CS-744: Big Data Systems

Assignment 1

Adarsh Kumar, Arjun Balasubramanian, Sonu Agarwal

October 1, 2018

Sorting using Apache Spark

Introduction

The first part of the assignment is to sort an IoT dataset present on HDFS in the form of a CSV file. First, we are required to sort the data firstly by the country code alphabetically and then by timestamp to break ties in country code.

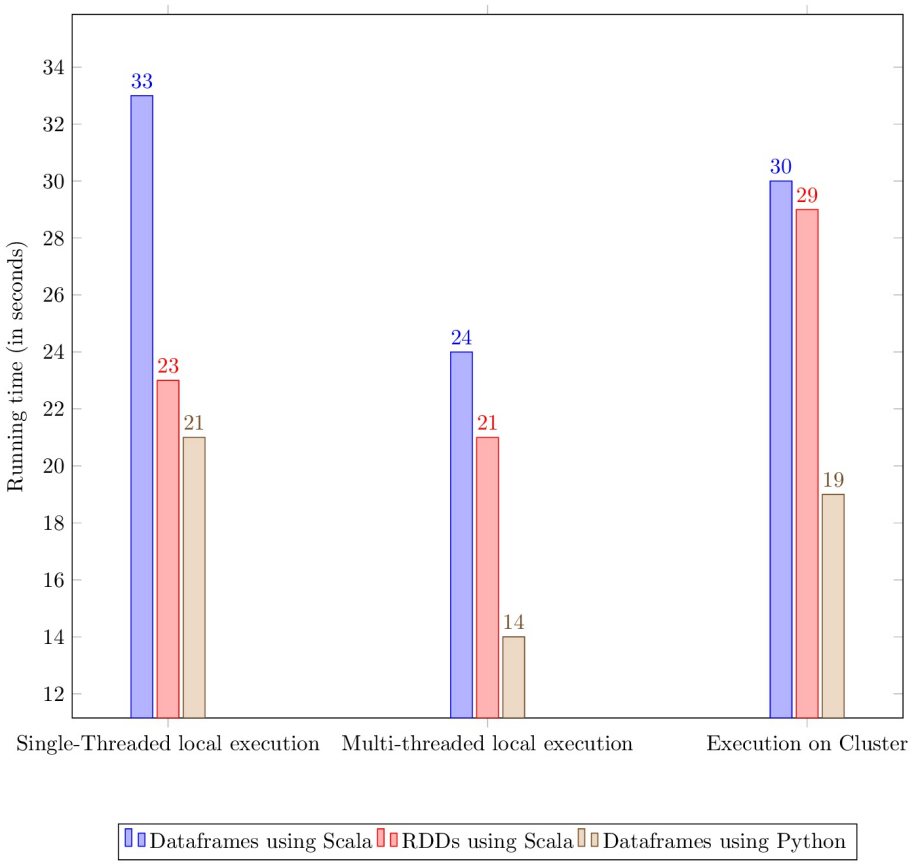
Different Implementations and their comparison

At a high level, we saw two different approaches to solve this problem -

1. **Using RDDs –** In this approach we first convert the dataset in the CSV file into an RDD by loading the CSV file and then splitting each line on the CSV into an array of strings. Let us call each array of strings as "array_strings". We then index *array_strings* such that we get a new RDD in the format <*array_strings[2], array_strings[14],array_strings*>. Essentially, this makes each *array_strings* indexed by country code and then by the timestamp. When a sortByKey() is done on this RDD, it automatically guarantees that the rows would be sorted first by country code and then by timestamp to break ties. Post this, the data is merged back to have the fields separated by commas and then written back to HDFS.
2. **Using Dataframes** – Dataframes essentially have the same properties as RDDs (lazy evaluation, immutability), but offer a nice and clean abstraction for manipulating structured data. Hence, they almost serve as a natural choice for operating on CSV files. Dataframes offer simple APIs for reading CSV files into memory and for sorting rows based on the values from multiple columns.

Through an experiment, we wish to find out which of the above approaches offers better performance. Moreover, we also wish to benchmark the performance of Spark in different programming languages like Python and Scala. We also wish to benchmark the performance of

our code on a local machine vs. Execution on a cluster. Below is a graph that shows our observations -



