University of Pisa

Department of Information Engineering

Master's Degree Artificial Intelligence and Data Engineering

Cloud Computing Project: Inverted Index

Professors:

Carlo Vallati

Carlo Puliafito

Students:

Andrea Zanin

Lorenzo Valtriani

Samuele Marchi

ACADEMIC YEAR 2024/2025

Contents

Introduction

1.1 Problem presentation

This project, developed within the field of *cloud computing*, focuses on implementing an *inverted index* on a large dataset composed by small-sized scientific papers (on the order of kilobytes).
The application was implemented in three ways: using **Hadoop MapReduce**, with both a *combiner* and an *in-mapper combining* variant; a distributed version with **Apache Spark** and a non parallel version developed in **Python**. The output maps each word (`word`) to the list of files in which it appears, along with the number of occurrences per file, in the following format:

```
cloud          file1.txt:3  file2.txt:4
computing      file2.txt:2  file3.txt:1
```

The project concludes with a comparative analysis of the three approaches, focusing on *execution time* and *resource management*.

1.2 Dataset description

The dataset was collected using a custom Python web scraping script that automatically extracted scientific articles from the arXiv website. The original PDF files were converted to plain text to enable processing with distributed frameworks such as MapReduce and Apache Spark.

To assess the performance and scalability of the implementations, six dataset versions were generated, varying in total size and number of files.

Dataset name	**Size**	**Number of files**
`dataset_kilobytes`	276 KB	6
`dataset_megabytes`	94 MB	1,622
`dataset_megabytes2`	353 MB	5,609
`dataset_megabytes3`	706 MB	11,218
`dataset_gigabytes`	1.1 GB	19,064
`dataset_gigabytes2`	2.2 GB	57,192

This segmentation allows for evaluating algorithm efficiency under increasing workload conditions, similar to those in modern search engines and large-scale text processing systems.

Pseudocode

2.1 External combiner version

Algorithm 1 Mapper

```
class InvertedIndexMapper extends Mapper
    Variables: stopWords ← new empty list
    method Setup(context)
        stopWordsFile ← load stopwords.txt
        for all line in stopWordsFile do
            w ← cleaned lowercase line
            if w ≠ empty then
                add w to stopWords
            end if
        end for
    end method

    method Map(key, value, context)
        fName ← get file name from key
        tokens ← Preprocess(value)
        for all t in tokens do
            EMIT(t, (fName, 1))
        end for
    end method
end class
```

Algorithm 2 Combiner

```
class InvertedIndexCombiner extends Reducer
    method Reduce(key, values, context)
        fileCounts ← new AssociativeArray
        for all val in values do
            fileName ← get the file name from val
            count ← get the counter from val
            fileCounts{fileName} ← fileCounts{fileName} + count
        end for
        for all file in fileCounts do
            count ← fileCounts{file}
            EMIT(key, (file, fileCounts{file}))
        end for
    end method
end class
```

Algorithm 3 Reducer

```
class InvertedIndexReducer extends Reducer
    method Reduce(key, values, context)
        fileCounts ← new AssociativeArray
        result ← new text
        for all val in values do
            fileName ← get the file name from val
            count ← get the counter from val
            fileCounts{fileName} ← fileCounts{fileName} + count
        end for
        for all file in fileCounts do
            result ← concat(file:fileCounts{file})
        end for
        EMIT(key, result)
    end method
end class
```

2.2 In-Mapper combining version

Algorithm 4 Mapper with In-Mapper Combining

```
class InvertedIndexMapper extends Mapper
    Variables: stopWords ← empty list, wordMap ← empty map, fileName ← empty
    method Setup(context)
        stopWordsFile ← load stopwords.txt
        for all line in stopWordsFile do
            w ← cleaned lowercase line
            if w ≠ empty then
                add w to stopWords
            end if
        end for
    end method

    method Map(key, value, context)
        curFile ← get file name from key
        if fileName is null then
            fileName ← curFile
        end if
        if fileName ≠ curFile then
            Flush(context), fileName ← curFile
        end if
        for all t in Preprocess(value) do
            if t ≠ empty then
                wordMap{t} ← wordMap{t} + 1
            end if
        end for
    end method

    method Cleanup(context)
        Flush(context)
    end method

    method Flush(context)
        for all w in wordMap do
            EMIT(w, fileName:wordMap{w})
        end for
        clear wordMap
    end method
end class
```

