Cloud Computing - Project Report
Cloud-Busters

Hajar Makhlouf
h.makhlouf@studenti.unipi.it

Lorenzo Vezzani
l.vezzani1@studenti.unipi.it

Francesco Panattoni
f.panattoni2@studenti.unipi.it

June 17, 2025

https://github.com/lorenzo-vezzani/inverted-index-and-search

1 Introduction

Our project for the Cloud Computing course involves developing a basic search engine backend through the construction of an **Inverted Index**, a fundamental data structure in information retrieval systems such as **Google Search**. The main goal is to process a collection of text files and efficiently map each word to the files in which it appears, along with the frequency of its occurrences. Subsequently we had to analyze and compare the performance of a Java-based application using the **Hadoop** framework with that of a Python-based application using the **Spark** framework. Finally we have to build a **search query** in Python based on the inverted indexes produced.

We tried to make the code as optimized as possible by doing a lot of tests to try to optimize the execution time and memory usage.

2 Equipment

The tests were conducted on three identical virtual machines, each configured with the following hardware and software specifications:

- **CPU**: 2 virtual CPUs (vCPUs), mapped to Intel(R) Xeon(R) Silver 4208 CPU @2.10GHz, provided via KVM virtualization
- **RAM**: 6.8 GB of system memory
- **Disk**: 40 GB of allocated virtual storage (ext4 filesystem)
- **Operating System**: Ubuntu 22.04.1 LTS (Jammy Jellyfish), 64-bit

3 Dataset

We selected a 1583.5 MB corpus of 2685 plain-text files from Project Gutenberg, covering diverse fields including philosophy, science, theology, psychology, literature and other cultural subjects, to stress and tests our indexer across a broad range of real-world texts. This variety tests the system against typical literary content as well as challenging patterns, mirroring real-world search engine demands on both natural language and specialized data. File sizes vary from 5 KB to 250 MB: most are under 1 MB (reflecting typical book chapters or short essays), 329 fall between 1 MB and 7 MB (full-length books) and one extreme outlier ("Human_Genome_Project-Chromosome_1.txt", 250 MB) contains raw nucleotide sequences. Including this genomic text deliberately exposes our inverted-index builder to vast, mostly unique tokens-mimicking workloads with high vocabulary cardinality and ensuring our system handles both common-word skew and near-unique string distributions. By including files ranging from kilobytes to megabytes, the dataset enables a rigorous evaluation of how indexing and search-query systems scale with input size.

4 MapReduce and Hadoop code

The system uses **MapReduce**, via the **Hadoop** framework, to process large-scale data efficiently. The Hadoop cluster is optimized for virtual machines with limited memory through customized YARN and MapReduce settings. YARN manages resources and memory (up to 5 GB per node), while MapReduce configurations allocate 2048 MB to key tasks, with JVM heaps limited ot 1536 MB.

```
class TokenizerMapperStateful

  method initialize()
    word_counts <= New Empty
        AssociativeArray()
  end method

  method map(offset o, doc d)
    Filename <=
        retrieve_file_name()

    for all term t in doc d do
      if word_counts[t] does not
          contain Filename then
        word_counts[t][Filename]
            <= 1
      else
        word_counts[t][Filename]
            <=
            word_counts[t][Filename]
            + 1
      end if
    end for

    if word_counts.size() >
        FLUSH_THRESHOLD then
      flush(context)
    end if
  end method

  method flush(context)
    for each word in word_counts do
      for each filename in
          word_counts[word] do
        value <= filename + ":"+
        +word_counts[word][filename]

        emit(word, value)
      end for
    end for
    clear word_counts
  end method

  method cleanup(context)
    flush(context)
  end method

end class
```

Figure 1: Stateful Mapper PseudoCode

```
class TokenizerMapper
  method map(offset o, doc d)
    Filename <=
        retrieve_file_name()
    for all term t in doc d do
      emit(term t, filename + ":1")
    end for
  end method
end class
```

Figure 2: Stateless Mapper PseudoCode

```
class DocumentCountReducer
  method reduce(term, postingsList)
    docCounts <= {}
    for all posting in
        postingsList do
      for pair in split(posting,
          ",") do
        doc, cnt <=
            splitLast(pair, ":")
        docCounts[doc]
            docCounts.get(doc, 0)
            + toInt(cnt)
      endfor
    endfor
  emit(term, format(docCounts))
  end method
end class
```

Figure 3: Reducer PseudoCode

The application offers two interchangeable MapReduce variants: a classic **Mapper with optional Combiner** and a **Stateful In-Mapper Combiner**, selectable via command-line flags for modularity and flexibility.

