Lab on SPARK
"Analyzing Data with Spark"

Alisa Gazizullina
gazizullina2010@yandex.ru

Vera Sosnovik
sosnovikvera@gmail.com

1 Introduction

The main goal of this laboratory to use Apache Spark for analyzing big data sets. In this lab the Google's data set was used. To be more exact, it represents 29 days of activity in a large scale Google machine featuring about 12.5k machines. Moreover, the data set includes information about the jobs that were done on cluster during this period of time and information about resources: their capacity and usage.

In most of the studied problems we have used the whole collection of traces.

2 Question 1

In the first question we wanted to analyze the distribution of the machines according to their CPU capacity.

In order to answer this question we used *"machine events"* table. After reading information from the files, we used function *'map'* for splitting data. Next, we removed rows with CPU = 0 and with missing values by using the function *'filter'*. The last step for data preparation was to sort data according to their value of *timestamp* by using *'SortByKey'*.

As long as machines with the same id may appear multiple times in the trace and the amount of resources allocated for each machine is static regardless of its place in the trace we removed rows with the duplicate $'machineID'$, thus resulting in a table with unique $'machineID'$. Then using map and reduce operations we aggregated the quantity of each unique value of $'CPU'$ column and constructed the histogram of the distribution of CPU's according to their capacity($figure\,1$). We observed that the amount of machines with the highest capacity (being 346) is smaller relative to the once acquiring twice as less CPU capacity (being 4979). However, there are only 87 out of 5412 machines with the smallest CPU capacity of only 25%.

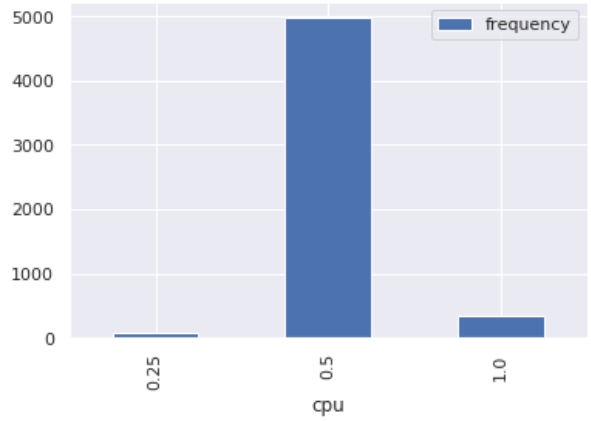


Fig1

Then we decided to analyze the distribution of machines with different CPUs by time, so as to see how the machines with different CPUs appear in the pre-selected window of the trace which is with the time stamp $\in (0, 2^6 3 - 1)$.

In the $figure\,2$ for the machines with the maximum CPU capacity being added over time we see that the graphic fluctuates significantly at the beginning of the window and shows the decline in the number of machines coming close to the end followed by the stabilized raise in the number of machines added. The graph for the machines with the 50% CPU capacity $figure\,3$ depicts the more stable trend with one peak at the end of the trace. Unlike for the 100% and 50% CPU capacity machines, machines with the 25% CPU capacity are added with some delays, so as we can observe in the yellow graph $figure\,4$.

