CS512: Final Project Report
Title: Word count and PageRank using MapReduce

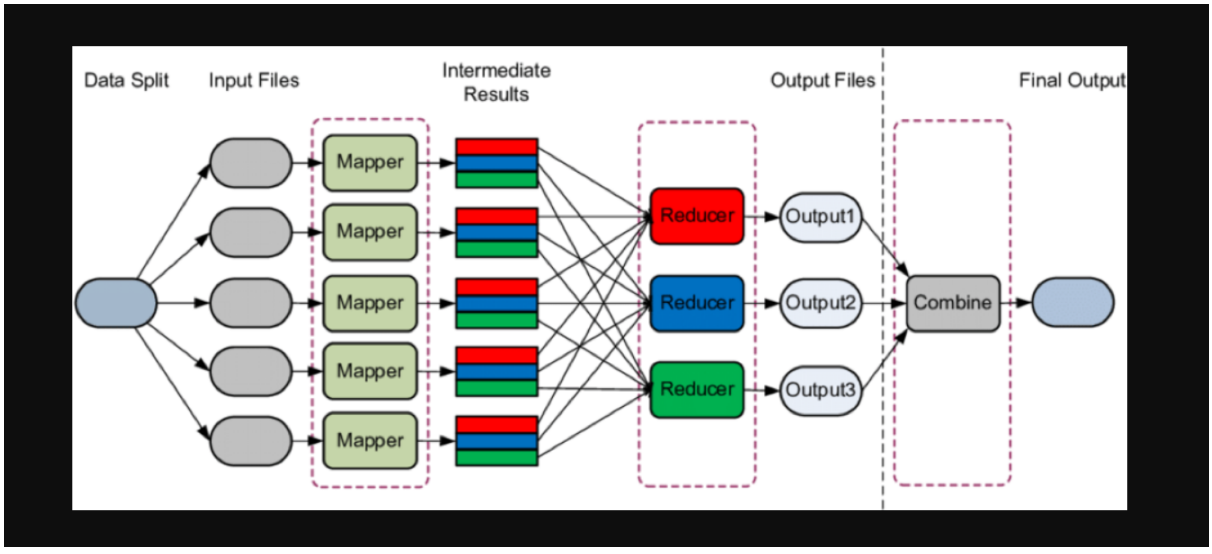
Gunjan Singh (gs896)
Meghana Tumkur Narendra (mt1080)
Praveen Pinjala (pp813)
Shikha Vyaghra (sv629)

Abstract

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

MapReduce is a programming model or pattern within the Hadoop framework that is used to access big data stored in the Hadoop File System (HDFS). It is a core component, integral to the functioning of the Hadoop framework.MapReduce facilitates concurrent processing by splitting petabytes of data into smaller chunks, and processing them in parallel on Hadoop commodity servers. In the end, it aggregates all the data from multiple servers to return a consolidated output back to the application. With MapReduce, rather than sending data to where the application or logic resides, the logic is executed on the server where the data already resides, to expedite processing.

Workflow of MapReduce:



So we try to make use of MapReduce to implement Word count and Page Rank algorithms. We ran the code on ilab machines using hadoop.

Description of the code

1. **Word Count**
   WordCount is a simple application that counts the number of occurrences of each word in a given input set.

   For the word count, we have three files named:
   - ➔ WordCount.java
   - ➔ WordcountMapper.java
   - ➔ WordcountReducer.java

- WordcountMapper.java:

```java
package com.example;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class WordcountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

        public static final IntWritable ONE = new IntWritable(1);

        @Override
        protected void map(LongWritable offset, Text line, Context context)
                        throws IOException, InterruptedException {
                for (String word : line.toString().split(" ")) {
                        context.write(new Text(word), ONE);
                }
        }
}
```

The Mapper implementation, via the WordcountMapper method, processes one line at a time, as provided by the specified TextInputFormat. It then splits the line into tokens separated by whitespaces,, and emits a key-value pair of < <word>, 1>.

