UNIVERSITY OF PISA

DEPARTMENT OF INFORMATION ENGINEERING

Cloud Computing Project Report

Inverted Index and Search

Work group:

Valentina Bertei

Alex Sgammato

Francesco Tarchi

ACADEMIC YEAR 2024/2025

1 Introduction

This project focuses on the design and implementation of a basic search engine backend built around the *inverted index* data structure.

Starting from a custom collection of text files, the system extracts every word and maps it to the set of documents in which it appears, along with its frequency. Additionally, a basic search component is provided to query the index in order to retrieve the list of files in which every word of a given string is included (could be also in a sparse way).

The objective is to explore different *Big Data* processing frameworks, namely **Hadoop** and **Spark**, by building two separate implementations of the same indexing task, assessing their performance across datasets of varying sizes. A comparison is conducted to analyze the efficiency and scalability of each approach.

2 Dataset Description

The dataset used in this project is derived from the *Common Crawl News* corpus, accessed via the *Hugging Face* `datasets` library. Specifically, the `cc_news` dataset was employed, which comprises a large-scale collection of news articles aggregated from a wide range of online sources. This dataset offers a rich and temporally diverse set of documents, making it well-suited for evaluating the performance of text processing pipelines such as inverted indexing.

To construct the document collection, a custom Python script was developed to stream the dataset and progressively write the articles into multiple plain text files. Each article was extracted by concatenating its `title` and `text` fields, and written to disk in 40MB-sized segments to facilitate manageable processing and distributed computation. The resulting corpus was saved into a dedicated directory, with each file named sequentially (e.g., `news_001.txt`, `news_002.txt`, etc.). The content was subjected to light pre-processing to remove extraneous elements and ensure textual uniformity across the dataset.

Multiple folders containing different numbers of text files were generated to support scalability testing and performance benchmarking of both the MapReduce and Spark-based indexing systems. Such folders are displayed in the following table.

Folders size	**Number of files**	**Average file size**
10 KB	3	4 KB
100 KB	5	21 KB
1 MB	6	175 KB
10 MB	8	1.25 MB
100 MB	10	10.00 MB
1 GB	26	40.50 MB
2 GB	53	40.50 MB
10 GB	150	68.27 MB

Table 1: Input dataset sizes and corresponding number of files used for testing

This range allows us to evaluate the scalability and efficiency of the indexing algorithms under different workloads, from very small datasets to considerably large collections. Each of these folders was processed independently to collect timing and resource usage metrics across the different implementations.

3 Hadoop Implementation

The Hadoop solution leverages the MapReduce paradigm to build the inverted index from a collection of text files. The *input split* size used is the HDFS default size.

3.1 Input format

The `FileInputFormat` used is the default `TextInputFormat`: an input split (`InputSplit`) is linked to a single file (`FileSplit`), but a file might be split into more input splits (in the case of big-sized files). Consequently, since each task works by default on a single `InputSplit`, each task works also on a single `FileSplit`. Moreover, each `InputSplit` (`FileSplit`) is processed line by line.

3.2 Mapper

The `TokenizerMapper` reads *a single line at time* from an `InputSplit`. It first removes non-alphabetical characters, converts the text to lowercase, and splits it into words. Each word is then combined with the corresponding file name to form the key `word@filename`. Eventually, the mapper emits the key-value pair `[word@filename, 1]` for each key occurrence.

```
map(key, line):
    words = tokenize_and_clean(line)
    filename = get_current_filename()

    for word in words:
        if word is not empty:
            emit(word + "@" + filename, 1)
```

Listing 1: Pseudocode for the Mapper

3.3 Combiner

The `SumCombiner` reads *a key at time* and acts as a local reducer in order to reduce network traffic by summing the values of each `word@filename` key before sending data to the actual reducers in the form of key-value pairs `[word@filename, sum]`.

```
combine(key, values):
    sum = 0
    for value in values:
        sum += value
    emit(key, sum)
```

Listing 2: Pseudocode for the Combiner

3.4 Reducer

The `InvertedIndexReducer` reads *a key at time.* It first sums all values of a key-value pair `[word@filename, listOf(value)]` in order to aggregate the frequency for each file in which a word appears. Then, it groups all filenames with the same word, and finally emits the word followed by a list of `filename:count` in a key-value pair `[word, listOf(filename: count)]`.

```
reduce(key, values):
    word, filename = split(key, "@")
    count = sum(values)
    if word != last_word:
        if last_word is not empty:
            emit(last_word, file_map)
        file_map = empty map
    file_map[filename] += count
    last_word = word
cleanup():
    emit(last_word, file_map)
```

Listing 3: Pseudocode for the Reducer

3.5 Output format

The final output has one line per word, followed by tab-separated pairs indicating in which files it occurs and how many times in that specific file. An example could be the following.

```
data    file1.txt:3    file7.txt:5    file8.txt:1
```

This structure is ideal for search queries and further text analysis tasks.