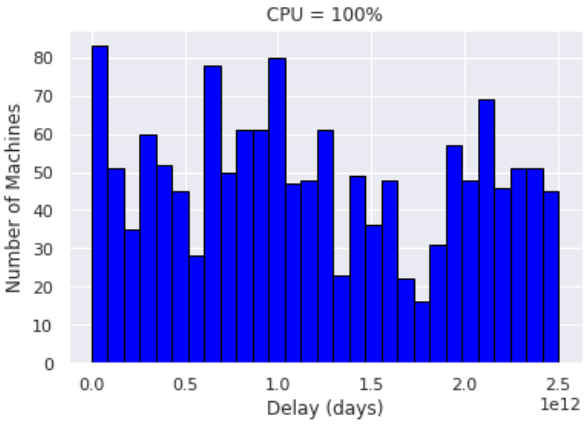


Fig2

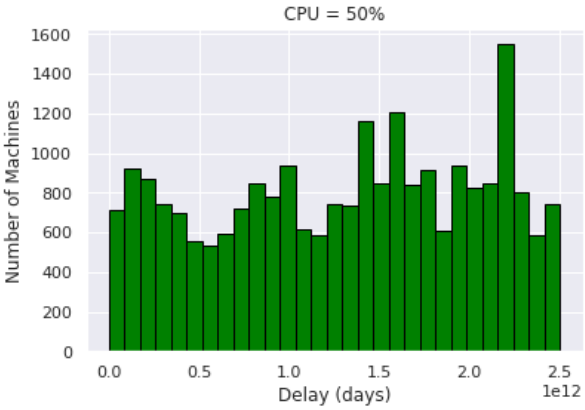


Fig3

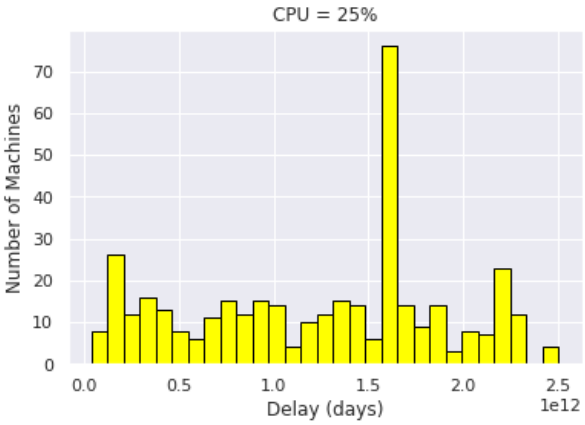


Fig4

3 Question 2

The second task was to analyze how many task compose the job.

We used *"task events"* table to calculate it. As long as jobs with the same appear multiple times in the trace with the different tasks, also resulting in the duplicates identified with the same pair of $'jodID'$ and $'taskID'$ (due to the fact that evicted tasks automatically restart continuously until killed) we dropped the duplicates.

Then we used grouped tasks by $JobID$ and $Aggregate$ them for matching every $JobID$ to all tasks($TaskID$) job comprised of. After that, we used function map to calculate the number of tasks that every $JobId$ has. Finally, the sum of all amount of tasks were divided by the total number of jobs.

We observed that the maximum number of tasks per job is 20010 and the minimum is 1, with the average being close to 37.

The average number of task that compose the job is 36.83.

4 Question 3

The next interesting question to investigate was to find out what is the percentage of jobs/tasks that got killed or evicted.

For the first part of this question, we used *"Job events"* table. We filtered out the jobs with the event types different to $EVICTED(2)$ or $KILLED(7)$, then as long as same jobs may be re-executed multiple times we removed duplicates and used the function $count$ to get the total amount of evicted or killed jobs in the case of Spark DataFrame API usage and map and $reduceByKey$ in the case of RDDs. Finally, the result was divided by the total amount of jobs in the trace.

For the second part of the question we used *"Task events"* table and the same approach.

We found out that the percentage of evicted jobs is quite small with the amount of killed tasks being significantly smaller, which means that distributing tasks to different sub entities in the hierarchy (nodes comprised of jobs, jobs of tasks) improves the stability of the whole system.

evicted and killed jobs - 13.53%

evicted and killed tasks - 0.18 %

5 Question 4

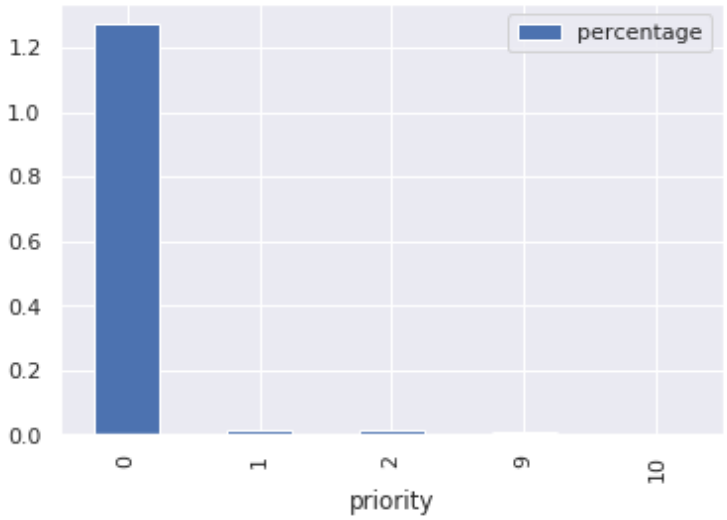
Do tasks with low priority have a higher probability of being evicted?

