Università di Pisa

Master Of Science in Computer Engineering

Cloud Computing

Inverted index search

Project members:
Francesco De Lucchini
Niccolò Mulè
Antonio Andrea Salvalaggio

Academic Year 2024/2025

1 Introduction

1.1 Problem description

An inverted index is a foundational **data structure used in information retrieval systems** like Google Search. Given a large collection of text files (e.g., articles, books, web pages), the inverted index maps each detected word to the files in which it appears, along with the number of times it appears in each file, for example:

```
cloud       file1.txt:4 file2.txt:1 file3.txt:1
computing   file1.txt:1 file3.txt:2
is          file3.txt:1
important   file1.txt:9 file3.txt:1
```

In this brief report, we compare three possible solutions to build an inverted index: a simple **python non-parallel script**, and two distributed applications, one based on the **Hadoop** framework (v 3.1.3) and one based on the **Spark** framework (v 3.4.4).

1.2 Hardware and software

The test bench is composed by a **cluster of by three virtual machines** deployed on the University of Pisa's datacenter. Each VM is equipped with 7 GB of RAM, approximately 40 GB of storage, a dual (V)core Intel Xeon Silver 4208 @ 2.10 GHz and runs Ubuntu 22.04.5 LTS.

1.3 Datasets

The datasets consist of **UTF-8 plain-text books** randomly sampled from the Gutenberg project:

Name	**Book count**	**Size** (approx.)
Tiny	6	500 KB
Small	19	5 MB
Medium	143	50 MB
Large	1328	500 MB
Huge	13490	5 GB

2 Pseudo-code

In this section, we briefly present the map-reduce pseudo-code of the Hadoop solution, abstracting from all the implementation details.

2.1 Mapper

For each input line of text, it splits the line into tokens, cleans the tokens by removing punctuation marks and lowercasing them, and then outputs couples `word, (filename, 1)`.

```
class Mapper
{
    function map(FilenameAndOffset key, Text line)
    {
        tokens = split(line)
        for (token t in tokens)
        {
            word = clean(t) // Remove punctuation marks and lowercase
            filenameAndCount = (key.filename, 1)
            emit(word, filenameAndCount)
        }
    }
}
```

2.2 Reducer

For each word, combines the `(filename, count)` values received from the mappers into a clean, well-formatted output string.

```
class Reducer
{
    function reduce(Text word, FilenameAndCount counts[])
    {
        // Given a word, keeps track of its occurrences in each file
        Map<filename, integer> globalCounts

        for FilenameAndCount c in counts
        {
            previousCount = globalCounts.get(c.filename)
            increment = c.count // Without a combiner it's always 1
            globalCounts.set(c.filename, previousCount + increment)
        }

        String outputLine = buildOutputLine(globalCounts);
        emit(word, outputLine);
    }
}
```

3 Results

3.1 Calibrating the Hadoop solution

3.1.1 Choosing the right input format

The widely-used `TextInputFormat` is designed to take textual input files, split them, and feed them line-by-line to a number of mappers equal to the total number of input splits. **When `TextInputFormat` is dealing with a lot of small files**, i.e., files smaller than the split size (which is exactly what happens in this project), it creates as many mappers as input files, which **is extremely inefficient**.

A better solution is to use `CombineTextInputFormat`, which automatically joins multiple small files together, and, therefore, creates way less mappers than input files, leading to far better performances (as shown in Figure 1). Note that with the biggest dataset, both solutions crash because of insufficient disk space to store the mappers' outputs, signaling a strong need for a (possibly in-mapper) combiner, which is studied in Section 3.1.2.

