Assignment 1
Group 11
CS744-Fall-2020

Name: Shing Ng, Jiun-Ting Chen, Matt Blakely

1. The python scripts for task 2 sorting are located in the task_2_sort folder.
   The scripts for part 3 and broken out into one folder per task named like part3_task* each folder contains a README, run.sh and the python script.

   Spark setup:
   We followed the assignment write-up and also edited the "spark-defaults.conf" in spark/conf/ folder with the following values:

spark.master	spark://node0:7077
spark.driver.memory	30g
spark.executor.memory	30g
spark.executor.cores	5
spark.task.cpus	1
spark.eventLog.enabled	true
spark.eventLog.dir	hdfs://10.10.1.1:9000/spark-events
spark.history.fs.logDirectory	hdfs://10.10.1.1:9000/spark-events
spark.local.dir	/mnt/data/temp

   We have also edited “spark-env.sh” in spark/conf/ folder with:

   SPARK_MASTER_HOST=10.10.1.1
   SPARK_LOCAL_DIRS=/mnt/data/temp

   Spark Health page: http://10.10.1.1:8080
   Spark History page: http://10.10.1.1:18080

1. **Introduction**
   Our goal for this project was to learn how to setup and run a spark application, and to write a complex enough application that we can gain a deep understanding of the Spark framework and

abstractions. Given Spark's complexity and amount of functionality we expected that it would be difficult to set up, however we were surprised that it was fairly straightforward to set up and to run a "hello world" script. The difficulties came in tweaking the parameters so that spark could function optimally and handle complex applications. Through implementing pagerank we gained hands-on experience using the Spark apis in python, the map-reduce coding paradigm, and investigating the RDD abstraction.

2. **PageRank algorithm**

The pagerank algorithm we used can be broken down into four phases:

- Import and clean the data
- Prepare the two main key value pair RDDs, rank and graph
- Iterate over our join-map-reduce transformations: we join the rank to the graph, flatMap the contributions from each outbound node, for each inbound node, and reduce the intermediate key value pairs to a single value and update the new ranks with the .85(contributions) + .15 formula
- Output the data to a file

The algorithm is only about 20 lines of code, however each line is dense and involves many transformations.

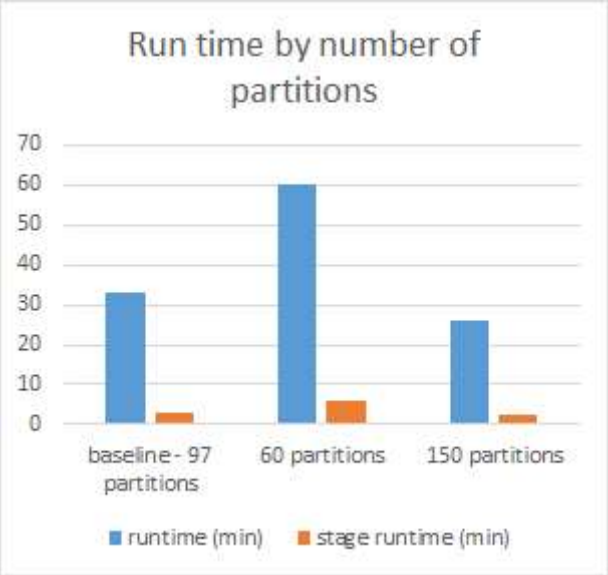
3. **Partition Analysis:**

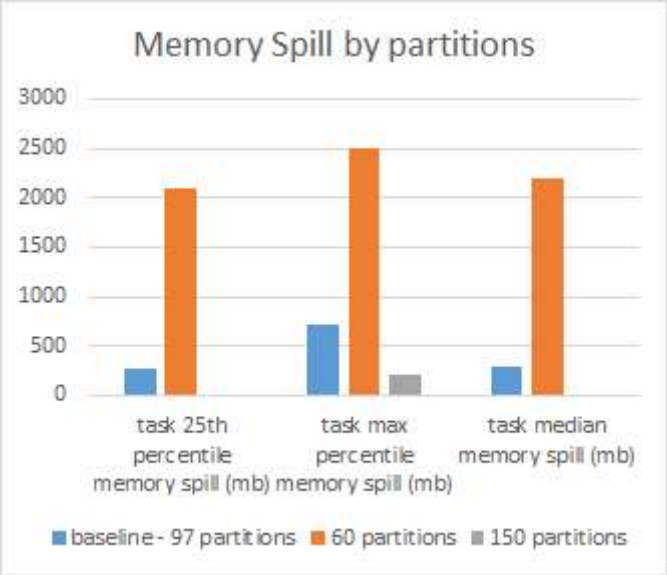
We found that partitions have two large effects on how the spark application processes the data: number/size of tasks, and, distribution of data.