We used *task events* table for this task. First, we selected evicted task by using functions $where$(to find tasks with event Type = 2) in the case of Spark DataFrames and filter in the case of Spark RDDs. And encountered that use of DataFrames to perform data cleaning and pre-processing results in a significant speedup. So we converted RDD to data frame instead and cleaned and filtered the data, also selected the single column containing priorities of each unique task there. Next, we in order to demonstrate the use of RDDs we converted data frame with the single $priority$ column back and performed map and reduce operations to aggregate the counts of each distinct priority value.

We investigated that only the tasks with the priorities $0, 1, 2, 9, 10$ were evicted. With the task with the priority 0 having the higher probability of being evicted and task with higher priorities $1, 2, 9, 10$ having probabilities of being evicted close to 0. In overall, the probability of the task being evicted is very low.

The results:

$$\begin{bmatrix} priority\,0 & pr(evict) = 0.012 & 1.27\% \\ priority\,1 & pr(evict) = 0.00015 & 0.01\% \\ priority\,2 & pr(evict) = 0.00016 & 0.0166\% \\ priority\,9 & pr(evict) \sim 0 & 0.0066\% \\ priority\,10 & pr(evict) \sim 0 & 0.0003\% \end{bmatrix}$$



Thus, as we can see, tasks with lower priority have a higher probability to be evicted.

6 Question 5

Is there a relation between the priority of a task and the amount of resources available on the machine it runs on?

In order to analyze the correlation between the priority of the task and the amount of available resources in the machine the task was executed on we leveraged the records from tables machine events and task events. We used the $broadcasted$ dictionary of machine events, that provide the mapping of machieID and resources available on it. Broadcasting results in a huge performance benefit, since before running each task, Spark computes the task's closure, so this closure is shipped along with the shared variable (dictionary in our case) to each Spark node n times corresponding to the total number of partitions. With the use of $broadcast$ the dictionary will be distributed once per node using efficient $p2p$ protocol. Next we use this dictionary in the mapping step to get the amount of available resources for each task. Then, having the distribution tables of CPU, RAM, priority for each of the unique pair of taskID and jobID we compute the Pearson correlation coefficients.

We result in a quite small negative correlation coefficient of -0.031 for priority and CPU, priority and memory. From that we may conclude that priority has no correlation with the amount of resources available figure 7, figure 8. However, CPU and RAM are in a strong correlation as can be observed in the figure 6.

