Analyzing Data with Spark - Lab Report

Hassan Eskandar[1], Hussein Awala[2]

[1] M2 MoSIG - AISSE – hassan.eskandar@grenoble-inp.org

[2] M2 MoSIG - DS – hussein.awala@grenoble-inp.org

January 21, 2019

1 Introduction

This assignment is about analyzing a large dataset using Apache Spark. The dataset has been made available by Google. It includes data about a cluster of 12500 machines, and the activity on this cluster during 29 days. This lab is an opportunity to process large amount of data and to implement complex data analyses using Spark. Both Python language and PySpark API (Resilient Distributed Datasets) are used to analyze the data.

2 About the Dataset

The dataset consists of multiple folders that are inter-correlated and the objective of the project is to analyze the data in order to discover the relation among different events in the dataset. Part of the dataset is described below by Google: A Google cluster is a set of machines, packed into racks, and connected by a high-bandwidth cluster network. A cell is a set of machines, typically all in a single cluster, that share a common cluster-management management system that allocates work to machines. Work arrives at a cell in the form of jobs. A job is comprised of one or more tasks, each of which is accompanied by a set of resource requirements used for scheduling (packing) the tasks onto machines. Each task represents a Linux program, possibly consisting of multiple processes, to be run on a single machine. Tasks and jobs are scheduled onto machines according to the lifecycle described below. Resource requirements and usage data for tasks are derived from information provided by the cell's management system and the individual machines in the cell.

3 Answering the Questions

1. What is the distribution of the machines according to their CPU capacity?

   In this question, we are asked to count the number of occurrences of all available CPU resources that are distributed over the machines. We started by filtering the RDD to remove the NULL values, then mapped the data as tuples with 2 columns ID and CPU. Reduce the RDD to remove the duplication by returning one value for each key, then mapping the dating by assigning the value 1 to each CPU instance, apply reducebykey to the current RDD and

sum up all the values to get the number of occurrences of each CPU. Figure 1 shows the result.

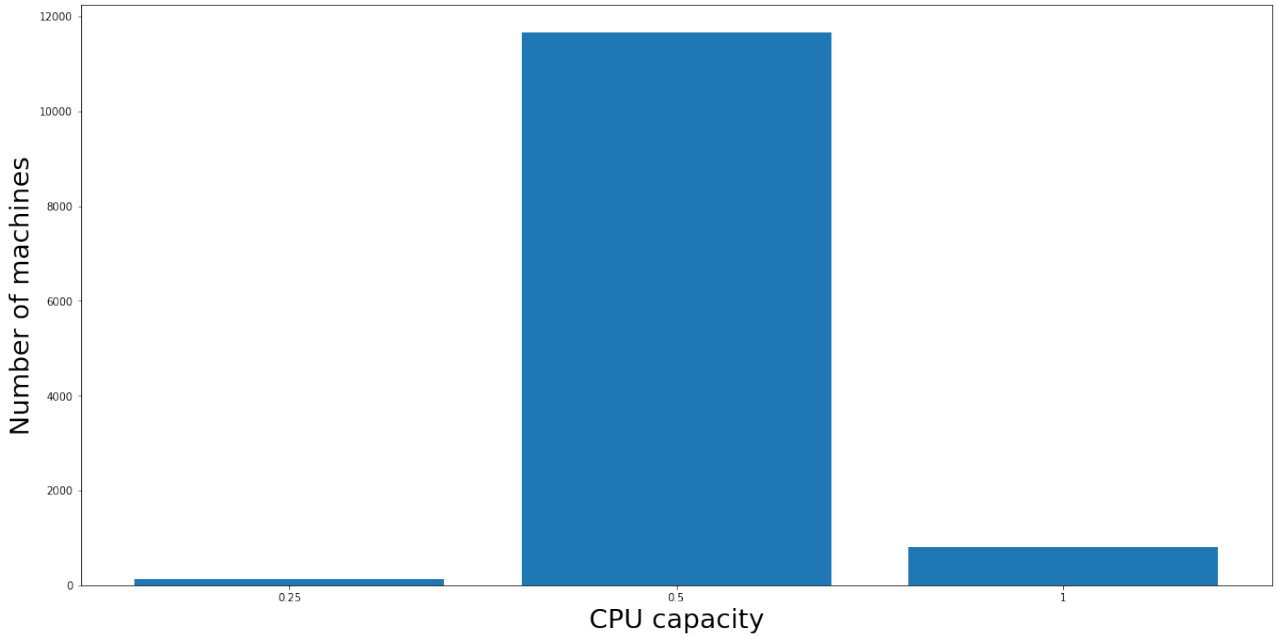


Figure 1: Distribution of Machines According to their CPU Capacity

2. On average, how many tasks compose a job?

   Here we want to compute the average number of tasks assigned to each job. First, retrieve the data(ID and task index as tuples) from the task event table, map the data as tuples(job ID, task index), calling distinct function will remove the duplicates. The output is the current RDD that contains tasks data, use it to map the job ID only, finally calculate the average by dividing the task number over the job number. After running the script on the whole data, result is **37.83%**

