DIT065 - Computational Techniques for Large-scale Data Assignment 4

Huw Fryer
Cem Mert Dalli
Stefano Ribes

May 3, 2022

1 Problem 1

Our implementation of the k-means algorithm using MRJob assumes two input file: a file containing the data points, one point per line, and a centroids file, in which each line reports a cluster center.

The `mapper` method computes the closest centroid for each of the data points, *i.e.* file lines. To do so, a `mapper_init` method is responsible for reading from a file the centroid values at the current iteration step. After that, it initializes a class member containing the centroid numerical values. To the best of our knowledge and understanding, the `mapper_init` method is called only once *before* starting the mapper. According to this assumption, this might mean that the centroid files doesn't need to be opened by the mappers nor by the tasks again, as the centroid array member should be eventually cached locally on the nodes.

The theoretical maximum speedup for the mapper is n, where n equals the number of points in the dataset. If we assume to utilize an HDFS, the data file would be stored as a sequence of blocks on multiple Datanodes. MRJob might therefore take advantage of that and exploit the Hadoop functionalities to map the tasks on several nodes.

On the other hand, the `reducer` does not require the old centroid values and expects tuples in the form of: `<centroid, list of data belonging to it>`. Because of that, its maximum theoretical speedup is equal to the number of centroids.

We believe that running a MRJob implementation of the k-means algorithm would be advisable in contexts where the data file to process and the number of clusters are both significantly high. In fact, the complex scheduling of tasks of the map-reduce paradigm might become a significant overhead in cases where the data points are too few, effectively causing slowdowns.

2 Problem 2

The problem was implemented in Pyspark, the overall speedup may be seen in Figure 1.

The code for problem 2a is included in file `problem2a.py`. Table 1 reports the descriptive statistics of the files with 1, 10, 100 million and 1 billion rows. The output for the graph is included in Figure **??**. Between 1 and 16 cores, for generating descriptive statistics in files with 1 million and 10 million rows, and the descriptive statistics obtained for each file is displayed in Table 1.

	1 million rows	**10 million rows**	**100 million rows**	**1 billion rows**
Min	3.141593	3.141593	3.141593	3.141593
Max	7.141593	7.141593	7.141593	7.141593
Mean	5.141801	5.141536	5.141704	5.141608
Std dev.	1.155144	1.154485	1.154656	1.154704
Bin 0 count	99919	999549	9999637	99996208
Bin 1 count	100115	999724	9996417	99997931
Bin 2 count	99842	999833	9996174	100009847
Bin 3 count	99955	999720	9998175	99988033
Bin 4 count	99959	1001143	10004386	100010791
Bin 5 count	99939	1001608	10003418	99998690
Bin 6 count	100055	999454	9999031	99985568
Bin 7 count	99822	999483	10004021	100006030
Bin 8 count	100041	1000164	9997405	100000631
Bin 9 count	100353	999322	10001336	100006271

Table 1: Descriptive statistics for one and 10 million rows files.

The following are the boundaries of the 10 bins for the 1M rows file and do not significantly differ from the 10M, 100M and 1B files:

3.14, 3.54, 3.94, 4.34, 4.74, 5.14, 5.54, 5.94, 6.34, 6.74, 7.14.

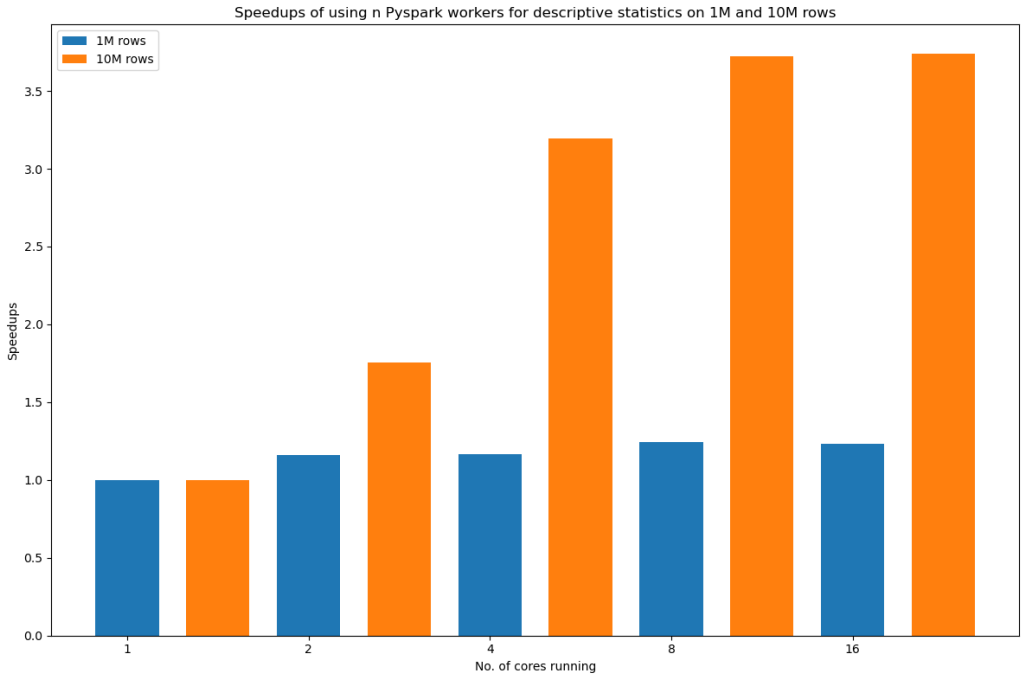

Figure 1: The speedups of the algorithm running on 1M and 10M rows data. The x-axis shows the running number of cores.

Median values in data files having 1, 10, 100 million and 1 billion rows are presented in Table 2.

Data size	**Median**
1 million	5.142466
10 million	5.141608
100 million	5.141801
1 billion	5.141582

Table 2: Median values and execution time for the provided data files.