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 8, 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 provides a local baseline for comparing with Hadoop and Spark. It builds an inverted index from text files on a single machine, tracking execution time and memory usage for fair performance evaluation with `psutils`.

The **non_parallel.py** script reads .txt files, tokenizes text, and constructs an index mapping words to document counts. Output is written in sorted, tab-delimited format. Optional size-limited processing and resource monitoring are included for controlled testing.

7 Search Query System

The accompanying `search_query.py` script is compatible with inverted indexes produced by all implementations. It uses `build_term_offset_index` to create an in-memory mapping of terms to file offsets, enabling random-access reads during search and minimizing I/O. The `search_optimized` function processes user queries by normalizing terms and retrieving relevant postings. It computes the intersection of filenames across query terms and returns a sorted list, omitting term frequencies in accordance with project requirements. The interface supports continuous querying and exits cleanly on `Ctrl+D`.

Although not scalable like Hadoop or Spark, the non-parallel solution offers a lightweight, easy-to-analyze benchmark for small datasets, helping assess the overhead introduced by distributed frameworks. Its performance metrics are evaluated alongside the parallel implementations in the following section.

8 Tests and Results

...