Academic Year: 2023/2024

Cloud Computing project
Letter frequency

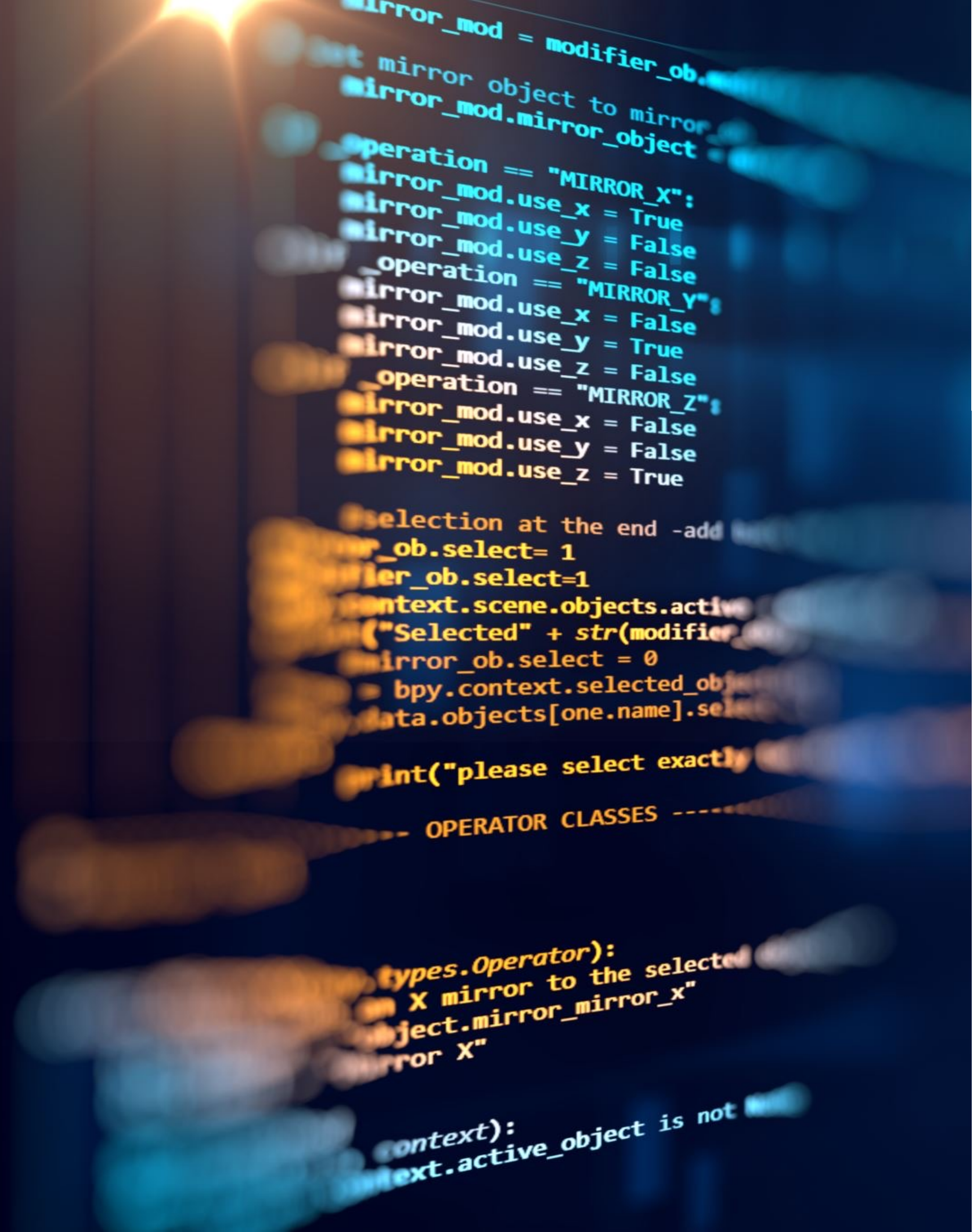
Fabio Piras

Giacomo Volpi

Guillaume Quint

(Sam Sepiol)

