Hadoop MapReduce Fundamentals

Shyam Mohan Kizhakekara
shyam.mk@outlook.com

1. Introduction

Hadoop is a framework that facilitates the storage of Big Data in a distributed environment, so that, it can be processed parallelly. There are two components of Hadoop – HDFS & YARN. HDFS (Hadoop Distributed File System) is used to store data of various formats across a cluster. YARN, on the other hand, is for resource management in Hadoop that enables parallel processing over the data stored across HDFS.

MapReduce is a programming framework that performs distributed and parallel processing on large data sets in a distributed environment. It consists of two distinct tasks – Map and Reduce. In the map phase, a block of data is read and processed to generate a set of key-value pairs as intermediate outputs. When the output of a Mapper is fed to the Reducer as input, it aggregates those intermediate key-value pairs into a smaller set of key-value pairs which is essentially the final output.

The project is all about setting up a Hadoop environment, creating certain jobs to process a given set of input files using MapReduce and collecting their outputs. This report explains each use case in scope, along with its corresponding command/code and output.

2. Part 02 – Hadoop

3.1. Overview

We have three different use cases for which three separate Hadoop jobs are to be developed. The master bash script *"HadoopExecutionPackage.sh"* which executes all the Hadoop jobs is developed as an interactive Unix application. Through this, the user can choose which use-case to execute and the corresponding output would be displayed on the console. Certain property variables used in this script are defined inside the file *"LoadHadoopProperties.sh"* and are loaded at the beginning of the execution.

3.2. Hadoop Jobs & Explanations

[1]. What is the number of distinct words in the corpus? How many words start with the letter Z/z? How many words appear less than 4 times?

- Here, the objective is to develop a single MapReduce job to find the count of unique words in the corpus, count of words starting with z/Z and the count of words appearing less than 4 times.
- To accomplish this**,** we start off with the WordCount example of MapReduce. We would have four classes namely, *HadoopMapper, HadoopReducer, HadoopCombiner* and *HadoopJobControl* defined in four different java files.

HadoopJobController:

- This class controls the entire execution of this job and is the entry point for the processing. This is the class where we define the mapper, combiner and reducer classes. The paths of input files and output files, and any other parameters passed to the jar would be received in this class and processed accordingly.

```
//Counters for the different use cases in Hadoop Q1
public static enum WORDCOUNT_COUNTER {
    KEY_WORDS_UNIQUE,
    KEY_WORDS_STARTING_WITH_Z,
    KEY_WORDS_LESS_THAN_4_TIMES
};
```

- Since we need to find the counts of words in all three tasks, we first define an *enum* with the required counters *KEY_WORDS_UNIQUE, KEY_WORDS_STARTING_WITH_Z & KEY_WORDS_LESS_THAN_4_TIMES*.

- Also, after the reduce phase, we fetch each of these counters and display their values on the console, which would be the required outputs.

HadoopMapper:

- In this class, we have defined the logic for mapping. Just as the WordCount example works, we use the respective word as the key and one (count as one) as the value.

```
/* Removing all the punctuations & special characters from the file
contents and converting it to the lower case. */
String valueString = value.toString().replaceAll("\r\n", "\n")
        .replaceAll("--", " ")
        .replaceAll("[—‘’“”•'\uFEFF]", "")
        .replaceAll(punctTokens, " ")
        .replaceAll("[\\s]+", " ")
        .toLowerCase();
```

- The map function first handles all punctuations, special characters and any whitespaces present int the text as required and then converts the whole text into lowercase.

```
/* Split the valueString, iterate through the different tokens and
map the unique words.    */
String[] valueArray = valueString.trim().split(splitDelim);
for (String element : valueArray) {
    if (element!=null && !element.trim().isEmpty()) {
        word.set(element.trim());
        context.write(word, one);
    }
}
```

- Splitting the text using a whitespace as delimiter and looping through the list of tokens, each token (if not null or empty) would be set as a key for the value one and would be written to the intermediate output file. These key-value pairs would be the input for the reducer.

