Anshu Verma | Arpit Jain | Arun Jose

Assignment - 1

Introduction

In this assignment, we deployed and configured Apache Spark (as the execution engine) and HDFS (as the underlying file system). Following this, we implemented two applications in Spark: one to sort some sample IoT data, and another to implement and run the PageRank algorithm on two datasets. We then ran a few experiments on this setup and discussed our observations and inferences below.

Experiment Setup

As part of the experiment setup, we used a 3 node cluster running on Ubuntu 16. We used HDFS (Hadoop 3.1.2) as our underlying filesystem and Spark 2.4.4 as the computational engine to perform data analytics. The configuration settings used in Spark in its *spark-defaults.conf* file has been listed below:

spark.driver.memory	29g
spark.executor.memory	29g
spark.executor.cores	5
spark.task.cpus	1
spark.submit.deployMode	client
spark.logConf	true

Since Spark stores some temporary shuffle data to disk, we changed the local directory of Spark to point to a mount location which has more space. This was done by setting the `SPARK_LOCAL_DIRS` parameter in the *spark-env.sh* file.

Implementation

- **A Simple Spark App**

Workload: http://pages.cs.wisc.edu/~shivaram/cs744-fa18/assets/export.csv (~110 KB)

The entire sample data is read from the input location and read into the system as an RDD. From this RDD, we filter out the headings and map the result into a new RDD in which each

row is an array of strings. This RDD is then sorted using the *sortBy()* operation first on the basis of country code and second on the basis of timestamp. Each row of this result is then mapped from an array of strings back to a string. This is the final result RDD which is written to the output location.

- **PageRank (*Berkeley-Stanford Web graph*)**

Workload: https://snap.stanford.edu/data/web-BerkStan.html (~18.2 MB)

We first read the input as an RDD and filter out the comments to get just the data. We then create two RDDs: *links* which have *(srcUrl, destUrl)* as its rows and *ranks* which stores the *(srcUrl, rank)*. The rank is initially set to 1 for all the URLs. We calculate the contribution of a webpage *p* by *rank(p)/num(neighbors of p)* and then update the *rank* to be *0.15 + 0.85*(sum of contributions)* in each iteration. The final resultant RDD is then saved to the output location as a text file. To try and get some interesting results, we tried custom partitioning to utilize data locality optimizations of Spark so that the *links* and *ranks* RDDs are stored on the same node making their *join* faster. Further, we tried persisting the *links* RDD in memory to see the performance advantages in this scenario. We didn't persist the *ranks* or *contributions* RDDs since a new RDD is produced for them at each iteration. Finally, we tested the fault tolerance of the system by killing one of the worker processes during execution.

- **PageRank (*Wiki Articles Dataset*)**

Workload: (CloudLab) /proj/uwmadison744-f19-PG0/data-part3/enwiki-pages-articles/ (~9.9 GB)

