CSE 687 Object Oriented Design Project MapReduce Phase #1

Team: Joseph Laible, Pedro Ortiz, Vimal ramnarain

Abstract

This paper details a standalone MapReduce system the team has designed for the efficient analysis of text data, specifically applying to the works of Shakespeare that was given as the data to use. This system functions within a single process architecture, which employs a command line interface to orchestrate the flow of data from input through processing to output.

Introduction

The architecture of this MapReduce system is structured to facilitate the processing of text datasets. By using a single threaded process, the system incorporates various components, each tasked to handle specific functions within the data handling and processing lifecycle. This design ensures that the system operates efficiently and effectively, adhering to the constraints of object oriented design principles. The development of this system incorporates the four fundamental principles of object-oriented programming encapsulation, abstraction, inheritance, and polymorphism, each playing a crucial role in ensuring the system's modularity and scalability.

System Architecture and Component Functionality

At the core of the system's architecture is the Executive component, which serves as the system's starting point. It manages command line inputs, validating and parsing the paths for input files, temporary storage, and the output directories. This initial step is critical for setting up the environment in which the MapReduce process will operate. Following this setup, the Workflow component takes charge, orchestrating the overall execution of the MapReduce process. It is responsible for initiating the mapping and reducing phases, managing the orderly processing of data. The Workflow component execute() method oversees the sequence of operations that ensures each phase transitions smoothly and efficiently. Supporting the system's operations is the File Management, which abstracts the complexities of file interactions. This component provides robust methods for opening (openFile), reading (readNextBlock), writing (writeFile), and closing (closeFile) files, along with capabilities to manage file deletions and checks for directory existence. These operations are essential for handling the data files that the system processes.

Encapsulation and Abstraction:

The File Management utility demonstrates encapsulation and abstraction by hiding the complexities of file operations from other components. It exposes a simple interface to other parts of the system, which do not need to be aware of the underlying implementations of file access, ensuring the modularity and maintainability of the system. Central to the data processing operation is the MapClass, tasked with the initial data processing phase. This component implements methods to start (start), process data (MapFunction), export processed data (ExportFunction), and conclude the mapping phase (end). The MapFunction handles the tokenization and normalization of text, converting text to lowercase and removing punctuation, and pairs each word with an initial count. The ExportFunction manages these pairs, writing them to a temporary file in batches.

Inheritance and Polymorphism:

While the system primarily focuses on component-based architecture without explicit use of inheritance, the design principles of polymorphism could be reflected in future extensions where different types of processing classes might inherit from a common base, allowing the system to process different types of data or perform different kinds of analyses seamlessly.

After mapping, the Reduce component aggregates the intermediate data to compile the final results. It implements methods to start the reduction phase (start), process the aggregated data (reduce), and conclude the reduction (end). The reduce function specifically processes the intermediate key-value pairs, summing up the occurrences of each word and writing the final counts to the output directory.

Detailed Process Description

The system's functionality is encapsulated within its components methods. In the MapClass, the MapFunction method processes text by breaking it into words, normalizing these words, and pairing them with initial counts. These pairs are managed by the ExportFunction, which efficiently buffers and writes this data to disk. This method of batch processing ensures that disk writes are minimized, enhancing the system's efficiency. In the Reduce class, the reduce method takes over to aggregate the counts of each unique word, compiling these into the final results, which are then written to the output directory. The management of file operations through the File Management utility's methods ensures that data integrity and system efficiency are maintained throughout the process.

Conclusion

The MapReduce system's design provides a functional and efficient method for processing textual data, which is demonstrated by processing of Shakespeare’s works. Each component of the system is specifically designed to perform integral roles within the workflow, ensuring effective and efficient data processing. This architectural approach not only supports the current project requirements but also offers a scalable framework for future phases that are to come.