Fig6 Graphic of correlation between CPU and memory

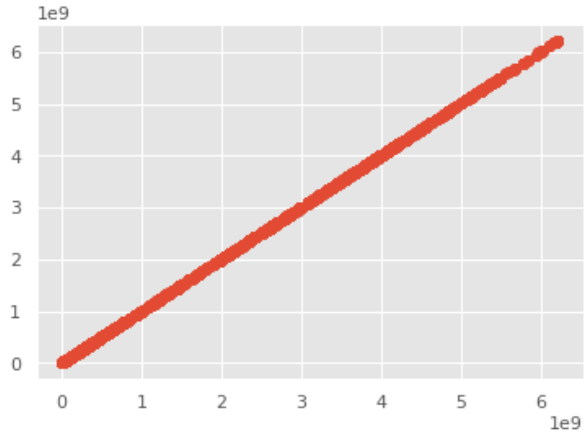


Fig7 Graphic of correlation between CPU and priority

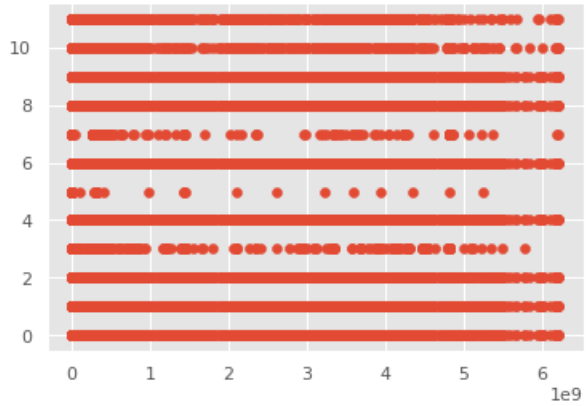
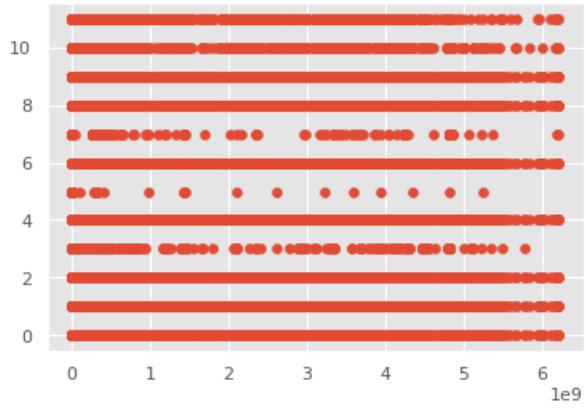


Fig8 Graphic of correlation between memory and priority



7 Question 6

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

For the question six we used *Task usage* table. First of all, we make a new data frame by selecting (function $select$) rows *'jobID', 'taskID', 'machineID', 'cpu_usage', 'mem_usage'*. After that, we created another data frame with *'jobID', 'taskID', 'machineID'*, but we selected only them with $CPUusage$ and $RAMusage$ are not equal to zero. Next step was to merge this two data frames

by *'jobID', 'taskID', 'machineID'* (function $join$). Finally, we added new columns with the values: used memory divided by request memory, used CPU dived by request CPU and the sum of these two values.

The results:

The number of machines, that consume less resources than request: 1661028

The number of machines, that consume significantly less resources than request: 1534055

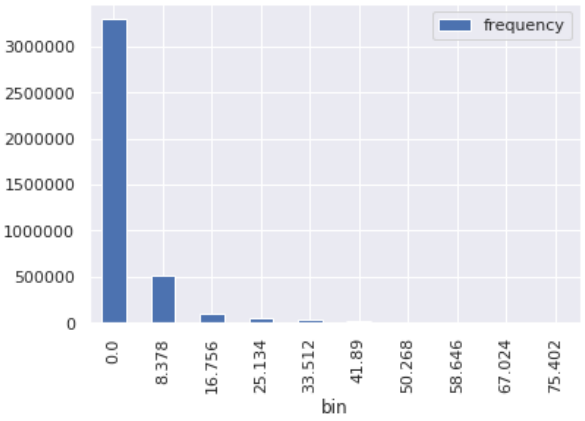
The number of machines, that consume the same amount of resources than request: 25

8 Question 7

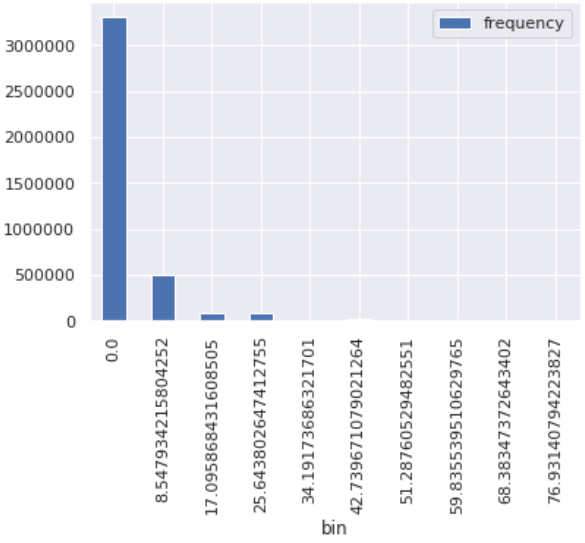
Are there machines for which the resources are largely underused?

For this question we used *Machine events* table and *Task usage* table. From the previous question we have a DataFrame with the following columns: $machineID$, $jobID$, $taskID$, cpu_usage, mem_usage, $timestamp$, $eventtype$, $platformID$, CPU, $memory$. We added new columns (by using function $withColumn$) that have information about the percentage of resources that each machine used.