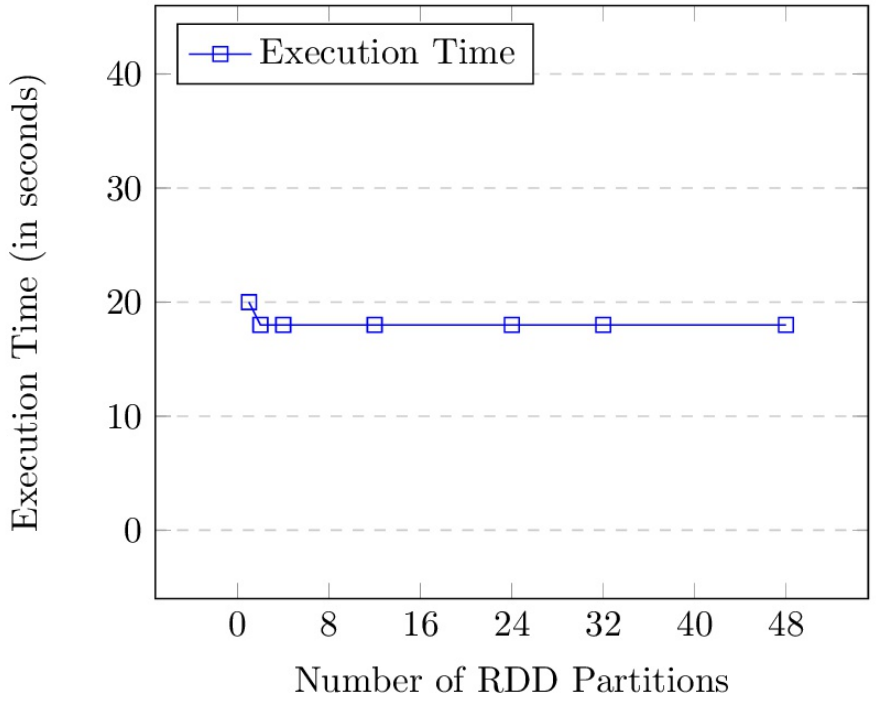
Below are some of the key takeaways from the above graph -

1. **Single-Threaded vs. Multi-Threaded execution:** Single-Threaded execution on a local machine is slower than Multi-Threaded execution on a local machine across all the different implementation. This is as per expectation and clearly delineates the superior performance of a 32-core processor.
2. **Local execution vs. Execution on Cluster:** Interestingly, the performance on a single machine using multiple threads is better than that of the cluster. Though counter-intuitive at first, this result is expected given that the input CSV had only 1100 records. When input size is low, network overheads in a cluster can dominate the runtime.
3. **RDDs vs. Dataframes:** We clearly notice that the execution using RDDs outperforms the execution using Dataframes. However, we also ran some re-executions using larger datasets and observed fluctuating results. An empirical analysis isn't going to settle the scores on this one!
4. **Python vs. Scala:** From the graph, we see that Python outperforms Scala. We largely believe that this is due to the high initial overhead of starting up the JVM in Scala. Interesting, we also noticed that Scala has better execution times if we discount the

initial overhead owing to the fact that it is faster to execute compiled code. Our take on this is to use Python for smaller programs and Scala for programs that are expected to run for a longer time.

Effect of manipulating number of RDD Partitions

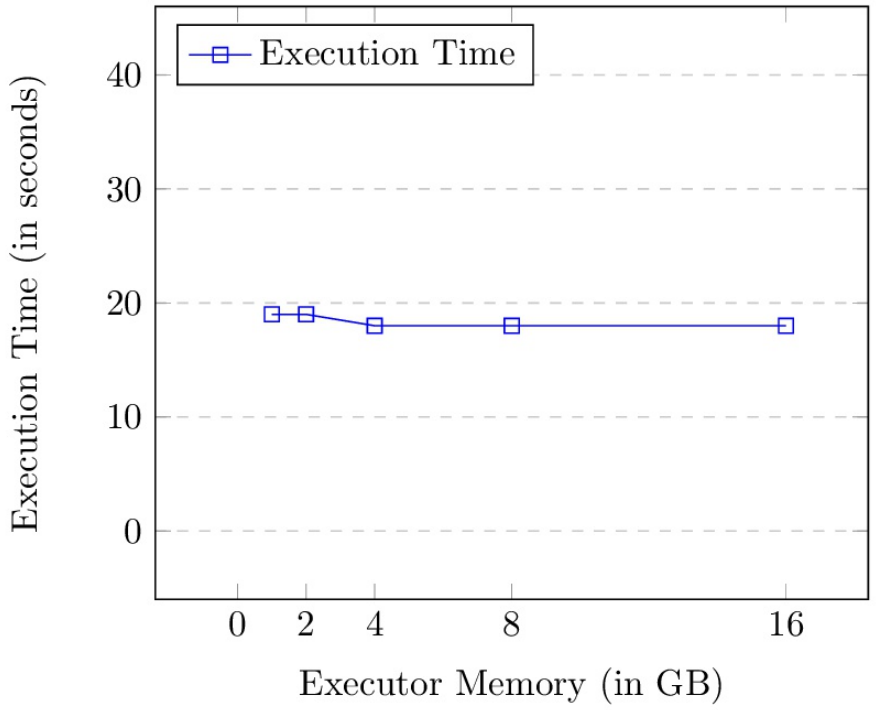
We try to manipulate the number of RDD partitions the input data is divided into before sorting. We believe that more partitions would induce more parallelism and hence afford greater performance. Below is a summary of our findings -



Execution time did not vary with the increase in the number of RDD partitions. We believe that this was because the input data was very small. As a consequence, inducing parallelism did little to change the effective sharding of input data. To understand further, there is very little difference in runtime between a multi-core machine processing a smaller number of moderately sized partitions against a multi-core machine processing a larger number of smaller sized partitions.