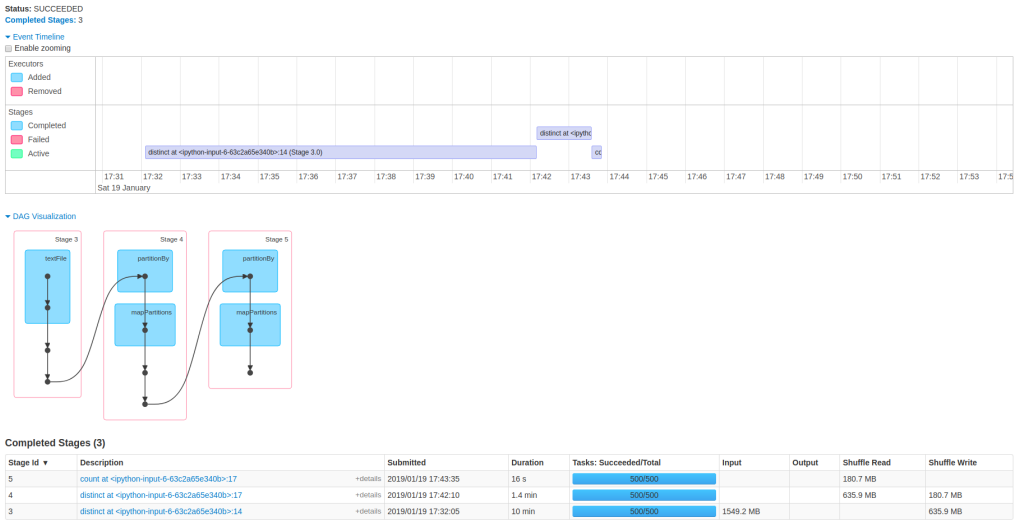


Figure 2: Capture from PySparkShell application UI

We can see that Spark created 3 stages one for each RDD, and all the transformations on the RDD are represented in form of points in the stage. The relation between the stages(fleshes) represent on which RDD we based to construct the new RDD(we can create a RDD by reading from file, by paralleling an a list, or using another RDD). Spark took long time to remove the duplication from the whole file. It took a little time to remove the duplication from the last RDD and compute the count, while it didn't take the same time to compute the count on the same RDD, we can conclude that the distinct action is very expensive on Spark.

3. What is the percentage of jobs/tasks that got killed or evicted?

   The idea here is to look at the task_event table, fetch and count the number of tasks column, and from the task_event RDD map the data as tuples((job id,task index), event type), filter the data to get the events evicted and killed, and finally to compute the percentage of tasks killed or evicted, divide the number of tasks that are either evicted or killed over the overall tasks. We got a percentage of **11.2%** after running the script on the whole data. We did the same study on the jobs, using the job_event file and only the job id as key for each job, the percentage of evicted or killed tasks is **13.54%**.
   If we want to analyze this difference we find that it's normal, because when the task of a job is evicted or killed, we consider that the job is evicted or killed also, that makes the percentage of evicted or killed jobs grater than that of tasks.

4. Do tasks with low priority have a higher probability of being evicted?

   Figure 3 and Figure 4 show the distribution of evicted tasks over all priority and over each priority respectively. To approach this, we first mapped the data in task_event table as tuples (priority, event type), group the data by priority, for each group calculate the percentage of tasks evicted from all tasks, sort the data according to their priority then compute correlation between the priority and the percentage evaluated by using the function np.corrcoef(). Analyze: there is a big correlation between the priority of the tasks and tasks evicted which is equal to **-0.7363035**.
   This result is very logical, because the reason of eviction is often to run a task with higher priority, so the higher the priority of a task, the less likely it is to evacuate.

   When the number of tasks with lower priority is much bigger than that with high priority, we studied this problem with other approach, this outcome may not represent reality, because if we have only 10 tasks with priority 11, and 5 of these 10 are evicted, in the first approach we will obtain a low percentage because 5/(number of evicted tasks) is very small, while the half of the tasks with priority 11 are evicted.
   Now we have to compute the percentage of evicted tasks for each priority from the number of tasks having this priority, to do this calculation, we got all the tasks with their priorities to group them by priorities and compute the number of tasks having each priority, then we filter the same RDD to got only the evicted tasks then do the same group by to obtain the number of evicted tasks having each priority, and finally we write a python loop to compute and plot our result.
   From this new result we cannot conclude if there is a relation or not, because more tang 25% of the tasks with priority 10 are evicted and the priorities 0,1,3 have high percentages also.