UNIVERSITÀ DI PISA



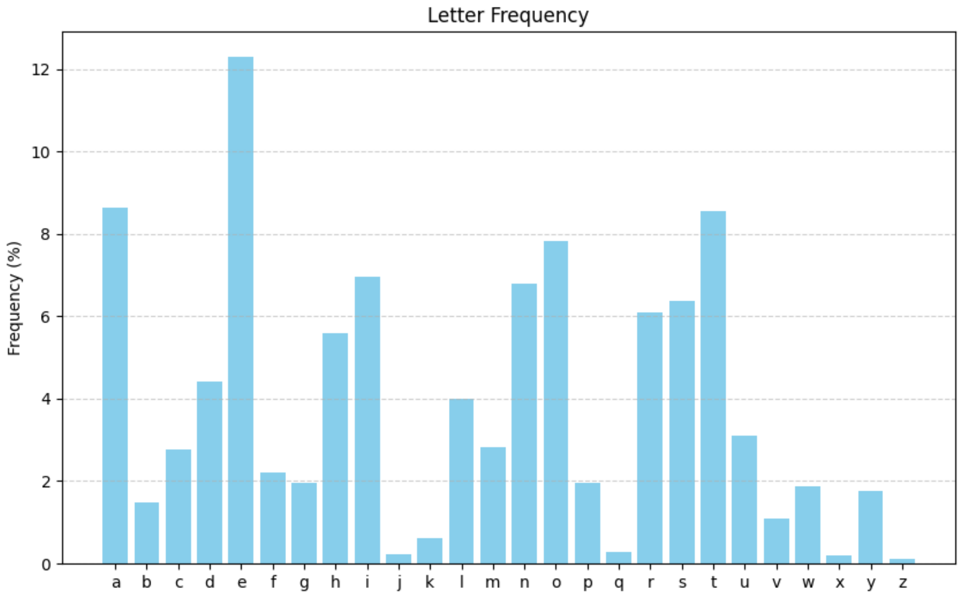
Problem and implementations

- The goal of this project was to analyze the frequency of letters in a text file using the Hadoop MapReduce.
- We implement 2 major versions of Hadoop MapReduce
- We also developed a pure Python, C and Spark implementation

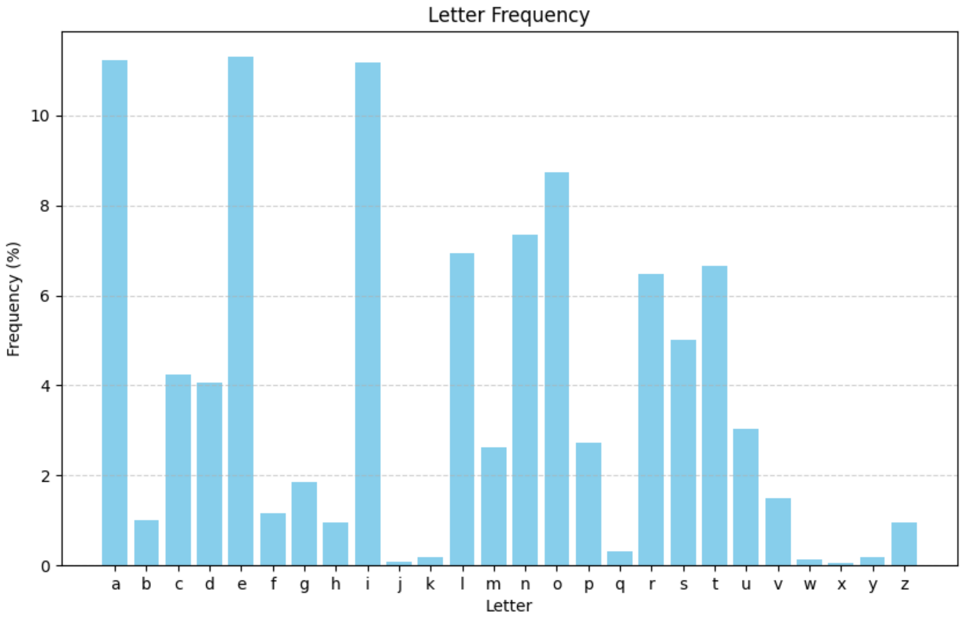
Dataset

Name	Language	Size
Project Gutenberg	English	1,2,5,10,20,50,100,654 MB
Wikipedia early June 2024 dump	Italian	3.63 GB
Base64 of OpenSSL random bytes	-	1GB and 6GB