Effect of manipulating Executor Memory size

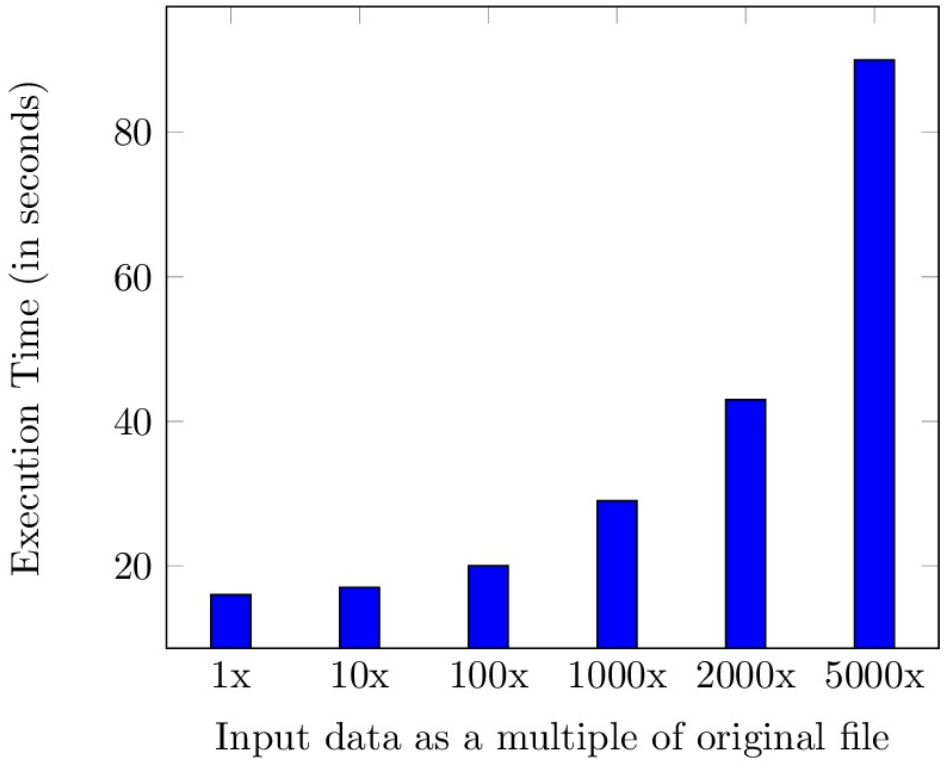
Since RDDs are processed in-memory, we try to analyze the impact of having restricted memory usage on the workers. We restricted memory by changing the setting named "spark.executor.memory" in the spark configuration file. Below is a summary of our findings -



We do see small improvements in performance when the executor memory is increased. However, the increase is not as pronounced as expected since the input data is not large. If input data was large and if the RDDs cannot be held in memory, they would spill over to the disk, thus resulting in degradation of performance.

Effect of Input size

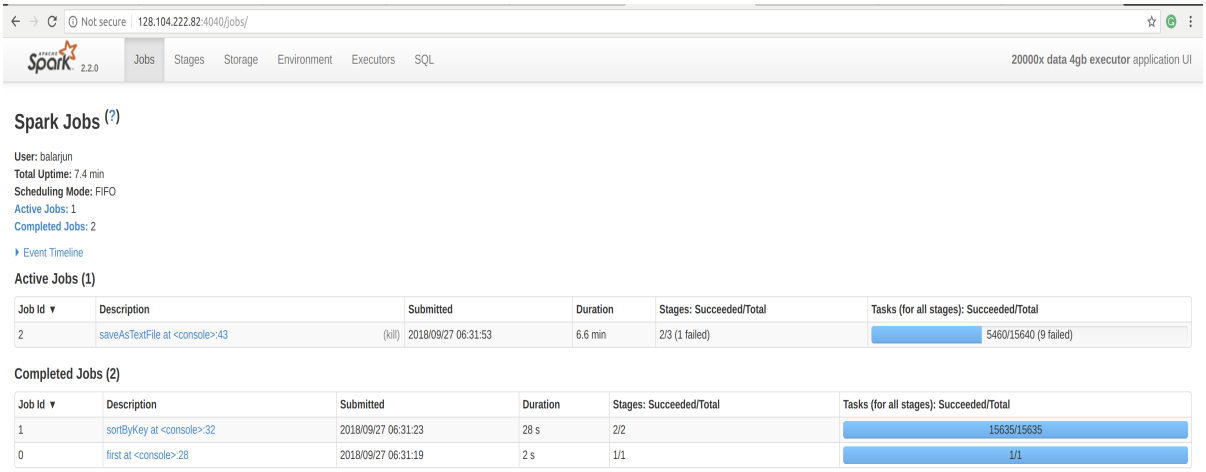
We expect the runtime to increase linearly with the input size. This is particularly true when the input size is such that the number of stages required to process them far exceeds the degree of parallelism available on the cluster. We use a Spark configuration having 4GB of executor memory and our experimentation on this configuration is summarized below -



As visible from the graph, we see pretty much a monotonically increasing correlation between the size of input data and the runtime. We achieved larger input sizes by replicating the same input CSV multiple times. We also cleaned up the disk cache and inode references before each execution to eradicate the effects of reads from cache.