Number of partitions

The number of partitions corresponds to the number of tasks which the application processes. For the default run, spark assigned 97 tasks per stage, for the other two runs we specified the number of partitions in various transformations to ensure that the application always used the specified number of partitions. We chose 60 (15 * 4) because the spark documentation recommended roughly 4 tasks per core, and we chose 150 (15 * 10) because it was a multiple of 15 substantially larger than the recommended amount.

The number of partitions corresponds to the size of each task, in other words, how much work each task needs to complete. We expect to see the runtime and spill time decrease as the number of partitions increase and the data showed that to be true. Future work could determine at what point the number of partitions causes a degradation in performance.

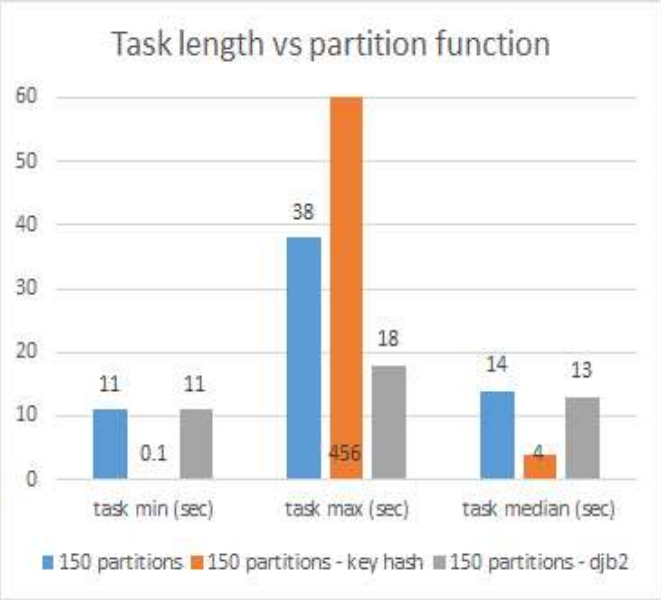


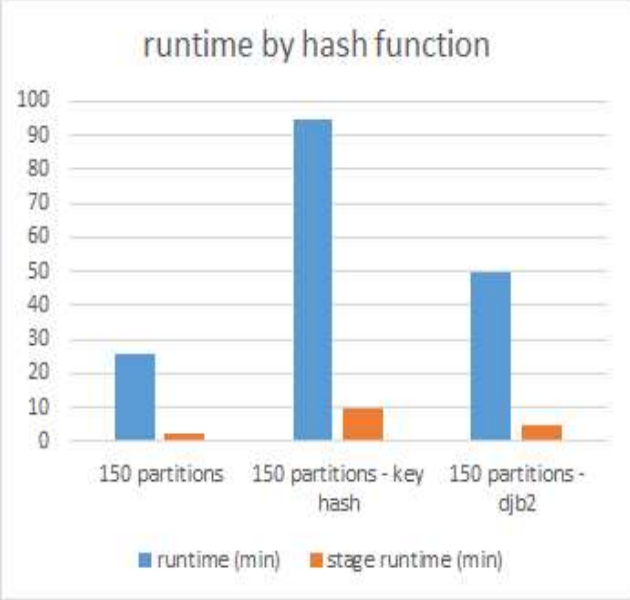


Data distribution

The partitioner function controls the size of each partition. Ideally, the data should be evenly distributed between each partition. However, creating partitions with an even distribution of elements can only be achieved if the partitioner function is built for that specific dataset. The default spark hash function is too general to work particularly well with the wiki dataset. As a result, there was a large variation in the task length of the baseline tasks.

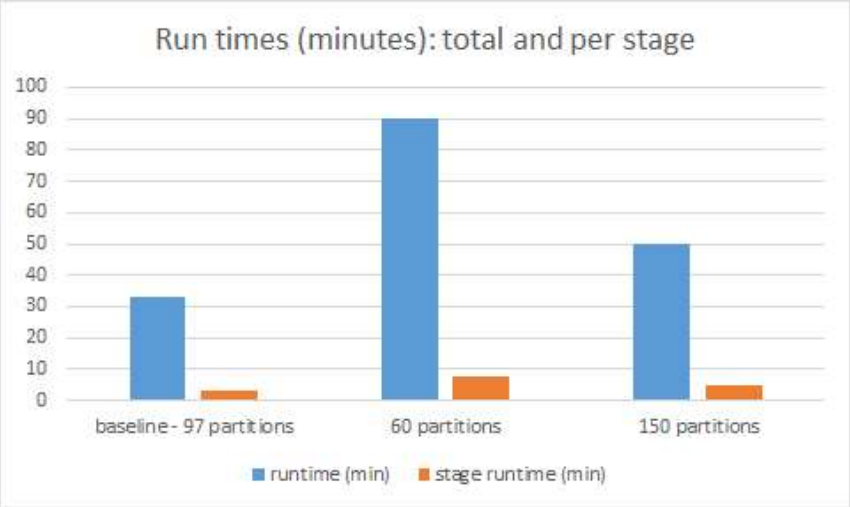
This meant that each job did not process an equal amount of data; the ones which processed the most data were stragglers. Stragglers cause a decrease in overall performance (run time increases for that task and for the overall stage) and decrease in utilization (some cores are processing nothing waiting for the stragglers to finish). To compare with the default hash function we tried to create a hash function based on the unicode number of the first character of each link, and using a hash function designed to distribute words evenly called djb2.
Note in the task length graph, the key hash (orange line) is 10 times larger than the other two. Also note that the runtime total for the hash functions is an estimate based off of the complete stages. Each stage represented one iteration of join-map-reduce cycle, since the hash functions always errored out, we took the average stage run time then multiplied it by 10 to be able to compare with the base run. More on the hash functions performance at the end of this section.