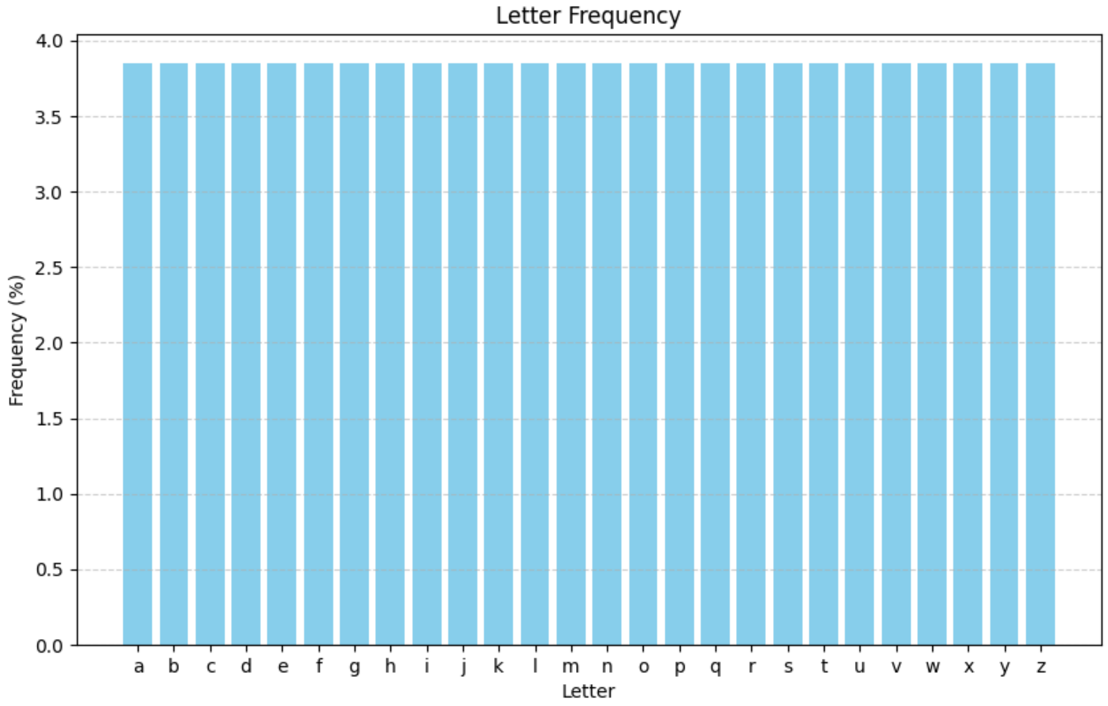
Frequency results



English



Italian



OpenSSL Rand

Hadoop MapReduce First implementation

2 MapReduce Jobs, the first counts the total number of letters in the document, the second counts the occurrence of each letter and divides it with the output of the first job

```
function Mapper1(documentFragment):
    for each character in documentFragment:
        if character is a letter:
            emit ("total", 1)

function Reducer1(key, values) :
    sum = 0
    for each value in values:
        sum += value
    emit (key, sum)

function Mapper2 (documentFragment):
    for each character in documentFragment:
        if character is a letter:
            emit (character.lower(), 1)

function Reducer2 (letter, letter_count) :
    total_letters = conf["total"] // This is the output from Reducer 1
    sum =0
    for each value in letter_count:
        sum += value
    emit (letter, sum/total_letters)
```

Hadoop MapReduce Second implementation

In-Mapper combiner strategy. Only one job, use an HashMap to collect the occurrence of each letter

```
class MapperWithCombiner:
    hashmap = {}

    function setup():
        hashmap.clear()

    function map(documentFragment):
        for each character in documentFragment:
            if character is a letter:
                char_lower = character.lower()
                if char_lower in hashmap:
                    hashmap[char_lower] += 1
                else:
                    hashmap[char_lower] = 1
                hashmap['#'] +=1

    function cleanup():
        for each key, value in hashmap.items():
            emit(key, [value, hashmap['#']]) // Emit letter count and total count

class Reducer
    function reduce(letter, values):
        if letter is '#':
            return
        sum = 0
        sumTotal = 0
        for each count,total in values:
            sum += count
            sumTotal += total
        emit(key, (sum/sumTotal))

```

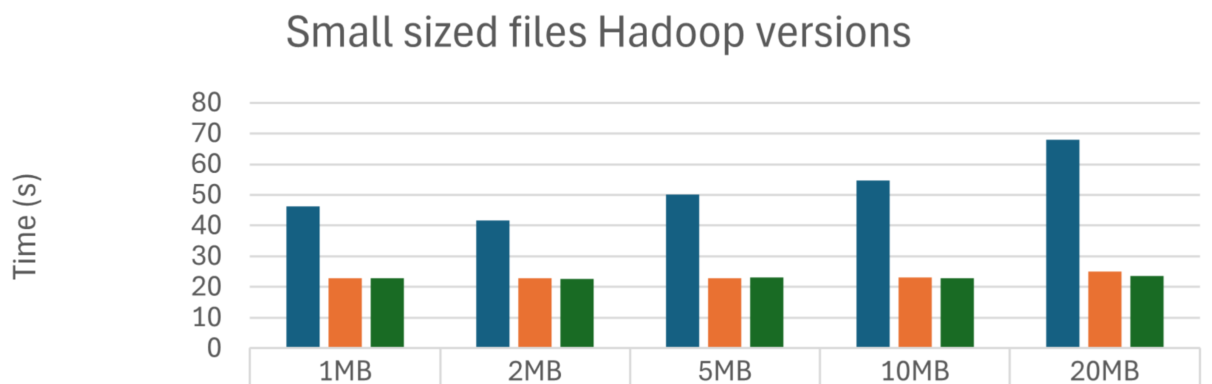
Hadoop MapReduce Optimizing Second implementation

Use EnumMap instead of HashMap since keys are known a-priori

```
class MapperWithCombiner:
    LetterEnum = a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z
    enumMap = EnumMap<Letter,Long>
    totalLetters=0

    function setup():
        for each letter in LetterEnum
            enumMap.set(letter,0)

    function map(documentFragment):
        for each character in documentFragment:
            if character is a letter:
                char_lower = character.lower()
                if char_lower in enumMap:
                    enumMap[char_lower] += 1
                else:
                    enumMap[char_lower] = 1
                totalLetters++

    function cleanup():
        for each key, value in enumMap.items():
            emit(key, [value, totalLetters]) // Emit letter count and total count

class Reducer
    function reduce(letter, values):
        sum = 0
        sumTotal = 0
        for each count,total in values:
            sum += count
            sumTotal += total
        emit(key, (sum/sumTotal))
```