Performance of Spark when a worker is under memory pressure

We setup Spark to have 4GB executor memory on the 3 workers. Prior to submitting the Spark job, we started running a simple C program on one worker that would continuously issue malloc() calls in frequent intervals. We did this to analyze the performance of a worker under memory pressure. The below diagram shows the stark reality of what was observed -



From the execution of stages, we found that initial stages proceeded normally without any hiccups. However, once the malloc() calls started starving the worker of memory, we noticed quite a bit of failures. In fact, the Spark master was trying to continuously allocate the task to the worker and it kept continuously failing due to lack of memory. We were really surprised that that the task was not offloaded to another worker. Eventually, we shut down the C program, which freed up all the memory and then the Spark job was able to complete successfully.

PageRank using Apache Spark

Introduction

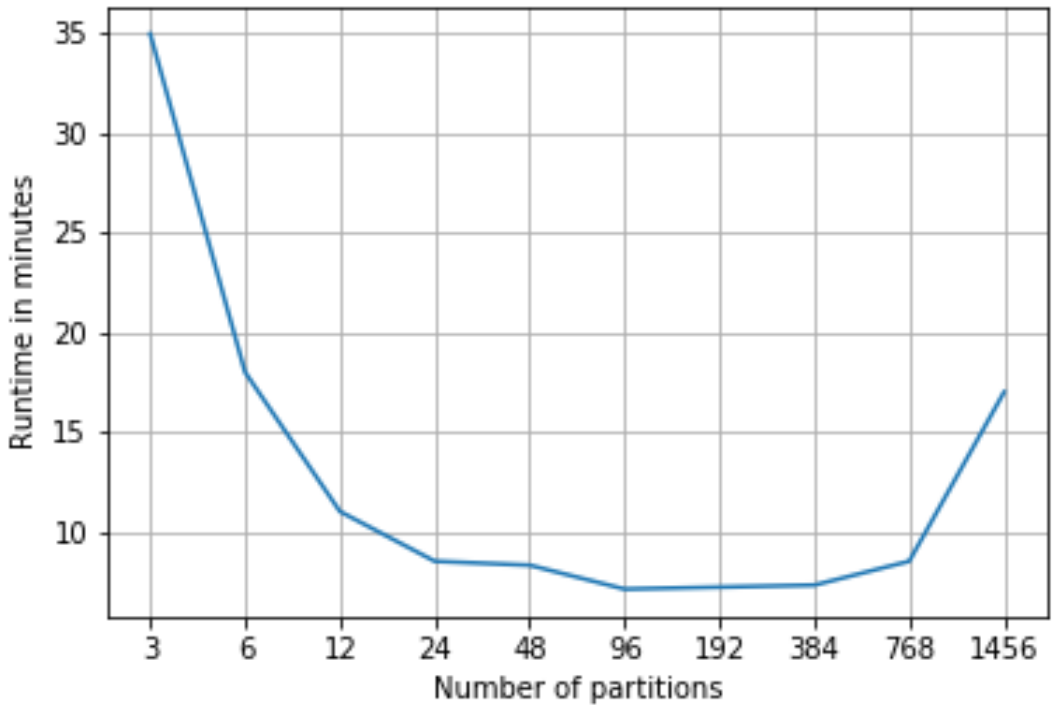
The second part of the assignment is to implement the PageRank algorithm using Apache Spark. PageRank is an algorithm used by Google to determine the quality of links while indexing web pages on the internet. We seek to determine the performance of Spark under various conditions that will be outlined below. For sake of brevity, we omit benchmarking against the small dataset in the assignment, since we believe that the observations would be akin to that of sorting. Hence, our observations below solely consider the large dataset provided in the assignment.

Impact of Number of RDD Partitions

We leverage RDD partitioning to partition both the graph data as well as the intermediate rank data. We partition it in this manner for two reasons -

