Practical Work #4 – Word Count (Mini MapReduce)

Nguyen Tien Dung – 23BI14116

December 4, 2025

1 Goal of the practical work

The goal of this practical work is to implement a *Word Count* application using a MapReduce-style design. Instead of using a full Hadoop or MapReduce framework, I build a small "mini MapReduce" engine in C++ that follows the classical Map–Shuffle–Reduce pattern.

The objectives are:

- Read text data from one or more input files.
- Apply a **Mapper** to emit key/value pairs `(word, 1)`.
- Group and aggregate pairs by key in a **Reducer**.
- Write the final list of `(word, count)` pairs to an output file.
- Document the design and implementation steps clearly.

2 Environment and approach

The implementation is done in C++ on the same Kali Linux VM used in previous practical works. There is no existing MapReduce framework for C/C++ in the lab setup, so I designed and implemented a small, single-process MapReduce pipeline:

- **Map phase**: process input lines, tokenize them into words, and emit `(word, 1)` pairs.
- **Shuffle / Sort**: collect all intermediate pairs and sort them by key.
- **Reduce phase**: scan the sorted list and aggregate counts for each unique word.

This design is not distributed, but it follows the same logical steps as a real MapReduce job and is sufficient for the purpose of the practical work.

3 System organization

The Word Count implementation is contained in a single C++ source file:

- `wordcount.cpp`: implements the mini MapReduce engine for Word Count.

The program is invoked as:

```
./wordcount <output_file> <input_file1> [input_file2 ...]
```

For example:

```
./wordcount output.txt input1.txt input2.txt
```

4 Mini MapReduce design

4.1 Key/value structure

I represent intermediate and final results as a simple structure:

Listing 1: Key/value struct for word counts

```
struct KeyValue {
    std::string key;
    int value;
};
```

Each intermediate pair produced by the Mapper is of the form:

- `key` = normalized word.
- `value` = 1.

4.2 Word normalization and tokenization

To handle words in a consistent way, I normalize them:

- Convert all letters to lowercase.
- Keep only alphanumeric characters.
- Treat any non-alphanumeric character as a separator.

The function `normalize_word` implements this:

Listing 2: Word normalization

```
std::string normalize_word(const std::string &w) {
    std::string res;
    res.reserve(w.size());
    for (unsigned char c : w) {
        if (std::isalnum(c)) {
            res.push_back(std::tolower(c));
        }
    }
    return res;
}
```

5 Implementation

5.1 Mapper: map_line

The Mapper operates on a single line of text and emits a list of `(word, 1)` pairs:

Listing 3: Mapper function for a single line

```
std::vector<KeyValue> map_line(const std::string &line) {
    std::vector<KeyValue> out;
    std::string current;

    for (unsigned char c : line) {
        if (std::isalnum(c)) {
            current.push_back(c);
        } else {
            if (!current.empty()) {
                std::string w = normalize_word(current);
```

```
            if (!w.empty()) {
                out.push_back({w, 1});
            }
            current.clear();
        }
    }
    if (!current.empty()) {
        std::string w = normalize_word(current);
        if (!w.empty()) {
            out.push_back({w, 1});
        }
    }

    return out;
}
```

Behaviour.

- The line is scanned character by character.
- A "current word" buffer collects alphanumeric characters.
- When a non-alphanumeric character is seen, `current` is normalized and emitted as `(word, 1)`.
- At the end of the line, any remaining word is also emitted.

5.2 Reducer: reduce_all

After the Map phase, I obtain a large vector of intermediate pairs. To aggregate counts per word, I:

1. Sort the vector by `key`.
2. Scan through the sorted list and sum the values for each group of identical keys.

This is implemented in `reduce_all`:

Listing 4: Reducer function to aggregate counts

```
std::vector<KeyValue> reduce_all(std::vector<KeyValue> &intermediate) {
    std::vector<KeyValue> result;
    if (intermediate.empty()) return result;

    std::sort(intermediate.begin(), intermediate.end(),
              [](const KeyValue &a, const KeyValue &b) {
                  return a.key < b.key;
              });

    std::string current_key = intermediate[0].key;
    int current_sum = 0;

    for (const auto &kv : intermediate) {
        if (kv.key == current_key) {
            current_sum += kv.value;
        } else {
            result.push_back({current_key, current_sum});
            current_key = kv.key;
            current_sum = kv.value;
        }
```

```
    }
    // push the last accumulated key
    result.push_back({current_key, current_sum});

    return result;
}
```

