UNIVERSITY OF PISA

Department of Information Engineering

Master's degree in Artificial Intelligence and Data Engineering

Cloud Computing Project

Design and Evaluation of Inverted Index and Search Systems using Hadoop, Spark, and Python

Work Group - BetterCallCloud:
Andrea Bochicchio
Ivan Brillo
Filippo Gambelli

ACADEMIC YEAR 2024/2025

Contents

1 Introduction and Dataset Description

In this chapter, we provide an overview of the project objectives and describe the dataset used to evaluate our inverted-index implementations. The GitHub repository can be found here.

1.1 Context and Objectives

This work falls within the domain of information retrieval systems, where the *inverted index* is the core data structure that enables efficient full-text search over large document collections. The goals of this project are to:

- build an inverted index using two distributed frameworks: Hadoop MapReduce (Java) and Apache Spark (Python);
- develop a standalone Python application for non-parallel index construction and a simple query tool that returns only the filenames containing the searched terms;
- introduce optimization techniques such as combiners and in-mapper combining to improve Hadoop computations;
- compare performance metrics (execution time, memory usage) across different dataset sizes and varying numbers of reducers or partitions.

1.2 Dataset Description

To ensure a robust experimental evaluation, we selected a corpus of English Wikipedia articles from Kaggle (Plain Text Wikipedia 2020-11)[1]. The raw data are provided in JSON format, with each record containing fields such as `"title"` and `"text"`.

1.2.1 Partitioning by Size and File Count

We created five subsets based on total data size: approximately 512 KB, 1 MB, 512 MB, 1 GB, and 2 GB. Each subset is further divided into two variants, containing either 10 or 20 text files. Files are generated by concatenating articles until each file reaches its target size.
To better simulate real-world document collections, we assign random sizes to individual files within each variant, resulting in non-uniform file size distributions.

1.2.2 Preprocessing

All the implementations perform an initial text-cleaning stage. We apply the following regular expression:

```
[^a-zA-Z0-9\s]
```

to remove punctuation, special symbols, and any non-alphanumeric characters, leaving only letters, digits, and whitespace.

[1] `https://www.kaggle.com/datasets/ltcmdrdata/plain-text-wikipedia-202011`

2 Hadoop MapReduce Implementation

In this chapter, we present our Hadoop-based implementation for constructing the inverted index. We focus on the high-level logic of the two approaches used: the standard combiner-based method and the stateful in-mapper combiner variant. The pseudocode provided captures the core mechanisms employed in each case to process input data, aggregate term frequencies, and generate the final inverted index.

2.1 Pseudocode Convention

In the pseudocode that follows, any lookup of a non-existent key in a map is assumed to yield a default value of zero. In actual Java code, this behavior corresponds to `map.merge(key, value, Long::sum);` which initializes the entry to `0 + value` if the key is absent.

2.2 Map–Combine–Reduce Pseudocode

The following pseudocode (Algorithm 1) abstracts the logic in `InvertedIndex.java`.

Algorithm 1 Map–Combine–Reduce pseudocode for Inverted Index

```
Require: input paths in, output path out
Ensure: inverted index mapping each word to list of (filename, total_count)
 1: class InvIndexMapper
 2:     fields:
 3:     outputKey : Text
 4:     filename : String
 5:     procedure SETUP
 6:         filename ← get name of current input split
 7:     procedure MAP(offset, line)
 8:         for all word in tokenize(line) do
 9:             emit(key=word, value=(filename, 1))
10: end class

11: class InvIndexCombiner
12:     procedure REDUCE(word, values)
13:         localCounts ← empty map from (word) to (fname, count)
14:         for all (fn, c) in values do
15:             localCounts[fn] += c
16:         for all (fname, count) in localCounts do
17:             emit(key=word, value(fname, count))
18: end class

19: class InvIndexReducer
20:     procedure REDUCE(word, values)
21:         totalCounts ← empty map from (word) to (fname, count)
22:         for all (fname, count) in values do
23:             totalCounts[fname] += count
24:         outputString ← empty string
25:         for all (fname, count) in totalCounts do
26:             outputString.append(fname : count)
27:         emit(key=word, value=outputString)
28: end class
```

2.3 Stateful In-Mapper Combiner Pseudocode

Algorithm 2 sketches the logic in `InvertedIndexInMapperCombiner.java`, where the mapper maintains an internal buffer and flushes it based on memory usage.