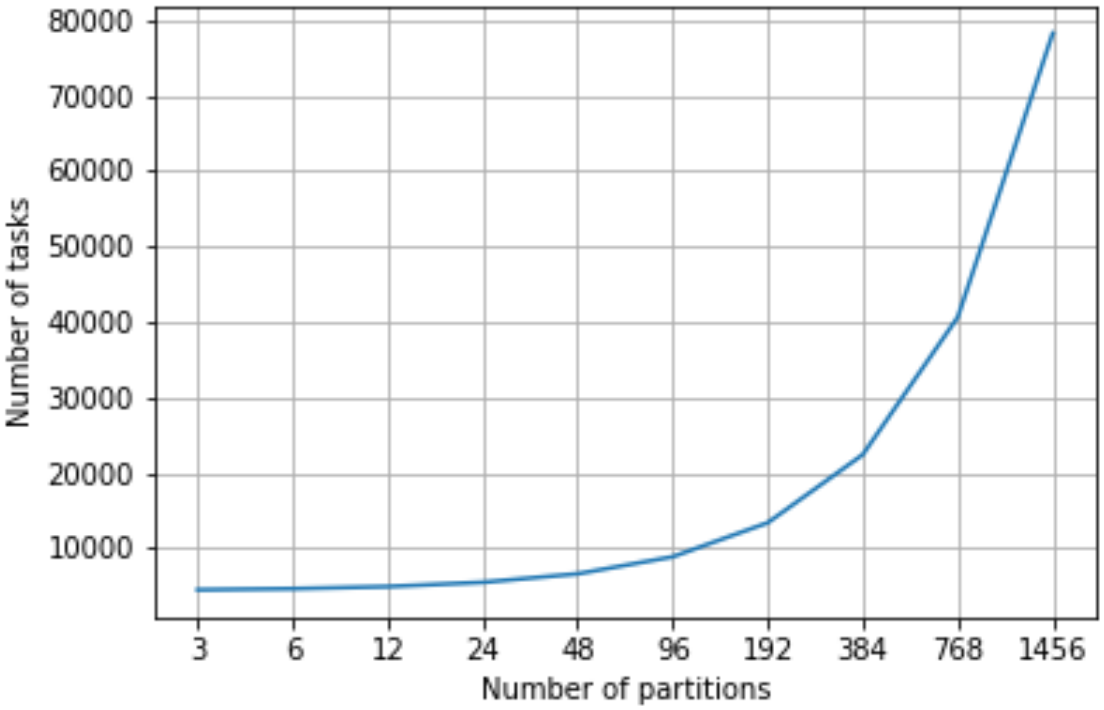
1. To reduce the shuffle when we need to join the graph RDD and the intermediate rank RDD.
2. To exploit all available threads in the cluster.

Below is the correlation between the runtime and the number of RDD partitions -



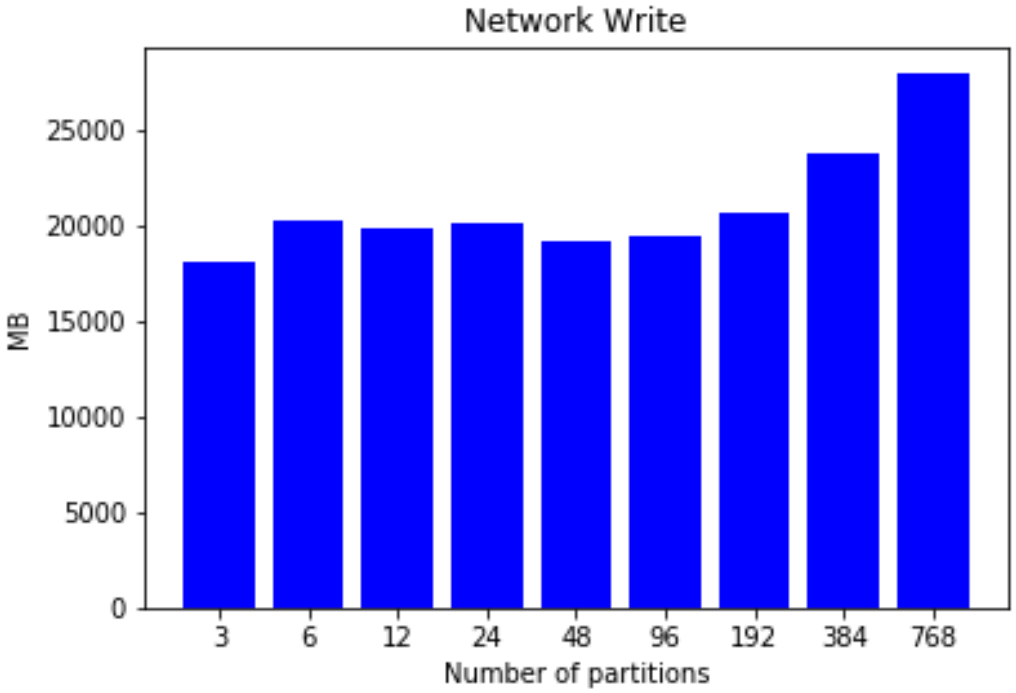
From the graph above, we can see that initially as the number of RDD partitions increase, the runtime decreases. This trend is observed till the number of RDD partitions becomes 96. Post this, we observe a region where the runtime is largely independent of the number of RDD partitions. Post 768 partitions, we observe a sharp increase in runtime.

