Università di Pisa

Cloud Computing project:
Letter Frequency Analysis with Hadoop MapReduce

Fabio Piras, Giacomo Volpi, Guillaume Quint

Academic Year: 2023/2024

1 Introduction

The goal of this project was to analyze the frequency of letters in a text file using the Hadoop MapReduce framework installed in fully-distributed mode on a cluster of virtual machines. The results of the analysis were then confronted with other solutions such as:

- (Pure) Python
- Spark
- C

Both pure Python and C solutions were tested on a single machine, while Spark was tested on a cluster of virtual machines. The programs were tested on files of different sizes, starting from 1MB up to 6GB. We used 3 different datasets, for files from 1 to 654 MB we used Project Gutenberg[1], the 1 GB and 6GB files were obtained from `openssl rand` (part of the OpenSSL library[2]). The 3GB file is obtained from the most recent Italian Wikipedia[3] dump converted into text format. The results were then compared in terms of execution time, in order to perform a statistically valid analysis we have automated the launch of the programs multiple times during the day in order to mitigate the bias introduced by the load on the cluster.

2 Hadoop MapReduce implementation

2 .1 2 Job combiner

In this version we use two MapReduce jobs: the first counts the total number of letters in the document, the second one calculates letter frequency, using the output of the previous job to divide the number of occurrences of the letter by the total.

2 .2 In-Mapper Combiner

We then developed the version with only one MapReduce job with In-Mapper Combiner strategy. We use an HashMap to count the occurrence of every letter, and we use the # symbol as a key to keep track of the total letters count.

2 .3 Optimized In-Mapper Combiner

A tiny optimization that we adopted on the previous version is the use of EnumMap instead of HashMap, since the key-space is known a-priori at launch of the application. In addition EnumMap uses array internally and did not need to build an hash table.

3 Pure Python implementation

3 .1 Simple file read

This solution was the simplest to implement, it reads the file and then uses a dictionary for letter-count mapping. Although simple, it is not very efficient as it is not parallelized. In addition we are limited by the memory available on the system.

3 .2 Memory mapping solution

In order to handle big files with low amount of memory we developed a solution that uses `mmap` to map the file in the virtual memory of the process.

[1]`https://www.gutenberg.org`
[2]`https://www.openssl.org/`
[3]`https://dumps.wikimedia.org/itwiki/20240601/`

4 C

Two main solutions were implemented in C, the first one is a multi-threaded application and it utilizes memory-mapped files for efficient file I/O and threads for parallel processing. The second solution tries to be an improvement to the first with moderate success: by moving the check for ascii characters from the threaded worker code to the main thread, we reduced the chances of branch-misprediction thus increasing the throughput. The C solutions are naturally more efficient and faster than the python ones, however the algorithms are still I/O bounded causing long waiting time for large files handling.