To address Hadoop's inefficiency with many small files, we use `CombineFileSplit` to reduce task overhead.

In Hadoop, the `CombineFileInputFormat` class does not know by itself how to read each individual file within a `CombineFileSplit`. For this reason, it requires a **custom RecordReader** for each combined file. This is the job of `MyCombineTextInputFormat`.

`MyCombineFileRecordReaderWrapper` is a wrapper around LineRecordReader, which reads one line at a time as in a normal Hadoop job. Its main function, however, is another: **it keeps track of the name of the file it is reading from**, using a `ThreadLocal` object. This is essential for an inverted index, because each word read from the line must be associated with the document (i.e. the file) in which it appears.

The `TokenizerMapStateful` class accumulates word counts in memory using a data structure initialized in `setup()`, mapping words to document-specific counts. Once a predefined **threshold** is exceeded, it triggers a **flush**, emitting partial results in the format: $\langle$**word**, **doc-id:count**$\rangle$

Residual data is emitted during `cleanup()`.

In contrast, the `TokenizerMapper` class—lacking in-mapper combining—directly emits key-value pairs of the form:

$$\langle \textbf{word}, \textbf{doc-id:1} \rangle$$

The `CombinerDocCounts` class implements the Combiner logic, aggregating intermediate values by summing occurrences per document:

$$\langle \textbf{word}, \textbf{doc-id:count} \rangle$$

Finally, the `DocumentCountReducer` consolidates all partial counts per word across files and formats the output as:

```
word \t filename1:count1 \t filename2:count2
```

5 Spark code

The Spark-related Python code was implemented based on the functional patterns and structure demonstrated during the lectures. **Apache Spark** is an open-source, distributed data processing engine designed for fast in-memory analytics and large-scale workload orchestration. Spark doesn't strictly use **MapReduce**, but it supports map and reduce operations but runs them within a more flexible **DAG** execution model rather than the rigid two-stage MapReduce paradigm.

Spark jobs were executed on YARN with event logging and the Spark History Server enabled. The configuration included Kryo serialization, dynamic allocation with 3 executors (2 cores and 3GB RAM each), and speculative execution to handle stragglers—ensuring effective resource utilization and monitoring.

RDD_inverted_index_search.py constructs an inverted index using `RDDs`. It loads documents from HDFS via `wholeTextFiles`, generating `(path, content)` pairs. Text is tokenized by lowercasing, removing non-alphanumerics, and splitting into words, producing key-value pairs of the form: $((\textbf{word}, \textbf{docID}), 1)$

These are aggregated using `reduceByKey`, mapped to `(word, (docID, count))`, and grouped by key to produce sorted postings lists. Partitioning is adjusted dynamically (1 partition per 44MB) to ensure workload balance. The final output can be saved as tab-delimited text, JSON, or Parquet, generating a distributed inverted index. This is the format of the final output:

```
word \t filename1:count1 \t filename2:count2
```

However this code had poor performance, performance that we expected better from Spark. So we build another version. So a second version was implemented using Spark `DataFrames`. Spark's **DataFrames** wrap RDDs with a schema and declarative API, letting Spark Catalyst optimizer and Tungsten execution engine apply column-level and query-plan optimizations for far better performance and memory use than raw RDDs.

In the new **inverted_index_search.py**, it first loads each specified path into a unified DataFrame annotated with a filename column, gracefully skipping any unreadable files. It then applies a sequence of Spark SQL transformations: all non-alphanumeric characters are stripped via `regexp_replace`, text is lowercased and split on whitespace and each word is exploded into its own row. Empty tokens are filtered out to ensure data quality. So we have a more optimized **tokenization**. In the next phase, the code groups by word and filename to compute per-document term frequencies, then concatenates these as filename:count strings. A second grouping by word collects and sorts the full postings list into an array, producing one row per unique term with its complete, ordered document list. Finally the output is either written as plain text (with words and tab-separated postings) JSON, or Parquet. This approach leverages Spark's built-in DataFrame optimizations and avoids manual RDD manipulations while delivering a scalable inverted index.

