Big Data Programming

Project Report 1

Project Report

University College Dublin

Submitted By

Akshat Rastogi

(17200430)

1. INRODUCTION

This report is the explanation document for the First Project of Big Data Programming Assignment. This will explain the logic along with the commands and code used to achieve the given tasks. I will also be using code snippets/screenshots wherever required.
To easily execute the command, there is a Readme file for explaining steps to execute it. There are two separate sections in the Readme files for the two sections of the project:

1. **CLI for Big Data** – This file is for the explanation text file to execute the bash script for CLI for Big Data Section.
2. **HADOOP** – This file is for the explanation text file to execute the bash script for CLI for Big Data Section.

This project involves text processing and finding counts, insights from a dataset of books. It involves parsing the input data to clean text and generate a list of valid tokens. Once we are ready with the tokens, we can perform a series of actions on them to get insights out of this. The tasks can be completed in several ways, we will be doing the tasks using the Bash on Linux command line interface(CLI) and Hadoop – Map Reduce.

CLI for Big Data: Bash (Bourne- again Shell) – Operating Systems give us the option to work on command line interface, which is faster than the Graphical User Interface and can be automated according to the needs of developer. On Linux installations, the shell is known as Bash which is one of the most primitive ways to process data and generate useful insights out of it. CLI commands can also use piped (|) communication where the output of one command is supplied as input for the next command. We will be using a series of commands to find out the output of our tasks.

Hadoop is an open source Java-based programming framework that supports storage and processing of extremely large data sets on a cluster of commodity hardware. Map Reduce is the programming paradigm and an associated implementation under the umbrella of Apace Hadoop to process and generate Big data sets using a distributive algorithm and a cluster of servers. The idea is to divide the input dataset into independent chunks which can be processed individually.

1.1. CLI FOR BASH

All the command to execute different questions are wrapped under a single bash script to make the script more automated. However, there are no different bash scripts for any single question, instead they are completed using single line commands with piped operations. The reason to write single line command with piped operations is that individual bash scripts for such tasks are inefficient and takes more time to execute as compared to the commands with piped operations. The bash script is used to download files, make necessary input/ output folders and execute commands based on the arguments provided to the script.
As a part of automation of scripts, I am following a set of pre-requisite commands to set up required directories and download files before executing any single command.

Step 1: Validation for Arguments
If no arguments are supplied to the code (arguments supplied are less than 1) then it will exit, and you will need to run it again by providing a set of valid parameters.

Step 2: Creating input directory
It will check if input folder exit or not. If the input directory does not exist, then it will create it.

Step 3: Download the books (Same as the links provided)
The books will be downloaded from a file named as url.txt. It will read the URLs from the file and download them in the input-data folder.

Kindly Note: *The provided URLs were Cache URLs and some of them were not working. Those which were giving junk value have been replaced by their original URL. If you get garbage value for files, download files manually* [A1]

Step 4: Tokenize the words

Text from the downloaded books is now used to create words token for further evaluation. Once the tokens are ready, save them in a temp file (*variable $TEMP which will be used later in commands*) for later use. This is done to avoid tokenizing again and again if multiple questions are executed together. This will save time in tokenizing words instead of tokenizing them again and again. Same words in lowercase or upper case are considered same.

Command to Tokenize text:

```
sed '/^[[:space:]]*$/d;s/[[:space:]]*$//;s/--/ /g;s/[—‘”“•´-]//g;s/^\xEF\xBB\xBF//g' $FILES | tr -s
'[:punct:][:blank:]\r\n' '\n'| tr '[:upper:]' '[:lower:]' > $TEMP
```

The way words are tokenized is explained below:

Detailed Description	Command
1. Remove Blank lines (having one or more spaces)	sed '/^[[:space:]]*$/d’
2. Remove trailing spaces from the text globally with a blank.	sed ' s/[[:space:]]*$// '
3. Replace special characters (not identified as punctuations) with blank globally. It includes [—‘”“•´-]	sed 's/[—‘”“•´-]//g'
4. Remove BOM characters from starting of line (characters which are neither blank nor space but white spaces)	sed 's/^\xEF\xBB\xBF//g'
5. Now, remove the punctuations and blank spaces with a newline character to move every word in a new line. Use of -s will squeeze repeats	tr -s '[:punct:][:blank:]\r\n' '\n'
6. Convert words from upper case to lower case	tr '[:upper:]' '[:lower:]'

Step 5: Use Case - Questions and Solutions

Once the tokens are ready to be used, execute the commands for the questions based on parameter provided while executing the script.

1.1.1. Question 1 – (Output Screenshot after explanations)

a. What is the number of distinct words in the corpus?

Use the temp file to read tokens and sort them alphabetically. After sorting, find unique elements. The total number of output lines will be the distinct number of words in the set of all files.

Command:

```
cat $TEMP | sort | uniq | wc -l

Output: 13095
```

Usage:

1. **cat $TEMP** - Read temp file having tokens
2. **sort –** Sort lines of text file
3. **uniq -** report or omit repeated lines
4. **wc** – print newline, words
   -l: print the newline counts

b. How many words start with the letter Z/z?