Algorithm 2 Stateful In-Mapper Combiner pseudocode

```
Require: input paths in, output path out
Ensure: inverted index mapping each word to list of (filename, total_count)
 1: class InvIndexMapper
 2:     fields:
 3:     localCounts ← empty map (map from word to count)
 4:     filename : String
 5:     flushThreshold ← 0.7
 6:     procedure SETUP
 7:         filename ← get name of current input split
 8:     procedure MAP(offset, line)
 9:         for all word in tokenize(line) do
10:             localCounts[word] ++
11:         if usedMemory() / maxMemory() > flushThreshold then
12:             FLUSH
13:     procedure CLEANUP
14:         FLUSH
15:     procedure FLUSH
16:         for all (word, count) in localCounts do
17:             emit(key=word, value=(filename, count))
18:         localCounts ← empty map
19: end class

20: class InvIndexReducer
21:     procedure REDUCE(word, values)
22:         totalCounts ← empty map from filename to count
23:         for all (fname, count) in values do
24:             totalCounts[fname] += count
25:         outputString ← empty string
26:         for all (fname, count) in totalCounts do
27:             outputString.append(fname : count)
28:         emit(key=word, value=outputString)
29: end class
```

3 Performance Evaluation

We conducted a comprehensive performance evaluation across three implementations: Hadoop, Spark, and a sequential Python version.

For the Hadoop implementation, we performed a series of experiments using two workloads (10 and 20 documents), five input sizes (512 KB, 1 MB, 512 MB, 1 GB, and 2 GB), and three reducer settings (1, 5, and 15). Each configuration was executed five times at different times of day to reduce the impact of load variations on the University of Pisa Datacenter.

For the Spark implementation, we used the same datasets and document counts but varied the number of RDD partitions instead of reducers. This allowed us to assess how Spark's parallelism model affects performance under the same input conditions as Hadoop.

For the Python version, a non-parallel sequential program, we executed tests using the same datasets and document counts, without any partitioning or parallel configuration.

All experiments were automated using Bash scripts, which were used across all implementations to run jobs, collect execution times, and extract memory usage where applicable. Metrics were saved to a CSV file and averaged over ten runs per configuration for consistency.

3.1 Standard Hadoop vs. Hadoop In-Mapper Combiner

To better understand the performance impact of in-mapper combining compared to the standard combiner, we analyze and compare execution time and memory usage across multiple workload configurations.

3.1.1 Execution Time

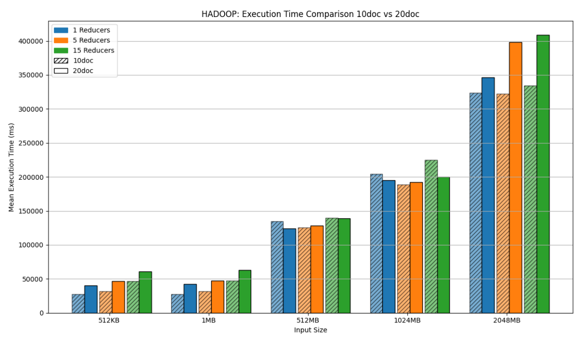


Figure 3.1: Standard Hadoop Combiner Execution Time

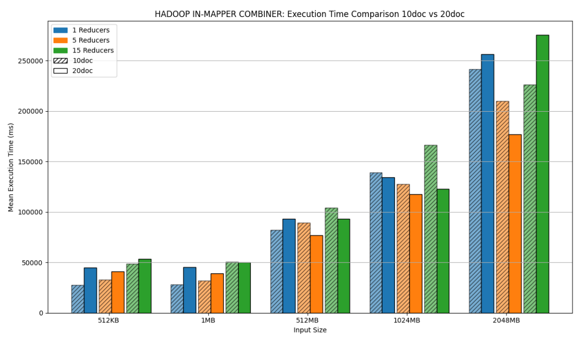


Figure 3.2: Hadoop In-Mapper Combiner Execution Time

Figures 3.1 and 3.2 show the average execution times for standard Hadoop and Hadoop with in-mapper combining, respectively. Overall, the in-mapper approach leads to consistent runtime improvements, especially for larger datasets (512 MB and above), where reductions of up to 25% are observed in some configurations.

