Cloud Computing - Project Report
Cloud-Busters

Hajar Makhlouf
h.makhlouf@studenti.unipi.it

Lorenzo Vezzani
...@studenti.unipi.it

Francesco Panattoni
f.panattoni2@studenti.unipi.it

May 30, 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**: 39 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

The application is structured to support two interchangeable variants of the MapReduce job: one using a classic **mapper with an optional combiner** and one based on an **statefull in-mapper combiner** approach. This modularity is achieved by dynamically selecting the Mapper class at runtime based on command-line flags, improving flexibility and maintainability.

Hadoop struggles with many small files, as each one generates a separate task using FileInputSplit, leading to significant overhead. To address this, we use **CombineFileSplit**, which merges multiple small files into a single input split, reducing the number of map tasks and improving efficiency. However, CombineFileSplit introduces a challenge: determining which file the Mapper is currently reading.

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.

`TokenizerMapStateful` accumulate counts internally, it maintains a data structure in memory (initialized as emply inside `setup()` method) that associates each word with a map that counts how many times it appears in each document. The actual counting is of course performed inside the `map` method. In order not to occupy too much RAM, a **threshold** has been established. If this is exceeded, which a **flush** operation is performed, that is, all the words with their counts for the various documents are emitted in the format: $\langle$**word, doc-id:count**$\rangle$. In the end, inside `cleanup()` the remaining data in memory is emit.

Instead, the `TokenizerMapper` class takes an input in the key-value form following the structure: $\langle$**offset, doc**$\rangle$. Since this version doesn't provide in-mapper combining, the output is simple as well: it's in the form $\langle$**word, doc-id:1**$\rangle$.

The combiner is implemented in the `CombinerDocCounts` class. It receives from the mapper a series of key-value pairs in the form described above and performs a local combination for each document, adding the occurrences of the word in the single document. This local aggregation leads to an intermiadate output in the form: $\langle$**word, doc-id:count**$\rangle$.

The `DocumentCountReducer`, finally, has the purpose of gathering all the occurrences of the same word, of finishing the sum relative to the single documents (since large files can be stored on different nodes) and of formatting the output correctly:

$\langle$**word**, [**doc-1:count**, ..., **doc-n:count**]$\rangle$.

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.

RDD_inverted_index_search.py was the first version of the **Spark** code. Its **build_index** method begins by reading one or more text files from **HDFS** into Spark as an **RDD** of (filepath, content) pairs using **wholeTextFiles**, then unites these RDDs into a single **collection**. It applies a **flatMap** transformation that lowercases and **tokenizes** each document via a regular expression, counts word occurrences locally with Python's Counter, and emits tuples of the form (word, filename: count). By performing **local counting** before the **shuffle**, the method minimizes network traffic. Next, it aggregates all partial maps for each word across partitions using **aggregateByKey**, merging dictionaries of filename counts into a complete postings list. The resulting RDD of (word, postingsDict) is converted into tab-delimited strings like:

"word f̂ile1:count1 f̂ile2:count2"

Finally, it writes the index either as plain text via saveAsTextFile or as structured JSON or Parquet through a Spark DataFrame, producing an efficient, fully distributed inverted index.

Spark configuration sets the cluster manager to YARN and enables event logging, storing logs in HDFS. It activates the history server with logs updated every 10 seconds and accessible via port 18080. Executor resources are set to 2 cores and 3 GB of memory, while the driver is allocated 1 GB of memory plus 1536 MB of overhead.

However this code had poor performance, performance that we expected better from Spark. So we build another version. 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.

This code works better with lower Spark configurations. Executor resources are set to 2 cores and 2 GB of memory, while the driver is allocated 1 GB of memory plus 512 MB of overhead.

6 Non-parallel code

...

7 Test and Results

...