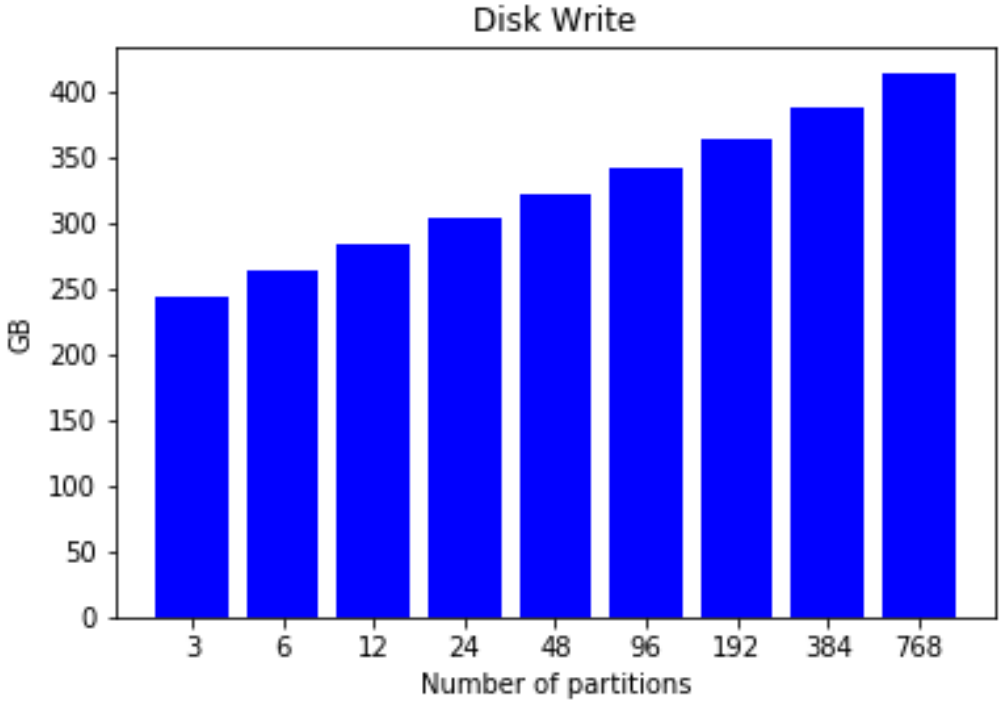
To understand this trend, it is useful to look at how Spark greedily creates tasks to exploit the parallelism of additional RDD partitions. The below graph captures the correlation between the number of RDD partitions and the total number of tasks that Sparks creates -



As visible in the above graph, the number of tasks has an exponential correlation with the number of RDD partitions. In general, parallelism is good up to the point that it can exploit the maximum number of available threads in the cluster. When the number of tasks goes extremely high, it essentially becomes a high number of tasks operating on finer shards of data. The hardware specification theoretically limits the number of tasks that can be run in parallel even if they have no dependency. Hence, it is better to have a limited number of moderately sized RDD partitions than many extremely small RDD partitions. For this workload, we can infer 96 partitions as an inflection point beyond which partitioning is detrimental.

Additionally, we monitored the disk interface on which the intermediate data is written and the network interface of the followers during the experiment. We have an interesting observation with respect to the network and disk usage as the number of partitions vary. The average values across workers are summarized below -


Network Write
MB
25000
20000
15000
10000
5000
0
3
6
12
24
48
96
192
384
768
Number of partitions


Disk Write
GB
400
350
300
250
200
150
100
50
0
3
6
12
24
48
96
192
384
768
Number of partitions

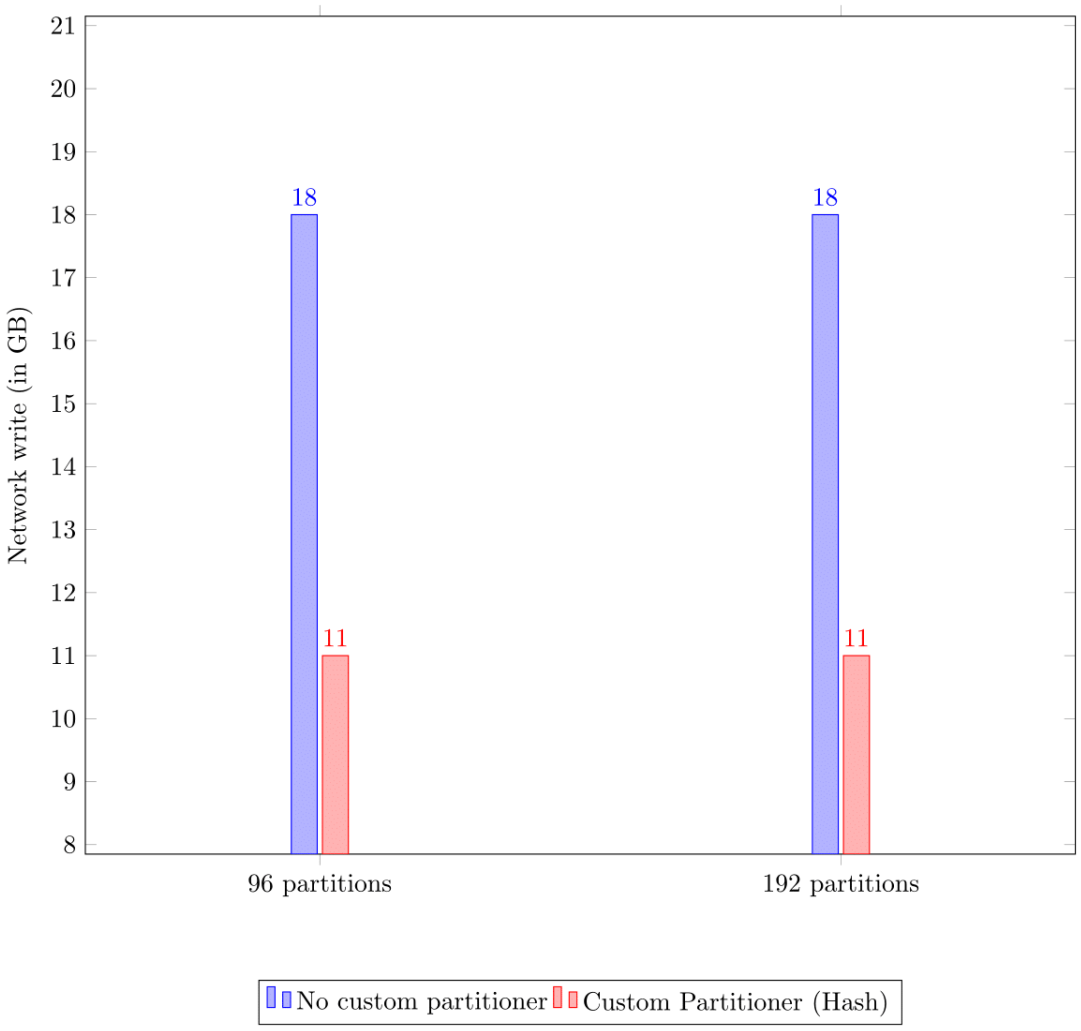
As the number of partitions increase, we notice a slight inflation in the amount of network and disk writes. This positively correlates to increasing runtimes observed beyond the inflection point of 96 RDD partitions.

