Lab Spark Google

Kawtar Laghdaf
Tijana Ninkovic

January 2020

1 Introduction

In this work we study data about Google machines clusters as an application of Apache Spark analysis on a large dataset. The study is about the general use of processing and storing resources for multiple jobs/tasks of the machines. Thanks to "Resilient Distributed Dataset" (RDD) abstraction, quick filtering, projections and juncture of information was done. In a second phase of the work, we study performances of Pyspark. We chose to study Spark DataFrames and compare them to RDDs.

2 Description of the conducted analysis

2.1 About data

The study was done on the data collected by Google about activities of 12500 machines during 29 days. Data is stored as CSV files in the folder `gs://clusterdata-2011-2/` that can be downloaded using GSUtil. The main folder contains the sub-folders of these tables:

- Machine events
- Machine attributes
- Job events
- Task events table
- Task constraints
- Task usage

Limited for time and resources, we only used the first file of each table in our analysis.

2.2 Our study

Our work focused on studying some aspects of the data. We mainly extracted the following information:

- The distribution of the machines according to their CPU and memory capacity.
- The percentage of computational power lost due to maintenance (a machine went offline and reconnected later).
- The distribution of the number jobs/tasks per scheduling class.
- The percentage of jobs/tasks that got killed or evicted depending on the scheduling class.
- The link between task's priority and eviction (Do tasks with low priority have a higher probability of being evicted?).

- The link between tasks from the same job and machines (Do tasks from the same job run on the same machine?).
- Are there tasks that consume significantly less resources than what they requested?

In order to extract these information, we used PySpark commands. More specifically, we read the CSV files using the function **textFile** and we applied the Spark function **map** which returns an RDD (fault-tolerant collection of elements that can be operated on in parallel [1]) from the data table. Then we do our analysis depending on what information we want to extract.

2.3 Description of the analysis

In the following section we will describe how we did each of our analysis and what we concluded from the results we got. A more detailed explanation including the code is given in a notebook we submitted.
For visualising data we used python's **matplotlib** and **seaborn** libraries.

2.3.1 Distribution of the machines according to their CPU and capacity:

Information about CPU and memory capacity of machines is stored in the **machine events** dataset. The way we found the distribution of machines according to CPU capacity is first by creating a new RDD containing only the machine ID and CPU capacity using the **map** function. The CPU capacity is normalized and divided into categories: 0.25, 0.5, 0.75 and 1 (one corresponding to the highest CPU capacity).
Therefore, what we did is count how many machines belong to each of the categories.

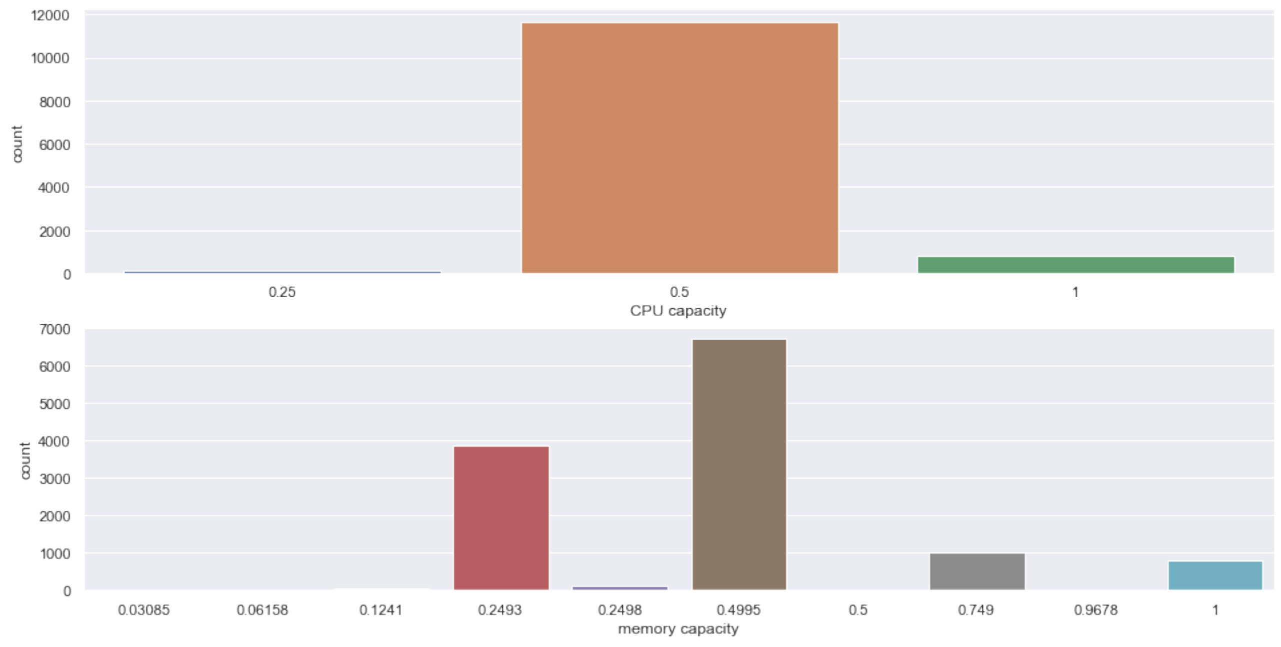
Below are the results we got:

CPU capacity	machines (%)
'0.25'	0.998
'0.5'	92.42
'1'	6.325
'Unknown'	0.256

We also look at the memory capacity, and the same way we found:

Memory capacity	machines (%)
'0.03085'	0.039
'0.06158'	0.007
'0.1241'	0.427
'0.2493'	30.629
'0.2498'	0.998
'0.4995'	53.335
'0.5'	0.023
'0.749'	7.946
'0.9678'	0.039
'1'	6.298
'Unknown'	0.253

To visualize this data we used the **countplot()** function from the **seaborn** package.



The countplot shows us the distribution of each CPU and memory capacity categories. Both of these values are normalized and take values from 0 to 1. For CPU we have different categories and we can see that the majority of machines falls into the '0.5' category. Memory capacity has more categories, but we still see that the majority of the machines fall in the middle.

2.3.2 Percentage of computational power lost due to maintenance:

When performing this analysis we used the **machine events** table, which contains information about when a machine connected or when it went off, as well as timestamps of when those events happened. What we did is create a list of events for each machine and then looked that *event type 1* which corresponds to a machine being removed and *event type 0* which corresponds to a machine being added. We use the timestamp to calculate how much time has passed between removing a machine and adding it again and then we multiply that by the CPU capacity of that machine.

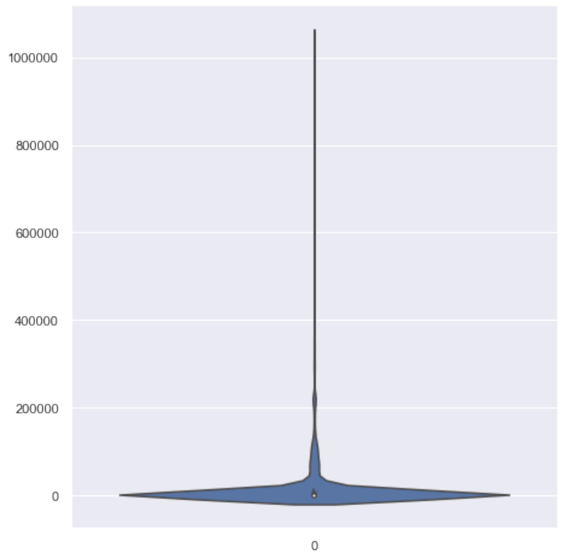
The sum of all of those values is our *lost capacity*. We also multiply the capacity of each machine with the maximum timestamp that we have in our RDD and assume that would be the *ideal capacity* if machines never went offline. Then, we divide these two numbers to get the percentage of computational power that was lost.