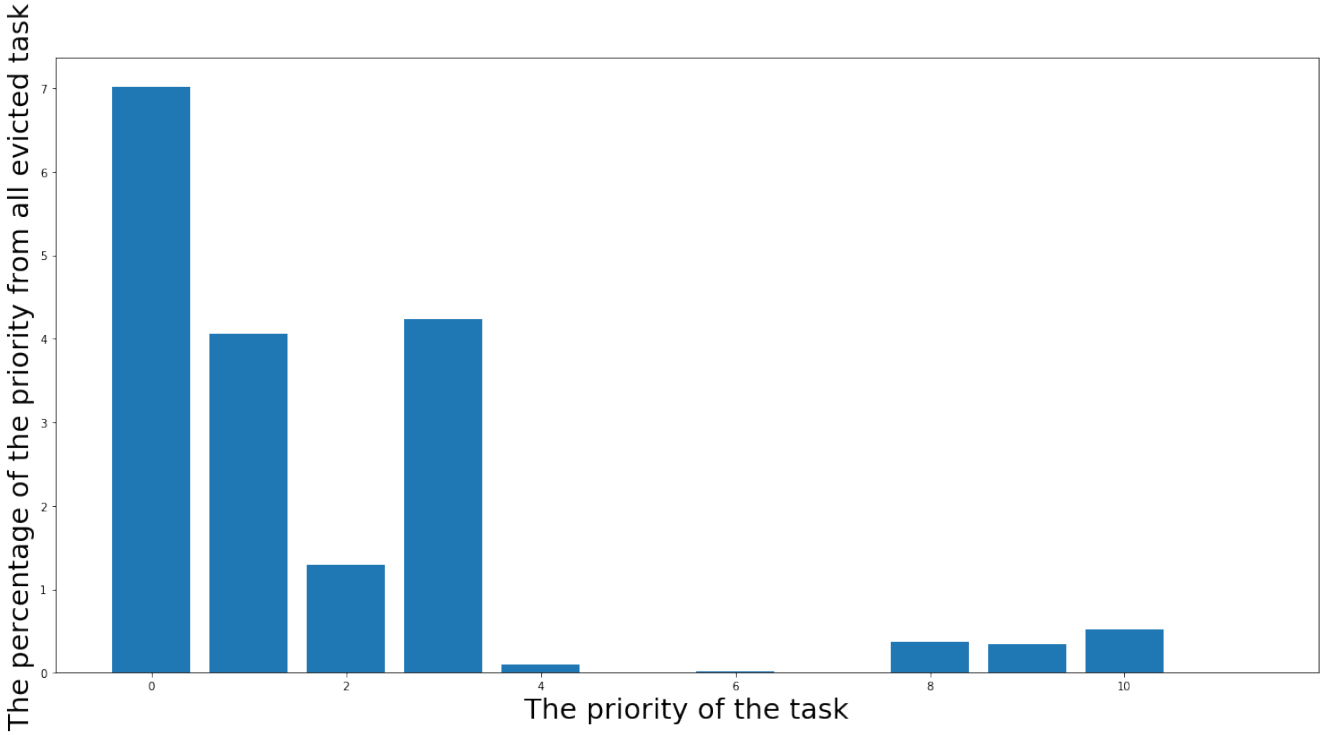


Figure 3: Distribution of evicted tasks over the priorities

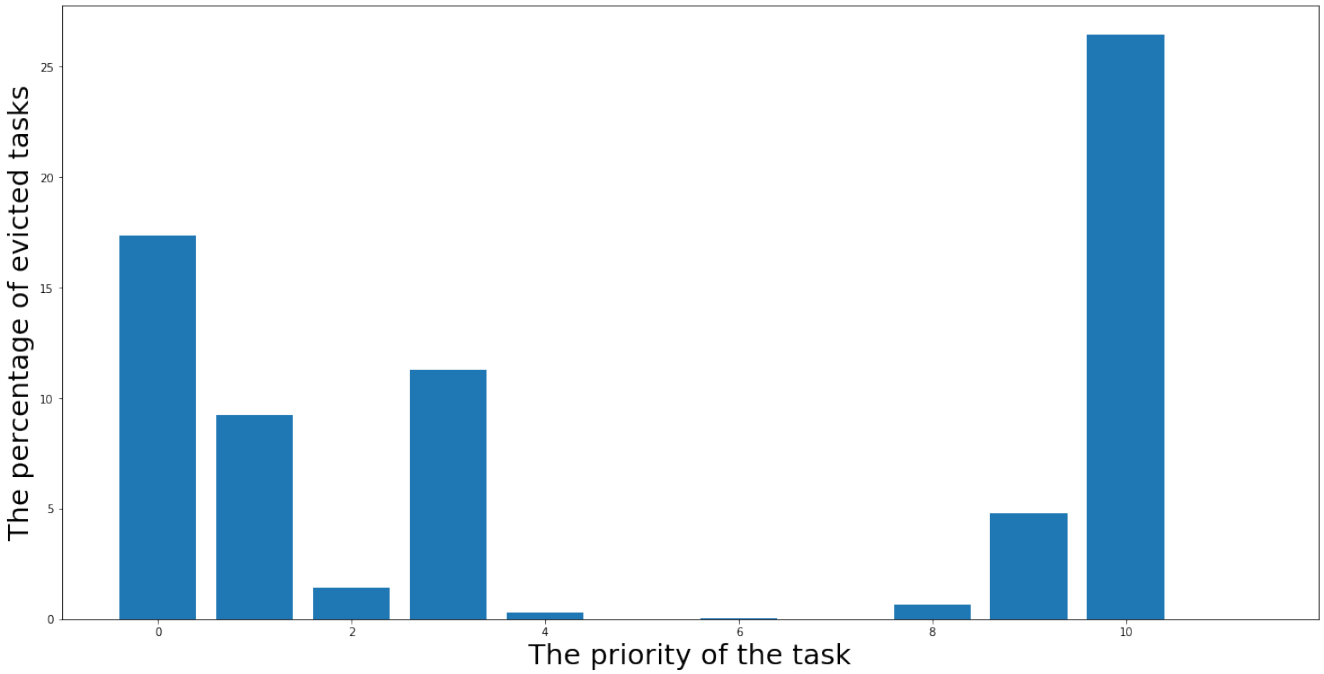


Figure 4: Percentage of evicted tasks for each priority

5. Is there a relation between the priority of a task and the amount of resources available on the machine it runs on?
   We have to compute the available resources for each machine at each time stamp. To do this we subtract the used resources from the total resources of the machine. And to compute the used resources for each machine at each time stamp we calculated the sum of resources used by each task running on this machine at this time stamp. Then, we created a RDD that contains the key of the task and the machine ID at the given time stamp. Then, we join this RDD with the priority of each task, then we re transformed the form of the RDD to ((machine ID, time stamp), priority) and we calculated the percentage of high priorities found on each machine at the given time stamp by setting a threshold of greater than 6 to be high priority, low priority otherwise. Finally, we joined the resource available RDD with the distribution of high priority RDD, and computed the correlation between them.
   After computing the correlations between the percentage of tasks with high priority over all task in each timestamp in each machine and the CPU and the memory available in this machine at this timestamp (-0.034 between CPU and priority and -0.05 between memory and priority), we conclude that there isn't any correlation between the priority of the tasks and the resources available in the machine.

6. Are there tasks that consume significantly less resources than what they requested?

   Here the key aspect the analysis of the usage of the resources at each execution time. Since it was not detailed in the question, 2 strategies were used to approach this question, first to compute the maximum usage of the resources and the second one is to compute the average usage of the resources.