4 Other Frameworks

In order to get some stats about our "classic" MapReduce application, we develop the following solutions.

- A "classic" *Spark* solution in Python, supporting RDDs and parallelism.
- A *Spark* solution with a sort of *"in-mapper combiner"* in Python, supporting RDDs and parallelism.
- A *Python non-parallel* solution.
- A *Hadoop* MapReduce solution with *in-mapper combiner*.
- Some "classic" *Hadoop* solutions with *more than one reducer* (2, 4 and 8 reducers).

Moreover, a *search algorithm* has been developed in order to search a list of words in the output of an execution of our application: function returns the files in which **all** the words are included (even if in different places of the file).

5 Experimental Results[1]

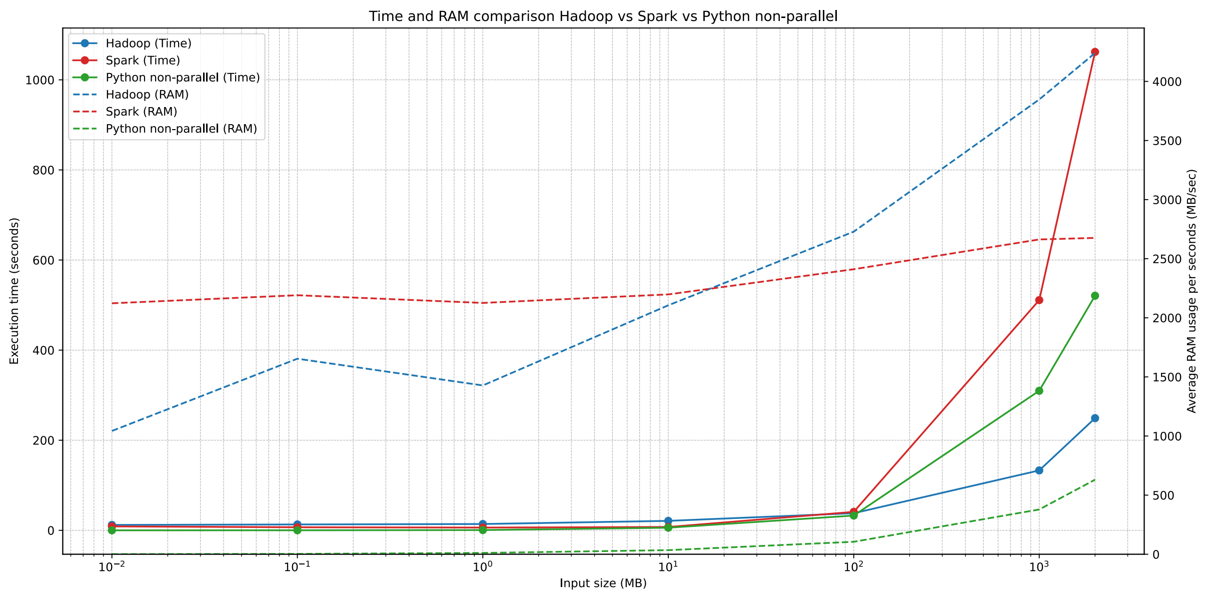

Figure 1: Comparison between the three main solutions (Hadoop vs. Spark vs. Python non-parallel).