Impact of using a Custom Partitioner

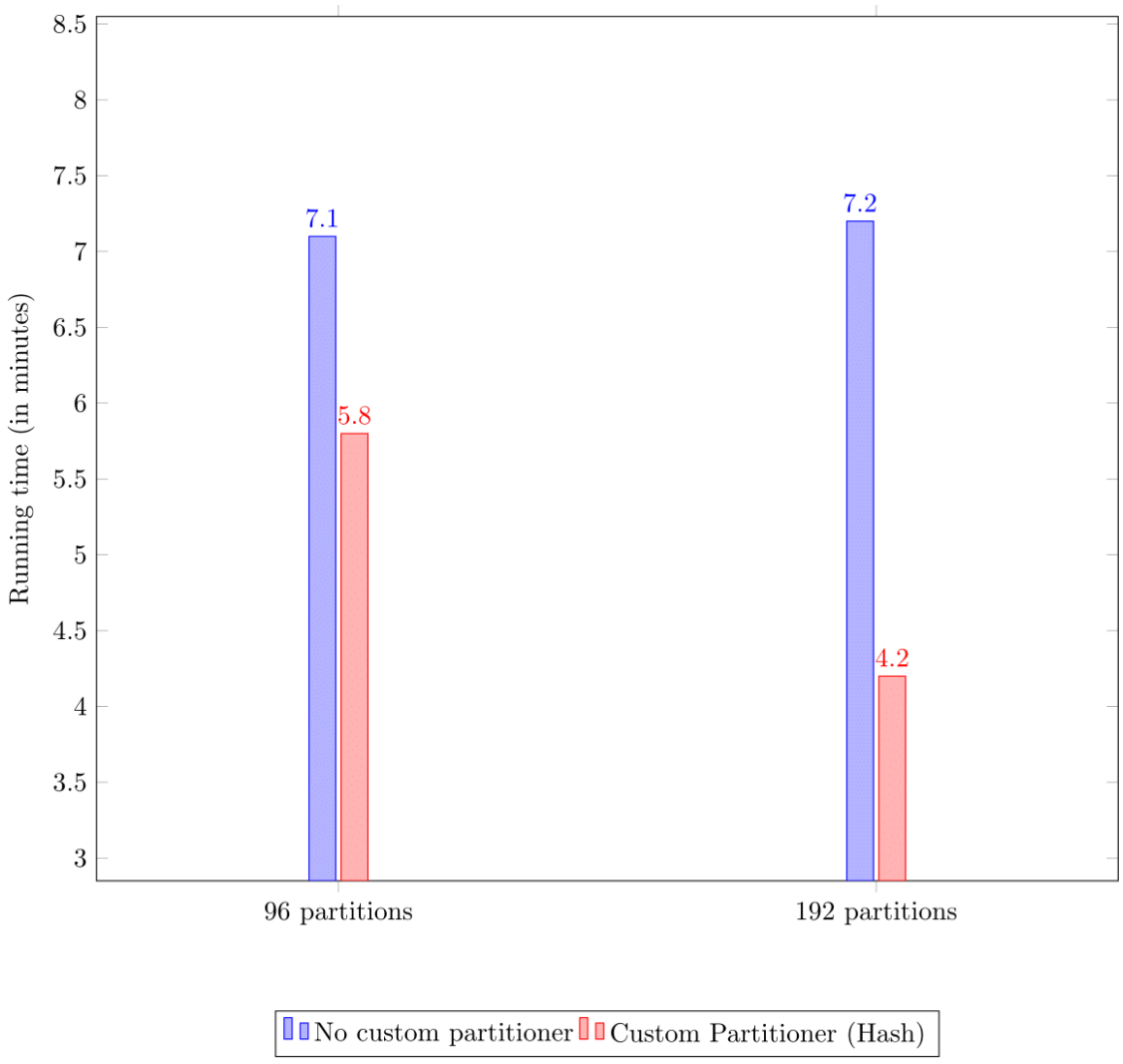
In our first experiment, we did not inform Spark how to shard the input data into partitions. Intuitively, this is bad for performance, since we would experience a large number of shuffles during operations like a join.