- WordcountReducer.java:

```java
package com.example;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class WordcountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

	@Override
	protected void reduce(Text key, Iterable<IntWritable> values, Context context)
			throws IOException, InterruptedException {

		int count = 0;
		for (IntWritable current : values) {
			count += current.get();
		}
		context.write(key, new IntWritable(count));
	}

}
```

The Reducer implementation, via the WordcountReducer method just sums up the values, which are the occurrence counts for each key .

- WordCount.java

    - We first instantiate configuration for our code which will be used by the jobs , which will be created in the next step.
    - In this file we create a job, which allows the user to configure it, submit it, control its execution, and query the state. The set methods only work until the job is submitted, afterwards they will throw an IllegalStateException
    - Our main function initiates a ToolRunner object which is a utility to help run Tools.ToolRunner can be used to run classes implementing Tool interface, which hence calls the run function.
    - The run method specifies various facets of the job, such as the input/output paths (passed via the command line), key/value types, input/output formats etc., in the Job. It then calls the job.waitForCompletion to submit the job and monitor its progress.
    - After creating an instance of job, different methods of job are called. The methods and its functions are as follows:

Job Method	Description
setJarByClass	Set the Jar by finding where a given class came from.
setMapperClass	Set the Mapper for the job.

setReducerClass	Set the Reducer for the job.
setCombinerClass	Set the combiner class for the job.
setOutputKeyClass	Set the key class for the job output data.
setOutputValueClass	Set the value class for job outputs.

- WordCount also specifies a combiner. Hence, the output of each map is passed through the local combiner (which is the same as the Reducer as per the job configuration) for local aggregation, after being sorted on the *key*s.
- We also give the paths for our input and output files in the arguments while running the code. For this purpose we use FileInputFormat .
- FileInputFormat is the base class for all file-based InputFormats.
- InputFormat describes the input-specification for a Map-Reduce job.
- FileInputFormat.setInputPaths - Set the array of Paths as the list of inputs for the map-reduce job.
- FileOutputFormat.setOutputPath - Set the Path of the output directory for the map-reduce job.
- We try to extract the output directory , if it already exists then we delete it to avoid any errors.

2. Page Rank

PageRank is an algorithm used by Google Search to rank web pages in their search engine results. PageRank is a way of measuring the importance of website pages.

For page rank we have the following code files:

- ➔ PageRank.java
- ➔ SortFloatComparator.java
- ➔ Step1Mapper.java
- ➔ Step1Reducer.java
- ➔ Step2Mapper.java
- ➔ Step2Reducer.java
- ➔ Step3Mapper.java

- The input file used for Page Rank includes: the first line as 2 values which are NodesCount (NC) and EdgesCount(EC) separated by a single space, the next NC lines will have the NodeID and URL of the node, the next EC lines will have NodeID and Outlink to the NodeID.

- ➢ Step1:
  - This step is the first step being run by our program. For this step a separate mapper and reducer has been made for a clear understanding of the algorithm.

  - Mapper: Step1Mapper.java reads the initial input file, ignores the first line and the NC next lines and outputs the EC edges <IntWritable, IntWritable> (e.g. <1, 2>).

  - Reducer: Step1Reducer.java receives the EC edges (Each node and an Iterable of its outlinks) and outputs for each node a concatenated list of its outlinks prefixed with an initial page rank (separated by a tab) <IntWritable, Text> (e.g. <1, "1.0 3,2">).

- ➢ Step2:
  - This is the most important step in the page rank algorithm. This code is made to run multiple times i.e. it will be run in iterations, since the output of this step will be used in the first step.
  - Mapper:
    - Step2Mapper.java reads the last output generated (it can be from the first step or from a previous iteration) and outputs for each *[node, rank, "outlink1,outlink2,..."]*:
    - For each outlink it outputs the outlink and rank / len(outlinks):
      - <Text, Text> (e.g. <"3", "0.5">.
    - The node and its outlinks prefixed with an open bracket:
    - <Text, Text> (e.g. <"1", "[3,2]">).

  - Reducer:
    - Step2Reducer.java receives for each node a list of values (the values can be either page ranks or the list of the outlinks), calculates the pagerank by using the formula
      (1 - d) + (d * sum(pageRank))
      and outputs the same structure as the first step:
      <Text, Text> (e.g. <"1", "0.7875 3,2">).