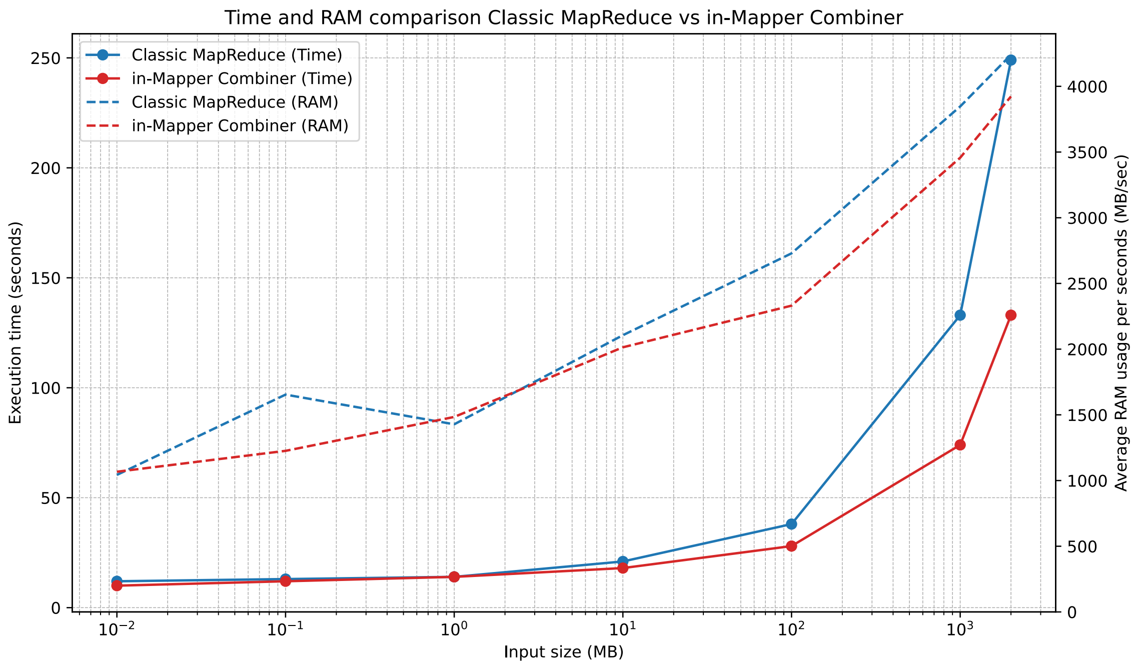

Figure 2: Comparison between classic Hadoop MapReduce and Hadoop MapReduce with in-mapper combiner.

[1]In all our time statistics for Hadoop we considered just the MapReduce time (for Spark, the total time of the stages): we did not considered the driver instantiation time because the Python non-parallel solution does not have it.

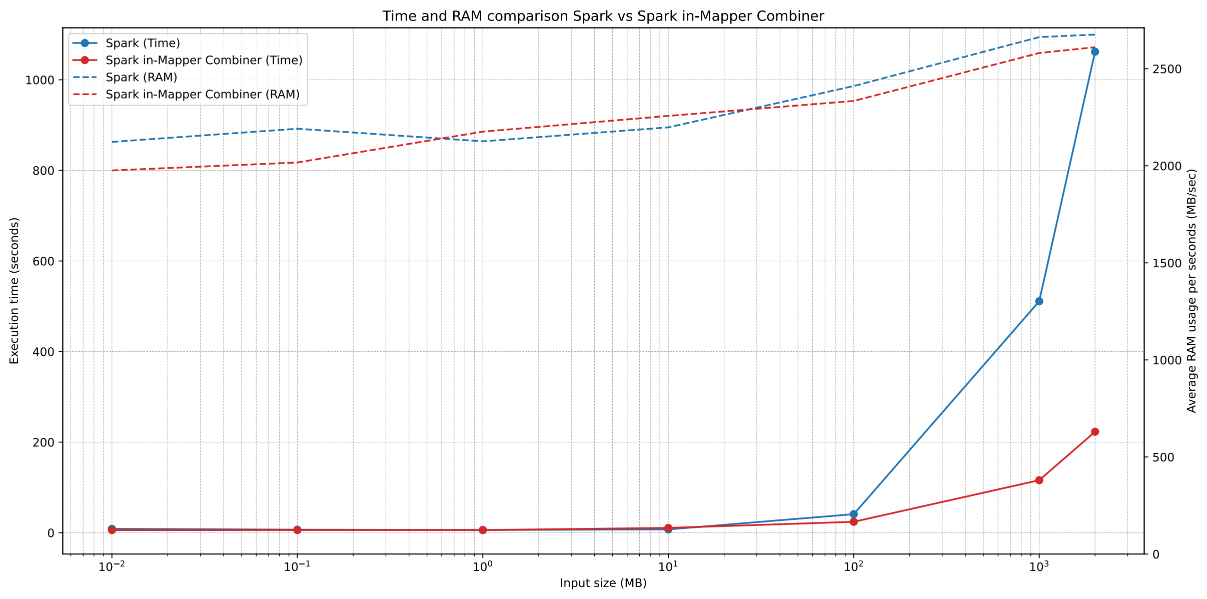

Figure 3: Comparison between classic Spark solution and Spark solution with "in-mapper combiner".