   - Computing the max: Using the task_usage RDD, first we data in the following form ((Job Id,task index),(maximum CPU rate,maximum Memory usage)), group the data with the key (Job Id,task index), then map the values according a function which takes the iterator of the elements resulted in the groupby operation as input and returns the max of this values as output. On the other hand, using create a request_resource RDD and map the data ((Job Id,task index),(requested CPU,requested Memory)) as tuples to know the requested resources, then join the 2 RDDs. After joining these two RDD, we mapped the ((Job Id,task index),(requested resources, difference between the requested and the used resources)), where this difference is positive when the task didn't use all the resources that it required, and negative when it used resources more that it required, after this mapping we filter the data to keep only the positive number(tasks used less than that required), then another filter ti keep only those that used less then the half of the required resources. Finally, we computed the percentage of these tasks over all tasks. We did this study for memory and CPU.
   - Computing the average: Same approach is used but with replacing the the max by the average and with tuples formed is ((Job Id,task index),(CPU rate,canonical memory usage)), and function takes the iterator of the elements resulted in the groupby operation as input and returns the mean of this values as output, and the same steps for the reset of the code.

   To define the case of "significantly less", we compute the percentage of the tasks that use

resources less than the half of that they require, the result was as follow:

- 42% of the tasks with maximum usage of CPU does not exceed half of what is requested.
- 68% of the tasks with average usage of CPU does not exceed half of what is requested.
- 70% of the tasks with maximum usage of memory does not exceed half of what is requested.
- 72% of the tasks with average usage of memory does not exceed half of what is requested.

