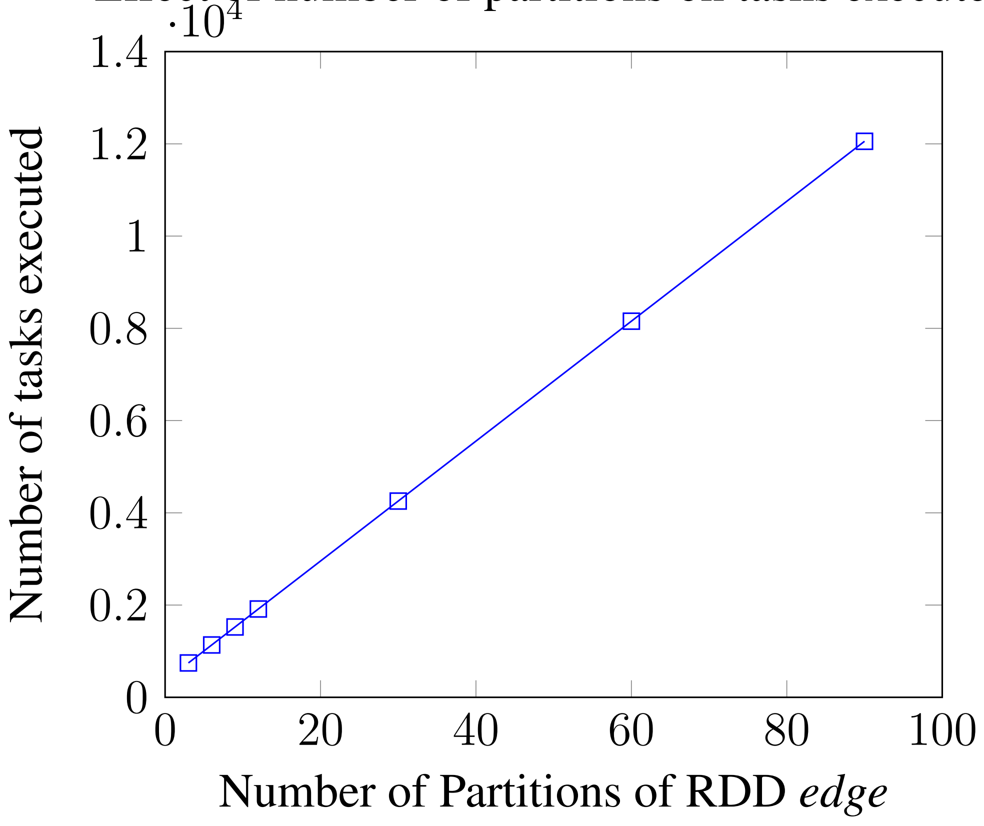
Table 3: Observing effect of persisting RDD in memory on the execution time of the pagerank algorithm and number of tasks executed by Spark workers

We present the above tabular information as a graph for easier inference. From the graph we can observe that the difference in execution times for larger number of partitions is more than for smaller number of partitions. Larger number of partitions indicates more number of chunks which can be processed in parallel by the workers.

Effect of number of partitions on execution time



Effect of number of partitions on tasks executed



We also present a plot of how the number of tasks executed changes with increasing the number of partitions. We observe an almost linear increase in the number of tasks to be executed with increasing the number of partitions as now the RDDs will be divided into many more chunks/blocks and all the operations now need to be done on each of these and this explains the linear increase in tasks executed

3.3 Effect of killing a Worker process

We set the number of partitions to 30 and 60 for this part and present results for only these two values. We observed that the an initial tasks took a longer time to execute than a single task towards the end of the execution. Hence we categorized 50% completion of task in two ways:

- Half of the total number of tasks have been executed [1]
- Execution is carried on for half the time that it takes for the complete execution when all workers are alive [2]

We summarize the results for both the above categorizations of 50% completion in the table below:

Partitions	Estimated Tasks	Executed Tasks	Time(min)[Worker 1 killed]	Time[Finish execution]
30	4255	5037	21	41
30	4255	4315	15	33
60	8155	9584	20	38
60	8155	8235	14	31

Table 4: Observing effect of persisting RDD in memory on the execution time of the pagerank algorithm and number of tasks executed by Spark workers

From the table we observed that on killing one of the worker processes at 50% completion of the task, some tasks start failing as these would have been the ones that the master would have initially allotted to the worker which was killed. Now the master has to re-execute these tasks as well as distribute the remaining tasks among the two other alive workers. This leads to increase in time of execution for completion as is visible from the table.

We also reasoned that if a worker fails towards the later portion in the algorithm, a lot more computation needs to be redone whose results were stored in that worker. On the contrary, if a worker fails earlier, the lineage graph for it is shorter and hence lesser computation needs to be re-performed though now we have lesser computing resources for a longer time. Based on the results, we can conclude that for the pagerank algorithm, late failing has a worse impact on execution time than early failing [since among the two cases for 30 and 60 partitions respectively, the time to complete execution is less for the cases when the worker was killed at less than half the execution time as compared to killing the worker at half execution time.]