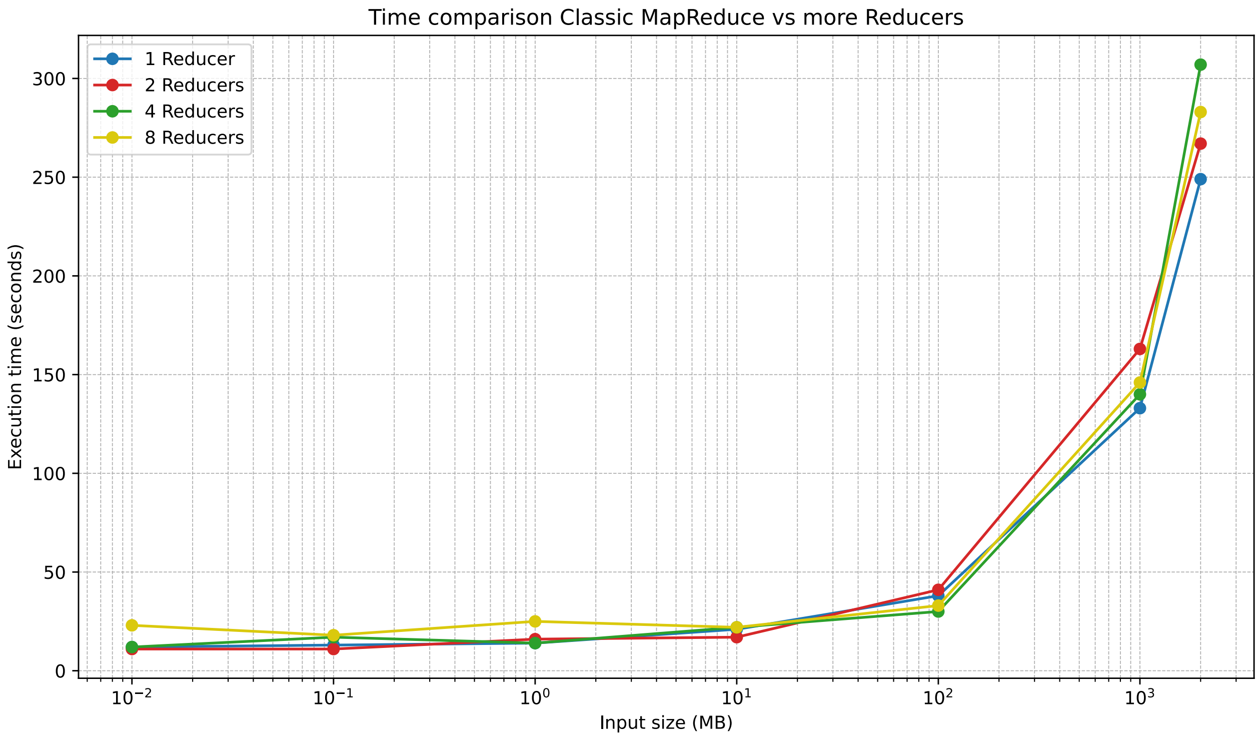

Figure 4: Comparison between classic MapReduce with growing number of reducers.

6 Conclusions

From an examination of the previous graphs, we can state the following assumptions.

- In terms of **execution time**, large differences between the three solutions are evident just with big datasets (from 100 MB on).

- In terms of **RAM usage**, large differences are clear with datasets of every size.

- The **best framework in terms of execution time** is *Hadoop*: even the "classic" MapReduce solution parallelizes the workload better than the competitors, instantiating a variable number of mappers based on dataset size.

  The "classic" *Spark* solution spends a large amount of time to move intermediate data in the form `[word@filename, 1]` between memory and "spill memory" or disk. This behavior starts with dataset sizes of 1 GB and 2 GB (which are too big to be entirely kept in executors' RAM), leading to a clear worsening in performances w.r.t. the other two solutions.

  The *Python non-parallel* solution positions itself in the middle between the other two solutions: it uses a quantity of RAM that allows its code to be executed without needing the disk (unlike what *Spark* does). Such trend is expected to change and become worse than *Spark* with much bigger datasets and machines with more available resources.

- The **best framework in terms of RAM usage** is clearly the *Python non-parallel* solution, because it does not need to instantiate containers with fixed amount of memory used in order to support drivers and executors.

  The "classic" `Spark` solution uses about the same amount of RAM along all dataset sizes, because the number of executors remains the same (2). On the other hand, the *Hadoop* solution is the worst in term of memory due to the hard parallelization it can reach (each mapper container uses 256 MB of RAM).

- The *in-mapper combiner* solutions for both *Hadoop* and *Spark* leads to an **expected hard improvement in execution time**: in particular for *Spark* the improvement is huge (about 80% with dataset seizes $>$ 1 GB). The reason is the smaller number of keys to exchange between containers (or stages).

  Such solutions did not report an improvement in RAM usage w.r.t their own versions without *in-mapper combiner*: for both frameworks, the number of mappers/executors/application managers remains the same (each of them uses a fixed amount of RAM).

- The *Hadoop* solutions with *more reducers* turned out to be **useless**. With our dataset sizes, the reduce part is not heavy at all and the improvement in reduce time with more reducers is not considerable. Moreover, the hashing of keys along with the redistribution of key-value pairs introduces an overhead that balances such improvement, even worsening performances with some combinations of number of reducers and dataset size.

  The different numbers of reducers did not improve RAM usage, so it was not reported inside the graph in order to make it more clear.

To summarize everything, *Hadoop* with *in-mapper combiner* and *one reducer* is the best solution for our application and our datasets.