6 Non-parallel code

The non-parallel Python implementation builds an inverted index on a single node using a SPIMI (Single Pass In-Memory Indexing) approach. It parses command-line arguments for local or HDFS input/output URIs, optional size limits, and verbosity via `argparse`. Text files are read (either from HDFS via `hdfs.InsecureClient` or from the local filesystem), normalized (punctuation and underscores replaced by spaces), tokenized, and accumulated in an in-memory index. Every `BLOCK_SIZE` files—or when the size limit is reached—the current postings are flushed to a sorted block file in a temporary directory. After all blocks are written, a multi-way merge using a min-heap combines block files into a single output (written back to local or HDFS), preserving term order and aggregating postings. The script prints a concise summary of block construction time, merge time, total runtime, and memory usage.

7 Search Query System

The `search_query.py` utility offers a unified CLI for querying inverted indexes generated by local, Hadoop or Spark runs. It defines mutually exclusive flags (`--folder`, `--hadoop`, `--spark`, `--spark-rdd`, `--non-parallel`) via `argparse`. Depending on the flag, it loads lines from local files or HDFS (`hdfs.InsecureClient`), then invokes `build_term_offset_index_from_lines` to map each lowercase term to its file-line offset. During interactive querying, input terms are normalized, their postings lists retrieved by offset, and filename sets intersected to yield a sorted result list. The REPL loops until Ctrl+D, printing "No matches found." if empty.

8 Tests and Results

8.1 A first comparison

For the performance evaluation, four versions of the MapReduce implementation were considered: `Hadoop-noimc`[1], `Hadoop-imc`, `Spark-Dataframe` and `Spark-RDD`. Three metrics were used: **Execution time**, **Aggregate resource allocation**[2] and **Shuffle bytes**. The comparison was conducted using datasets of: `128MB`, `256MB`, `512MB`, `1024MB`, and `1583MB`.

From the execution time histograms (Fig. 4) it's clear that Hadoop-noimc combiner performs poorly on both small and large datasets. Spark-RDD, instead, turns out to have comparable results to Hadoop-imc combining and Spark-dataframe for small datasets. However, when increasing the dataset, it becomes even worse than Hadoop-noimc. The execution time of Hadoop-imc and Spark-Dataframe is comparable for each input size. **Non-Parallel Execution**, on the other hand, consistently shows the worst performance as dataset size grows, clearly highlighting the benefits of distributed computing for large-scale data processing. The aggregate resource allocation graph (Fig. 5) closely resembles the execution time graph. This is because aggregate memory usage is strongly correlated with execution time. However, it is also evident that, for the same execution time, Spark programs use more memory than Hadoop ones. This confirms that Spark relies more heavily on in-memory processing, whereas Hadoop performs more disk-based I/O operations. Dividing average aggregate resource allocation by average execution-time, we can obtain average resource allocation. From the following table, relative to the 1583MB case, it is also clearer that the Spark versions exploit more RAM than the Hadoop ones.

Version	Avg. Memory (MB)
NO-IMC	9785.71
IMC	10297.36
DataFrame	11008.17
RDD	12745.41

The last histograms (Fig. 6) are dedicated to shuffle bytes: the amount of MB received by reducers. For small datasets the result is pretty much the same for every version of the application. Starting from 512 MB the shuffle bytes of the RDD version increase significantly until they are extremely more in the version with 1583MB. This is due to the fact that a lot of **wide transformation** are used, such as: `repartition` (used to load balancing data), `groupByKey`, `reduceByKey` and `sortyByKey`.

Indeed, the CPU time[3], relative to the 1583MB case, as shown in the following table, is very low in the RDD-version, due to high number of I/O operations resulting from wide transformations. It is also evident that the CPU time is very high in Hadoop-noimc, the main reason behind this is the high amount of `emit()` called in the map phase.