Times

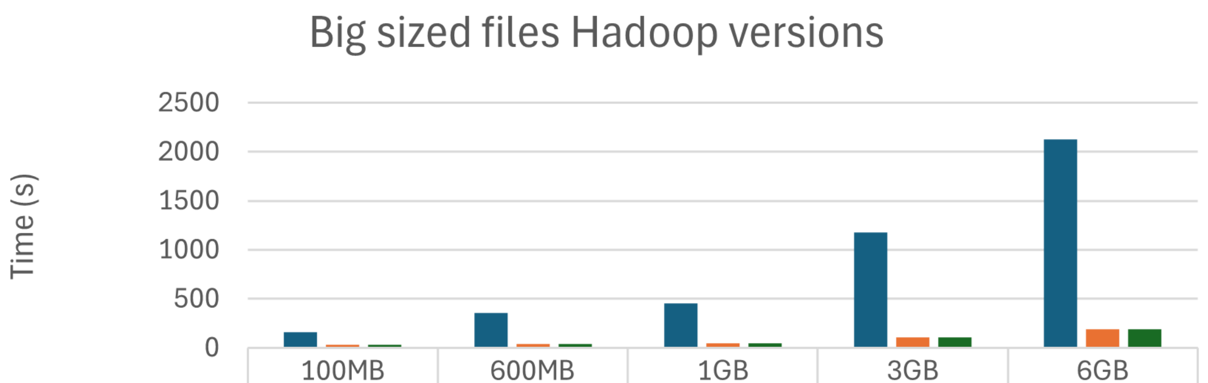
- In order to obtain statistically valid results, we automated the launch of programs multiple times during the day



	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

File size

2 Job Combiner In Mapper Opt In mapper



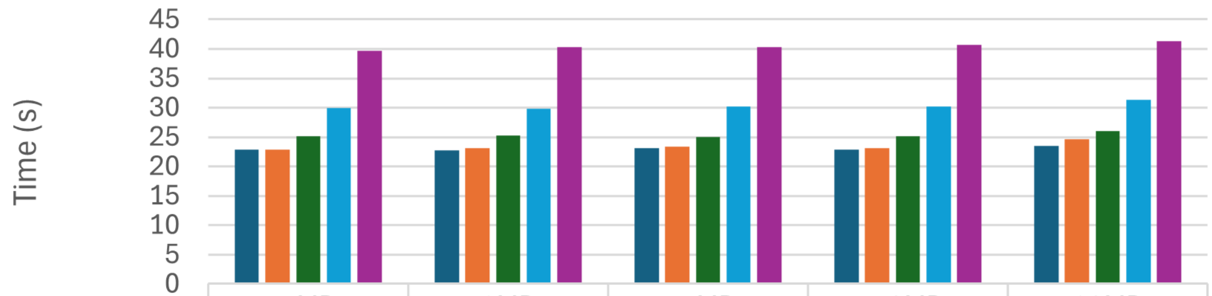
	100MB	600MB	1GB	3GB	6GB
2 Job Combiner	163.60	355.69	450.94	1174.45	2125.71
In Mapper	33.19	42.16	49.14	106.81	188.87
Opt In mapper	31.44	42.65	50.20	105.88	190.23

File size

2 Job Combiner In Mapper Opt In mapper

Increasing the number of reducers

Small sized files #reducers

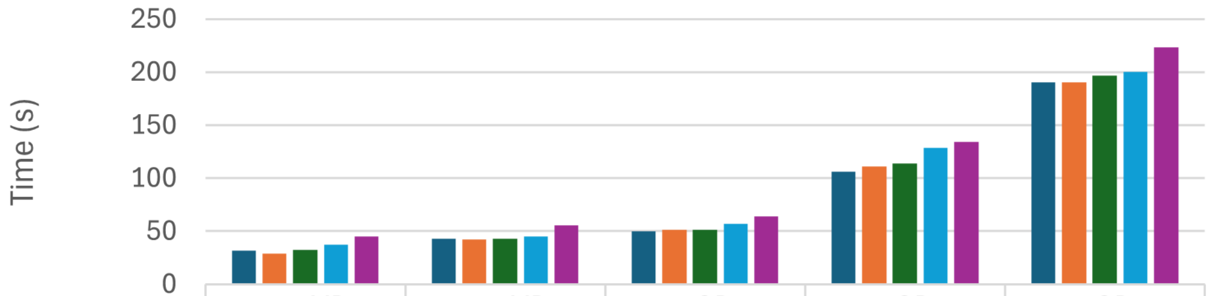


	1MB	2MB	5MB	10MB	20MB
1 reducer	22.81	22.74	23.09	22.83	23.53
2 reducers	22.91	23.15	23.36	23.17	24.64
4 reducers	25.16	25.24	24.98	25.13	26.05
8 reducers	29.98	29.78	30.23	30.25	31.31
16 reducers	39.69	40.29	40.37	40.74	41.28