  - ➔ Pseudo-code for the above explanation:

  - ★ Map(Offset, Text):

```
node, rank, outlinks = Parse(Text)

if outlinks == None:
    return

for (outlink in outlinks):
```

```
                Write(outlink, rank / len(outlinks))

        Write(node, '[' + outlinks)
```

★ Reduce(Node, Text[]):

```
        outlinks = []
        totalRank = 0

        for (text in Text):
                if text.startswith('['):
                        outlinks = text[1:]
        else
                totalRank += text

        totalRank = (1 - 0.85) + (0.85 * totalRank)
        Write(Node, totalRank + '\t' + outlinks)
```

This step is made to run y times (The y is given in the arguments) or until a minimum score is met. The score is calculated as follows:
∑i(|lastRanks[i] - newRanks[i]|).

➢ Step3:

- This step basically only needs a mapper and the use of the shuffle sort phase to print the rankings but can use a Reducer to, for example, print the top N ranked pages.
- Shuffling and Sorting in MapReduce: The process of transferring data from the mappers to reducers is known as shuffling i.e. the process by which the system performs the sort and transfers the map output to the reducer as input. The keys generated by the mapper are automatically sorted by MapReduce Framework, i.e. Before the start of reducer, all intermediate key-value pairs in MapReduce that are generated by mapper get sorted by key and not by value. Shuffling-Sorting occurs simultaneously to summarize the Mapper intermediate output.

- Setup:
  - Before mapping, we read the input data (the NC nodes lines) and fill a HashMap with each node and its corresponding url, this is simply to be able to print the url of the pages in the ranking instead of just the node ids.

- Mapper:
  - Step3Mapper.java receives the last rank's output (node, rank, outlinks) and outputs for each:
  - <FloatWritable, Text> (e.g. <1.6375, "http://www.google.com/">).

        - We write the rank as the key so that the shuffle sort phase sorts our entries in a descending order (thus not needing a Reducer).

    - Here to implement the Shuffling and Sorting we have developed our customized SortComparator in SortFloatComparator.java. Since we are dealing with float values, we have customized the sort comparator which will handle the float variables and compare them accordingly. For this purpose job.setSortComparatorClass has been used. setSortComparatorClass defines the comparator that controls how the keys are sorted before they are passed to the Reducer.

- PageRank.java:
    - It is the main driver of the algorithm.
    - The main function calls step1 , step2 and step3 functions and each of the functions work according to the previously discussed steps. Like word count, we have created a job object and used its methods to initialize the mappers and reducers. The function “showResults” is used to display the final result.

Data transformation and usage

1. Word Count

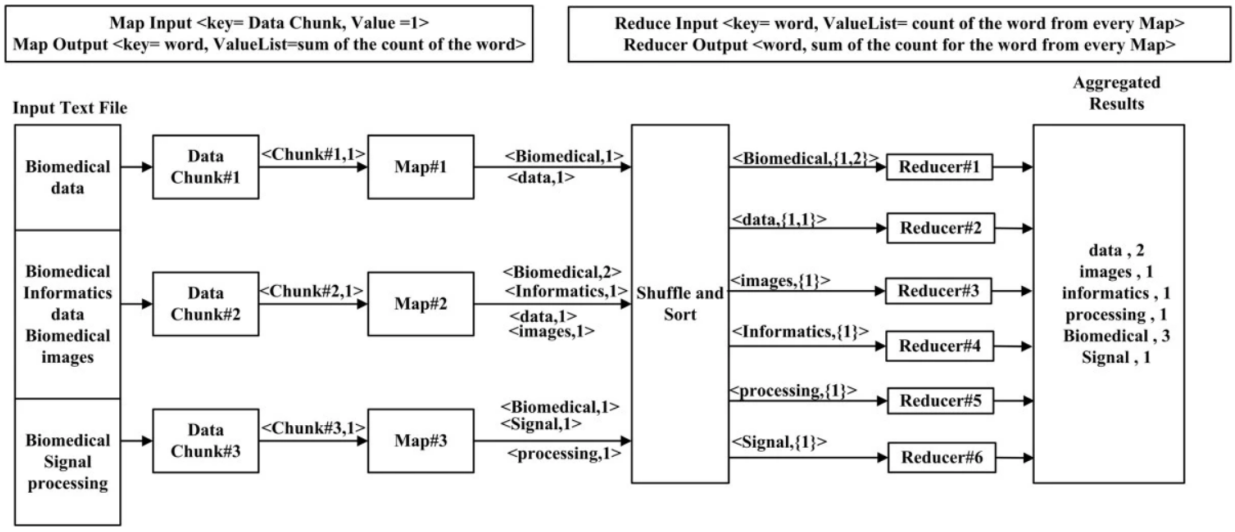
We are using input files which consist of line(s) , each having a bunch of strings. Each string will be considered as a word and our code will count how many times each word has been repeated.