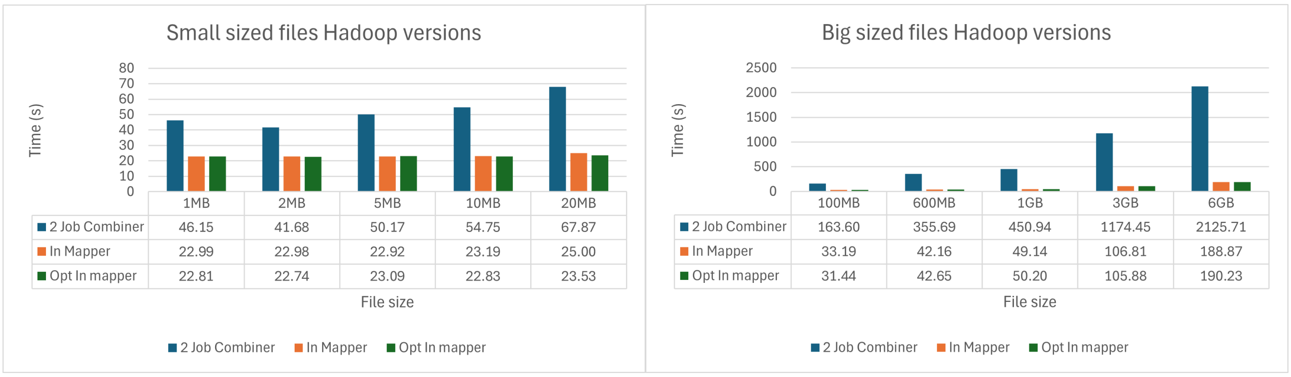
5 Spark

Additionally a Spark distributed solution was developed, ideally this provided us with the benefit of a cloud based solution, but instead of relying only on the disk like Java Hadoop, the usage of RAM should improve performances. This simple solution only uses transform functions like `flatMap` for the initial file splitting, `map` and `reduceByKey`, and as actions the `count` function to retrieve the total number of characters and `collect` to finalize the result computation.

6 Results

As shown in Fig. 1 the optimized solution with *In-Mapper Combiner* is the fastest, followed by the non-optimized one and finally by the *2JobMapper*. This is the expected result since the *In-Mapper Combiner* is a more efficient solution than using two map-reduce jobs. On smaller files the optimization of the EnumMap gives us a little advantage, but on larger files the difference is less noticeable. We can see that increasing the number of reducers does not give us any advantage, in fact it increases the execution time as shown in Fig. 2. This is due to the fact that the reducers are handling a very light task, and the additional overhead of managing more reducers is not compensated by the work done by them. The letter frequency is shown in Fig. 3. We can see that the frequency of the letters is consistent with the expected values[4]. The execution time of the Hadoop implementation is compared with the pure Python implementation in Fig. 4. We can see that the in regards of smaller files the python implementation is faster, but as the file size increases the Hadoop implementation becomes more and more efficient. The execution time of the Hadoop implementation is then compared with the C one in Fig. 5. We can see that the C implementation is faster than the Hadoop one, however the C implementation is not able to handle large files due to the memory limitation while Hadoop, thanks to the distributed nature of the system, can overall handle files with bigger sizes. Additionally to improve the performance of the C implementation we would need to scale vertically meanwhile the Hadoop one can be scaled horizontally. To further comment on the non-distributed implementations, given that both the C and pure Python solutions use the mmap feature, the former takes more advantage of it meanwhile the for the latter it seems to have no differences. This can be seen in Fig. 6, where the overall times for the C solution are almost null, apart for very large files that do not fit in the available RAM, when the file is already in memory, meanwhile the Python presents no particular differences. Finally the Hadoop implementation is compared with the Spark one, we can see that the Spark implementation is slower than the Hadoop one in Fig. 7. For smaller files this can be attributed to a bigger general overhead, but for larger files it should not scale so bad. Further investigation showed that each computational block of the Spark job takes a lot of CPU time (100 seconds each for a file of 6Gb) as documented in Fig. 8. Additionally the hadoop dashboard showed us (Fig. 9 and Fig. 10 respectably) that the Java Hadoop resource manager tries to maximize the number VCPU meanwhile Spark is stuck at 2, this causes CPU bottleneck and the result is longer completion times.

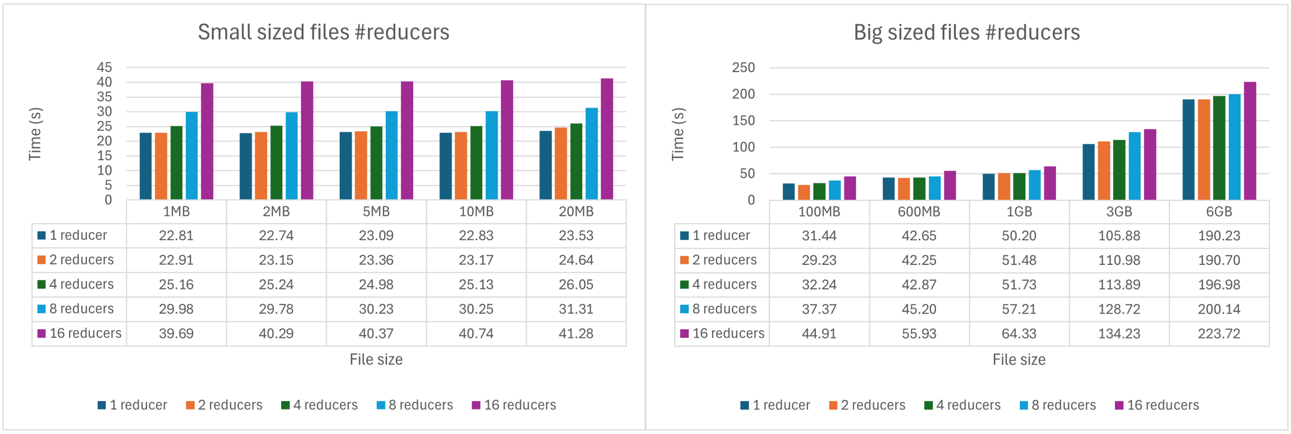
[4]`https://en.wikipedia.org/wiki/Letter_frequency`