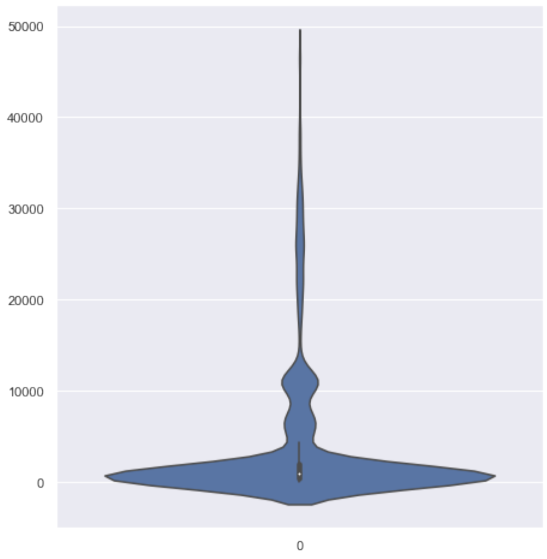
The result that we got when running this analysis was that the percentage of computational power lost due to maintenance is **4.81%**.

Another thing we looked at is the distribution of time a machine was offline for maintenance. We found this time as a difference between timestamps of when machine was removed and timestamp of when machine was added again. These timestamps represent microseconds, so we converted them to seconds.

We saved all these values in an array and when observing the results we got, we saw that there are a lot of outliers when machine was offline for significantly longer than average.

In the graph below we used a **violin plot** to represent how these times are distributed. The first one is before removing outliers. We can see that most values are concentrated around zero but because of these extreme cases where machines were offline for a long time (which aren't numerous) we don't see that distribution well. Therefore, in the second graph we represent the distribution without those values. Once again we see that the majority is concentrated around zero.





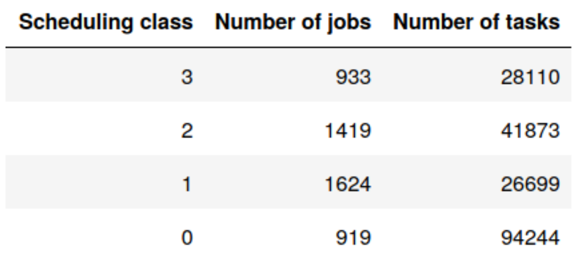
This tells us that when machines in Google cluster go offline for maintenance that time is usually not long and therefore not a lot of capacity is lost. They always came back on shortly enough that the capacity that was lost is not that significant, which corresponds to the result of 4.81% which we got above.

2.3.3 Distribution of the number jobs/tasks per scheduling class:

For this analysis we used the **task events** table, which contains information about scheduling classes for all tasks. First, we studied the distribution of jobs, by counting how many different jobs are in each scheduling class.

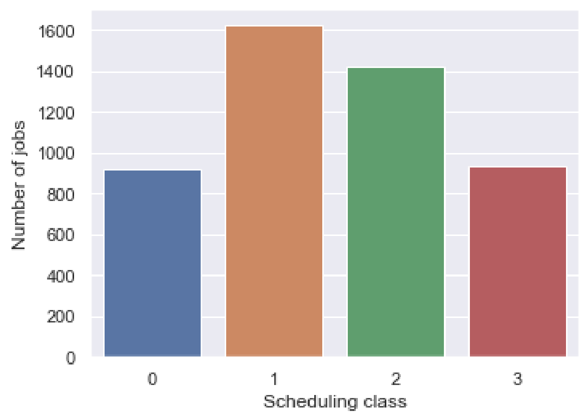
Next, we count the number of different tasks in each scheduling class, taking into account that tasks are identified by both the job ID and the task index.

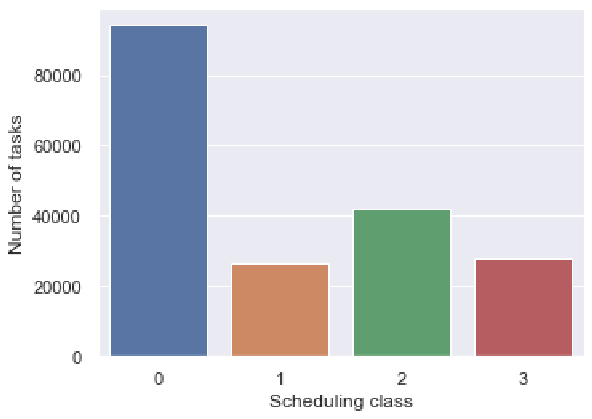
We obtained the following results:



Scheduling class	Number of jobs	Number of tasks
3	933	28110
2	1419	41873
1	1624	26699
0	919	94244

We can represent this graphically with a **barplot**





From this graph we see that, in terms of jobs, most of them belong to classes 1 and 2 and there are fewer in classes 0 and 3. However, when it comes to tasks, significantly more belong to class 0 than to other three classes.