Example: Input text: Hello Hadoop Goodbye Hadoop

Output from the mapper :
< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
< Hadoop, 1>

Final Output:
< Goodbye, 1>
< Hadoop, 2>
< Hello, 1>

Another example with a flow diagram:



2. Page Rank

The input used in this implementation (inputs) is as follows: Note: Each line is 2 values separated by a space.

First line: NodesCount EdgesCount (e.g. "5 9")
NC next lines: NodeID NodeURL (e.g. "1 http://example.com")
EC next lines: NodeID OutlinkToNodeID (e.g. "1 2")

After running the page rank algorithm, the output is displayed with the link , which is given in the input text file, and the calculated rank associated with it.

Time and Space Complexity

For MapReduce algorithms, we can divide the complexity as follows:

- ➢ Key Complexity: This itself consists of three parts:
  - The maximum size of a <KEY, VALUE> pair input to or output by a Mapper/Reducer,
  - The maximum running time for a Mapper/Reducer for a <KEY, VALUE> pair.
  - The maximum memory used by a Mapper/Reducer to process a <KEY, VALUE> pair
- ➢ Sequential Complexity: This time, we sum over all Mappers and Reducers as opposed to looking at the worst.
  - The size of all <KEY, VALUE> pairs input and output by the Mappers and the Reducers,
  - The total running time for all Mappers and Reducers.

1. **Word Count**:

   Assume N documents, M words, total document size S, and word frequencies f1, f2, . . . , f_M. We get the following complexity (assuming batched Reducers):

   - Key complexity: The size, time, and memory are all $O(f_{MAX})$ where f_{MAX}= maximum of all the frequencies .
   - Sequential complexity: The total size and running time are both O(S).

   With streaming Reducers, the key complexity becomes $O(f_{MAX})$ (size and time), and O(1) (memory), whereas the sequential complexity remains the same.

2. **Page Rank:**

   Given a directed graph G = (V, E) with M edges, N nodes, and maximum in- or out-degree d_{MAX}, each iteration of PageRank (assuming each edge is already annotated with the out-degree of its source node) for batched Reducers is:

   - Key complexity: The size, time, and memory are all $O(d_{MAX})$.
   - Sequential complexity: The total size and running time are both O(M).

   With streaming Reducers, the key complexity becomes $O(d_{MAX})$ (size and time), and O(1) (memory), whereas the sequential complexity remains the same

Applications of MapReduce:

- MapReduce is useful in a wide range of applications, including distributed pattern-based searching, distributed sorting, web link-graph reversal, Singular Value Decomposition,web access log stats, inverted index construction, document clustering, machine learning and statistical machine translation. Moreover, the MapReduce model has been adapted to several computing environments like multi-core and many-core systems, desktop grids, multi-cluster, volunteer computing environments, dynamic cloud environments, mobile environments, and high-performance computing environments.

- At Google, MapReduce was used to completely regenerate Google's index of the World Wide Web. It replaced the old *ad hoc* programs that updated the index and ran the various analyses. Development at Google has since moved on to technologies such as Percolator, FlumeJava and MillWheel that offer streaming operation and updates instead of batch processing, to allow integrating "live" search results without rebuilding the complete index.

- MapReduce's stable inputs and outputs are usually stored in a distributed file system. The transient data are usually stored on local disk and fetched remotely by the reducers.