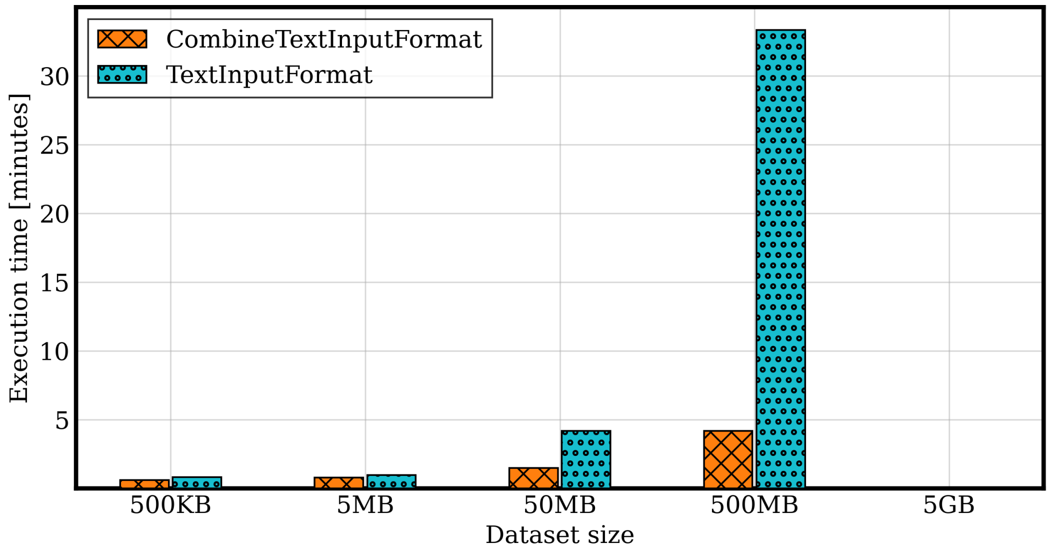

Figure 1: Execution times of `TextInputFormat` and `CombineTextInputFormat` as the size of the dataset increases

3.1.2 Choosing the right combiner design

Figure 2 shows that, as expected, **implementing an in-mapper combiner avoids the emission of unnecessary intermediate results**, reducing both network traffic and local disk writes. Compared to a standard combiner (or no combiner at all), **this comes at the cost of a more complex implementation**. In fact, it was necessary to implement a manual flush mechanism that periodically checks how much memory the mapper is using and flushes its aggregation map whenever a certain threshold is exceeded.

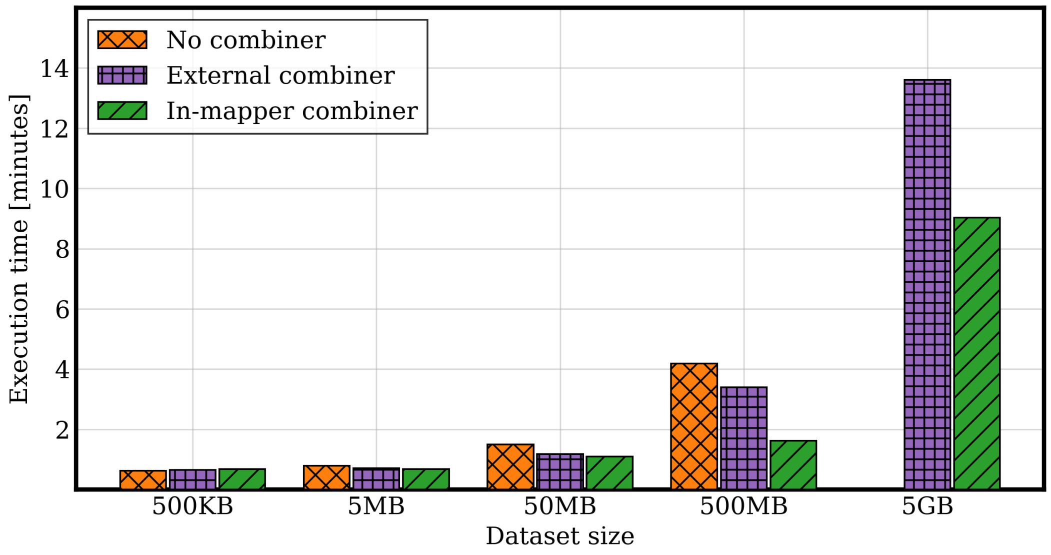

Figure 2: Execution times of different combiner designs as the size of the dataset increases

3.1.3 Choosing the right number of reducers

With the configuration illustrated in Section 3.2, each node in the cluster can run up to 4 concurrent Hadoop tasks. Since there are 3 nodes in the cluster, the optimal number of reducers is expected to be around 12. As Figure 3 shows, **execution times are higher with fewer than 12 reducers**, indicating that the cluster's resources are not being used to their full potential. **Having more than 12 reducers provides better load balancing**, as the standard deviation is lower, however, it **also introduces more scheduling overhead**. For the selected configuration, **16 reducers seem to be a good compromise**.