To improve performance, we can leverage the Hash Partitioner that Spark provides. It basically computes the RDD partition on which a data record should reside by computing the hash on the key. In effect, a Hash Partitioner would ensure that keys having the same value reside on the same RDD partition, which leads to less network shuffle and consequently better performance.

First, we benchmark Hash partitioning against the default partitioning by empirically showing that network shuffles are reduced. The backing data is presented below -

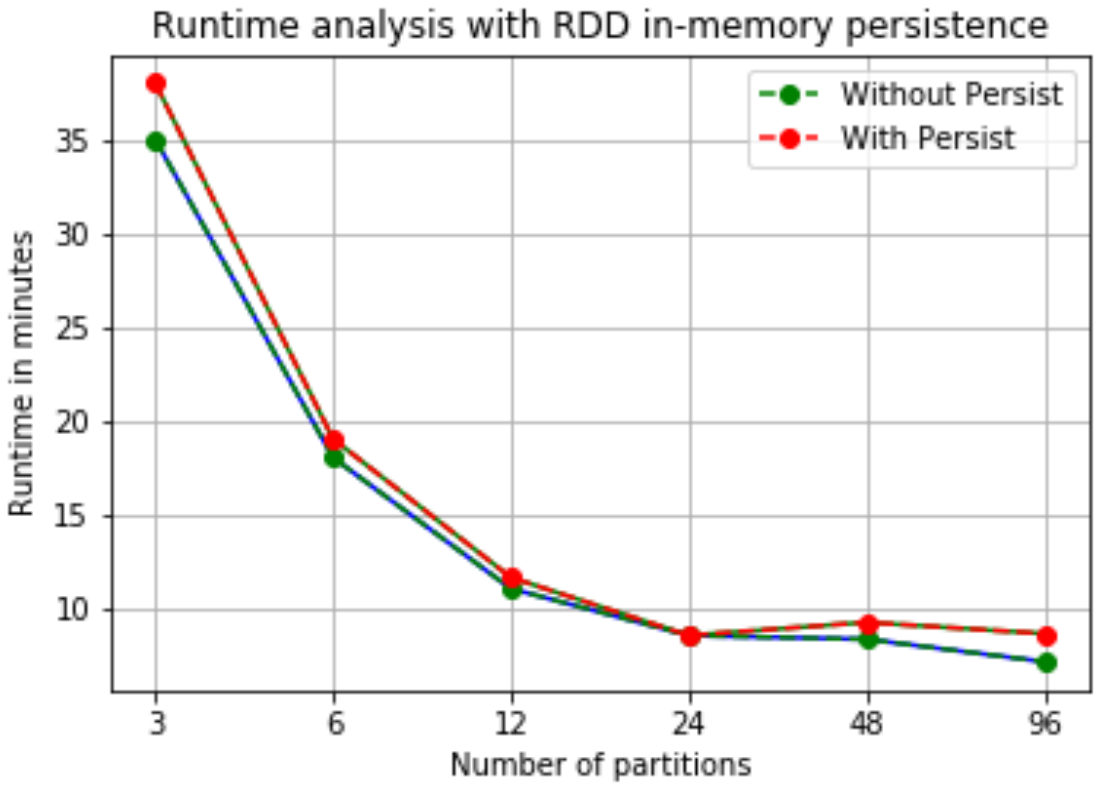


Consequently, we also notice a reasonable decrease in the runtime by using Hash Partitioning as evidenced from the graph below -



Impact of persisting RDDs in memory

Explicitly persisting RDDs in memory using the persist() operation is one technique to improve the performance. We try to persist the input graph RDD since this would optimize the join with intermediate rank RDDs. Additionally, we try to persist the intermediate rank RDD across iterations for the same reason. We expect the performance to improve by using persist. Below are our observations -



Contrary to our expectations, we see that the persist operation does not give us performance gains. In fact, it seems to be detrimental to performance in a few scenarios. We believe that this is simply due to the large size of the RDDs. Large-sized RDDs overflow into the disk and hence the advantage of the persist operation is not realized.

Impact of using a Custom Partitioner along with persisting RDDs in memory

We wanted to observe the combined effect using custom partitioning with persisting RDDs in memory. For this we chose the number of partitions to be 96 because this gave us optimum performance in most of the above experiments. We used default Hash Partitioner as above for partitioning. Runtime for this experiment was **4.2 minutes** which is a significant improvement from individual optimizations in which case runtimes were around 7 minutes.

This implies that we can use different optimization techniques in tandem to improve the overall performance.

Impact of killing a Spark Worker midway through the task execution

We kill one of the Spark worker tasks midway through the execution of the Spark task. As expected, we were able to see logs that indicated tasks executing on the killed worker that were lost. These tasks were then re-executed on the remaining available workers. By killing a worker, we expect the runtime to increase due to two main reasons as below -

1. Lost tasks need to be re-executed.
2. Remaining tasks need to be executed on 2 workers instead of 3 workers and this reduces the degree of available parallelism.

The below graph captures the impact of killing a worker -