Algorithm 5 Reducer

```
class InvertedIndexReducer extends Reducer
    method Reduce(key, values, context)
        result ← convert values in serializzable form
        EMIT(key, result)
    end method
end class
```

Performance Evaluation

Before analyzing the performance of the different applications, it is important to note that the cluster, provided by the University of Pisa, consists of three VMs, each equipped with **7 GB of RAM** and **40 GB of disk**. Each VM has been configured with the following Hadoop and Spark settings:

Configuration	Value
`yarn.nodemanager.resource.memory-mb`	1536
`yarn.scheduler.maximum-allocation-mb`	1536

Configuration	Value
`yarn.app.mapreduce.am.resource.mb`	512
`mapreduce.map.memory.mb`	256
`mapreduce.reduce.memory.mb`	256

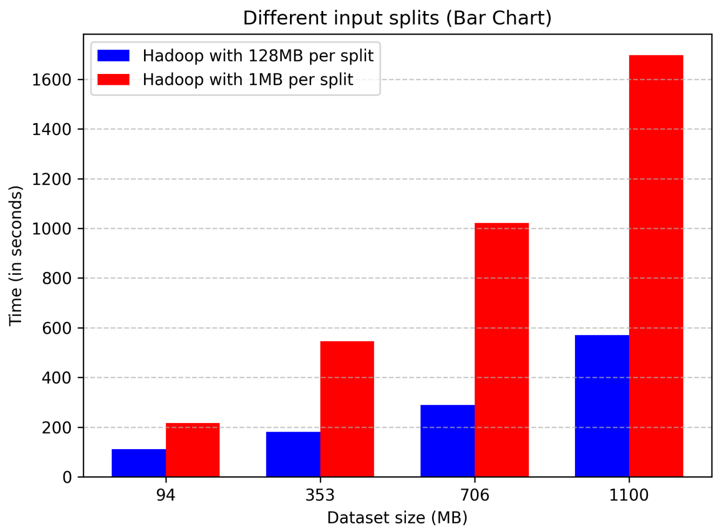
Configuration	Value
`spark.master`	yarn
`spark.driver.memory`	512m
`spark.yarn.am.memory`	512m
`spark.executor.memory`	512m

3.1 Different input split sizes

In this section, the application was experimentally evaluated by varying the *input split size*, thereby affecting the number of mappers. The use of `CombineTextInputFormat` allowed multiple small files to be grouped into a single input split, optimizing processing in scenarios with large numbers of small files.

Smaller input splits result in *fewer spilled records per map task*, reducing disk I/O and *lowering the average mapper execution time*. However, this setup leads to a *longer overall job duration* and a *noticeable increase in memory usage*.

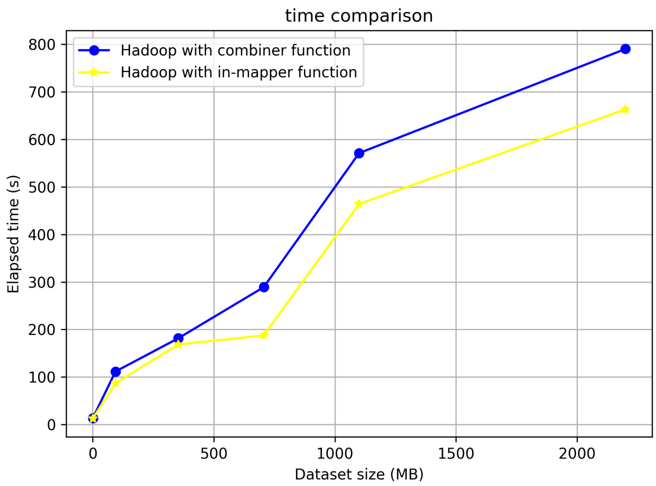
In conclusion, for applications dealing with many small files, as in this case, using `CombineTextInputFormat` with an input split size equal to the HDFS block size is an effective strategy.



