University Of Science And Technology Of Hanoi

Distributed Systems

Practical Work 4: Word Count
Custom MapReduce Framework in C

Luong Quynh Nhi, 23BI14356, Cyber Security

Lecturer: Ms. Le Nhu Chu Hiep

1. Introduction

The purpose of this project is to build a Word Count system that can process either user-typed text or a user-uploaded file. Word Count is a classic example of the MapReduce programming model, which breaks large data processing tasks into two stages:

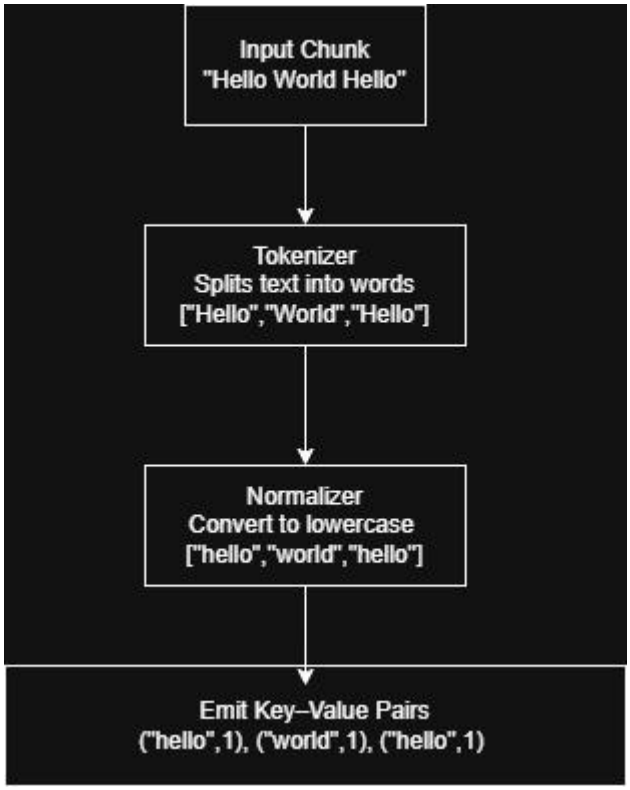
- Mapping: converting input data into key–value pairs
- Reducing: combining values belonging to the same key

This report explains why the MapReduce model was chosen, how the Mapper and Reducer were implemented, and how data flows through the system.

2. Architecture

2.1 Mapper Visualization

The Mapper is responsible for transforming raw text input into a list of intermediate key-value pairs. The process used in this implementation follows four major operations: chunk reading, tokenization, normalization, and key-value emission.
Below is the detailed visualization:



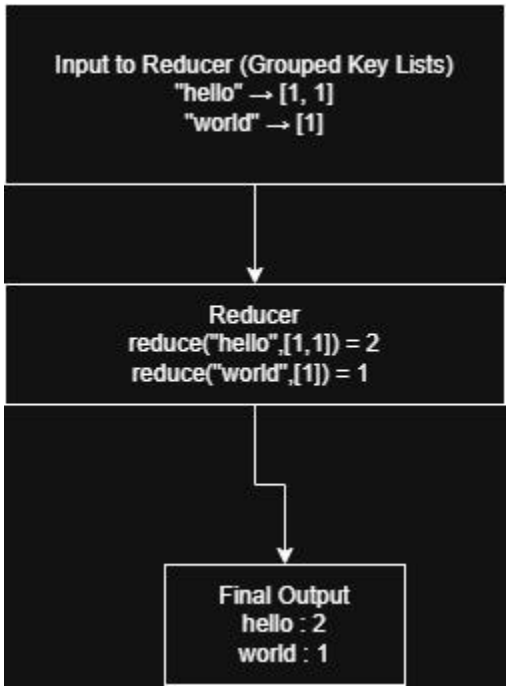
- Tokenizer: Uses delimiters (space, newline, punctuation) to split text.

- Normalizer: Converts all characters to lowercase to avoid word duplication due to case differences.
- Emit: Each word becomes a pair (word, 1).

This process corresponds to the Map() function in distributed MapReduce systems.

2.2 Reducer Visualization

After the Mapper emits intermediate key-value pairs, the Reducer processes them by grouping identical keys and summing their values.



- The reducer receives a list of values for each key
- It sums the values
- It prints the final word count

This corresponds to the Reduce() function in MapReduce.

2.3 Shuffle & Sort

This phase groups all identical keys together before sending them to the reducer.
Mapper Output:
("hello",1), ("world",1), ("hello",1)