HadoopCombiner:

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

    int sum = 0;
    for (IntWritable val : values) {
        sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
}
```

- When a MapReduce job is run on a large dataset, the mapper generates large chunks of intermediate data which is fed to the reducer for further processing. This leads to massive network congestion, which can be resolved using a combiner. The primary job of a combiner or mini-reducer, is to process the output data from mapper, before passing it to the reducer. Technically, both the combiner and reducer use the same code logic.

HadoopReducer:

- This class defines the same aggregation logic as defined in the combiner class. It processes the input from mapper (fed through combiner) and increments the counters, eventually forming our final outputs. The final outputs would be written to the output file corresponding to this job and this file would be present inside the output files directory path passed to the jar as one of the input parameters.

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

    context.getCounter(HadoopJobControl.WORDCOUNT_COUNTER.KEY_WORDS_UNIQUE)
            .increment(1);
```

- In MapReduce, the no. of reducer instances is equal to the no. of keys being processed. As such, we can increment the counter *KEY_WORDS_UNIQUE* as soon as a reduce function is invoked.

```
if (key.toString().startsWith("z")) {
    context.getCounter(HadoopJobControl.WORDCOUNT_COUNTER
            .KEY_WORDS_STARTING_WITH_Z).increment(1);
}
```

- If the key (word) being processed, starts with 'z', we increment the defined counter *KEY_WORDS_STARTING_WITH_Z*. We need not consider 'Z' as all the words are already normalized.

```
if (sum < 4) {
    context.getCounter(HadoopJobControl.WORDCOUNT_COUNTER
            .KEY_WORDS_LESS_THAN_4_TIMES).increment(1);
}
```

- If the count of occurrences of a key (word) as calculated by the reducer is less than 4, we can increment the counter *KEY_WORDS_LESS_THAN_4_TIMES.*

Output:

```
// Get the counter values and display them as output
Counters counters = job.getCounters();
Counter cntObj = counters.findCounter(WORDCOUNT_COUNTER.
        KEY_WORDS_UNIQUE);
System.out.println("\nNo. of unique words: " + cntObj.getValue());

cntObj = counters.findCounter(WORDCOUNT_COUNTER.
        KEY_WORDS_STARTING_WITH_Z);
System.out.println("\nNo. of words starting with z/Z: "
        + cntObj.getValue());

cntObj = counters.findCounter(WORDCOUNT_COUNTER.
        KEY_WORDS_LESS_THAN_4_TIMES);
System.out.println("\nNo. of words appearing less than 4 times: "
        + cntObj.getValue());
System.out.println();
```

- This piece of code defined inside the main() method of HadoopJobController class displays the final values of each counter defined earlier, on the console. These values combined with the strings (for easy understanding of the counts), form our final output.

[2]. How many terms appear in only one single document? Such words may appear multiple times in one document, but they must appear in only one document in the corpus.

- Here, the objective is to develop a single MapReduce job to find the count of unique words in the corpus, that appear only in one file, among the set of 6 input files in our scope.
- Like the last use-case, this case also uses the the WordCount example of MapReduce as a baseline. We would have three classes namely, *HadoopMapper, HadoopReducer* and *HadoopJobControl* defined in three different java files.

HadoopJobController:

- This class controls the entire execution of this job and is the entry point for the processing. This is the class where we define the mapper and reducer classes. The paths of input files and output files, and any other parameters passed to the jar would be received in this class and processed accordingly.

```
//Counters for the use case in Hadoop Q2
public static enum WORDCOUNT_COUNTER {
    KEY_WORDS_UNIQUE_TO_A_FILE,
};
```

- Since we need to find the count of words unique to a specific file, we first define an *enum* with a counter *KEY_WORDS_UNIQUE_TO_A_FILE,* which would be used in the reducer.

- Also, after the reduce phase, we fetch this counter and display its value on the console, which would be the final output.