File size

1 reducer 2 reducers 4 reducers 8 reducers 16 reducers

Big sized files #reducers



	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

File size

1 reducer 2 reducers 4 reducers 8 reducers 16 reducers

Alternative solutions

Python script: on a single machine

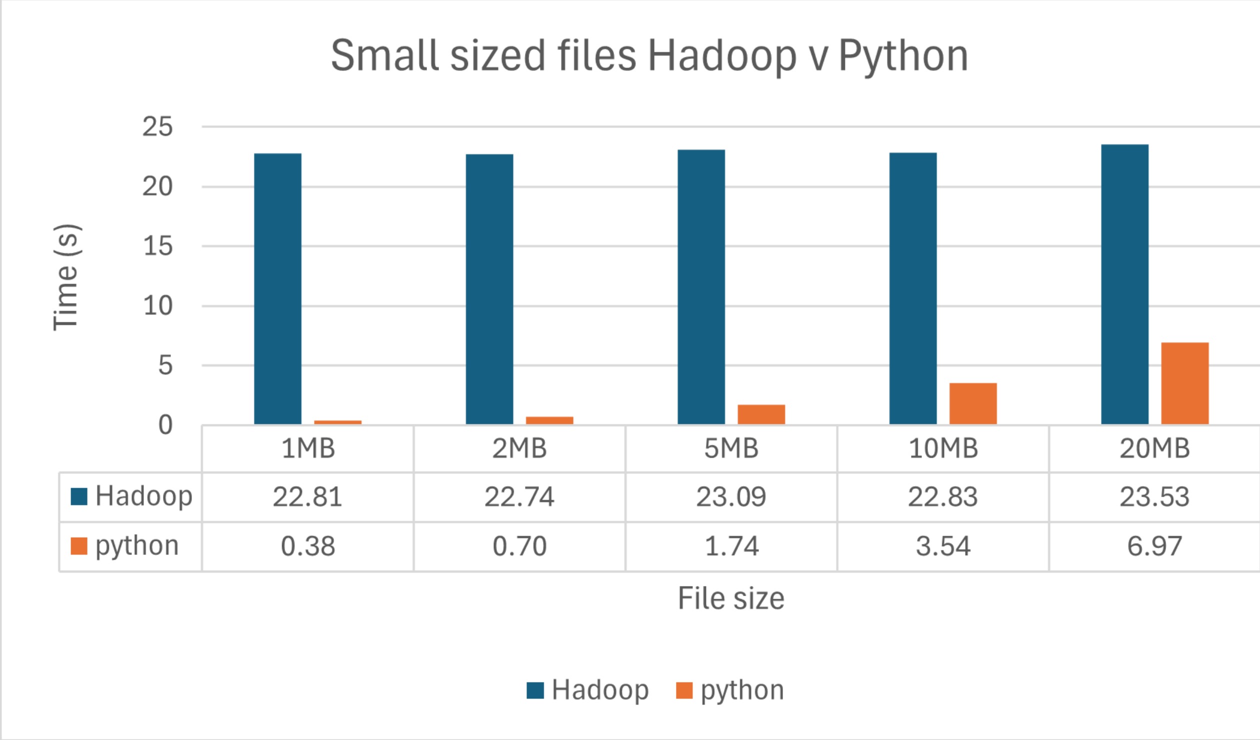
C program: on a single machine

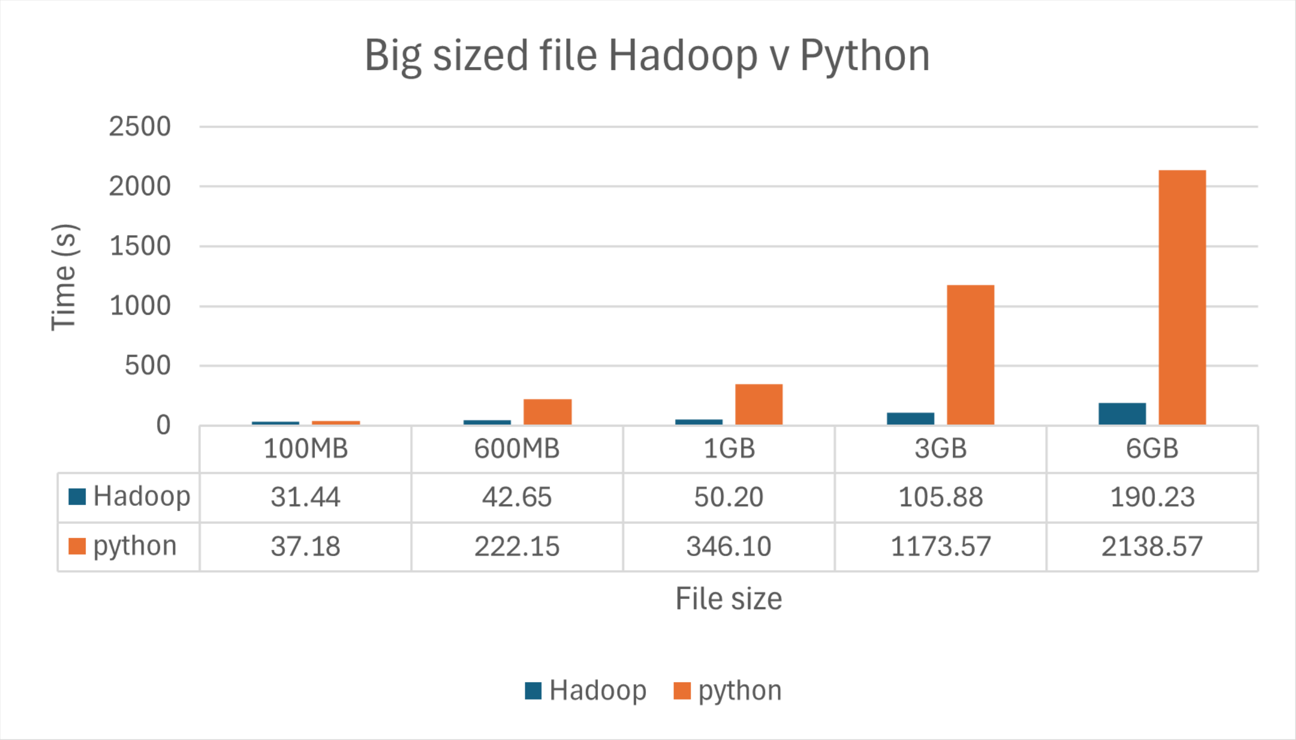
Python spark: used on the cluster

Pure Python implementations

- **First implementation**: Simple, use file.read() and dictionary for letters count
- **Second implementation**: Use mmap to map file in the virtual address space of the process, can process files bigger than RAM limits