Use the temp file to read tokens and sort them alphabetically. After sorting find unique elements. From the list of unique words, find the words starting with z/Z, since the tokens were converted to lower case already, small ‘z’ is enough. The count of the of the number of lines will give the unique words starting with z/Z

Kindly Note: *This output contains count of words starting with z/Z, not the count of their occurrences*.

Command:

```
cat $TEMP | sort | uniq | grep "^[zZ]" | wc -l
```

Output: 4

Usage:

1. **Similar as Part a, Question 1**
2. **grep -** print lines matching a pattern
   ^[zZ]: Regex for lines starting with z or Z

c. How many words appear less than 4 times?

Use the temp file to read tokens and sort them alphabetically. After sorting, find unique elements but this time we need their count as well. From the list of unique words, find sum of lines for words whose frequency count is less than 1 using awk command.

Command:

```
cat $TEMP | sort | uniq -c | awk 'BEGIN{ sum=0} {if
($1 < 4) sum=sum+1} END {print sum}'
```

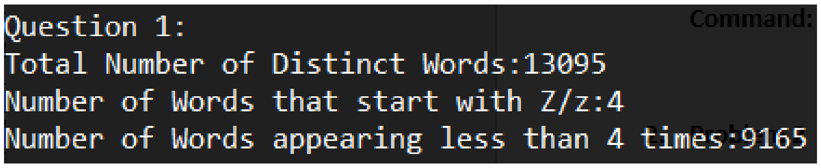
Output: 9165

Usage:

1. **cat $TEMP** - Read temp file having tokens
2. **sort – sort lines of text file**
3. **uniq -** report or omit repeated lines
   -c: prefix lines by the number of occurrences
4. **awk -** pattern scanning and processing language

Awk Parameters(Explained):
Default delimiter: white space
BEGIN{ … } : Executes before parsing data (Variable declaration, sum = 0)
{ … } : Executes for every line (add 1 to sum if count, column 1, is less than 4)
END{ … } : Executes after reading all lines (print sum, count of words occurring less than 4 times)

Output Screenshot:



```
Question 1:
Total Number of Distinct Words:13095
Number of Words that start with Z/z:4
Number of Words appearing less than 4 times:9165
```

1.1.2. Question 2 - What are the most frequent words that end in -ing?

Use the temp file to read tokens and sort them alphabetically. After sorting, find unique elements along with their count(frequencies). From the list of unique words, find the words ending with ing. Sort the results again in descending order based on count. Get top 10 lines to find the most frequent words ending in 'ing'.

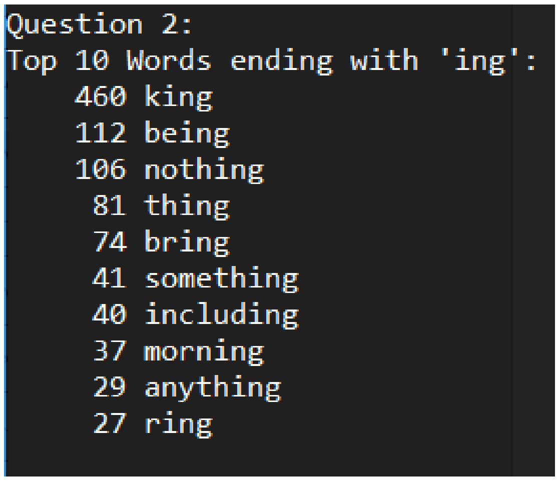
Command:

```
cat $TEMP | sort | uniq -c | grep "ing$" | sort -nr | head
-10
```

Usage:

1. **cat $TEMP** - Read temp file having tokens
2. **sort –** sort lines of text file
3. **uniq** - report or omit repeated line
   -c: find unique with count
4. **sort –** sort lines based on count
   -n: numeric sort
   -r: descending order
5. **head -10 –** get top N (10) lines

Output Screenshot:



```
Question 2:
Top 10 Words ending with 'ing':
    460 king
    112 being
    106 nothing
     81 thing
     74 bring
     41 something
     40 including
     37 morning
     29 anything
     27 ring
```

1.1.3. Question 3 - Which is more frequent: me/my/mine/I or us/our/ours/we?

Use the temp file to read tokens and sort them alphabetically. After sorting, find unique elements along with their count(frequencies). From the list of unique words, find the sum of words in groups me/my/mine/I and us/our/ours/we. Whichever group is more frequent, display that as output.

