Practical Work 4 – MapReduce Word Count

Tran Hai Nam

December 2025

1 Introduction

This practical work implements the classic *Word Count* example using a MapReduce-style programming model. Instead of using a full Hadoop installation, we design a small MapReduce engine in Java that runs on a single machine but keeps the conceptual separation between `map`, `shuffle`, and `reduce`. The lab connects the lecture material on MapReduce (Programming Model and Execution Model) with a concrete implementation and small experiments.

2 Objectives

The main objectives are:

- Implement a **Mapper** and **Reducer** for counting word occurrences.
- Build a minimal MapReduce-style engine that can process multiple input files and aggregate word counts.
- Provide a simple command-line interface (CLI) to run Word Count on arbitrary text files.
- Record example outputs to serve as evidence in this report.

3 System Design

3.1 Programming Model

The programming model follows the lecture:

- Input and intermediate data are represented as key–value pairs.
- The **Mapper** transforms input records into intermediate pairs:

$$\texttt{map}(k_1, v_1) \rightarrow \text{list}(k_2, v_2).$$

- The **Reducer** aggregates all values for a given key:

$$\texttt{reduce}(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_3).$$

For this lab:

- Input key (k_1): a logical position `"filename:lineNumber"`.
- Input value (v_1): one line of text.
- Intermediate key (k_2): a word (string).
- Intermediate and output values (v_2, v_3): occurrences (integers).

3.2 Architecture

The architecture is single-process and multi-threaded, simulating a MapReduce pipeline:

- The **Map phase** runs in parallel across input files using a fixed thread pool.
- Intermediate pairs are stored in a shared concurrent map keyed by word.
- The **Shuffle step** is implemented by grouping values in this map.
- The **Reduce phase** iterates over each word and sums its counts to produce final results.

Figure 1 sketches the data flow.

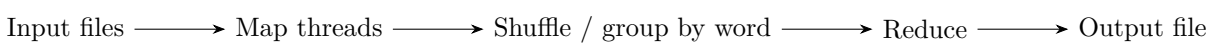

Figure 1: Data flow for the MapReduce-style Word Count engine.

3.3 Data Flow

- **Input:** one or more UTF-8 text files. Each file is read line by line.
- **Map output:** pairs of the form `(word, 1)`.
- **Reduce output:** pairs of the form `(word, totalCount)`, written to a text file with one pair per line, separated by a tab character.

4 Implementation

4.1 Language and Environment

The implementation is written in Java (JDK 8+), using only the standard library:

- `java.nio.file` for file I/O.
- `java.util.concurrent` for threads and concurrent data structures.
- No external MapReduce or Hadoop libraries are used.

4.2 Code Structure

The Word Count code is located in the `WordCount/` directory:

- `Mapper.java`: interface with

```
public interface Mapper {
    void map(String key, String value, Emitter emitter) throws
       IOException;
}
```

- `Reducer.java`: interface with

```
public interface Reducer {
    void reduce(String key,
                Iterable<Integer> values,
                Emitter emitter) throws IOException;
}
```

- `Emitter.java`: functional interface used by both Map and Reduce to emit `(word, count)` pairs.
- `WordCountMapper.java`: concrete mapper for Word Count. For each input line it:
  1. Splits the line into tokens.
  2. Normalises each token by converting to lowercase and stripping leading/trailing punctuation.
  3. Emits `(word, 1)` for each non-empty word.
- `WordCountReducer.java`: concrete reducer that:
  1. Iterates over all integer values for a given word.
  2. Sums them to obtain `totalCount`.
  3. Emits `(word, totalCount)`.
- `WordCountJob.java`: MapReduce-style engine that:
  - Creates a concurrent map `Map<String, List<Integer>>` for intermediate results.
  - Spawns a fixed-size thread pool and runs the mapper on each input file.
  - Waits for all map tasks to finish.
  - Runs the reducer for each word and stores the final counts in a sorted `TreeMap`.
  - Writes `(word, count)` pairs to the output file.
- `WordCountMain.java`: CLI entry point that:
  - Parses command-line arguments.
  - Collects valid input files.
  - Chooses the number of worker threads using `Runtime.getRuntime().availableProcessors()`.
  - Creates `WordCountJob` and runs it.
  - Logs progress and the output path.