This improvement stems from a reduction in the number of intermediate key-value pairs emitted by mappers, and consequently, fewer spill files are created, reducing I/O operations. Even the volume of transferred data may be reduced, since the execution of the standard combiner in Hadoop is optional. The benefits become more prominent as the input size grows, where the overhead of data transfer and I/O operations in the standard implementation becomes increasingly significant.

3.1.2 Memory Usage

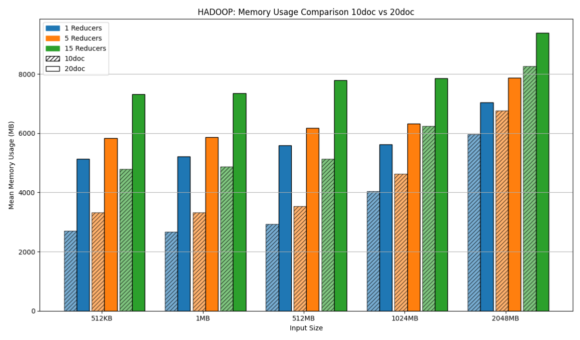


Figure 3.3: Standard Hadoop Combiner Memory Usage

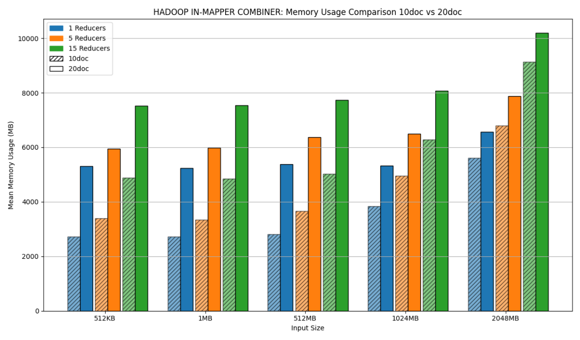


Figure 3.4: Hadoop In-Mapper Combiner Memory Usage

Figures 3.3 and 3.4 show the physical memory usage at the end of execution for the two approaches. We can conclude that there is no statistically significant difference between the two approaches.

3.1.3 Performance Overview

The in-mapper combiner provides a simple yet effective optimization for MapReduce jobs, particularly when processing large volumes of input data. By reducing the number of stored and emitted intermediate pairs, it minimizes expensive I/O operations, communication overhead, and shuffle costs, resulting in noticeable improvements in runtime performance.

3.2 Platform Comparison

To gain a comprehensive understanding of how our implementations compare, we analyze execution times across four approaches: standard Hadoop, Hadoop with in-mapper combining, Spark, and a sequential Python implementation.

For this comparison, we selected the most performant configuration for each framework based on preliminary testing: Hadoop and its in-mapper variant were run with 5 reducers; Spark used automatic partitioning; and Python was executed sequentially without parallelism. Execution times represent the average of five runs per configuration.

3.2.1 Execution Time

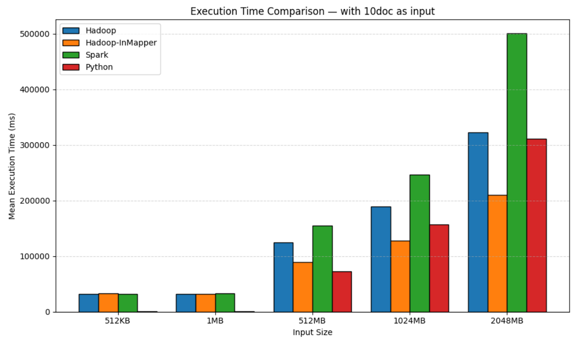


Figure 3.5: Execution Time Comparison with 10 files

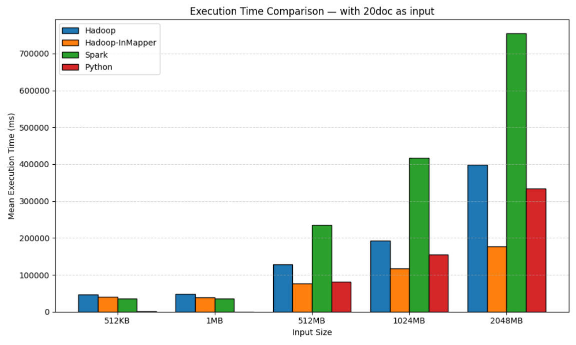


Figure 3.6: Execution Time Comparison with 20 files

As illustrated in Figure 3.5 and Figure 3.6, the performance comparison reveals key characteristics of each implementation:

- **Python excels on small datasets**: the sequential Python implementation significantly outperforms the distributed systems for small input sizes (512 KB, 1 MB) due to the absence of communication, coordination, and task scheduling overhead. However, its runtime increases sharply with larger datasets, becoming impractical beyond hundreds of megabytes.
- **Hadoop with in-mapper combining is best for large data**: among all tested systems, Hadoop with in-mapper combining yields the lowest execution times on large datasets (512 MB and above).
- **Standard Combiner Hadoop performs consistently but slower**: the baseline Hadoop implementation maintains stable performance but suffers from higher overhead than its in-mapper-enhanced counterpart.
- **Spark underperforms for large inputs**: despite Spark's reputation for speed, our results show slower execution times on large datasets. Profiling suggests that the bottleneck lies in RDD creation and input reading, probably exacerbated by large file sizes.

The Python version's speed advantage on small inputs confirms that distributed frameworks have non-negligible startup and communication costs, which dominate in lightweight scenarios. However, these costs scale poorly as input sizes grow, where parallelization and data locality become essential.
Spark's underperformance likely stems from the overhead of constructing RDDs and reading large text files using suboptimal methods (e.g., `wholeTextFiles` or `spark.read.text`), which do not fully leverage Spark's capabilities for efficient input splitting. Further tuning and alternative input-handling strategies may improve Spark's scalability in future work. More detailed results of the Spark execution time tests are presented in Figure 3.8b.
Lastly, we perform a comparison with a larger dataset, specifically created as a stress test, consisting of 100 documents totaling 2 GB. This experiment aimed to evaluate how the different implementations of Spark and Hadoop react under increased load. The mean computation time for every implementation increased by almost 100% compared to the average results for the 10-document dataset (Figure 3.7).

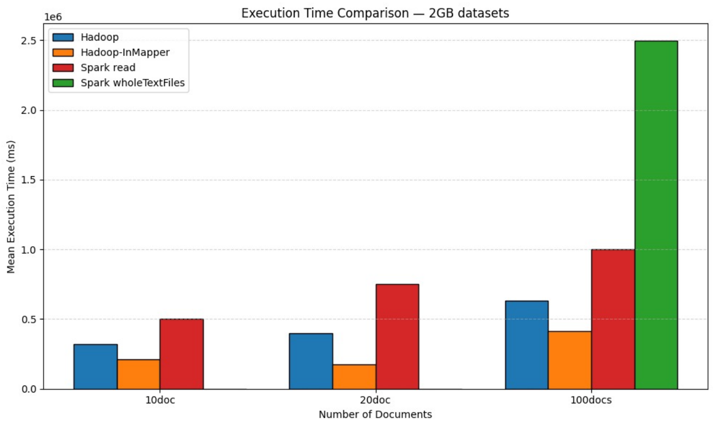
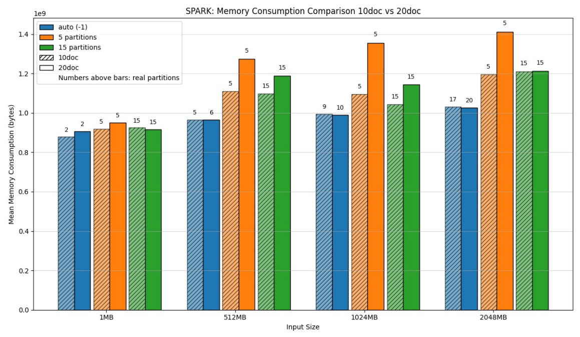


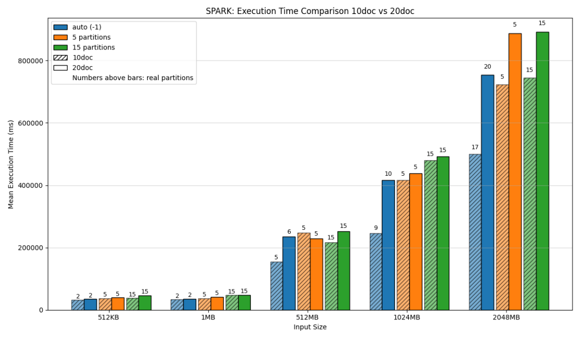
Figure 3.7: Execution time (ms) comparison across Spark (auto-partitioning) and Hadoop (5 reducers) implementations with 2GB datasets

3.2.2 Memory Usage

In addition to comparing execution times, we also performed an analysis of memory usage between Hadoop and Spark. First of all, we perform a study on spark memory utilization. We calculate the memory used adding the driver max memory and the sum of the executors peak process memory. The result is reported in the Figure 3.8a[1]. From the figure, we can observe that although there is a slight decrease in memory usage with the 1MB dataset, the amount of memory used in the other cases remains relatively consistent, despite the increasing dataset sizes.