HadoopMapper:

- In this class, we have defined the logic for mapping. Just as the WordCount example works, we use the respective word as the key and its source (name of file) as the value.

```
@Override
public void setup(Context context) throws
    InterruptedException, IOException {

    super.setup(context);

    /* Fetching the source of the contents i.e., the file from which the
    content is obtained */
    localname = ((FileSplit) context.getInputSplit()).
                getPath().getName();
}
```

- Our first step is to obtain the source (document) of a word. For this we define the setup() method and use the getInputSplit() method to identify from which file, the word is obtained.

```
/* Removing all the punctuations & special characters from the file
contents and converting it to the lower case. */
String valueString = value.toString().replaceAll("\r\n", "\n")
        .replaceAll("--", " ")
        .replaceAll("[—‘’“”\•’]", "")
        .replaceAll(punctTokens, " ")
        .replaceAll("[\\W]+", " ")
        .toLowerCase();
```

- The map function then handles all the punctuations, special characters and any whitespaces present in the text and then converts the whole text into lowercase.

```
/* Split the valueString, get the elements in an array and push the
corresponding key and value pairs into the map.   */
String[] valueArray = valueString.trim().split(splitDelim);
for (String element : valueArray) {
    if (element!=null && !element.trim().isEmpty()) {
        word.set(element.trim());
        value.set(localname);
        context.write(word, value);
    }
}
```

- Splitting the text using a whitespace as delimiter and looping through the list of tokens, each token (if not null or empty) would be set as a key for which the value would be its source (file name). This is written to the intermediate output file. These key-value pairs would form the input for the reducer.

HadoopReducer:

- This class defines the aggregation logic for this task. It processes the input from mapper and increments the counter, eventually forming our final output.

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

    HashSet<Text> tempSet = new HashSet<>();

    for (Text val : values) {
        tempSet.add(val);
    }
```

- For each key being processed, keep adding the values (source file names) to a corresponding HashSet. As we know, HashSet will eliminate all the duplicate values when being inserted.

```
if (tempSet.size() == 1) {

    context.getCounter(HadoopJobControl.WORDCOUNT_COUNTER.
            KEY_WORDS_UNIQUE_TO_A_FILE).increment(1);
    result.set(tempSet.iterator().next());
    context.write(key, result);
}
```

- If the size of the HashSet corresponding to a specific key is 1, it means that the set contains only a single element i.e., one file name. Hence, we can conclude that this key (word) appears only in one file and increment the counter.

- The final outputs would be written to the output file corresponding to this job and this file would be present inside the output files directory path passed to the jar as one of the input parameters.

Output:

```
Counters jobCounters = job.getCounters();
Counter counter = jobCounters.findCounter(
        WORDCOUNT_COUNTER.KEY_WORDS_UNIQUE_TO_A_FILE);
System.out.println("No. of Words Unique to a specific File: " +
        counter.getValue());
```

- This piece of code defined inside the main() method of HadoopJobController class displays the final value of the counter defined earlier, on the console. This value is our final output.

[3]. Take one stop-word (e.g., the, and) and compute the five words that appear the most after it. E.g. "the cat belongs to the old lady from the hamlet!", "cat ", "old" and "hamlet" would be candidates. The output should contain 5 lines with the words and their frequency.

- Here, the objective is to develop a single MapReduce job to find the five most frequent words appearing after a specific stop-word.
- As done in the previous use cases**,** we start off with the WordCount example of MapReduce. We would have four classes namely, *HadoopMapper, HadoopReducer, HadoopCombiner* and *HadoopJobControl* defined in four different java files.
- Apart from the paths of input and output files, this job would have an additional input parameter, which is the stop-word. The master bash script executing this jar would receive the stop-word as an input from the user.

HadoopJobController:

- This class controls the entire execution of this job and is the entry point for the processing. This is the class where we define the mapper, combiner and reducer classes. The paths of input files and output files, and the stop-word parameters passed to the jar would be received in this class and processed accordingly.

```
Configuration conf = new Configuration();
conf.set("inputStopWord", args[2]);
Job job = Job.getInstance(conf, "COMP47470-Hadoop Q3");
job.setJarByClass(HadoopJobControl.class);
job.setMapperClass(HadoopMapper.class);
job.setCombinerClass(HadoopCombiner.class);
job.setReducerClass(HadoopReducer.class);
```

- The stop-word obtained as the third input parameter would be confined with the configuration object, for future user in the HadoopMapper class.

- The reduce phase would write the final output to an output file. To display the output on the console, we fetch the output from this file.

HadoopMapper:

- In this class, we have defined the logic for mapping. Just as the WordCount example works, we use the respective word as the key and one as the value.