Version	CPU Time (s)
NO-IMC	1154.49
IMC	642.85
DataFrame	521.85
RDD	82.59

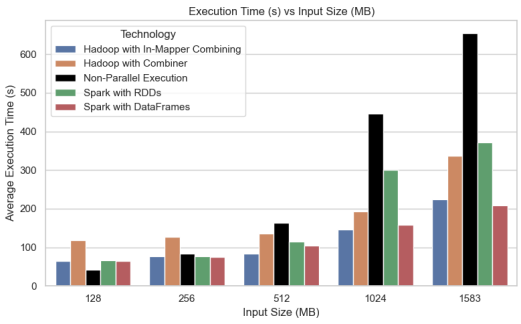

Figure 4: Execution time plot

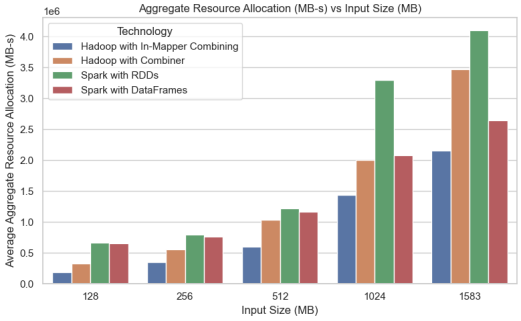

Figure 5: Aggregate resource plot

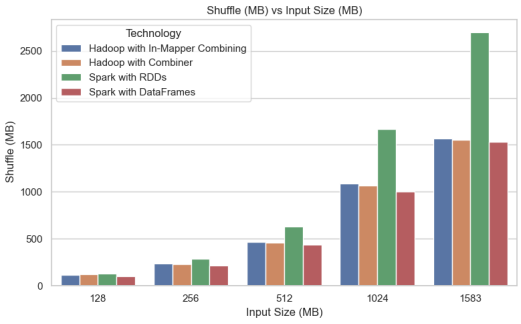

Figure 6: Shuffle plot

[1] IMC stands for *in-mapper combining*

[2] Aggregate resource allocation shows the total amount of primary memory occupied by the application over time [MB·s].

[3] CPU time is the total time the CPU actually executed instructions from a process, summed across all cores used.

8.2 Effect of Reducers on Performance

The Execution Time image (Fig. 7) shows the impact of varying the number of reducers on the execution time of the Hadoop application. As input size increases, the difference in performance becomes more pronounced. For small datasets (128MB, 256MB and 512MB), the reducer count has minimal impact. However, for larger datasets (1024MB and 1583MB), using more reducers generally leads to lower execution times, with the best performance observed when using 4 reducers. Interestingly, performance slightly degrades when moving from 4 to 8 reducers, likely due to overhead from task coordination and context switching. This suggests that, while increasing reducer count improves parallelism and reduces execution time up to a point, beyond that point the benefits diminish or even reverse.

For Aggregate Resource Allocation (Fig. 8), despite some small irregularities in the graph, it is easy to see that there is a pattern: it increases as the number of reducers increases. This behavior is expected, as increasing the number of reducers leads to more concurrent tasks executing in parallel, each requiring its own memory space. As a result, the total memory footprint of the application increases with the number of reducers. Although this allows for better task distribution and reduced execution time, it comes at the cost of higher memory consumption. Therefore, there is a trade-off between parallelism and resource utilization, and selecting the optimal number of reducers involves balancing execution efficiency with memory availability.

Figure 7: Reducers Execution Time

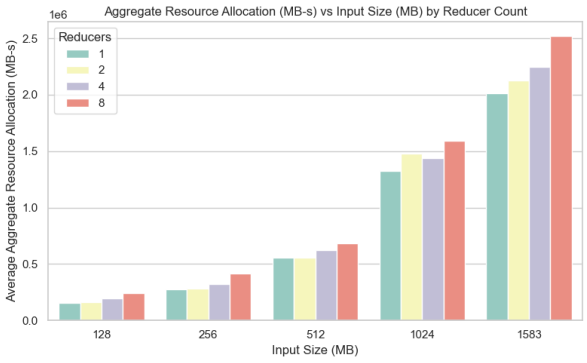

Figure 8: Reducers Aggregate Resource Allocation

8.3 Final Considerations

The comprehensive evaluation across execution time, memory usage and shuffle volume shows that no single implementation universally dominates every metric, but trade-offs emerge clearly. `Hadoop-noimc` suffers from both high CPU time (1154.49 s) and poor scalability, while `Spark-RDD` achieves the lowest CPU time (82.59 s) at the expense of excessive shuffle bytes and aggregate resource allocation (12745.41 MB · s on average). `Hadoop-imc` improves over `Hadoop-noimc` by reducing both execution time and memory footprint, yet still lags behind the Spark-based approaches for large datasets.

`Spark with Dataframes` consistently delivers near-optimal execution times while keeping both aggregate resource allocation (11008.17 MB · s) and shuffle volume moderate. This balance of low execution latency, controlled memory consumption and efficient shuffling makes the DataFrame API the best choice for large-scale word-count workloads on our cluster. Furthermore, tuning the reducer count to four provides additional speedup in Hadoop jobs, but does not bridge the gap to Spark's in-memory processing advantages.

In conclusion, for batch analytics on medium to large datasets, `Spark with Dataframe` offers the strongest overall performance profile. However, if minimizing memory usage is paramount and data volumes remain moderate, `Hadoop with In-Mapper Combining` remains a viable alternative.