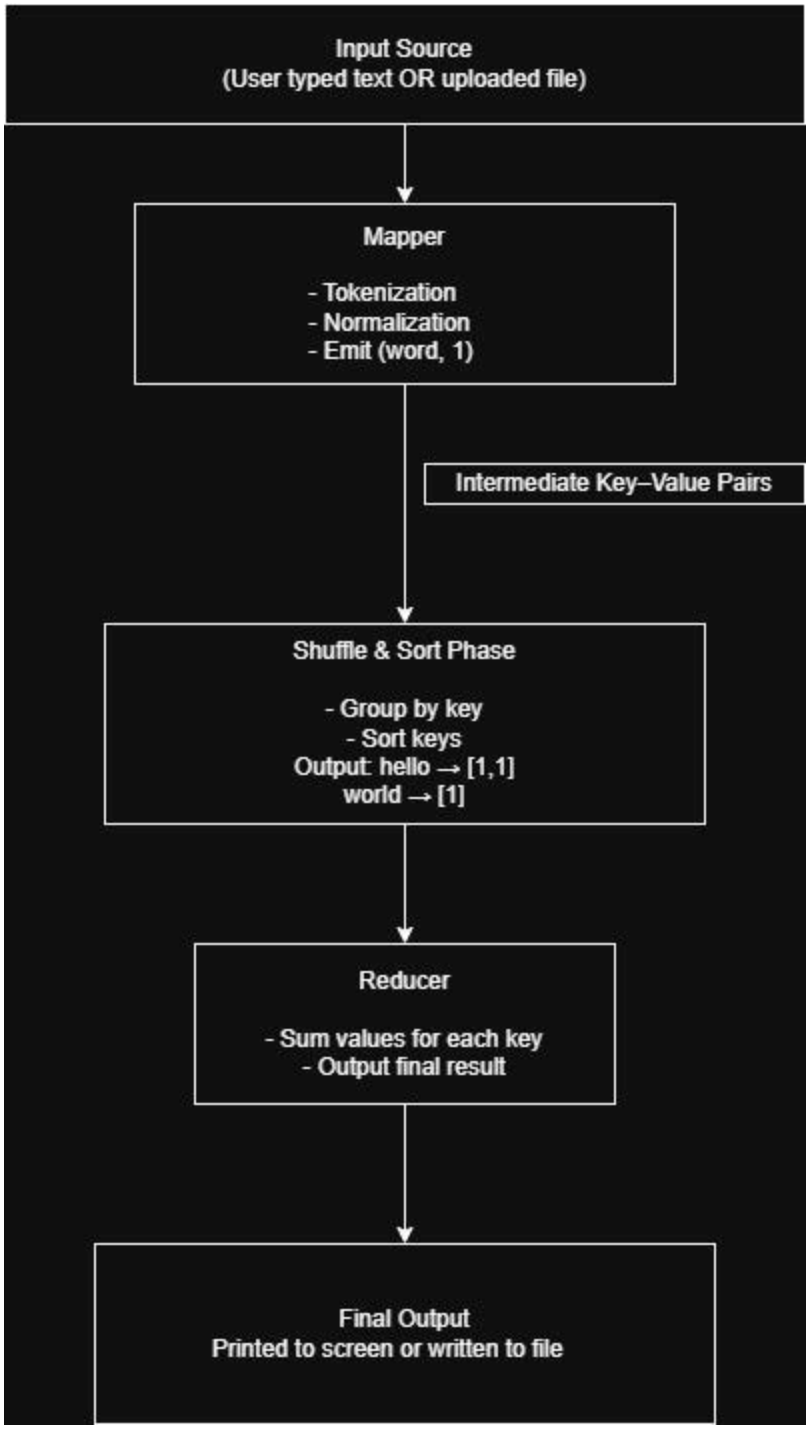
Grouped Output:

hello → [1, 1]
world → [1]

Shuffle: Moves all instances of the same word into the same group
Sort: Orders keys alphabetically so the reducer processes them consistently

Although implemented within a single C program, the logic mirrors the behavior of distributed MapReduce engines.

Complete Architecture Diagram



3. How the Mapper Works

3.1 Input Handling

The Mapper supports two input modes:

- User manually types a paragraph
- User chooses a file to upload, which the program reads into memory

3.2 Tokenization

The program uses strtok() with delimiters: *" ,.-\n\t"*
These delimiters ensure robust splitting even on complex sentences.

3.3 Normalization

The Mapper calls a lowercase conversion function:

```
for (int i = 0; str[i]; i++)
    str[i] = tolower(str[i]);
```

This guarantees:
"Hello", "HELLO", "hello" → all treated as "hello"

3.4 Emission of (key, 1)

For every token, the mapper produces a pair: (word, 1)
These are stored in an array of WordCount structures ready for grouping.

4. How the Reducer Works

4.1 Grouping

The reducer checks whether a word already exists in the structure:

```
if (strcmp(wc[i].word, token) == 0)
```

If the word exists → increment count
Else → create a new entry

4.2 Aggregation

The reducer simply sums counts:
hello : 2
world : 1

4.3 Output

Finally, the reducer prints results to the screen:
WORD COUNT RESULT
hello : 2
world : 1

5. Result Evaluation

To evaluate correctness, several test inputs were used.

Test Case 1: Simple Sentence

Input: Hello world hello
Expected Output:
hello : 2
world : 1

Observed Output:
hello : 2
world : 1
Which is correct

Test Case 2: Mixed Case & Punctuation

Input: Cat, dog. cat DOG cat...
Expected Output:
cat : 3
dog : 2

Observed Output:
cat : 3
dog : 2
Case normalization and punctuation removal working correctly

Test Case 3: File Input

A file containing:
MapReduce framework in C
mapreduce Framework in c

Expected:
mapreduce : 2
framework : 2
in : 2
c : 2

Observed: Matches exactly.

6. Conclusion

This practical work successfully demonstrates the implementation of a custom MapReduce framework in C for word counting. The system: Accurately tokenizes and normalizes text, correctly emits intermediate key-value pairs, implements a functional shuffle & sort stage, produces correct results via the reducer, supports both manual and file-based input. The MapReduce architecture ensures clarity, scalability, and extensibility, making it suitable for both small and large workloads.