7. Are there machines for which the resources are largely underused?

   To study the usage of each machine, we have to compute the average of resources used during the running time of the machine, to compute this mean, we use the task_usage RDD and at every timestamp, we can retrieve the resources running on each machine. Map the result as tuples of machine ID as the key and both sum of CPU and sum of Memory as the value. apply group by on the machine ID in order to calculate the mean of tuples over each time stamp, and from the machine_events table we fetch the available resources in order to clean it by filtering all the NULL values and removing and duplicates and save it in a new RDD. Finally, apply join function over the 2 RDDs to compute the average of available resources over the time. Then we computed the mean of the unused resources on each machine by subtract the mean of used resources from all resources of each machine. Finally we calculated the percentage of unused resources from all resources and we sorted the result. After collecting the result, we had a list contains the percentage of unused resources for each machine sorted from high unused CPU to low, by analyzing the result, we observed that 56% of machines have more than 80% of unused CPU which is a big percentage, and 39% machines have more than 80% of unused memory which is also a big percentage, so the answer of your question is yes, there are a lot of machines for which the resources are largely underused.

4 Added Questions for Analysis

We extend our analysis to answer the following question that is related to question 6:

Are there tasks that consume more resources than what they requested?

To approach this, We computed the max of resources used during the execution time, and we get the required resources for each task from the task_event table, we joined these 2 RDDs and we subtracted the used resources from the required resources and finally we computed the percentage the negative valued(negative values represent the tasks which use more than they required).

second approach: we applied same algorithm but instead of using the max of resources used during the execution time, we used the average of resources...

The surprise was that 48% of the tasks at an instant(max use) use CPU more than they required, and 19% of the tasks use memory more than they required. And by the second approach that's more logical and efficient to make a conclusion, we observed that 33% of the tasks use CPU more than the requested, and 5.6% of the tasks use memory more than the requested.

After this study, we can say that there is a problem in the estimation of the resources required

to execute for each task, that's can cause many problems. For example, a task ask a worker to execute with required memory 500MB, and this worker has 550MB and accept this request, then the task during the execution needs 600MB, that's will stop the execution of this task and maybe another tasks with an out of memory exception.

5 Lazy Evaluation

As the name itself indicates its definition, lazy evaluation in Spark means that the execution will not start until an action is triggered.

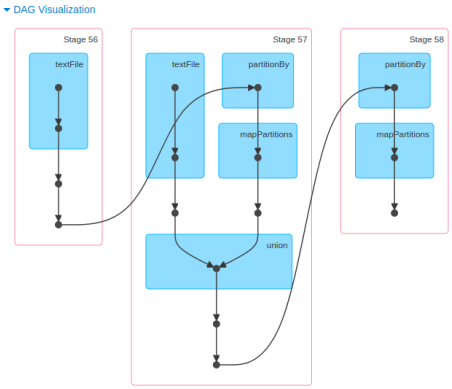
Transformations are lazy in nature meaning when we call some operation in RDD, it does not execute immediately. Spark maintains the record of which operation is being called(Through DAG). We can think Spark RDD as the data, that we built up through transformation. Since transformations are lazy in nature, so we can execute operation any time by calling an action on data. Hence, in lazy evaluation data is not loaded until it is necessary.

We computed the execution time of the transformations, the actions and the actions after caching the result of the transformation stage in the question 7, the duration of the transformations is 0.22 seconds, that of actions is 38.35 seconds and that of actions after persisting the result of the transformations is 1.032 seconds. We can conclude that Spark toke 0.22 seconds to create the DAG of the transformations sequence, 37 seconds to do these transformations, and 1 second only for the actions. The testing of the same code after persisting prove the Spark's lazy evaluation, where in this case the time of execution is the time of actions only.

6 Persistence of RDD

By default, a RDD is recomputed for each action run on it. Calling the function *persist()* will cache the RDD being used in memory for later usage, this will save time especially if the data under analysis is huge. Also, note that a call to persist does not trigger transformations evaluation and will prove this in an example. Please refer to the figure below for more information. In memory computing is proved to be 10k times faster than that of the Disk [From slides]

As you can see in Figure 5, the time taken to finish the task is 19 min (1.4 min to read the file and group by the key, 15 min to read another file and join it with the first RDD where the join action is very expensive, and 2.6 min to compute the count) whereas in Figure 6 it took only 2.5 min to finish the task where we persist the RDD after doing the join, now if want to compute the count, collect the rows, or use this RDD to create another RDD, spark will use the saved RDD directly without recreating it using its DAG, that can save 16.4 min in our simple example for each use of this RDD.



Completed Stages (3)

Stage Id ▼	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
58	count at <ipython-input-78-e7c967cd292c>:11 +details	2019/01/19 19:28:27	2.6 min	505/505			2.0 GB	
57	join at <ipython-input-78-e7c967cd292c>:8 +details	2019/01/19 19:13:10	15 min	505/505	1549.2 MB		272.1 MB	2.0 GB
56	groupByKey at <ipython-input-77-f8755f46dda9>:36 +details	2019/01/19 19:11:48	1.4 min	5/5	426.9 MB			272.1 MB

Figure 5

7 Performance Evaluation

To study the benefits of working with multiple thread on Spark, we try to compute the number of distinct tuple (job ID,task index) in the task_events RDD with differnt number of cores(from 1 to 4), but while we cannot change the number of cores after defining the SparkContext, we had to restart the server after each job to change the number of cores.