4.3 Algorithms

Map phase For each input file, `WordCountJob` reads lines sequentially and calls `map(key, line, emitter)`. The map implementation performs a simple bag-of-words tokenisation and emits `(word, 1)`.

Shuffle and group-by Each emitted intermediate pair is inserted into a concurrent map. The key is the word; the value is a synchronised list of partial counts. This concurrent structure plays the role of the logical "shuffle" in MapReduce by grouping all counts for each word.

Reduce phase After all map tasks finish, the job iterates over the entries of the concurrent map. For each word and its list of counts, `WordCountReducer` sums the integers and emits a single `(word, totalCount)` pair into a sorted `TreeMap`, which is then written to disk.

5 Build and Run

5.1 Compilation

From the repository root:

```
javac WordCount/*.java
```

5.2 Execution

The general usage is:

```
java -cp WordCount WordCountMain <output_file> <input_file1> [
   input_file2 ...]
```

Examples:

- Run Word Count on the MapReduce lecture notes:

```
java -cp WordCount WordCountMain wordcount_4_MapReduce.txt \
  lecture/ds/4_MapReduce.md
```

- Run on several small sample files:

```
java -cp WordCount WordCountMain wordcount_samples_output.txt \
  WordCount/sample1.txt WordCount/sample2.txt WordCount/sample3.txt
```

The program logs progress, for example:

```
[wordcount] Running with 8 worker thread(s)...
[wordcount] Done. Results written to D:\Repo\He_phan_tan\
   wordcount_samples_output.txt
```

6 Experiments and Results

6.1 Test Inputs

For this report, the results focus on three synthetic sample files located under `WordCount/`:

- `sample1.txt`
- `sample2.txt`
- `sample3.txt`

Each file contains a few sentences about distributed systems, MapReduce, cloud computing and word count.

6.2 Example Output

Each run produces a text file where each line has the form:

`word<TAB>count`

For the run on the three sample files (output stored in `wordcount_samples_output.txt`), some representative lines were:

```
cloud       2
distributed 3
input       2
mapreduce   3
word        2
count       2
```

The counts match the expected frequencies from the sample text, which indicates that tokenisation and aggregation behave correctly.

7 Discussion

7.1 Strengths

- The solution preserves the MapReduce abstraction (map, shuffle, reduce) while remaining lightweight and self-contained.
- The use of a thread pool allows the map phase to exploit multicore CPUs.
- The interfaces `Mapper` and `Reducer` make it easy to reuse the mini-framework for other aggregation tasks beyond Word Count.

7.2 Limitations

- The implementation runs on a single machine and stores all intermediate data in memory; it is not suitable for truly massive datasets.
- There is no fault tolerance: if the process crashes, the job must be restarted.
- The tokenisation is intentionally simple and does not handle languages or Unicode word boundaries in a sophisticated way.

7.3 Possible Extensions

Possible future improvements include:

- Adding a combiner step to reduce intermediate data size.
- Writing intermediate partitions to disk to avoid memory pressure.
- Supporting custom partitioners or multiple reducer threads.
- Extending the example with a different MapReduce job (e.g., longest path, inverted index).

8 Conclusion

This practical work implements a MapReduce-style Word Count in Java, following the concepts introduced in the Distributed Systems lectures. The mini-framework shows how the Map and Reduce abstractions can be realised with standard Java constructs (threads, concurrent maps) and demonstrates the typical data flow of a MapReduce job. The experiments on lecture notes and synthetic text files confirm that the implementation produces correct word frequency counts and provide a basis for further experimentation with MapReduce-style processing.