Hadoop vs Python times





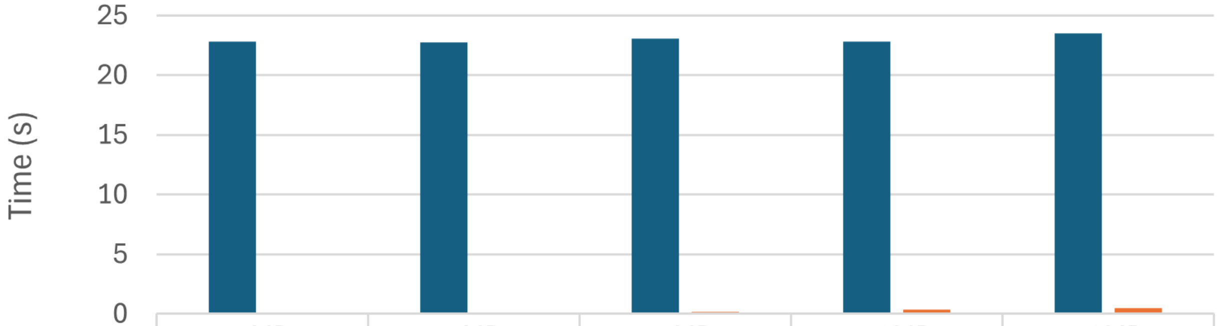
C implementations

- **First implementation:** Use multithread approach with mmap
- **Improving first implementation:** Remove the check for ascii characters from the thread code. This results in more computation (counting non-ascii chars too) but reduces chances of branch misprediction

All solutions were compiled with enabled optimizations (gcc -O2 flag)

Hadoop MapReduce vs C times

Small sized file Hadoop v C

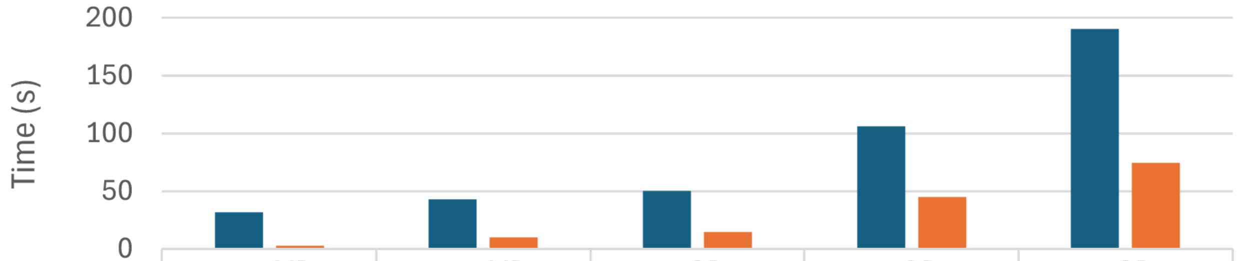


	1MB	2MB	5MB	10MB	20MB
Hadoop	22.81	22.74	23.09	22.83	23.53
C	0.08	0.05	0.18	0.38	0.48

File size

Hadoop C

Big sized files Hadoop v C

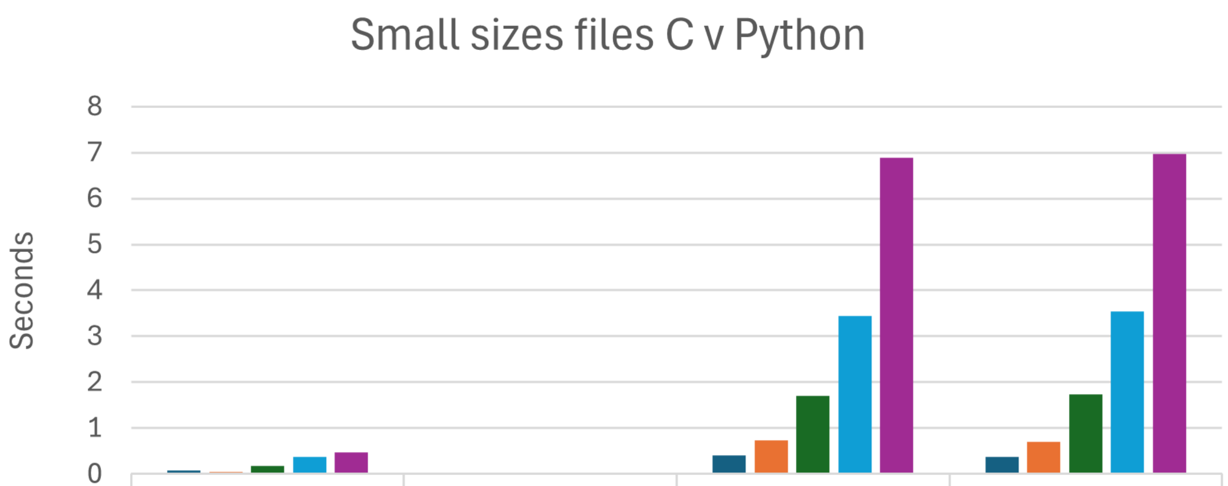


	100MB	600MB	1GB	3GB	6GB
Hadoop	31.44	42.65	50.20	105.88	190.23
C	2.60	10.03	14.73	45.13	74.75