Number of Cores	1	2	3	4
Time (in Seconds)	1297.84	753.86	678.17	637.93

Table 1: Performance Evaluation

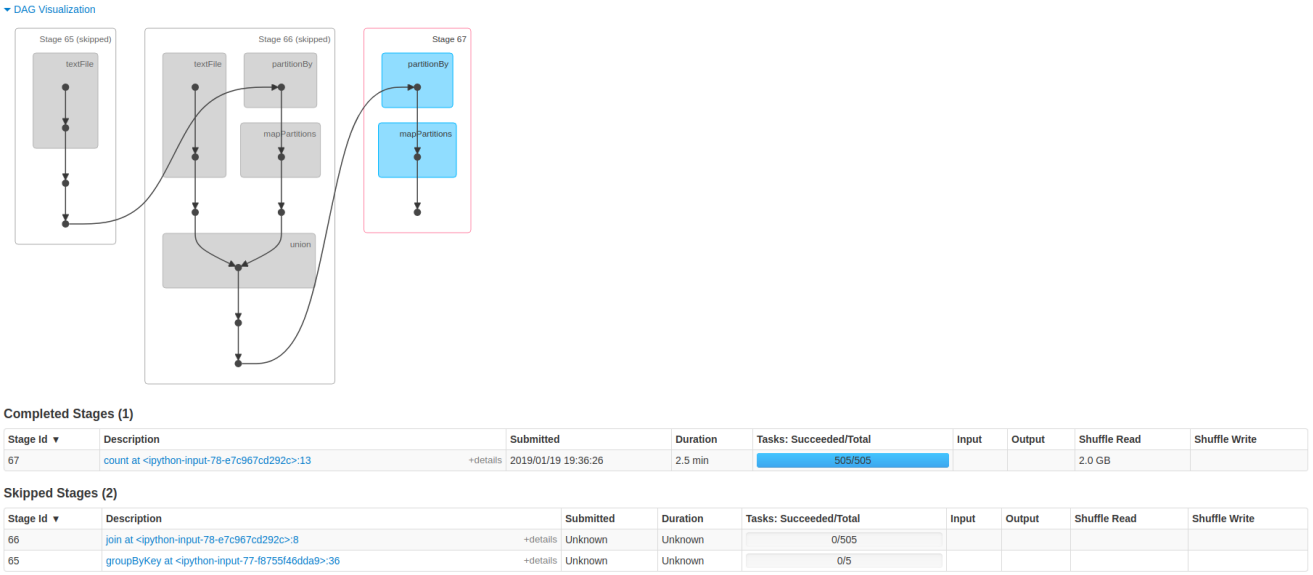


Figure 6

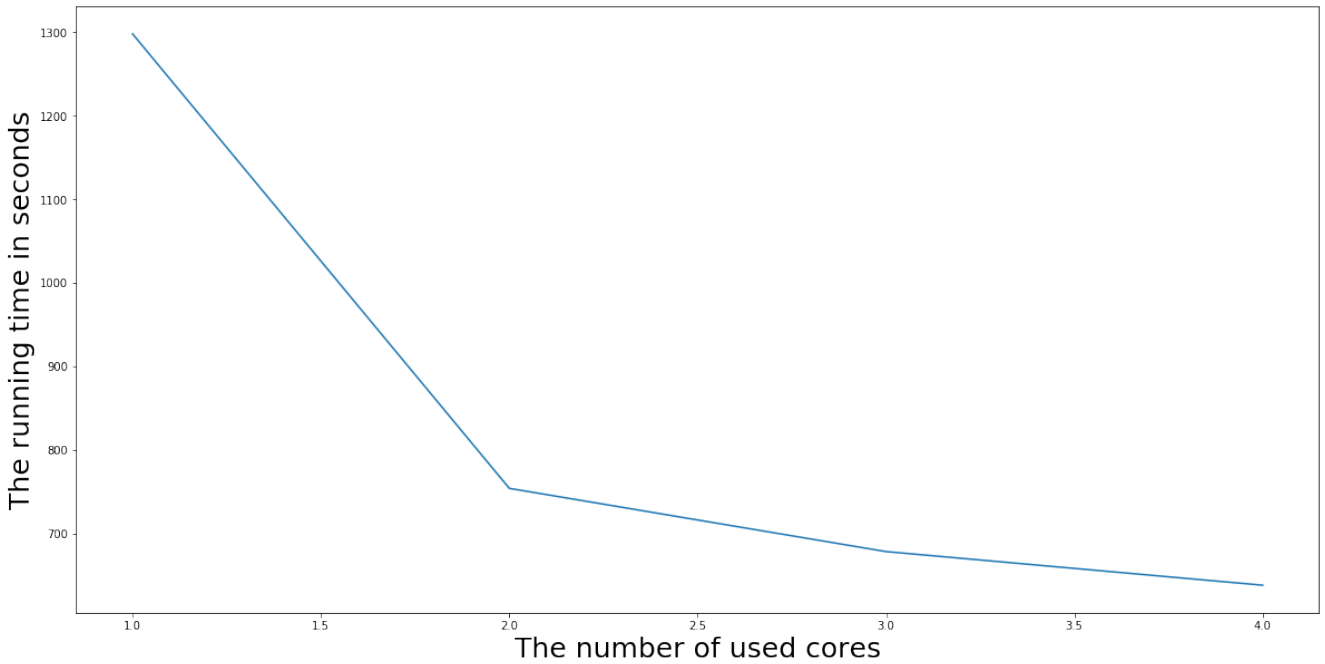