The scheduling classes represent how latency sensitive jobs/tasks are. Class 3 represents more latency-sensitive tasks (e.g., serving revenue-generating user requests) and class 0 represents non-production tasks (e.g., development, non-business-critical analyses). Here we can see that most tasks are non-production.

One thing to note here is that some tasks change their scheduling class. What this means is that we counted some tasks several times in different classes. Considering a task just once would, in our opinion, make less sense because there is no right scheduling class to consider in that case.

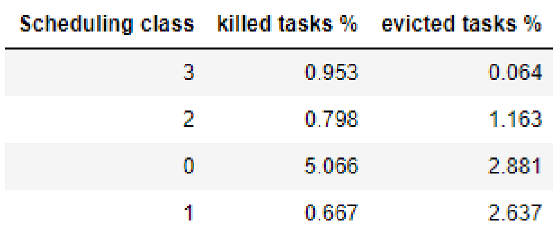
2.3.4 Percentage of jobs/tasks that got killed or evicted depending on the scheduling class

For this analysis we used **task events** again to get a list of all the events that happened on a certain task. The tasks events we were interested in are *event type 5* which corresponds to a task being killed and *event type 2* which corresponds to a task being evicted.

We counted the number of killed tasks in each scheduling class, the number of evicted tasks in each scheduling class and also the total number of tasks in each scheduling class.

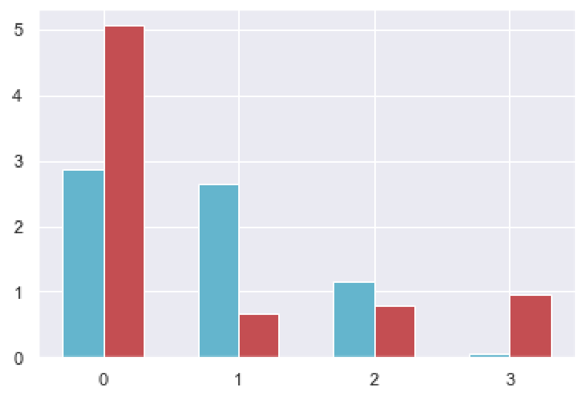
By dividing the number of killed/evicted tasks per scheduling class with the total number of tasks in that class, we got the percentages of tasks that were killed/evicted in each class.

These percentages are given in the table below:



Scheduling class	killed tasks %	evicted tasks %
3	0.953	0.064
2	0.798	1.163
0	5.066	2.881
1	0.667	2.637

Below they are represented graphically. The x-axis represents different scheduling classes and the y-axis represents the percentage of jobs that were killed or evicted in that scheduling class. Red bars are killed tasks and blue bars are evicted tasks.

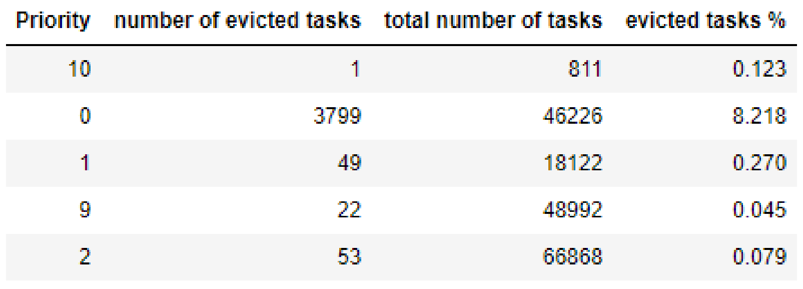


The percentage of killed tasks in scheduling class 0 is significantly bigger than in other classes. For the remaining three the number is similar, below 1 percent. For evicted tasks we can see that the number in classes 0 and 1 is the biggest and that in class 3 it is very small. We conclude that a task has the highest probability of being killed or evicted if it is a non-production task (belongs to class 0).

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

Here we used the **task events** table again because it also has information about the priority of the tasks. We did a similar analysis as before, but instead of using the scheduling class column, we are used the priority column. We again filtered the tasks for which we have *event type 2* (which means they were evicted) and then for each priority divide the number of evicted tasks of given priority with the total number of tasks of that priority. This gives us the probability of a task being evicted if it has a certain priority.