(a) Spark Memory Usage



(b) Spark Execution Time

Figure 3.8: Spark memory consumption and execution time comparison.

To evaluate memory consumption in Spark and Hadoop, we used the aggregate resource utilization metric. This metric, computed by YARN and shared between both frameworks, enables a robust comparison of results. It measures the total memory allocated for the computation, multiplied by the time required to complete it. This is an important metric as it reflects not only the amount of memory consumed but also the duration for which it is allocated within the cluster. As shown in Figure 3.9, the Hadoop in-mapper implementation remains the most efficient.

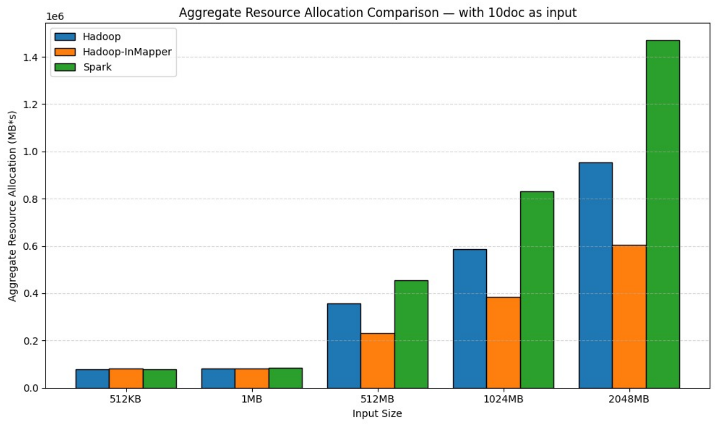


Figure 3.9: Aggregate Resource Allocation Comparison

3.3 Summary

Each technology presents unique trade-offs: Python is ideal for small-scale, quick analyses; Hadoop with in-mapper combining excels at large-scale batch processing. Spark requires careful optimization to realize its performance benefits, especially on large datasets.

[1]The 512KB dataset is not included in this figure because the command used to retrieve executor metrics did not return results, likely due to the execution time being too short to capture peak memory usage.

4 Search Query System

In addition to our parallel index-building pipelines, we implemented two variants of a simple, non-parallel search-query system in Python. Both systems consume the inverted-index files produced by Hadoop, Spark, or our standalone Python implementation, but they differ in when and how they load term postings into memory.

4.1 In-Memory Preloaded Search

The first implementation performs a one-time load of the entire inverted index into an in-memory dictionary. Upon startup, each partition file is read line by line; each line contains a term followed by a list of "filename:count" entries. We parse out the term and extract the set of filenames in which it appears. These per-partition dictionaries are merged into a global `Dict[str, Set[str]]` mapping each term to the set of all documents that contain it.

- **Advantages.**
  - Very fast per-query response time, since all postings are already in memory.
  - Query time scales with the number of query terms (and the size of the smallest posting list), not with dataset size.
- **Drawbacks.**
  - High initial loading cost proportional to the size of the full inverted index.
  - Requires enough RAM to hold all postings for the largest dataset.

4.2 On-Demand Disk-Based Search

The second implementation defers loading postings until a query arrives. Instead of building a global in-memory index at startup, it maintains only a map from query terms to empty result sets. When the user issues a query, the system scans each partition file on disk, line by line, checking whether the current line's term matches any of the query terms. If so, it extracts the filenames from the "filename:count" entries and merges them into the corresponding result set. Finally, the result sets for all terms are intersected to produce the output.

- **Advantages.**
  - Low startup overhead and minimal memory footprint, since only one line is buffered at a time.
  - Suitable for very large indices that cannot fit entirely in RAM.
- **Drawbacks.**
  - Slower per-query response, as each query requires scanning every index file on disk.
  - IO cost grows linearly with both the number of partitions and the size of each posting file.