	1MB	2MB	5MB	10MB	20MB
🟦 2 Job Combiner	46.15	41.68	50.17	54.75	67.87
🟧 In Mapper	22.99	22.98	22.92	23.19	25.00
🟩 Opt In mapper	22.81	22.74	23.09	22.83	23.53

	100MB	600MB	1GB	3GB	6GB
🟦 2 Job Combiner	163.60	355.69	450.94	1174.45	2125.71
🟧 In Mapper	33.19	42.16	49.14	106.81	188.87
🟩 Opt In mapper	31.44	42.65	50.20	105.88	190.23

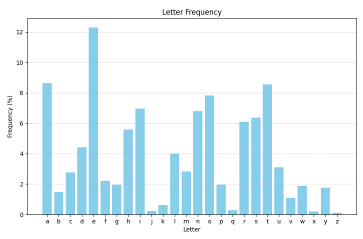
Figure 1: Time results for Hadoop implementations



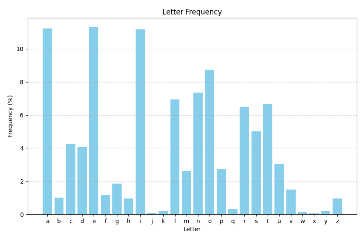
	1MB	2MB	5MB	10MB	20MB
🟦 1 reducer	22.81	22.74	23.09	22.83	23.53
🟧 2 reducers	22.91	23.15	23.36	23.17	24.64
🟩 4 reducers	25.16	25.24	24.98	25.13	26.05
🟦 8 reducers	29.98	29.78	30.23	30.25	31.31
🟪 16 reducers	39.69	40.29	40.37	40.74	41.28

	100MB	600MB	1GB	3GB	6GB
🟦 1 reducer	31.44	42.65	50.20	105.88	190.23
🟧 2 reducers	29.23	42.25	51.48	110.98	190.70
🟩 4 reducers	32.24	42.87	51.73	113.89	196.98
🟦 8 reducers	37.37	45.20	57.21	128.72	200.14
🟪 16 reducers	44.91	55.93	64.33	134.23	223.72

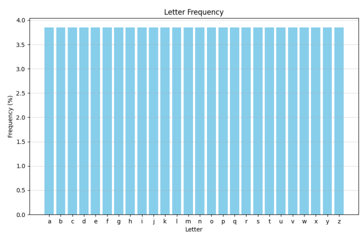
Figure 2: Time results varying number of reducers