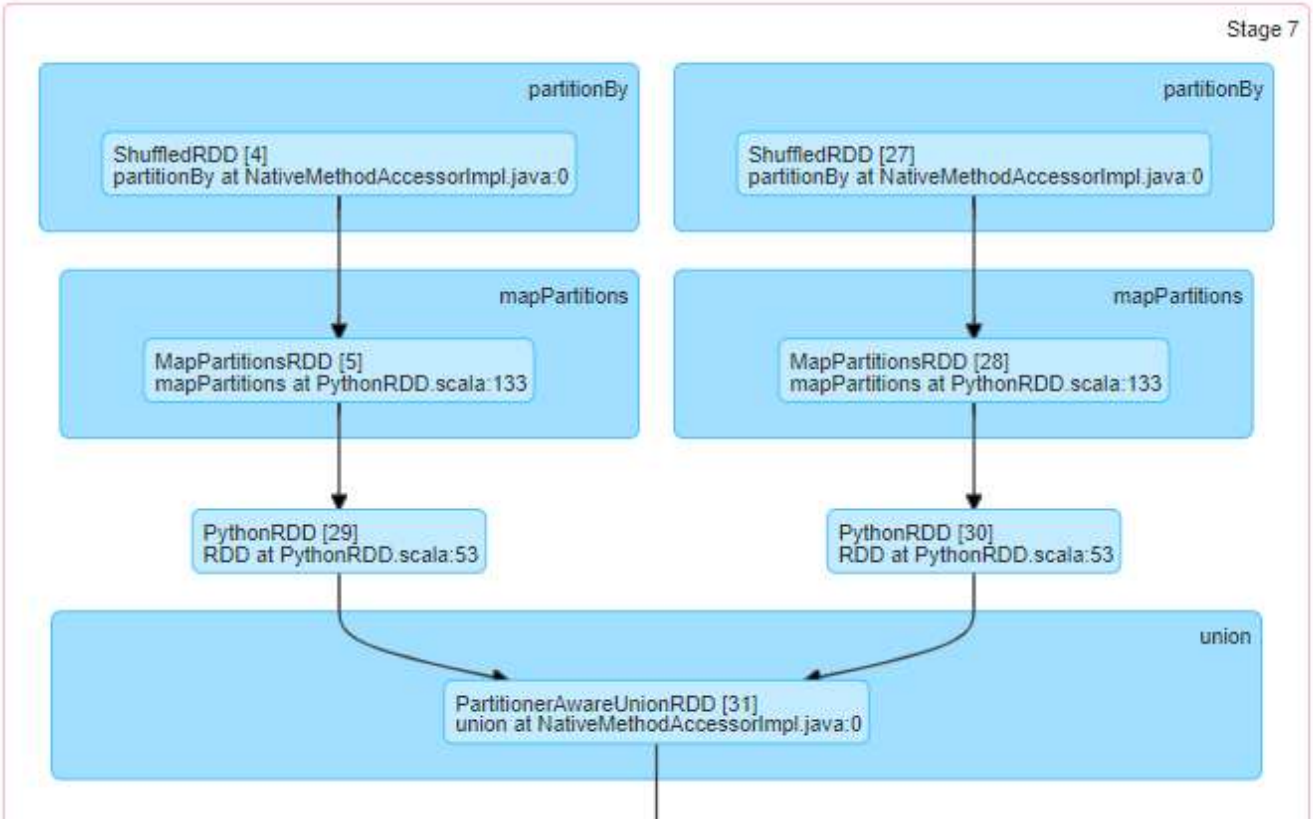


The hash function based on the first character of the link performed worse than the spark's and upon further investigation, it did a very poor job of distributing the links. Links starting with 's' account for nearly 10% of the links whereas links starting with 'u' were only about .004%. This led to an uneven distribution and a poor runtime.

We then found a hash function called djb2 which was designed to evenly distribute words. The hash function based on djb2 did an admirable job of distributing the links: each partition contained about 1.6% of links (when using 60 partitions). This led to a faster run time and balanced task execution time in each stage.
Strangely, the use of a custom hash function increased the runtime. Our hypothesis is that when it is evaluated, spark must leave its native Scala to call into python. Since it has to leave the JVM, the runtime is increased. Future work could investigate if the use of a custom hash function in Scala also increases the runtime of the application.