File size

Hadoop C

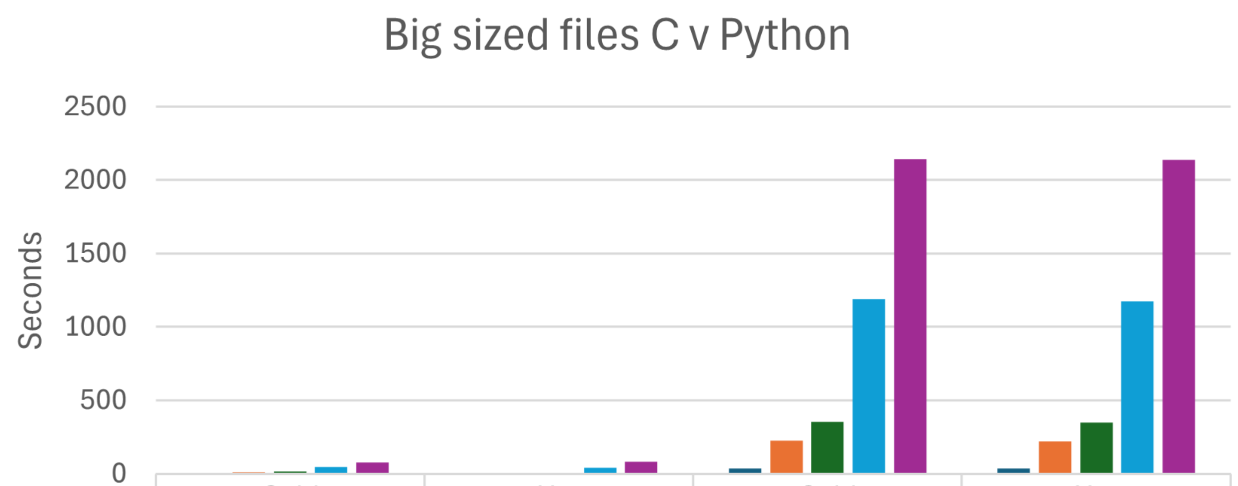
C vs Python times



	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

file size

1MB 2MB 5MB 10MB 20MB



	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

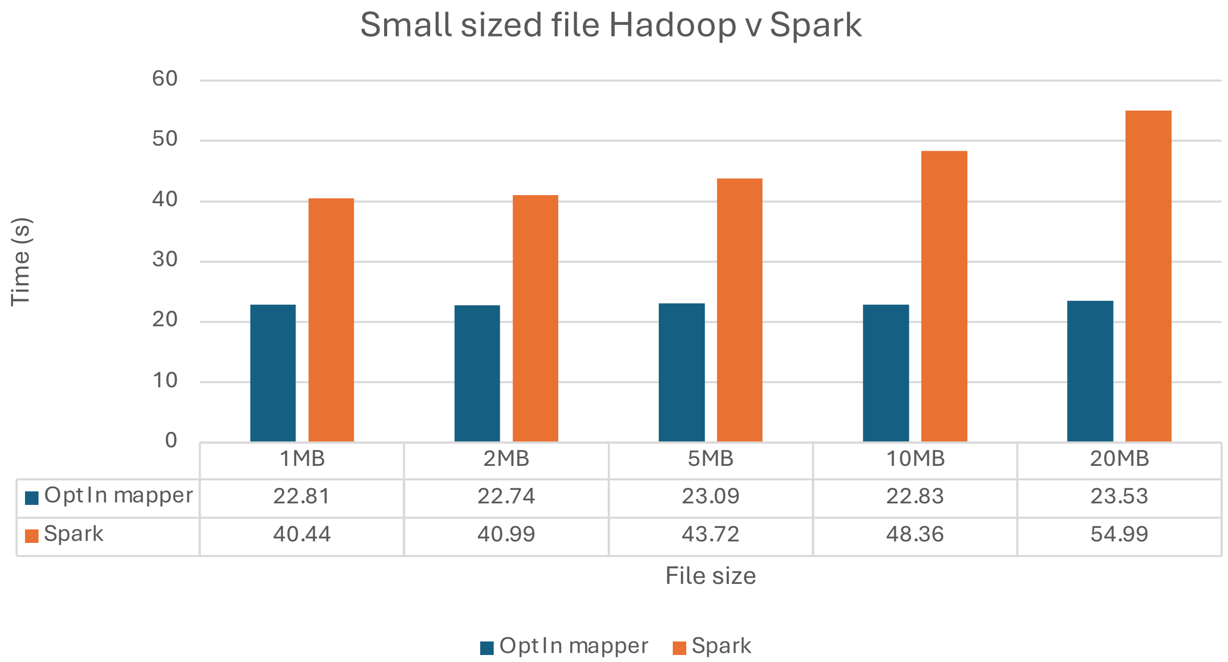
file size

100MB 600MB 1GB 3GB 6GB

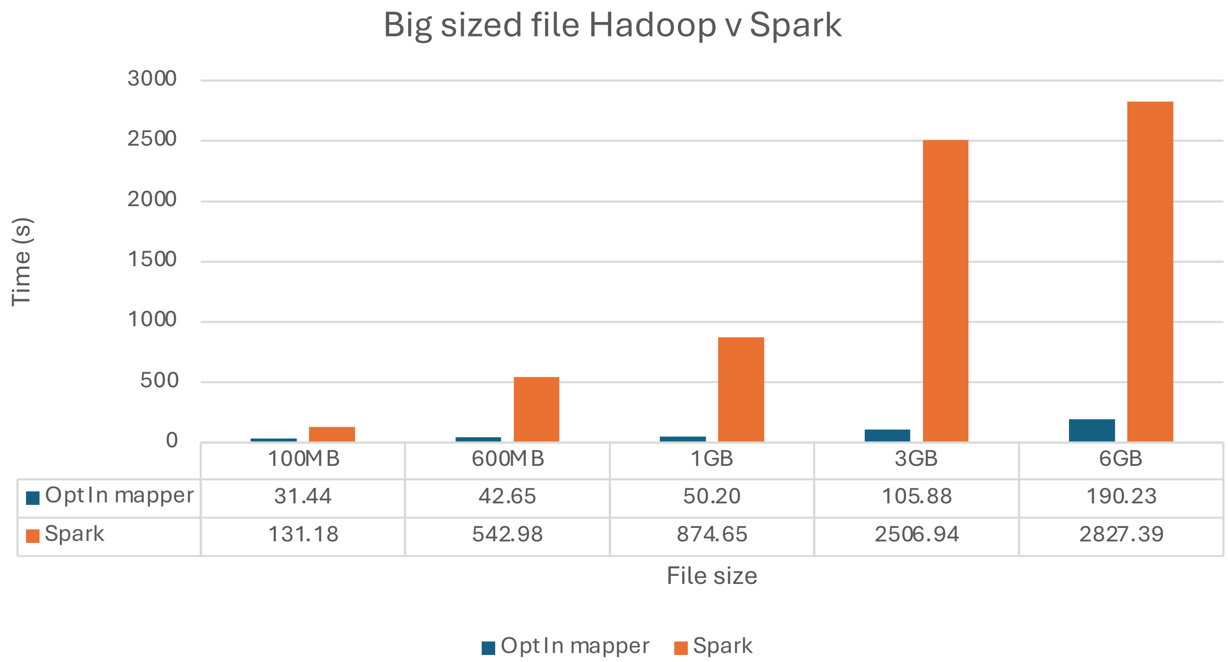
Spark implementation

- Use **RDD** and lineage tracking which allows for failure handling
- The usage of **RAM** should improve performances

Hadoop vs Spark times



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

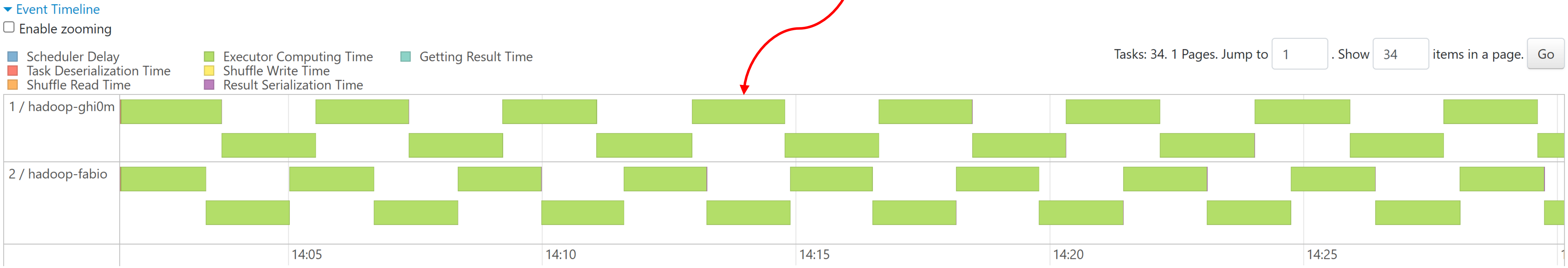


	100MB	600MB	1GB	3GB	6GB
OptIn mapper	31.44	42.65	50.20	105.88	190.23
Spark	131.18	542.98	874.65	2506.94	2827.39

On smaller files Spark can have a bigger overhead, but on larger ones it should not scale this bad, why?

Hadoop vs Spark times

~100 seconds of computing time for each block (6Gb file)



Unfortunately, the Spark deployment is CPU bounded on the cluster causing long delays

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

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

Key points

Hadoop MapReduce

- Only need to redefine few functions
- Distributed file system, can scale, fault tolerant
- Overhead of the JVM is visible with smaller files

Python

- Easy to implement, few lines of code
- Overhead of VM is visible w.r.t. C, MapReduce is faster with large files
- Doesn’t fully exploit mmap’d files
- No failure handling, files are limited by the disk size

C

- Faster, no virtual run-time
- Hard to implement a *fast version*
- No failure handling, files limited by the disk size