5.3 Main program

The `main` function orchestrates the Map and Reduce phases:

Listing 5: Main function of wordcount.cpp

```
int main(int argc, char *argv[]) {
    if (argc < 3) {
        std::cerr << "Usage: " << argv[0]
                  << " <output_file> <input_file1> [input_file2 ...]\n";
        return 1;
    }

    std::string output_file = argv[1];

    // ------------- Map phase -------------
    std::vector<KeyValue> intermediate;

    for (int i = 2; i < argc; ++i) {
        std::string input_file = argv[i];
        std::ifstream in(input_file);
        if (!in) {
            std::cerr << "Error: cannot open input file: "
                      << input_file << "\n";
            continue; // skip files that cannot be opened
        }

        std::string line;
        while (std::getline(in, line)) {
            auto kvs = map_line(line);
            intermediate.insert(intermediate.end(), kvs.begin(), kvs.end());
        }
        in.close();
    }

    std::cout << "[MapReduce] Mapped " << intermediate.size()
              << " key-value pairs.\n";

    // ------------- Reduce phase -------------
    auto result = reduce_all(intermediate);
    std::cout << "[MapReduce] Reduced to " << result.size()
              << " unique words.\n";

    // ------------- Write output -------------
    std::ofstream out(output_file);
    if (!out) {
        std::cerr << "Error: cannot open output file: "
                  << output_file << "\n";
        return 1;
    }
```

```
    for (const auto &kv : result) {
        out << kv.key << "␣" << kv.value << "\n";
    }
    out.close();

    std::cout << "[MapReduce]␣Result␣written␣to:␣" << output_file << "\n";
    return 0;
}
```

6 Build commands

To compile the program with g++:

Listing 6: Compilation command

```
g++ -Wall -O2 wordcount.cpp -o wordcount
```

This produces the executable `wordcount` in the same directory.

7 Execution and test results

7.1 Creating test input files

I created two small input files to test the Word Count:

```
echo "hello␣world␣hello␣suikawaii" > input1.txt
echo "world␣of␣distributed␣systems␣world" > input2.txt
```

7.2 Running the program

I then ran the mini MapReduce Word Count:

```
./wordcount output.txt input1.txt input2.txt
```

The console output shows how many intermediate pairs were mapped and how many unique words were found after reduction:

```
[MapReduce] Mapped 9 key-value pairs.
[MapReduce] Reduced to 6 unique words.
[MapReduce] Result written to: output.txt
```

7.3 Inspecting the output

The file `output.txt` contains the final counts in the format `word count`. For the test above, a typical output is:

```
distributed 1
hello 2
of 1
suikawaii 1
systems 1
world 3
```

This matches the expected counts:

- `hello`: appears 2 times.
- `world`: appears 3 times.
- `suikawaii`, `distributed`, `systems`, `of`: appear once.

8 Discussion

Even though the implementation runs in a single process, it closely follows the MapReduce model:

- The **Mapper** is a pure function from a line to a list of `(word, 1)` pairs.
- The **Shuffle/Sort** stage groups all intermediate pairs by key using `std::sort`.
- The **Reducer** aggregates all values for a given key into a total count.

This design could be extended in several directions:

- Using threads or MPI to parallelize the Map phase across multiple input files.
- Adding stop-word filtering or stemming to the normalization function.
- Adapting the code to process very large files in chunks.

For the scope of this practical work, the current implementation is simple, deterministic and easy to explain.

9 Personal work

For this Word Count practical work I:

- Designed a mini MapReduce pipeline (Map, Shuffle/Sort, Reduce) in C++.
- Implemented the Mapper, Reducer, and main orchestration logic in `wordcount.cpp`.
- Wrote normalization and tokenization code to handle words in a robust way.
- Compiled and tested the program with multiple input files.
- Prepared this report describing the design, code structure, build commands and test results.

10 Conclusion

In this practical work I implemented a Word Count application following the MapReduce paradigm. By writing a small MapReduce engine in C++, I could focus on the conceptual steps of mapping, grouping and reducing, without relying on an external framework. This exercise connects well to distributed data processing concepts while remaining lightweight and easy to run in a simple lab environment.