Finally, partitions affect how the data is stored within an RDD. If two RDDs are partitioned using the same partitioner function then some operations are able to process each partition completely in parallel leading to decreased shuffling and thus decreased runtimes. This is shown in the DAG as a “partition aware” operation:



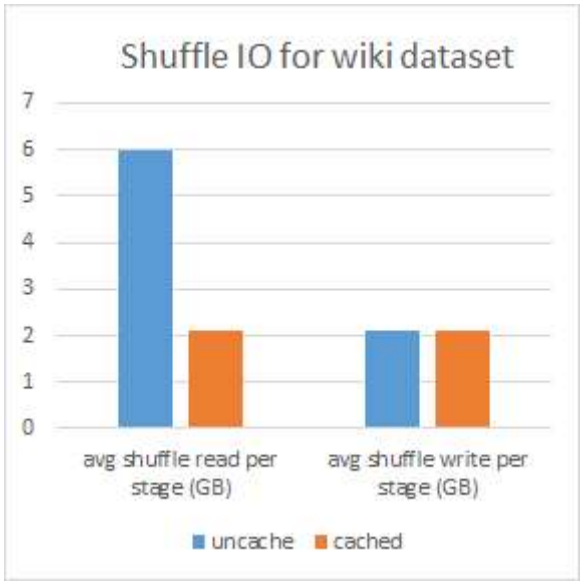
In this graph we see that it was able to process each RDD independently, then efficiently join them together using their shared partitioning function. The longer that you are able to keep the RDDs partitioned using the same partitioner, the more efficient the spark application will be as

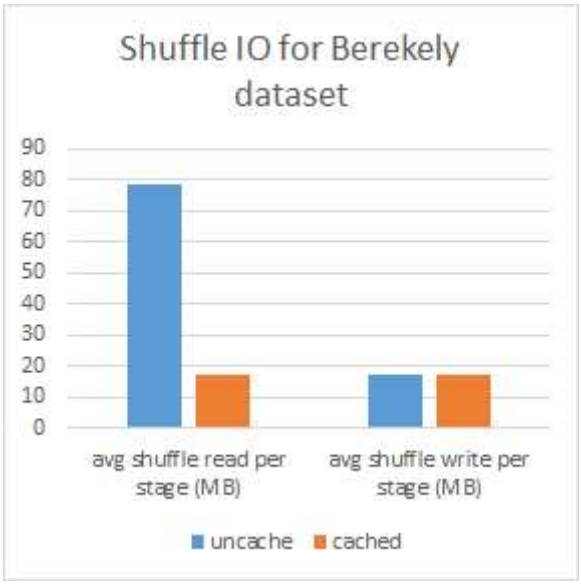
this will maximize the number of narrow dependencies. Transformations like: join, union, mapValues and flatMapValues are able to keep the partitioner applicable.Some transformations such as flatMap can't guarantee that they will output an RDD using the same partitioner function because the flatMap transformation can modify the key of the RDD. Our use of flatMap in pagerank is an example of this because it switched the key from the outbound node to the inbound node in preparation for the reduceByKey stage. ReduceByKey is an interesting example as by default it won't keep the partitioner but if you pass in a partitioner function, it will guarantee that the resulting RDD has that partitioner, presumably it does this by an embedded partitionBy call.

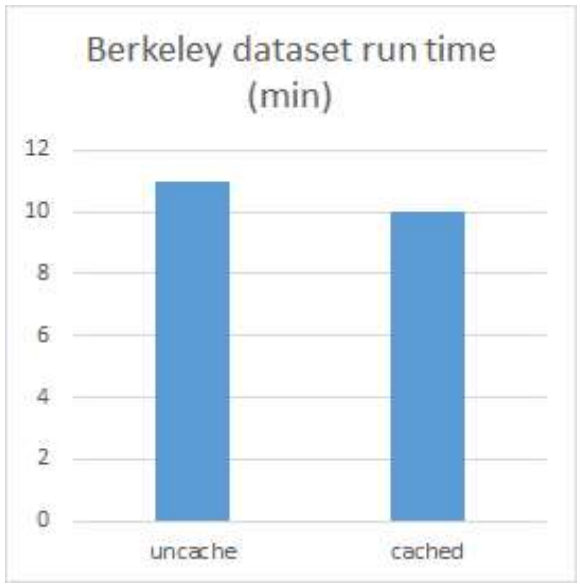
Throughout the testing we encountered many device out of space errors and we followed the class discussion and implemented the mitigation steps (directing spark to use a tmp directory on the mount drive, rebooting the instance, clearing out tmp folders, etc.). Eventually we discovered that the use of any custom python hash function greatly impacted performance and nearly guaranteed that the spark submission would hit the zero space error. Even when spark wrote two tmp files per run to the mnt drive, one tmp file was written to /tmp and node1 and node2 ran out of space even when tmp was empty prior to the run.