(a) English

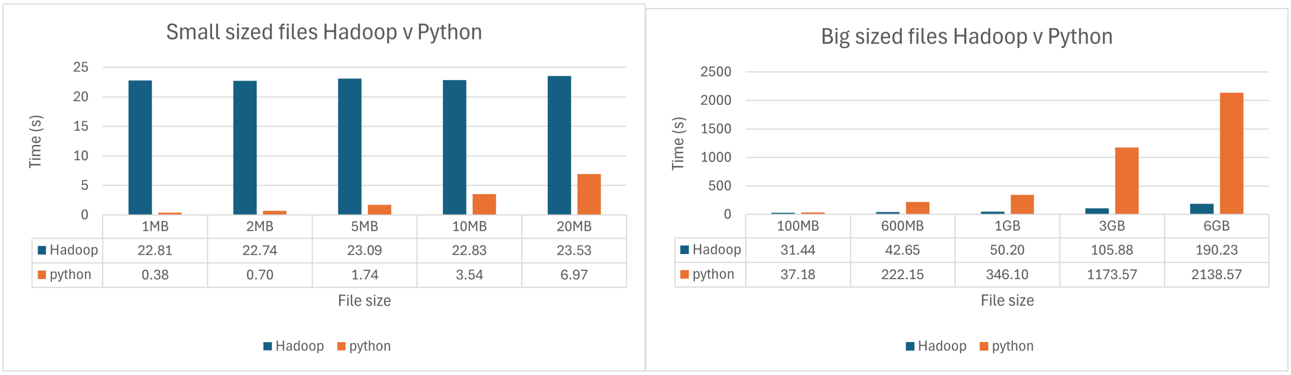


(b) Italian



(c) openssl rand

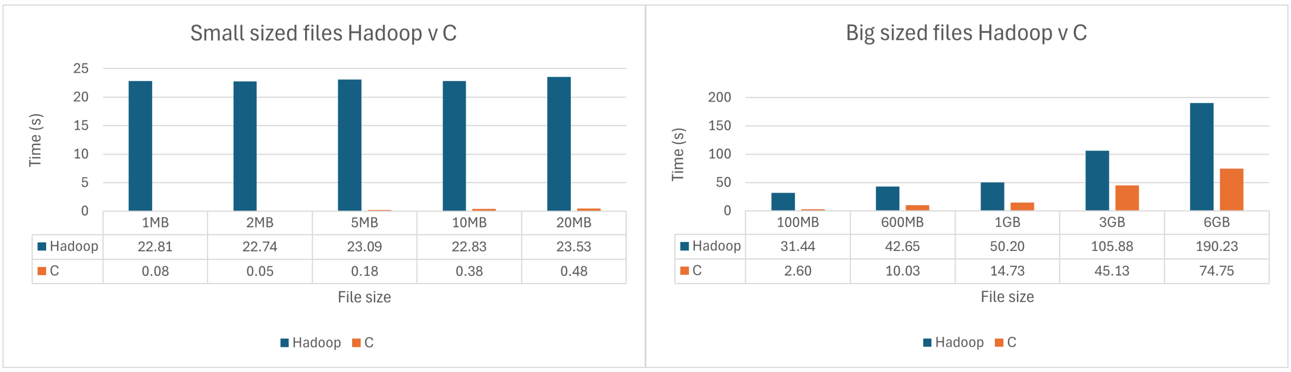
Figure 3: Letter frequency



	1MB	2MB	5MB	10MB	20MB
🟦 Hadoop	22.81	22.74	23.09	22.83	23.53
🟧 python	0.38	0.70	1.74	3.54	6.97

	100MB	600MB	1GB	3GB	6GB
🟦 Hadoop	31.44	42.65	50.20	105.88	190.23
🟧 python	37.18	222.15	346.10	1173.57	2138.57