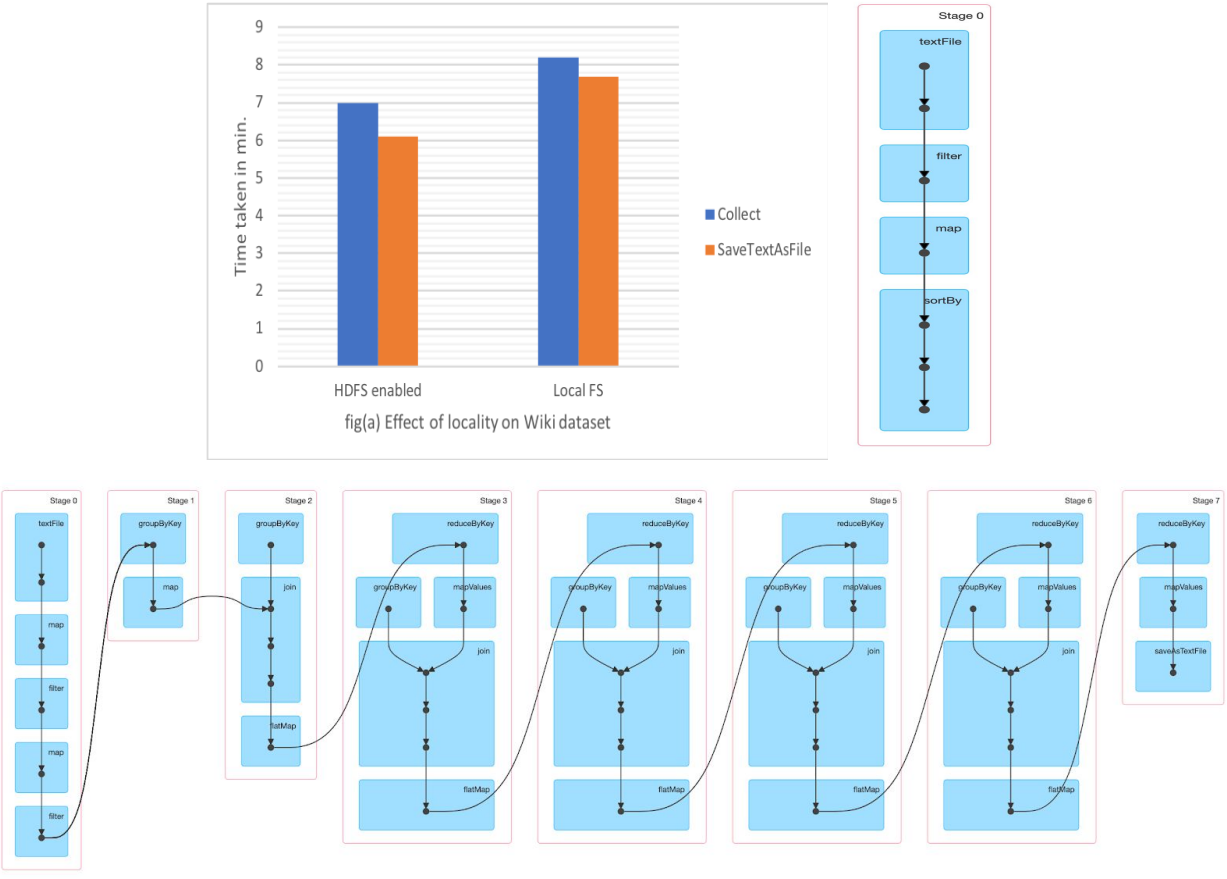
The algorithm is roughly the same as above with some extra pre-processing to filter the relevant data from the dataset. Again, similar experiments were run and the observations were noted.

Observations and Inferences

Task - 1

In this task, we have written the PageRank algorithm on the Berkeley-Stanford dataset and Wiki dataset. In the case of the Berkeley-Stanford dataset, there were a total of 26 tasks, whereas, in the case of the Wiki dataset, the task count was 776.

We experimented with input data location and action type in this task and studied their effect on total execution time. We noticed that *collect* action is taking more time than *saveTextAsFile* because *collect* action compiles the data and return it to the driver which incurs network cost. On the other hand, when we are doing *saveTextAsFile* the writes are distributed across multiple worker nodes. This parallelization of writes offsets the disk IO incurred in this case.



Task - 1: Lineage graphs of Berkley-Stanford dataset and wiki dataset

Task - 2

RDD Partitioning with Cache

In the case of the Berkeley-Stanford Web Graph, the amount of data is very less and probably resides on a single node of HDFS. Thus, while running PageRank, only one worker was being used. As a result, partitioning will not make any difference in this case.