4. **Caching Analysis:**

For caching the RDD that would be reused multiple times in memory, the effect did not seem to make a significant difference for both sorting and pagerank. For a side note, the partitioning was kept default for the comparisons, and I called an action (count()) after caching to ensure the caching is performed due to lazy execution of the spark framework. However, I have observed that by caching, some stages were skipped as shown in the Spark history page. This might be because the caching helped the program avoid recreating the RDD that has been cached and it can be fetched from memory. It is also observed that the shuffle read has decreased significantly in the cached version compared to the uncached one which is possibly due to the program getting the data it needed from memory instead of walking the lineage map and recreating the RDD.







5. **Fault-Tolerance Analysis:**
   We had unresolved “no space” issues due to Spark creating excessively large temporary files in the /tmp folder (even after we changed the local directory variables). When we killed a node at 25% and 75% this increased the IO activity on the remaining nodes leading to more large temporary files. As a result our pagerank process did not run to completion after a node failure. However, the task runtime on any given node is about the same so we were able to extrapolate how long the process would have taken if it had completed. We believe that our results were solid enough to do some basic performance evaluation.

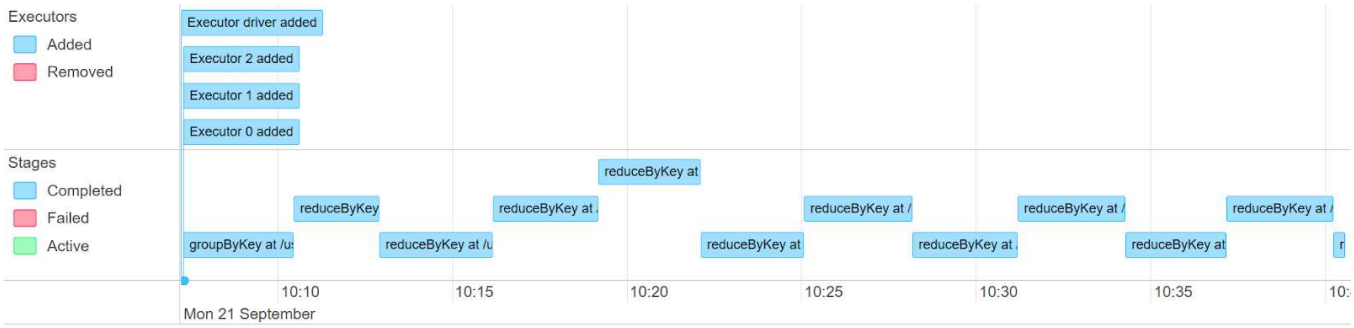
When some worker fails during execution, we found:

a. **Fault Resilient Strategy:** The tasks executed by the failed node and the failure task would be re-executed by existing nodes once spark noticed the failure. Figure 1 and 2 shows the timeline and the tasks list (From top to bottom are normal, failure at 25% and failure at 75%). Figure 3 shows the trend of I/O and shuffle read/write of each node through the process.

b. **Task Distribution**: The remaining tasks were fairly distributed to the existing nodes after the failure of a node. Figure 4 shows the alive executors have similar I/O and shuffle read/write.

c. **Impact on Network/Disk IO**: Figure 6 shows some network/disk information of nodes, which was collected by dstat. The network and disk activity of the existing nodes do not have an obvious change after the failure except a spike on disk read when the failure happened.

d. **Impact on Execution Time**: It is intuitive that a normal case would be completed faster than the failure cases. However, it is not obvious if a node failing at 25% or at 75% will finish faster. Although the existing nodes should re-execute more tasks if the failure happens later, it's also demanding for the existing nodes to cover more pending tasks if a node fails in the early phase. Figure 7 shows that the execution time of the process would be shorter when failure happened at 75% rather than 25% in our test.



