Practical Work 4: Word Count

Doan Dinh Khai - 22BA13167

December 24, 2025

MapReduce Implementation Choice

For this practical work, we implemented a Word Count application using a Python-based MapReduce pattern that follows the standard MapReduce programming model. This implementation can run standalone on a single machine or be adapted to work with Hadoop Streaming for distributed execution.

Operating System and Setup: The implementation was developed on Windows 10 using Python 3.x. The system uses standard Python libraries (`sys`, `re`, `subprocess`) and does not require any external MapReduce framework installation for basic operation. This makes it accessible and easy to test without setting up a Hadoop cluster.

Installation and Configuration: No special installation is required beyond Python 3.x. The implementation consists of three Python scripts:

- `mapper.py`: Implements the map phase
- `reducer.py`: Implements the reduce phase
- `word_count.py`: Orchestrates the MapReduce pipeline

For distributed execution, this implementation is compatible with Hadoop Streaming, which allows running any executable as a mapper or reducer. To use with Hadoop, one would run:

```
hadoop jar hadoop-streaming.jar \
  -mapper mapper.py \
  -reducer reducer.py \
  -input input_dir \
  -output output_dir
```

Advantages:

- Simple and lightweight: no complex framework setup required for basic testing
- Follows standard MapReduce pattern: mapper reads from stdin, reducer processes sorted key-value pairs
- Compatible with Hadoop Streaming for scaling to distributed systems
- Easy to understand and modify for educational purposes
- Uses standard Unix-style I/O (stdin/stdout), making it portable

Limitations:

- Single-machine execution does not leverage distributed computing benefits
- No built-in fault tolerance or task retry mechanisms
- Manual orchestration required (handled by `word_count.py`)
- For production use, a full MapReduce framework like Hadoop or Spark would be more appropriate

Mapper and Reducer Design

Input Format: The system accepts plain text files where each line represents a document or text segment. The mapper processes input line by line from standard input, which is the standard MapReduce input pattern.

Mapper Logic: The mapper (`mapper.py`) performs the following steps:

1. Reads each line from standard input
2. Converts text to lowercase for case-insensitive word counting
3. Tokenizes the line using regular expressions to extract words (sequences of alphabetic characters)
4. Filters out punctuation and non-alphabetic characters
5. Emits key-value pairs in the format `(word, 1)` for each word found

The tokenization uses the regular expression `\b[a-z]+\b` to match word boundaries and extract only alphabetic words. This approach:

- Normalizes words to lowercase
- Removes punctuation automatically
- Handles multiple spaces and special characters gracefully
- Does not filter stop words (all words are counted equally)

Intermediate Key-Value Pairs: The mapper emits pairs in the format `word\t1` (tab-separated), where:

- Key: The word (lowercase, alphabetic only)
- Value: The count (always 1 for each occurrence)

Shuffle and Sort Phase: In a full MapReduce framework, the shuffle phase would automatically group all values by key and sort them. In our standalone implementation, the reducer receives sorted input because Python's subprocess pipes maintain order, and the reducer processes keys sequentially.

Reducer Logic: The reducer (`reducer.py`) performs aggregation:

1. Reads key-value pairs from standard input (tab-separated)
2. Maintains state for the current word and its running count
3. When a new word is encountered, outputs the previous word's total count
4. Accumulates counts for the same word
5. Outputs the final word and its total count at the end

The reducer assumes input is sorted by key (word), which is guaranteed by the MapReduce framework's shuffle phase in distributed systems, or by the sequential processing in our standalone implementation.

Output Format: The final output consists of tab-separated pairs `word\tcount`, where each word appears once with its total occurrence count, sorted alphabetically by word.

Data Flow Diagram: The MapReduce word count process follows this flow:

1. **Input**: Text files with multiple lines

2. **Map Phase**: Each mapper processes lines independently, emitting `(word, 1)` pairs
3. **Shuffle/Sort**: Framework groups pairs by word and sorts them
4. **Reduce Phase**: Each reducer receives all pairs for a word (or group of words), sums the counts
5. **Output**: Final word count results `(word, total_count)`

Implementation Details

Programming Language and Libraries: The implementation uses Python 3.x with only standard library modules:

- `sys`: For reading from stdin and writing to stdout
- `re`: For regular expression-based tokenization
- `subprocess`: For orchestrating mapper and reducer processes (in `word_count.py`)

Running the Job: The system can be executed in two ways:
Standalone execution using the orchestrator script:

```
python word_count.py input.txt output.txt
```

This script pipes input through mapper and reducer processes automatically.
Manual execution (simulating Hadoop Streaming):

```
cat input.txt | python mapper.py | sort | python reducer.py > output.txt
```

The `sort` command simulates the shuffle/sort phase that would be handled by the MapReduce framework in a distributed system.

Code Structure:

Listing 1: Mapper implementation

```
import sys
import re

def tokenize(line):
    line = line.strip().lower()
    words = re.findall(r'\b[a-z]+\b', line)
    return words

for line in sys.stdin:
    words = tokenize(line)
    for word in words:
        print(f"{word}\t1")
```

Listing 2: Reducer implementation

```
import sys

current_word = None
current_count = 0

for line in sys.stdin:
    line = line.strip()
    if not line:
        continue
```

```
    parts = line.split('\t', 1)
    if len(parts) != 2:
        continue

    word, count = parts[0], parts[1]

    try:
        count = int(count)
    except ValueError:
        continue

    if word == current_word:
        current_count += count
    else:
        if current_word is not None:
            print(f"{current_word}\t{current_count}")
        current_word = word
        current_count = count

if current_word is not None:
    print(f"{current_word}\t{current_count}")
```

Optimizations:

- **Case normalization**: All words are converted to lowercase in the mapper, ensuring "The" and "the" are counted together
- **Efficient tokenization**: Uses regex for fast word extraction
- **Streaming processing**: Both mapper and reducer process data line by line, minimizing memory usage
- **Error handling**: Reducer includes validation for malformed input lines

For a production system, additional optimizations could include:

- **Combiner**: A local reducer in the mapper phase to reduce network traffic
- **Multiple reducers**: Partitioning output by word hash to parallelize reduction
- **Stop word filtering**: Removing common words (the, a, an, etc.) if not needed
- **Custom partitioner**: For better load balancing across reducers

Group Work

Choice and Installation of MapReduce Framework: The decision to use a Python-based MapReduce implementation was made to keep the solution simple and educational. The team evaluated Hadoop, Spark, and standalone Python approaches, ultimately choosing the Python approach for its simplicity and compatibility with Hadoop Streaming if scaling is needed later.

Implementation of Mapper and Reducer: The mapper and reducer were implemented following the standard MapReduce pattern. The mapper handles text tokenization and normalization, while the reducer performs count aggregation. Both components use standard I/O streams (stdin/stdout) for compatibility with MapReduce frameworks.

Experiments and Testing: Testing was performed using sample text files of various sizes:

- Small files (few KB) for quick validation

- Medium files (hundreds of KB) to verify correctness
- Edge cases: empty files, files with only punctuation, files with special characters

The implementation correctly handles:

- Case-insensitive word counting
- Punctuation removal
- Multiple spaces and line breaks
- Empty lines and malformed input

Performance measurements showed linear scaling with input size, as expected for a MapReduce implementation. The standalone version processes files efficiently for educational purposes, though distributed execution would be needed for very large datasets.

Writing and Formatting of the LaTeX Report: The report was written in LaTeX following the same structure as previous practical works. Code snippets were formatted using the `listings` package for syntax highlighting. The report documents the implementation choice, design decisions, code structure, and testing approach, providing a complete overview of the Word Count MapReduce implementation.