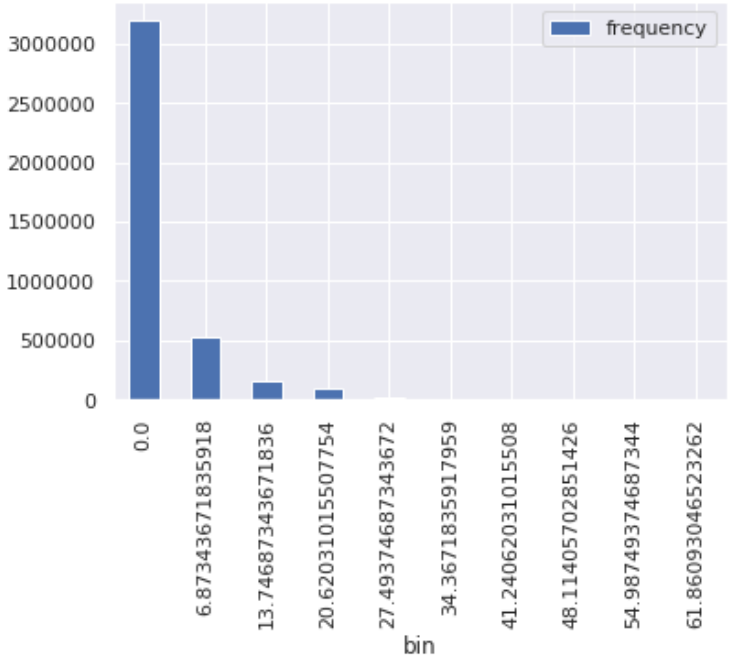
Let's look at the histogram of the percentage of CPU usage



Let's look at the histogram of the percentage of memory usage



Let's look at the histogram of the percentage of memory+CPU usage



As we can see, the majority of machines have underused resources.

9 DataFrame vs RDD

DataFrames are more efficient than RDD, as they are converted into an optimized RDD. Spark optimizes the executions of the operations applied to DataFrames. For example when we are performing group previous to filter on RDDs Spark would execute group firs and only then perform filtering resulting in a poor performance. However, in the case of Spark Data Frames the optimizer decides the order of operations itself, thus scheduling filtering prior to grouping.

Also, while performing simple grouping and aggregation operations RDD API is slower. In performing exploratory analysis, creating aggregated statistics on data, DataFrames are faster.

We executed the code for the same problem both using data frames and RDDs and measured the time taken. We discovered that while for the RDDs it takes 21 seconds, for data frames it takes only 3.5 seconds to come up with the result.

RDD

$$distinct_count = cpu_all.RDD.map(lambdax : (float(x[0]), 1)).$$

$$reduceByKey(lambdax, y : x + y).map(lambdax : list([x[0], x[1]])).collect()$$

Data Frame

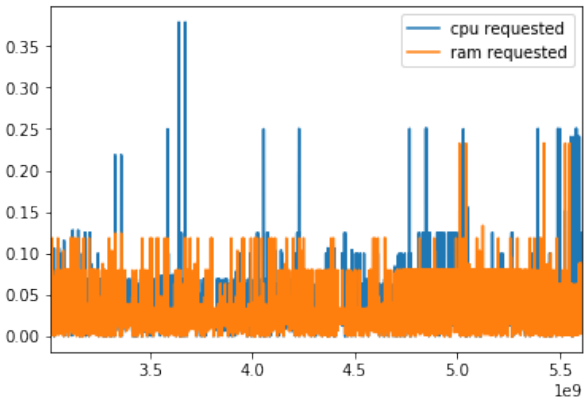
$$distinct_count = cpu_all.groupBy('CPU').count().toPandas()$$

10 Analysis

First of all, we decided to compare Spark with Pandas in terms of the speed of computation the maximum of columns. For this task, Spark was **1.9 times faster** than Pandas. This was done just to see how much Spark is faster.