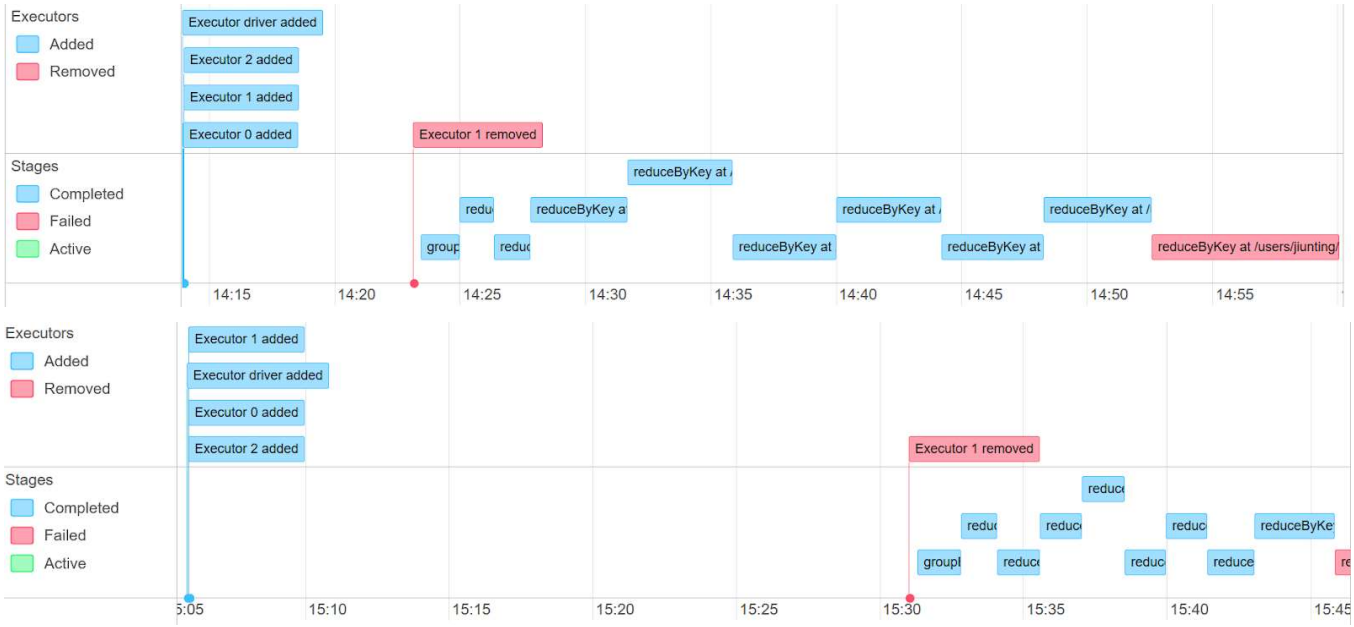


Figure 1. Timeline (Normal vs. stop at 25%, vs stop at 75%)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
8 (retry 1)	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:43:03	2.8 min	65/65			3.9 GB	1464.7 MB
7 (retry 1)	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:41:25	1.6 min	34/34			2.0 GB	763.0 MB
7	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:26:47	3.0 min	97/97			5.9 GB	2.1 GB
6 (retry 1)	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:39:58	1.4 min	30/30			1861.9 MB	681.4 MB
6	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:23:46	3.0 min	97/97			5.9 GB	2.1 GB
5 (retry 1)	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:38:32	1.4 min	30/30			1859.5 MB	678.8 MB
5	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:20:40	3.1 min	97/97			5.9 GB	2.1 GB
4 (retry 1)	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:37:03	1.5 min	31/31			1929.4 MB	705.8 MB
4	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:17:38	3.0 min	97/97			5.9 GB	2.1 GB
3 (retry 1)	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:35:35	1.5 min	32/32			2019.0 MB	726.3 MB
3	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:14:33	3.1 min	97/97			6.0 GB	2.2 GB
2 (retry 1)	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:34:05	1.5 min	30/30			2.0 GB	717.2 MB
2	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:11:16	3.3 min	97/97			6.5 GB	2.3 GB
1 (retry 1)	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:32:50	1.3 min	32/32			2.5 GB	958.7 MB
1	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:08:48	2.5 min	97/97			7.4 GB	2.8 GB
0 (retry 1)	groupByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:29 +details	2020/09/21 15:31:19	1.5 min	30/30	3.2 GB			1209.9 MB
0	groupByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:29 +details	2020/09/21 15:05:56	2.9 min	97/97	9.9 GB			3.7 GB

Skipped Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
11	runJob at SparkHadoopWriter.scala:78 +details	Unknown	Unknown	0/97				
10	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	Unknown	Unknown	0/97				

Failed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write	Failure Reason
9	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:45:50	34 s	7/97 (7 failed)			865.0 MB	159.0 MB	Job 0 cancelled as part of cancellation of all jobs
8	reduceByKey at /users/jiunting/spark-2.4.6-bin-hadoop2.7/pagerank.py:39 +details	2020/09/21 15:29:49	1.5 min	43/97 (24 failed)			2.6 GB	968.2 MB	org.apache.spark.shuffle.FetchFailedException: Failed to connect to /172.16.155.2:45760 +details