```
@Override
public void setup(Context context) throws
        InterruptedException, IOException {

    super.setup(context);

    // Get the respective stop-word received as input from the user
    Configuration conf = context.getConfiguration();
    stopWord = conf.get("inputStopWord").toLowerCase();
}
```

- Our first step is to obtain the input stop-word. For this, we define the setup() method inside which we create a configuration object, fetch the stop-word value confined in it and then convert it to lowercase. This is because, we would later normalize all the words.

```
/* Removing all the punctuations & special characters from the file
contents and converting it to the lower case. */
String valueString = value.toString().replaceAll("\r\n", "\n")
        .replaceAll("--", " ")
        .replaceAll("[—‘’“”•']", "")
        .replaceAll(punctTokens, " ")
        .replaceAll("[\\W]+", " ")
        .toLowerCase();
```

- The map function then handles all the punctuations, special characters and any whitespaces present in the text and then converts the whole text into lowercase.

```java
/* Split the valueString, get one word array and compare it against the
stop-word received as input. If they match, then fetch the next word
and set it as the key. Set the value as one. */
String[] valueArray = valueString.trim().split(splitDelim);
String currentToken;
String nextToken;
int j;
for (int i = 0; i < valueArray.length; i++) {
    currentToken = valueArray[i].trim();
    j = i + 1;
    nextToken = (j < valueArray.length) ? valueArray[j].trim() : "";
    if (currentToken.equalsIgnoreCase(stopWord)) {
        if (nextToken != null && !nextToken.isEmpty()) {
            word.set(nextToken);
            context.write(word, one);
        }
    }
}
```

- We split the text using a whitespace as delimiter and loop through the list of tokens. Each token (if not null or empty) would be compared against the input stop-word. If they match, we pick the immediate next word, set it as a key and set one as its value. This is written to the intermediate output file. These key-value pairs generated, would form the input for the reducer.

HadoopCombiner:

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

    int sum = 0;
    for (IntWritable val : values) {
        sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
}
```

- We use this aggregation logic inside the combiner to reduce the possibility of a network congestion and the load on the reducer.

HadoopReducer:

- This class defines the aggregation logic for this task. It processes the input from mapper (fed through combiner) and writes the final output to the corresponding output file.

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

    int sum = 0;

    for (IntWritable val : values) {
        sum += val.get();
    }
    valueMap.put(key.toString(), sum);
}
```

- For each key being processed, keep summing the values and store the final in sum. After this, instead of writing the output to the output file, put this key and its corresponding occurrences count to a HashMap.

```java
// Method to sort the contents of the map based on the values.
public static Map<String, Integer> returnSortedMap(Map<String, Integer> tempMap) {

    Map<String, Integer> sortedMap = new LinkedHashMap<>();
    tempMap.entrySet().stream()
            .sorted(Map.Entry.<String, Integer>comparingByValue().reversed())
            .forEachOrdered(x -> sortedMap.put(x.getKey(), x.getValue()));

    return sortedMap;
}
```

- Define a method to sort the HashMap in descending order based on its values. Note that this method is developed using the features available in Java 8 and hence, having a JDK of version 1.8 or above is very important.

```java
@Override
protected void cleanup(Context context) throws IOException, InterruptedException {

    Map<String, Integer> sortedMap = returnSortedMap(valueMap);
    int c = 0;

    /* Get the map containing the values given by the reducer. Sort the map
    using the method defined for reverse-sorting the obtained map based
    on its values. Once sorted, display the first five key-value pairs. */
    for (String key : sortedMap.keySet()) {

        if (c < 5) {
            result.set(sortedMap.get(key));
            context.write(new Text(key), result);
        } else {
            break;
        }
        c++;
    }
}
```

- Now that all the key-value pairs are available in the defined HashMap, we invoke the sorting method on it. This step is done inside the cleanup() method. Since we need only 5 most frequent words, we write only the first five key-value pairs inside the sorted map to the output file.
- These 5 words and their occurrences would be displayed on the console through the master shell script.