Figure 7: The variation of running time with number of cores

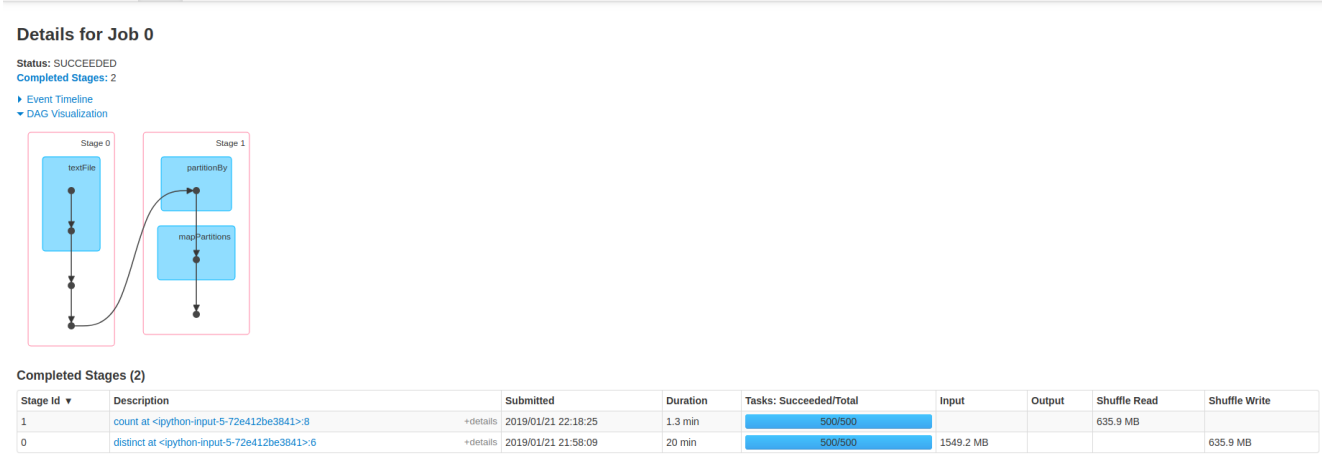


Figure 8: The execution time with 1 core

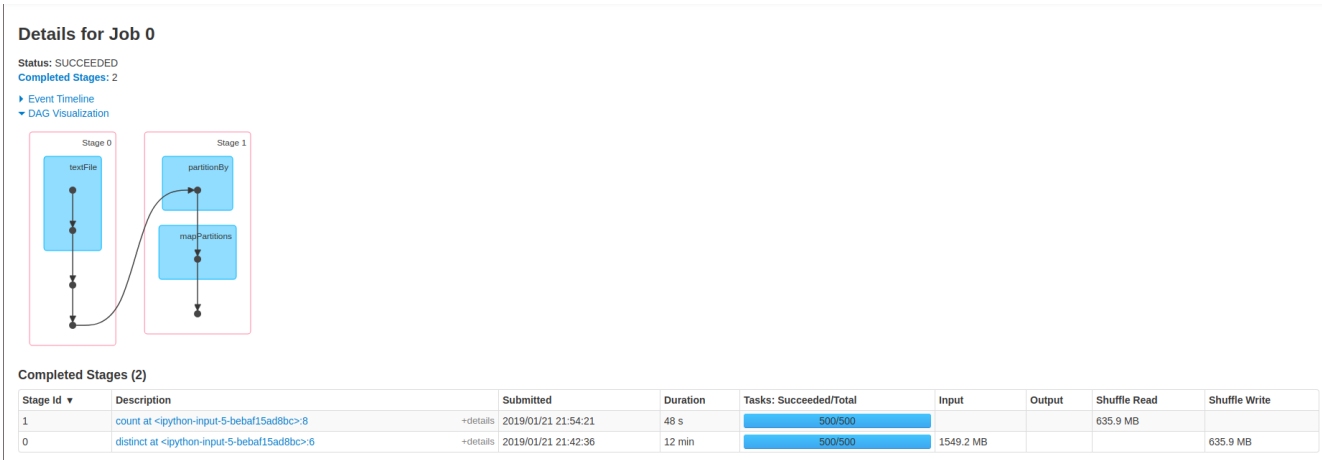