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(4)	0	84.3 KB / 65.7 GB	0.0 B	15	1	0	3	4	5 s (0.4 s)	186 KB	0.0 B	38.1 KB	0
Total(4)	0	84.3 KB / 65.7 GB	0.0 B	15	1	0	3	4	5 s (0.4 s)	186 KB	0.0 B	38.1 KB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0

Executors

Show 20 entries Search:

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	Thread Dump
0	172.16.62.3:34163	Active	0	28.9 KB / 16.4 GB	0.0 B	5	0	0	3	3	5 s (0.4 s)	186 KB	0.0 B	38.1 KB	stdout stderr	Thread Dump
driver	c220g1-030612vm-1.wisc.cloudlab.us:34559	Active	0	28.9 KB / 16.4 GB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump
1	172.16.62.1:35696	Active	0	26.4 KB / 16.4 GB	0.0 B	5	1	0	0	1	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
2	172.16.62.2:44297	Active	0	0.0 B / 16.4 GB	0.0 B	5	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump

Task - 2: Only one worker for BerkStan dataset

In the case of the Wiki Articles Dataset, we added a custom partitioner and used it on links and ranks (generated after each iteration) RDD. We noticed that intermediate data generated is significantly high and results in higher GC execution estimates (red color). Please note that such intermediate states were seen in other tasks as well.

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(4)	0	221.7 KB / 65.7 GB	0.0 B	15	6	0	488	494	1.5 h (29 min)	10.6 GB	26.2 GB	14.6 GB	0
Total(4)	0	221.7 KB / 65.7 GB	0.0 B	15	6	0	488	494	1.5 h (29 min)	10.6 GB	26.2 GB	14.6 GB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0