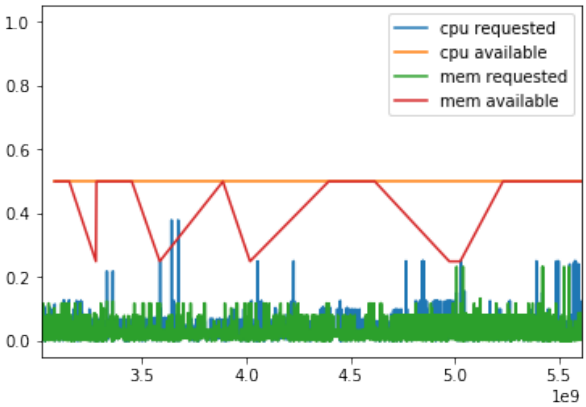
10.1 Request resources

In order to have the distribution of request resources, we used the *Task events table* and functions map for data pre-processing and $filter$ for collection of information. For plotting this information, we used $matplotlib$.



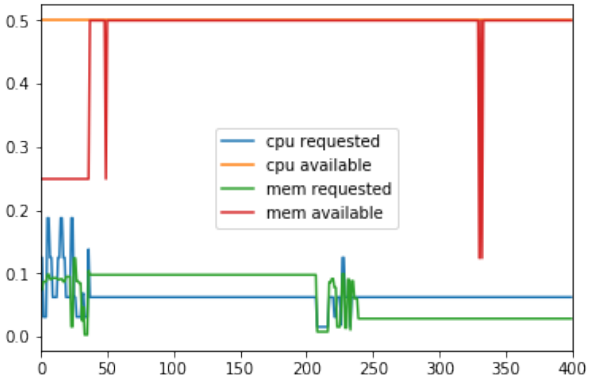
10.2 Available resources vs Request resources

To plot the graph the request resources vs available resources, we used *Machine events table*. For data pre-processing and information collecting were used the same functions as were mention before. For time stamps the broadcast function was used.



According to the graphic, there were always much more resources available than used.

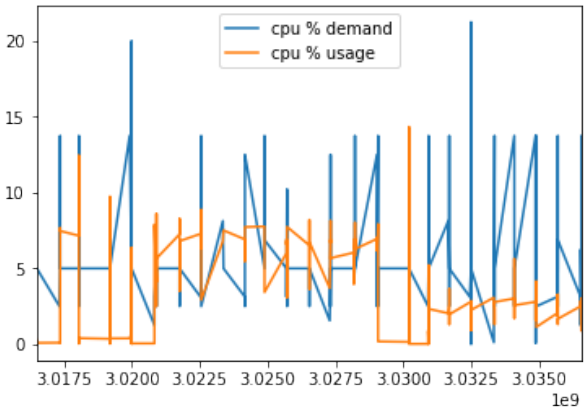
This graphic show how the distribution of available and request resources looks at the beginning of the trace window (when time stemp is equal to zero).



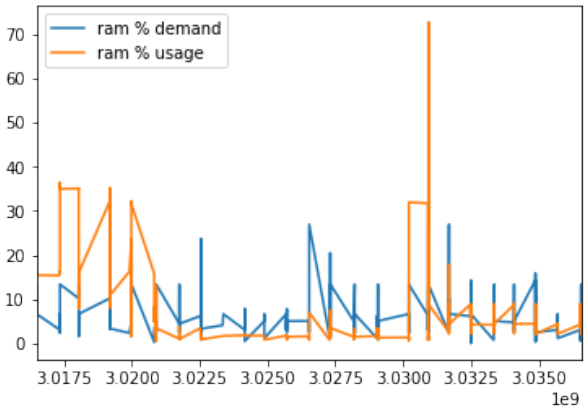
10.3 How well Google scheduler allocates resources

One of the most important question is how well Google scheduler allocates resources. In order to do it, we used *Task usage* table. We used functions map and $filter$ for reading data and avoiding overlapping.

We did two separates graphs for more clear picture.



For CPU the demand is higher than the real usage of it in the majority of the time. However, for memory the situation is different.



The density of task per machines is about 100.2 tasks per one machine. If we delete all tasks with request CPU = 0 or RAM = 0 the density would be little lower: 99.8 tasks per machine.