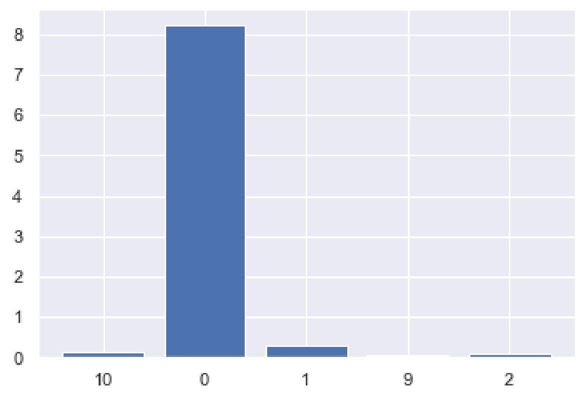
The results we obtained are given in the table below.



Priority	number of evicted tasks	total number of tasks	evicted tasks %
10	1	811	0.123
0	3799	46226	8.218
1	49	18122	0.270
9	22	48992	0.045
2	53	66868	0.079

Here we only have information about 5 priorities because we are not using all of the files and we are also filtering only the evicted tasks, so some classes get lost after this filtering.

We represent the percentage of evicted tasks by priority in the following graph:



We see that tasks with lower priority have a higher chance of being evicted, especially those with priority 0, which are significantly more likely to be evicted than those with higher priority.

2.3.6 In general, do tasks from the same job run on the same machine?

Our idea for this analysis was to count the number of *different tasks each job has*, and then also count the number of *different machines that tasks of each job run on.*

Next, we compared these two values by subtracting the number of machines per job from the number of tasks per job. If these two numbers are equal that means that all tasks of a given job run on different machines. However, if they are different, that means that we have more tasks running on the same machine.

We then did a sum of these differences and divided it by the total number of tasks. This gives us a percentage of *how often tasks of the same job were running on the same machine.*

The results we obtained by doing this computation is that **tasks of the same job run on the same machine 6.801% of the time.** This gives us information about how tasks of the same job are distributed on a Google cluster and we see that generally they **do not** run on the same machine.

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

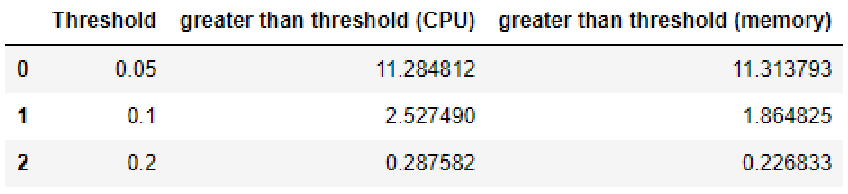
From the **task events** table we extracted the information about the requested CPU and memory for each task into separate RDDs. Since there can be multiple different values due to several measurements, we only consider the maximum value for each task.

From the **task usage** table we extracted information about the usage of CPU and memory for each task in a similar manner. Since this value was measured several times for each task (indicated by start and end event time fields), we extracted a list for each task, and we again looked for the maximum value.

Next, we used the **join** function to create a single RDD which will contain information about both requested and used CPU and after we computed the difference between these two values. We did the same thing for requested and used memory.

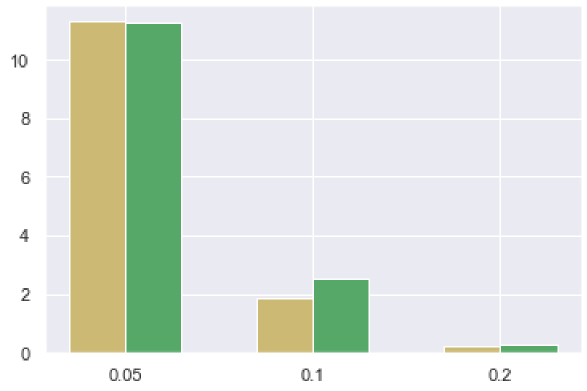
In order to compute how many tasks consume less resources than they requested, we needed to set a certain threshold. Depending on this threshold the results we get are different and since these values are normalized and sort of abstract it is hard to tell what is the right threshold to choose.