Figure 9: The execution time with 2 cores

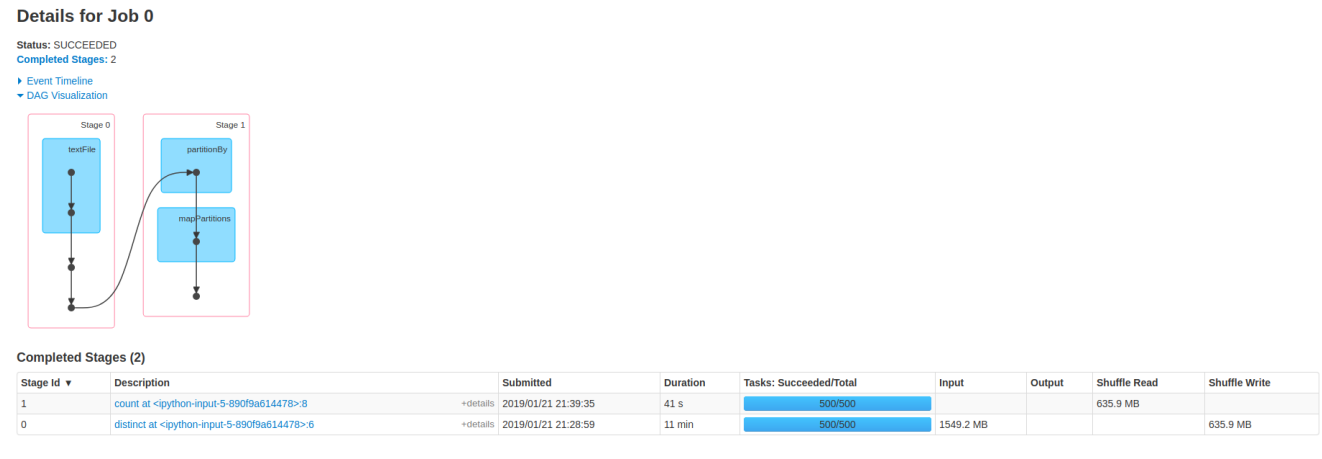


Figure 10: The execution time with 3 cores

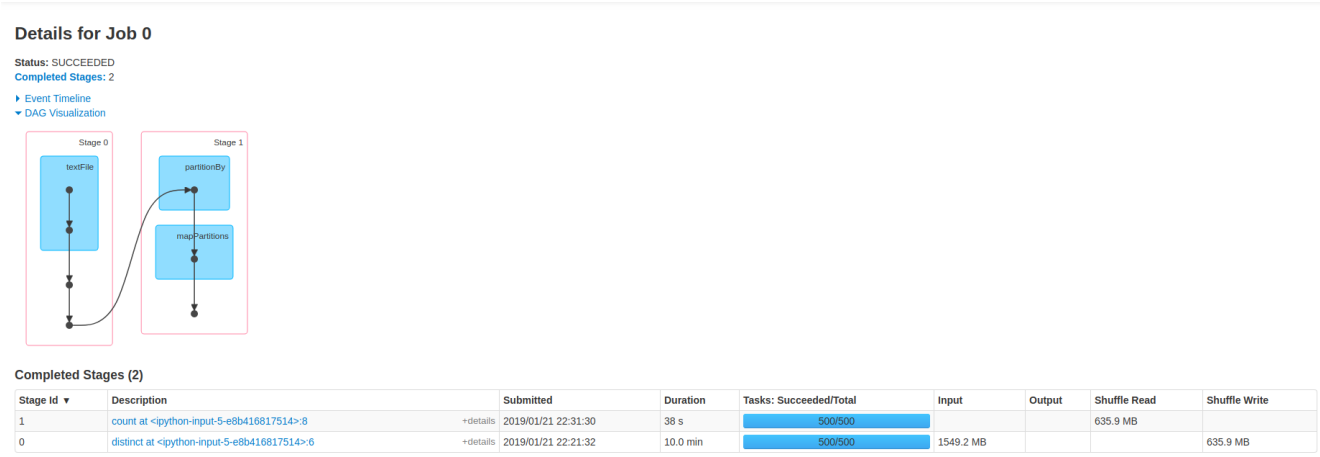


Figure 11: The execution time with 4 cores