Figure 2. Task list of the process stopping at 75%

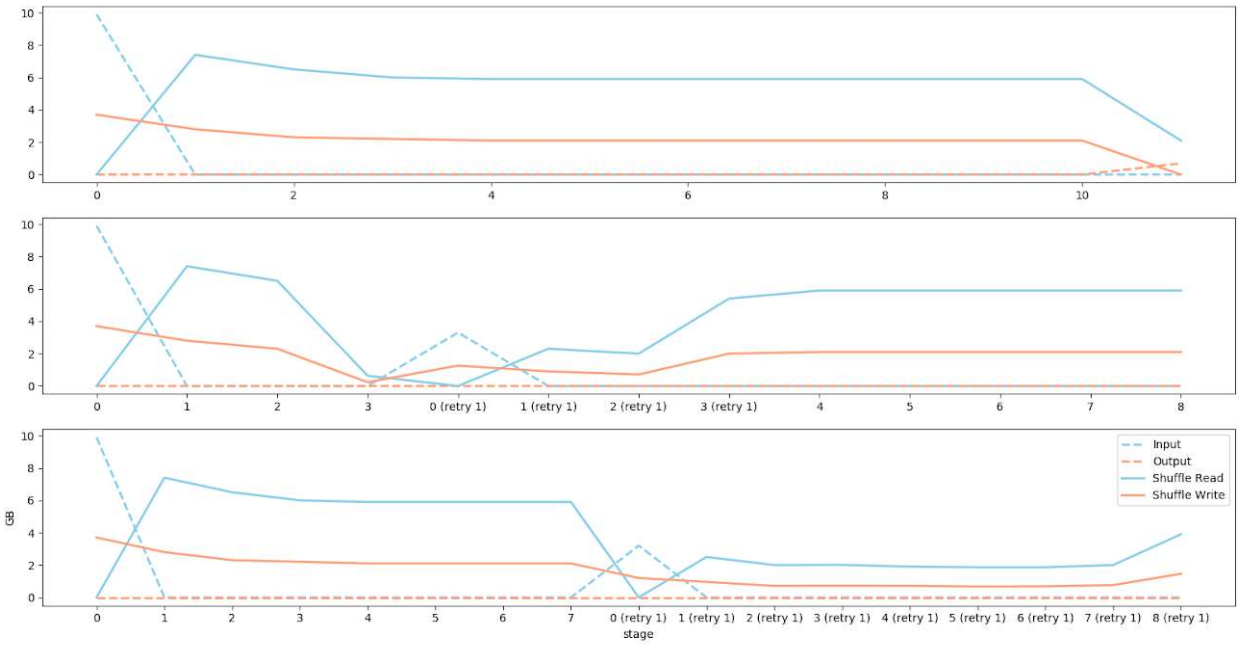


Figure 3. I/O and shuffle Read/Write (normal vs. stop at 25%, vs stop at 75%)

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs
0	172.16.155.3:38634	Active	0	0.0 B / 17 GB	0.0 B	5	0	0	413	413	2.6 h (1.3 min)	3.7 GB	24 GB	9.8 GB	stdout stderr
driver	c220g5-110917vm-1.wisc.cloudlab.us:44640	Active	0	0.0 B / 17 GB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	
1	172.16.155.2:45536	Active	0	0.0 B / 17 GB	0.0 B	5	0	0	373	373	2.5 h (59 s)	3.4 GB	21.7 GB	9 GB	stdout stderr
2	172.16.155.1:38955	Active	0	0.0 B / 17 GB	0.0 B	5	0	0	378	378	2.4 h (55 s)	3.5 GB	22.1 GB	9.1 GB	stdout stderr

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
0	172.16.155.3:33460	Active	0	0.0 B / 17 GB	0.0 B	5	5	16	447	468	3.0 h (1.5 min)	5.3 GB	28.1 GB	11.9 GB
driver	c220g5-110917vm-1.wisc.cloudlab.us:33021	Active	0	0.0 B / 17 GB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
1	172.16.155.2:36441	Dead	0	0.0 B / 17 GB	0.0 B	5	0	5	90	95	40 min (11 s)	3.5 GB	4.7 GB	3 GB
2	172.16.155.1:39817	Active	0	0.0 B / 17 GB	0.0 B	5	2	0	516	518	3.4 h (1.0 min)	5.3 GB	31.5 GB	13.5 GB

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
0	172.16.155.3:44186	Active	0	0.0 B / 17 GB	0.0 B	5	5	12	450	467	3.0 h (1.2 min)	5.3 GB	27.9 GB	11.9 GB
driver	c220g5-110917vm-1.wisc.cloudlab.us:36588	Active	0	0.0 B / 17 GB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B
1	172.16.155.2:45760	Dead	0	0.0 B / 17 GB	0.0 B	5	-7	12	260	265	1.9 h (35 s)	3.4 GB	15.8 GB	7 GB
2	172.16.155.1:38232	Active	0	0.0 B / 17 GB	0.0 B	5	1	3	430	434	2.9 h (59 s)	5.3 GB	26 GB	11.5 GB

Figure 4. Workload of Each Executors (Normal vs. stop at 25%, vs stop at 75%)