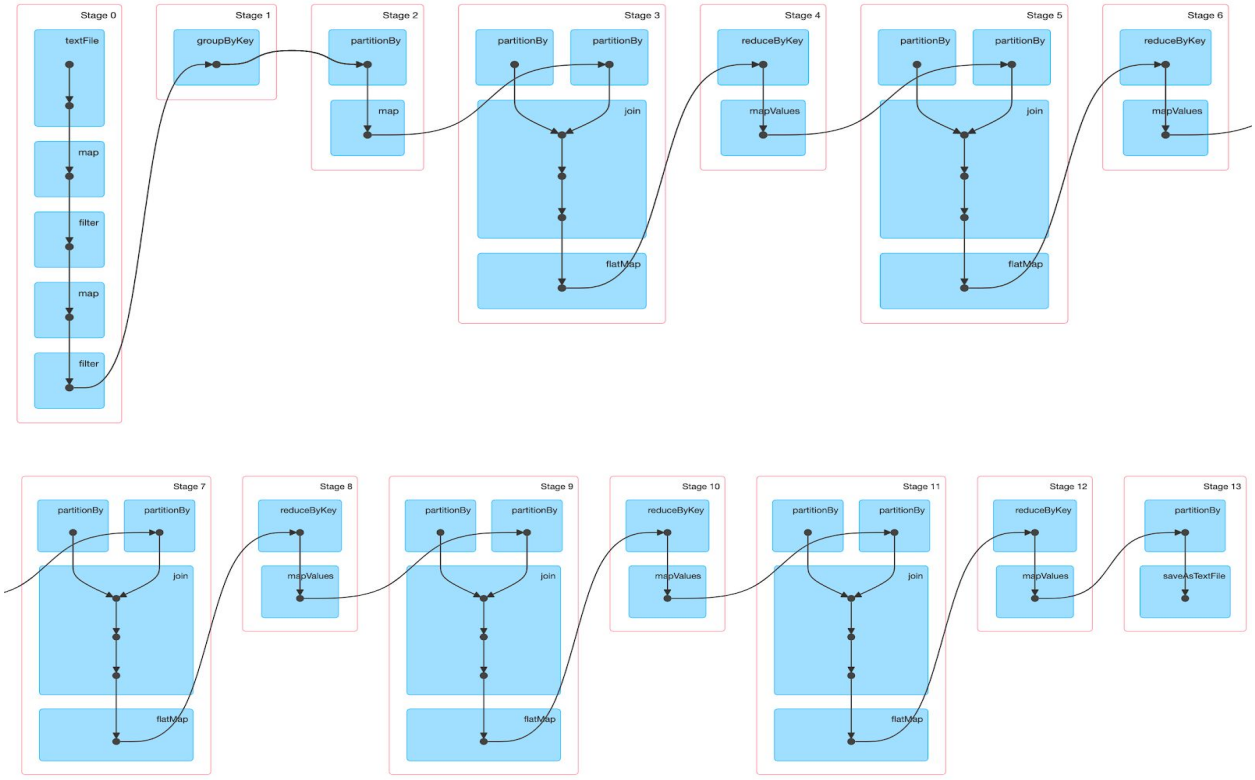
Executors

Show 20 entries Search:

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	Thread Dump
0	172.16.62.3:34509	Active	0	55.4 KB / 16.4 GB	0.0 B	5	1	0	166	167	30 min (9.4 min)	3.7 GB	8.9 GB	5 GB	stdout stderr	Thread Dump
driver	c220g1-030612vm-1.wisc.cloudlab.us:39187	Active	0	55.4 KB / 16.4 GB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump
1	172.16.62.1:36552	Active	0	55.4 KB / 16.4 GB	0.0 B	5	0	0	166	166	30 min (9.5 min)	3.5 GB	8.9 GB	5 GB	stdout stderr	Thread Dump
2	172.16.62.2:45715	Active	0	55.4 KB / 16.4 GB	0.0 B	5	5	0	156	161	31 min (10 min)	3.4 GB	8.4 GB	4.6 GB	stdout stderr	Thread Dump

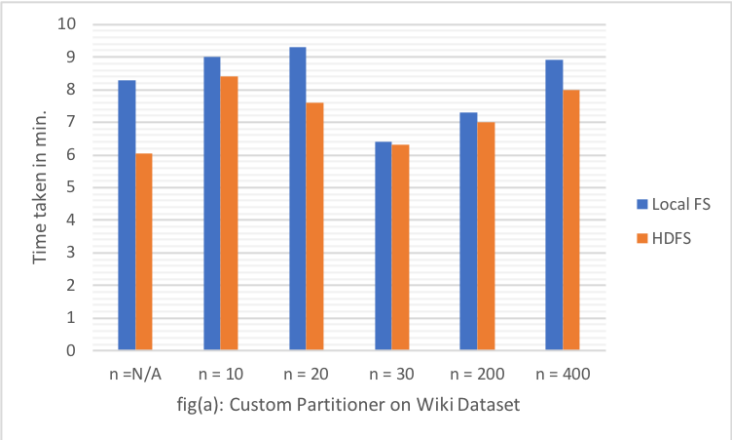
Task - 2: Intermediate data generation rate is very high, resulting in high GC time estimates.

We also noticed that partitioning resulted in the addition of more wide dependencies in the lineage graph. As a result, the number of stages in the DAG increased from 7 (without partitioning) to 13 (with partitioning).



Task - 2: DAG of Wiki data with 30 partitions per RDD

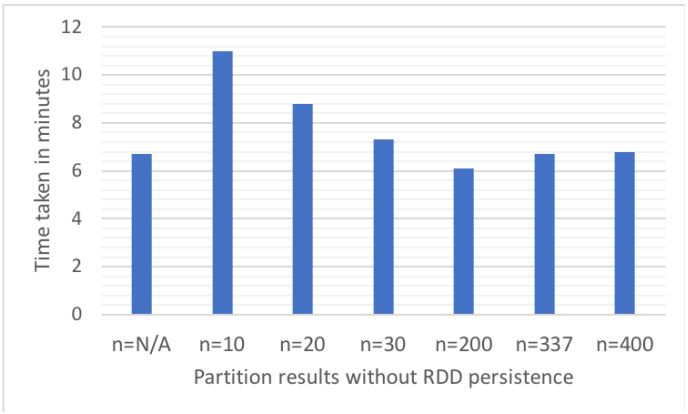
Performance can be enhanced by using correct partitioning. We were able to find the optimal partition count while reading from the local FS (30), but not in the case of HDFS.