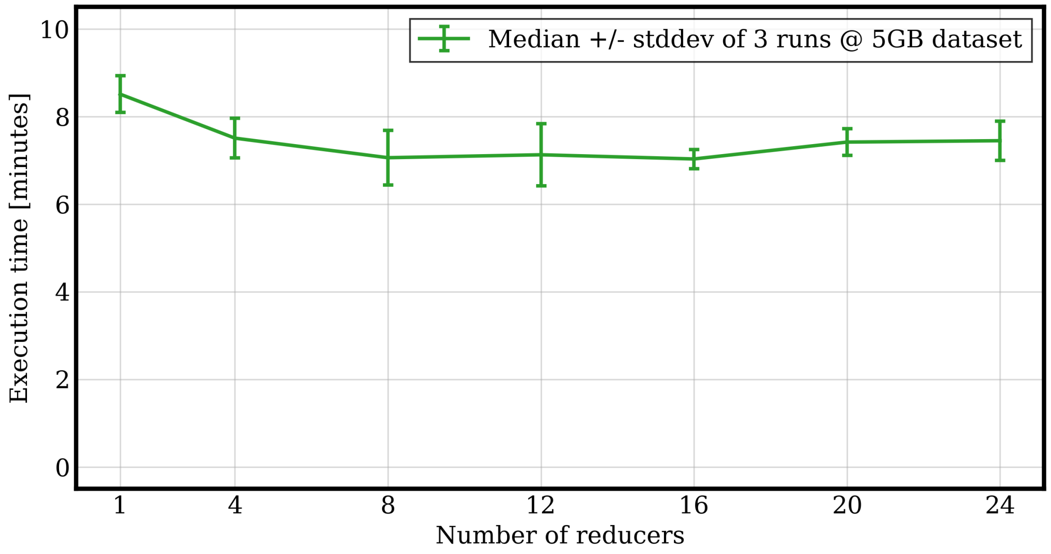

Figure 3: Execution time of the in-mapper combiner as the number of reducers increases

3.2 Comparing the Hadoop, Spark and Python solutions

Hadoop and Spark are both configured to run on a **fully distributed YARN installation**:

Hadoop

yarn.nodemanager.resource.memory-mb	2048
yarn.app.mapreduce.am.resource.mb	512
mapreduce.map.memory.mb	512
mapreduce.reduce.memory.mb	512

Spark

yarn.nodemanager.resource.memory-mb	4096
spark.yarn.am.memory	1024
spark.driver.memory	1024
spark.executor.memory	1024
spark.executor.memoryOverhead	1024
spark.executor.instances	5
spark.executor.cores	2

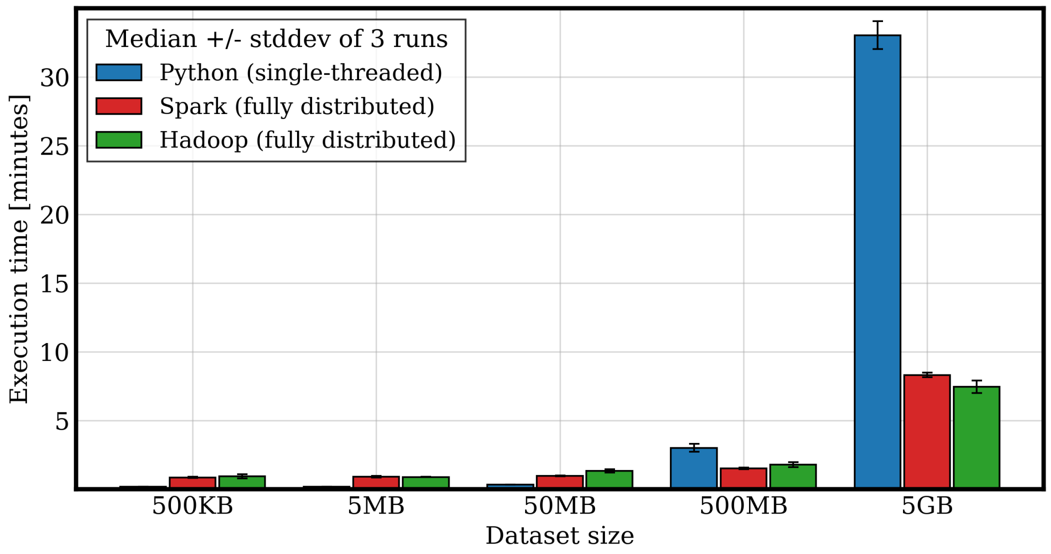

Figure 4: Execution times of different solutions as the size of the dataset increases

As shown in Figure 4, the **Hadoop** solution, even with half the total dedicated RAM of the **Spark** solution (6 GB vs 12 GB), achieves **similar execution times**. This comes at no surprise, since we are operating in a **memory-constrained environment** with a **single-pass algorithm**, the ideal use case for a MapReduce job. Moreover, as reported in Section 3.1, the **Hadoop solution was carefully fine-tuned** for the input datasets, instead, the Spark solution only uses off-the-shelf library components.

The single-threaded python solution, thanks to the reduced overhead, works remarkably well for small datasets, but **gets easily outperformed with heavier loads**. Nevertheless, **its memory footprint remains the lowest**, consistently using roughly as much memory as the output index size (precisely: 1 MB, 11 MB, 59 MB, 622 MB and 4820 MB). Finally, it should be highlighted that this solution **only required** 59 **lines of code**, which is 1.3 times fewer than the Spark one and around 10 times fewer than the Hadoop one. This makes it ideal for quickly developing a working prototype of the distributed application.