3.2 External combiner vs In-Mapper combining

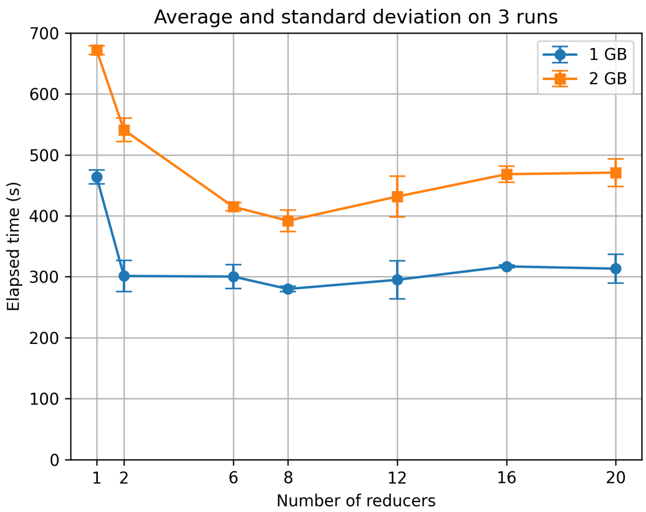
We ran the MapReduce application, first using an external combiner and then implementing in-mapper combining. We then compared the performance of the two approaches.

The In-Mapper solution offers several advantages. In all the cases analyzed, its execution time decreases, both in terms of CPU time and overall elapsed time. Additionally, memory utilization is reduced, not only in terms of peak memory usage but also in terms of the total memory consumed.



3.3 Different number of reducers

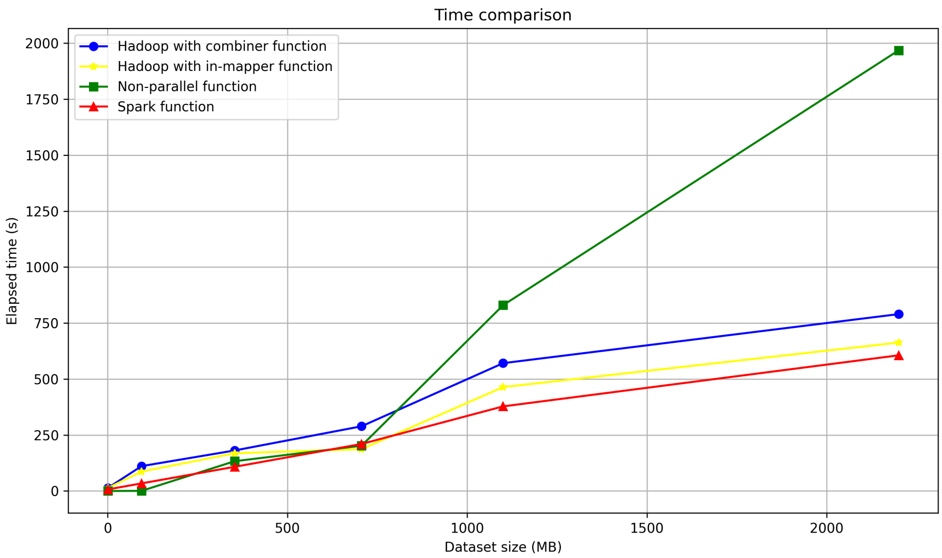
We now analyze how performance changes as the number of reducers increases. Specifically, we first tested the application on the 1 GB dataset and then on the 2 GB dataset, increasing the number of reducers up to 20 and performing 3 different runs for each configuration. This approach allowed us to compute the average elapsed time and its standard deviation for each setting. The results show that, for both datasets, **the fastest execution was achieved when using 8 reducers** which seems to be a good compromise *between load balancing and overhead.*



3.4 Time comparison between different solutions

Regarding execution time, it can be observed that **Spark proves to be the fastest solution**. This is particularly true when dividing the RDDs into partitions of approximately 750 files. This number was determined experimentally, aiming to reduce the number of partitions to lower overhead, while still leveraging only RAM for data processing, thus taking advantage of Spark's architecture.

It is evident that, although the non-parallel solution performs very well on small datasets, its execution time increases significantly as the data size grows. In contrast, both **Hadoop and Spark exhibit a growth trend that is similar to a logarithmic curve**, making them substantially more efficient when handling large-scale datasets.



3.5 Resources comparison between different solutions

Regarding the number of resources allocated by each of the compared solutions, the analysis considers the aggregate resource allocation, a metric available for both Hadoop and Spark. This value represents the **total amount of memory allocated over time**, calculated as the sum of the memory used by each container multiplied by the duration of its activity. For small datasets, resource usage remains nearly identical across all approaches. However, with increasing dataset size, a clear trend emerges: **the Hadoop solution with the external combiner allocates a significantly higher number of resources** compared to the other two solutions.