Therefore, we ran the analysis with different thresholds (**0.05**, **0.1** and **0.2**) and compared the results we got.



	Threshold	greater than threshold (CPU)	greater than threshold (memory)
0	0.05	11.284812	11.313793
1	0.1	2.527490	1.864825
2	0.2	0.287582	0.226833

The values in the tables represent the percentage of tasks that used less resources than requested with respect to a certain threshold. The numbers we are getting are almost the same for CPU and memory in each case, which is expected. It means that there is a correlation between CPU and memory usage.



From this graph we can see that if we put **0.05** as the threshold, we get that around 10% of tasks use more resources than they requested. If we increase the threshold to **0.1**, the percentage drops significantly, to about 2% of tasks. Finally, we see that only a very small number of tasks - around 0.2% - exceeds the threshold of **0.2**.

Important thing to note is that we also noticed that there were some tasks that used more resources than they requested. We did not filter them out of the analysis, since we are only counting those that used significantly less and not looking at an average difference bewteen requested and used resources. However, we are aware that they exist in our data.

3 Performance analysis

3.1 RDDs vs DataFrames in Spark

We chose to study **SQL DataFrames** and compare them to **RDDs**.

RDD is a distributed collection of data elements spread across many machines in the cluster. It is Read-only partition collection of records. RDD is the fundamental data structure of Spark. It allows a programmer to perform in-memory computations on large clusters in a fault-tolerant manner.

DataFrame is a distributed collection of data organized into named columns. It is conceptually equal to a table in a relational database. DataFrame in Spark allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction. DataFrame was built upon RDD, inspired from SQL. It allows you to deal with data by selecting columns, grouping them, etc.

Both RDDs and DataFrames use *lazy evaluation*, meaning that execution won't start until an action is triggered. They are also both *immutable*, which means that once created, they cannot be changed. The values can, however, be transformed by applying transformations.

RDDs in Apache Spark have two main limitations compared to DataFrames [2].

- First, they do not have an input **optimization engine**. Optimization should be done manually as RDDs cannot use Spark advance optimizers like **catalyst optimizer** and **Tungsten execution engine**. In the contrary, DataFrame uses Catalyst to generate optimized logical and physical query plan.
- Secondly, RDD does not provide **schema view** of data as in DataFrames, and thus has no provision for handling structured data. In DataFrames, the Schema view of data is a distributed collection of data organized into named columns, allowing higher-level abstraction.

We decided to compare through real examples Spark RDDs and DataFrames and prove those differences.

3.1.1 Comparison we made

In order to study the performance and compare it, we used SQL queries for Spark DataFrames and applied equivalent operations on RDDs.

First we looked at simple select SQL queries:

- selecting and filtering data
- sorting
- grouping

We then compared the performance on some of the algorithms that we used in the first part of the lab.

3.1.2 Results

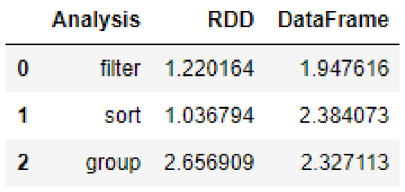
Comparison of structures:

- We first remark that DataFrames are more structured than RDDs.
- The DataFrames containt a header that appears when we display their content them.
- The function "withColumnRenamed" allowed as change the column names of the scheme and as a result have a more significant table.
- Instead of using the map function to project the data on the needed columns using index as in RDDs, we can easily use the function "select" in DataFrames directly with the names of the columns or use a SQL query.
- Extracting information from DataFrames is easier than RDDs when data has a definite scheme.

Comparison of time performance:

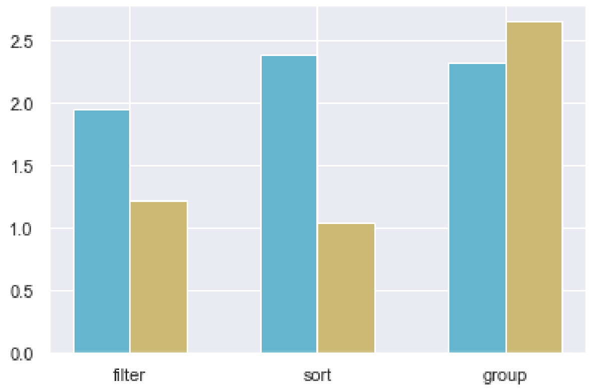
For comparing time we used a simple timer from the **timeit** Python library and measured the time it took to execute certain functions.