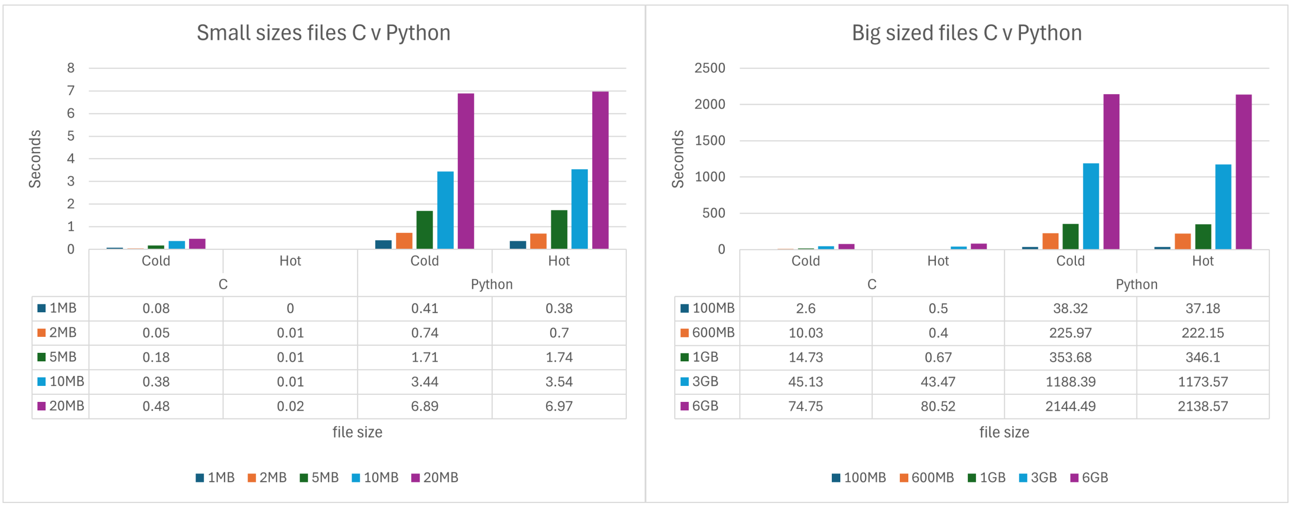
Figure 4: Time results for Hadoop and Python implementations



	1MB	2MB	5MB	10MB	20MB
Hadoop	22.81	22.74	23.09	22.83	23.53
C	0.08	0.05	0.18	0.38	0.48

	100MB	600MB	1GB	3GB	6GB
Hadoop	31.44	42.65	50.20	105.88	190.23
C	2.60	10.03	14.73	45.13	74.75

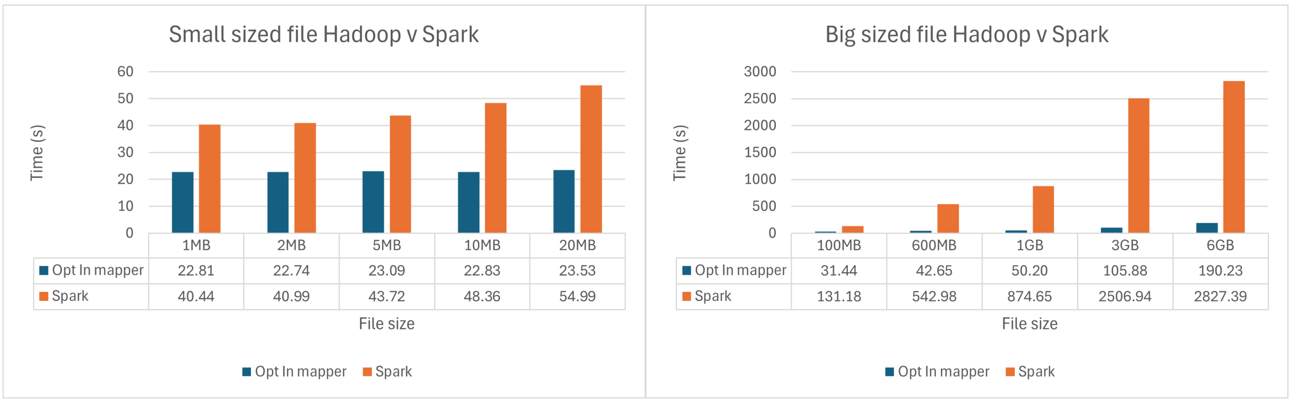
Figure 5: Time results for Hadoop and C implementations



	C		Python	
	Cold	Hot	Cold	Hot
1MB	0.08	0	0.41	0.38
2MB	0.05	0.01	0.74	0.7
5MB	0.18	0.01	1.71	1.74
10MB	0.38	0.01	3.44	3.54
20MB	0.48	0.02	6.89	6.97