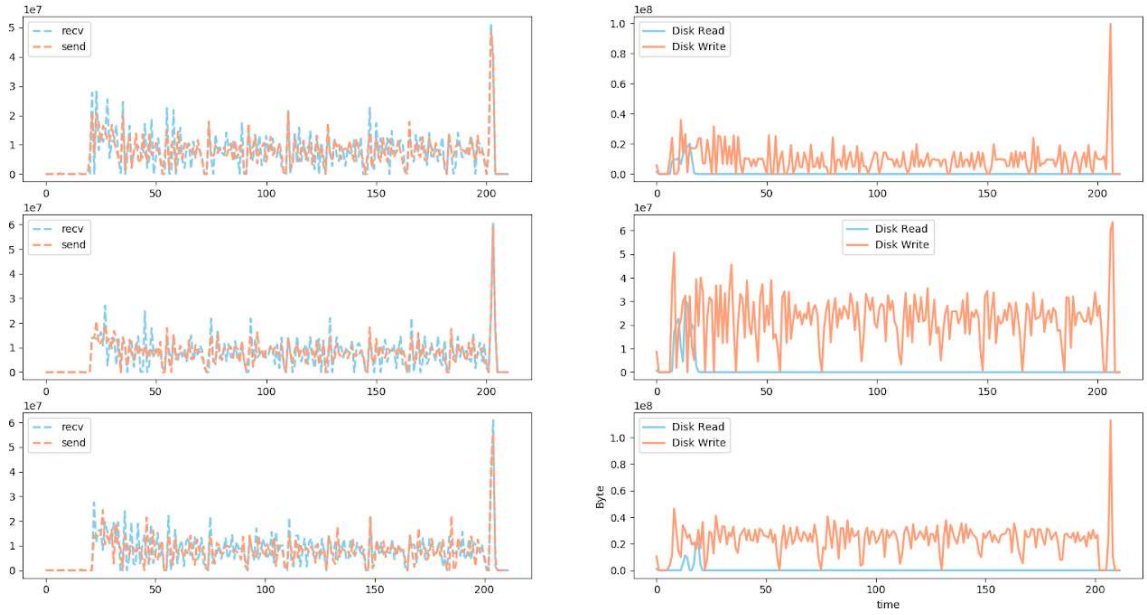


Figure 5. Network and Disk Activity (normal)

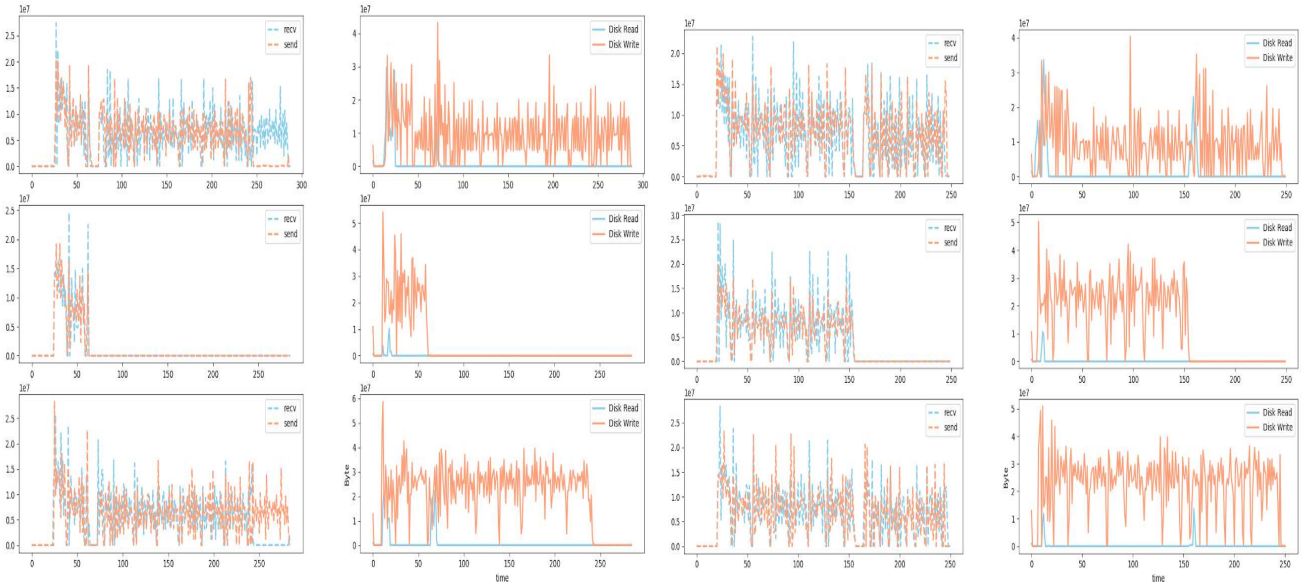


Figure 6. Network and Disk Activity (stop at 25% and stop at 75%)

	Execution time (min)
No failure	33
Node failure at 25%	62 (estimated)
Node failure at 75%	52 (estimated)

Figure 7. Execution time comparison between normal and failure cases

Contributions:

Jiun-Ting:

- Environment Setup and deployment
- Writing setup script
- Code/Log review, Solving major configuration issues,
- Task 4 in part 3;
- Final report

Shing Ng:

- spark environment setup
- part 2 sort algorithms, README, run.sh
- part 3 pagerank algorithm, README, run.sh
- task 3 in part 3
- final report

Matt Blakely:

- Spark environment setup and investigation
- Part 3 task 2 code and section in report
- Part 2 sort algorithm
- Part 3 pagerank algorithm
- Final report
- Standardizing report and submission formats