In the table below is the time it took for each of these simple queries, given in seconds.



	Analysis	RDD	DataFrame
0	filter	1.220164	1.947616
1	sort	1.036794	2.384073
2	group	2.656909	2.327113

We can see that in these simple functions the performance was quite similar. RDDs were overall a bit faster. We represented these results graphically below using a barplot.



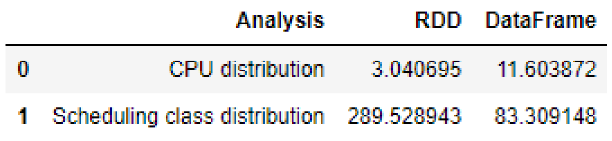
Blue bars represent the performance of Spark SQL DataFrame and yellow bars are performance of RDDs.

Next thing we did was rerunning some of the analysis we did in the first part of our work, but this time with Spark SQL and DataFrames.

These analysis were:

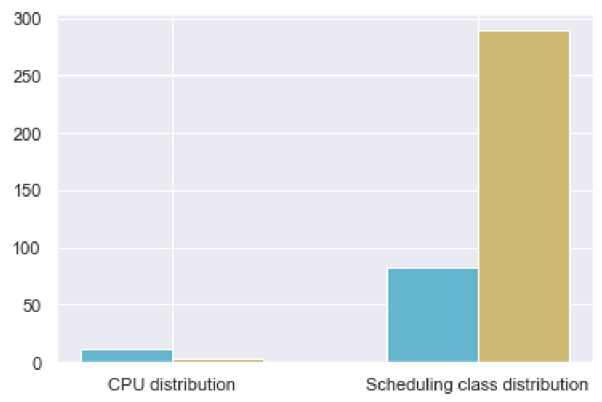
- distribution of machines according to CPU capacity
- distribution of tasks according to their scheduling class

In the table below we can see how they performed.



	Analysis	RDD	DataFrame
0	CPU distribution	3.040695	11.603872
1	Scheduling class distribution	289.528943	83.309148

For CPU distribution we were using the Machine events table which is very small. On that analysis, RDDs outperformed DataFrames. However, for the scheduling class distribution we were using the Task events table which has a lot more files and here DataFrames and Spark SQL significantly ourperformed RDDs.



Important thing to note is that the execution for both was quite long, but that is largely due to the limitations of the machine these jobs were run on. Only 50/500 files from the table were used for the analysis. Originally we were planning to use all files and get a more realistic picture, but since the computations were taking very long and, in case of bigger files, couldn't be performed at all, we chose to do them like this, which was still enough to illustrate the difference in performance.

4 Conclusion

In this work we studied a Google cluster usage trace dataset. What our analysis reveals is that machines in a Google cluster are mostly homogeneous - 92% of them have the same CPU capacity (0.5) and 83% of them have one of the two memory capacities (0.5 or 0.25). We also find that machines often go offline for maintenance but that period is never long and doesn't result in a lot of lost capacity.

We looked at the distribution of jobs and tasks according to their scheduling class and observed that most jobs belong to scheduling classes 1 and 2, but there are the most tasks in scheduling class 0. The scheduling class of a task or job does, however, change. Next, we find that scheduling class and priority have impact on whether a task will be killed or evicted. The highest probability of being evicted is for jobs of scheduling class 0 and priority 0, which is expected. We also saw that, generally, tasks of the same job are distributed over different machines in the cluster and they do not run on the same one.

When comparing RDDs with Spark SQL DataFrames, we found that DataFrames are better for structured data, make data manipulation easier and also perform better on larger datasets.

The Google dataset is large and contains a lot of information, which we only analyzed a small part of. This work could be extended by trying to cluster jobs or tasks depending on their duration, scheduling class and failure probability. It could also be extended by comparing RDDs and DataFrames on more complex algorithms.

References

[1] Apache spark rdd. `https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm`.

[2] Apache spark rdd vs dataframe vs dataset. `https://data-flair.training/blogs/apache-spark-rdd-vs-dataframe-vs-dataset/`.