Task - 2: Time taken with different number of partitions using local FS and HDFS

RDD Partitioning without Cache

We have also taken another set of results with RDD persistence disabled. All the reads were made on the HDFS file system and no writes on the file system. We used collect() action which sends the final RDD to the driver hence, incurring the network cost.



Task 2: Time taken with different number of partitions using HDFS and cache disabled

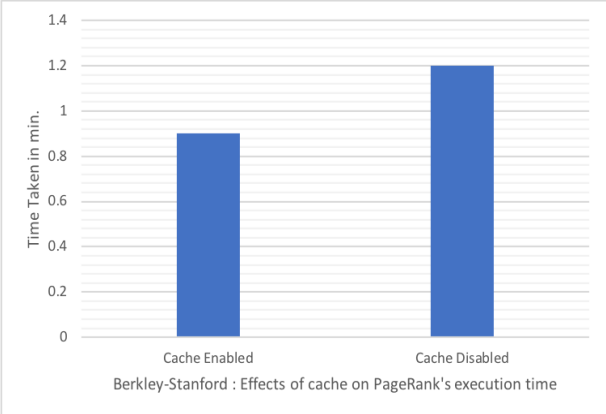
We noticed that after disabling the RDD cache the optimal partition count was somewhere around 200. Performance degrades if we increase or decrease the partition.

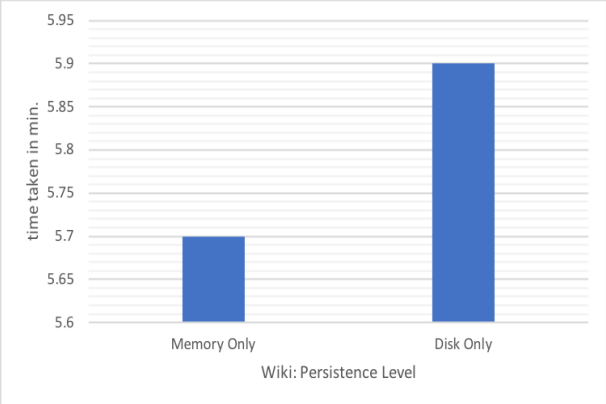
Increasing the partition count allows for a greater degree of parallelism and optimal utilization of executors. Although, higher partition count might become a problem when reshuffling happens as it might result in data transfer among JVMs causing network overhead.

An ideal partition count would be 2-3x of the total no. of cores available. Also, the lower bound of partition count is limited by the amount of memory available to the executors.

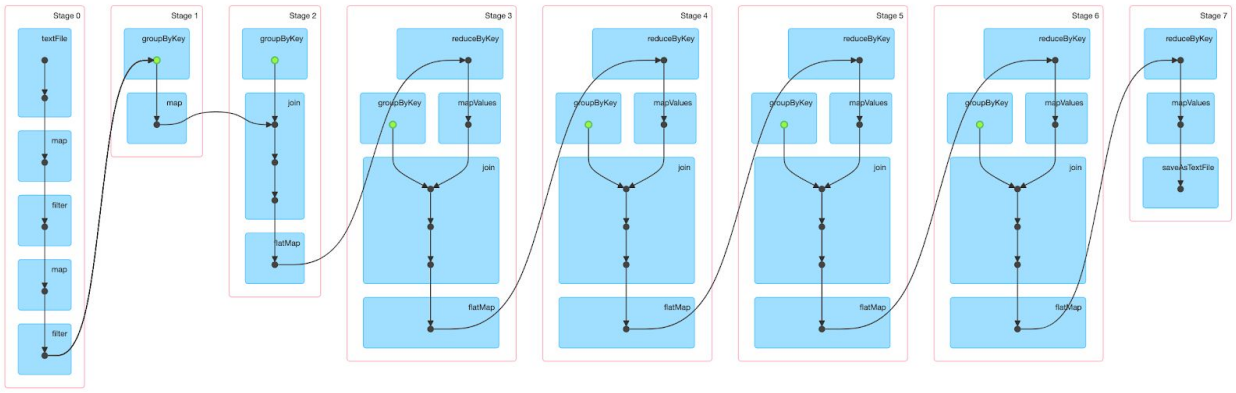
We also notice a proportional increase for all the partitions when compared to the corresponding graph with the caches. This shows that irrespective of partition count, caching always helps to optimize the job in Spark.