	C		Python	
	Cold	Hot	Cold	Hot
100MB	2.6	0.5	38.32	37.18
600MB	10.03	0.4	225.97	222.15
1GB	14.73	0.67	353.68	346.1
3GB	45.13	43.47	1188.39	1173.57
6GB	74.75	80.52	2144.49	2138.57

Figure 6: Time results for C and Python with memory bias



	1MB	2MB	5MB	10MB	20MB
Opt In mapper	22.81	22.74	23.09	22.83	23.53
Spark	40.44	40.99	43.72	48.36	54.99

	100MB	600MB	1GB	3GB	6GB
Opt In mapper	31.44	42.65	50.20	105.88	190.23
Spark	131.18	542.98	874.65	2506.94	2827.39

Figure 7: Time results for Hadoop and Spark implementations

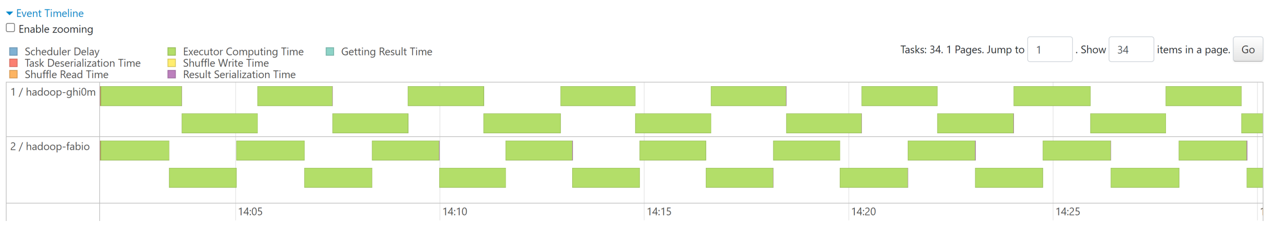
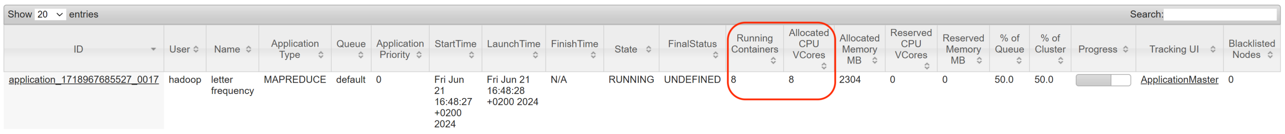
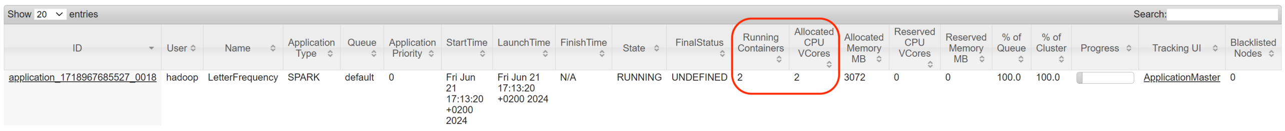


Figure 8: Spark computational blocks


Show 20 entries Search:

ID	User	Name	Application Type	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	Reserved CPU VCores	Reserved Memory MB	% of Queue	% of Cluster	Progress	Tracking UI	Blacklisted Nodes
application_1718967685527_0017	hadoop	letter frequency	MAPREDUCE	default	0	Fri Jun 21 16:48:27 +0200 2024	Fri Jun 21 16:48:28 +0200 2024	N/A	RUNNING	UNDEFINED	8	8	2304	0	0	50.0	50.0		ApplicationMaster	0

Figure 9: Hadoop VCPU usage


Show 20 entries Search:

ID	User	Name	Application Type	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	Reserved CPU VCores	Reserved Memory MB	% of Queue	% of Cluster	Progress	Tracking UI	Blacklisted Nodes
application_1718967685527_0018	hadoop	LetterFrequency	SPARK	default	0	Fri Jun 21 17:13:20 +0200 2024	Fri Jun 21 17:13:20 +0200 2024	N/A	RUNNING	UNDEFINED	2	2	3072	0	0	100.0	100.0		ApplicationMaster	0

Figure 10: Spark VCPU usage