Task - 3



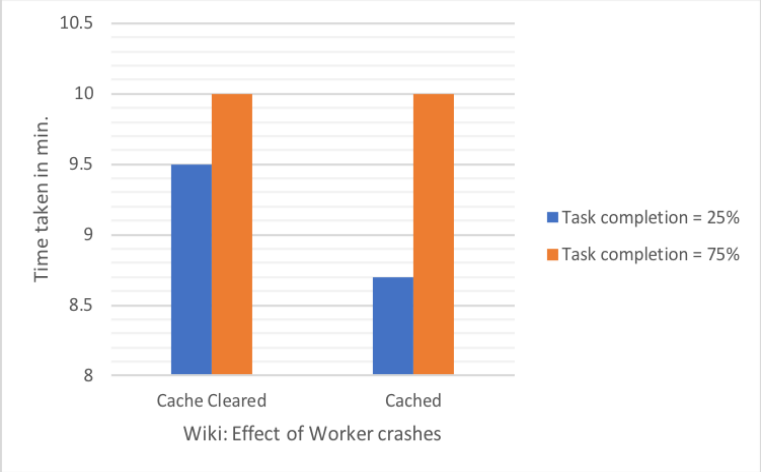


In this task, we persist the *links* RDD in memory and see how the performance compared with reading it from disk each time. As expected, the total runtime reduced in both datasets. We noticed a 25% improvement in the Berkeley-Stanford dataset. In the Wiki dataset, we tried three different persist approaches: in-memory, in-memory(serialized) and disk persistence. We observed a 14% improvement for the in-memory and disk persistence and around 18% for serialized in-memory persistence tests. Ideally, the in-memory performance should have been better than the other two, but since the results are comparable it could be due to some minor internal Spark differences in functionality.



Task - 3: DAG of Wiki data with RDD persistence

Task - 4



Finally, we wanted to test the fault tolerance of the system by killing a worker process mid-execution. We killed the process at two stages in the execution cycle (25% and 75%), both without and after clearing the cache in the to-be killed worker machine. As expected, the total time to complete execution was longer than the previous tasks in all these mentioned cases. Furthermore, an interesting observation was that killing the process at 75% had a bigger impact than killing it at the 25% mark. Although this seems non-intuitive (as fewer executors need to run more tasks), it can be explained by the fact that Spark reconstructs the lost RDDs from its lineage graph. As a result, it was faster in bringing up the RDDs at 25% as it requires lesser transformations as opposed to bringing them up at the 75% mark. This implies that the cost of reconstructing the RDDs was relatively more

expensive as compared to running the extra computations from the dead worker process in this case. Further, not clearing the cache ran slightly faster than clearing the cache.

Conclusion

After running these experiments, we have made the following inferences about RDD and Spark:

1) In order to optimize the tasks, Spark tries to place the execution code as close to the processed data. We noticed that all the tasks running in the executor were NODE_LOCAL.
2) Increasing the partition count does not always decrease the job's execution time. We have noticed that the optimal count is around (*core count*) * (*executor count*).
3) In general, caching improves the job's execution time. The RDD which is repeatedly being used in the code should be marked persisted. There are multiple options available for persist() method and depending on the data, one can choose to serialize it or write it on memory or disk.
4) RDD collect() method should be used wisely as it will trigger the flow of data from workers to the driver over the network. Whereas, in case of saving the RDD as a text file, the file system can keep the control and data flows separate and attempt some level of data locality (Note that file system's fault tolerance network consumption will still be there).