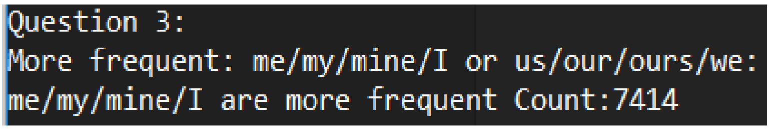
Command:

```
cat $TEMP | sort | uniq -c | awk 'BEGIN{ sum1=0; sum2=0;} {if($2 ~ /^me$|^my$|^mine$|^i$/ ) sum1=sum1+$1;
else if($2 ~ /^us$|^our$|^ours$|^we$/) sum2=sum2+$1} END{ if (sum1 > sum2) printf "\nme/my/mine/I are
more frequent Count:%d",sum1; else printf "\nus/our/ours/we are more frequent Count:%d",sum2;}'
```

Usage:

1. **cat $TEMP** - Read temp file having tokens
2. **sort –** sort lines of text file
3. **uniq -** report or omit repeated line
   -c: find unique with count
4. **awk -** pattern scanning and processing language
   Default delimiter: white space
   BEGIN{ … } : Executes before parsing data (Variable declaration, sum1 = 0, sum2 =0 where sum1 is the count for words in group me/my/mine/I and sum2 is the count for words in group us/our/ours/we)
   { ... } : Executes for every line (add 1 to sum1 if it is me/mine/my/I else add to sum2 if us/ours/our/we)
   END{ … } : Executes after reading all lines (compare sum1 and sum2, if sum1 is more me/mine/my/I are frequent else us/ours/our/we are frequent)

Output Screenshot:



```
Question 3:
More frequent: me/my/mine/I or us/our/ours/we:
me/my/mine/I are more frequent Count:7414
```

1.1.4. Question 4 - Take one stopword (e.g., the, and) and compute the five words that appear the most after it. The output should contains 5 lines with the words and their frequency.

First, read a stop word from the command line when prompted. The stop word will be converted to lower case since the lines will be converted to lower case. Instead of generating every word as token, read books line by line and for any occurrence of the stop word take the next word as token.

Command:

```
printf "Enter stopword:"
read stopword
stopword=`echo $stopword | tr '[:upper:]' '[:lower:]'`

sed '/^[[:space:]]*$/d;s/[[:space:]]*$//;s/--/ /g;s/[—“”“•´-]//g;s/^\xEF\xBB\xBF//g' $FILES | tr '\r\n' '\n' | tr -s
'[:punct:][:blank:]' ' ' | tr '[:upper:]' '[:lower:]' | awk '{ for(i = 1; i < NF; i++) { if($i=="'$stopword'") {i=i+1;print
$i;} } }' | sort | uniq -c | sort -nr | head -5
```

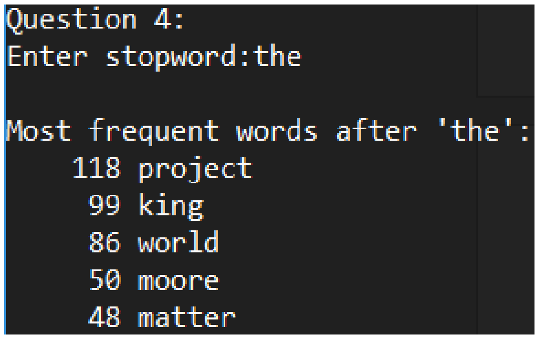
Clean lines are generated using the following logic:

1.	**sed '/^[[:space:]]*$/d’**	Remove blank lines
2.	**sed '/[[:space:]]*$//g’**	Remove trailing spaces from lines
3.	**sed ‘s/[—“”“•´-]//g’**	Remove special characters which are not recognized as punctuations
4.	**sed ‘s/^\xEF\xBB\xBF//g’**	Remove BOM characters from beginning of the line
5.	**tr '\r\n' '\n'**	Replace carriage return and newline with just newline (for windows linux subsystem)
6.	**tr -s '[:punct:][:blank:]' ' '**	Replace punctuations and blanks with a single space (with -s or squeezing repeats)
7.	**tr '[:upper:]' '[:lower:]'**	Convert from upper case to lower case

Once the clean lines are ready, generate tokens and find frequent words after stop word using the following rules:

1. **awk (loop over every word of each line)** - If word in the line matches stop word, print the immediate next word.
2. **sort** - Sort the tokens to bring same words together
3. **uniq** - Find unique words along with their count
   -c: find unique with count
4. **sort –** sort lines based on count
   -n: numeric sort
   -r: descending order
5. **head -5 –** Finally print top lines

Output Screenshot:



```
Question 4:
Enter stopword:the

Most frequent words after 'the':
    118 project
     99 king
     86 world
     50 moore
     48 matter
```

1.2. HADOOP

The next part of this project is to perform the text processing tasks using Map reduce tasks on Hadoop. For successful execution of map-reduce task, we first need to ensure Hadoop is working properly. If it is working properly, then we will create another set of folders in the Hadoop distributed file system (hdfs). We will also download and copy the books in hdfs since we will be reading the input from hdfs only.
For successful execution of Map-Reduce tasks, I have performed some commands using a small bash script. The use of bash script is to automate most of the process.

Step 1: Validation for Arguments
If no arguments are supplied to the code (arguments supplied are less than 1) then it will exit, and you will need to run it again by providing a set of valid parameters.

Step 2: Check if Hadoop is running
Checks if Yarn, dfs and job history server are running or not using jps. It will exit, if they are not running and you will need to run it again by starting all of them.

Step 3: Creating input and output directories
It will check if input directories exit or not. If the input directories do not exist, then it will create them locally and on Hadoop as required. Output directories will be created at runtime while executing commands.

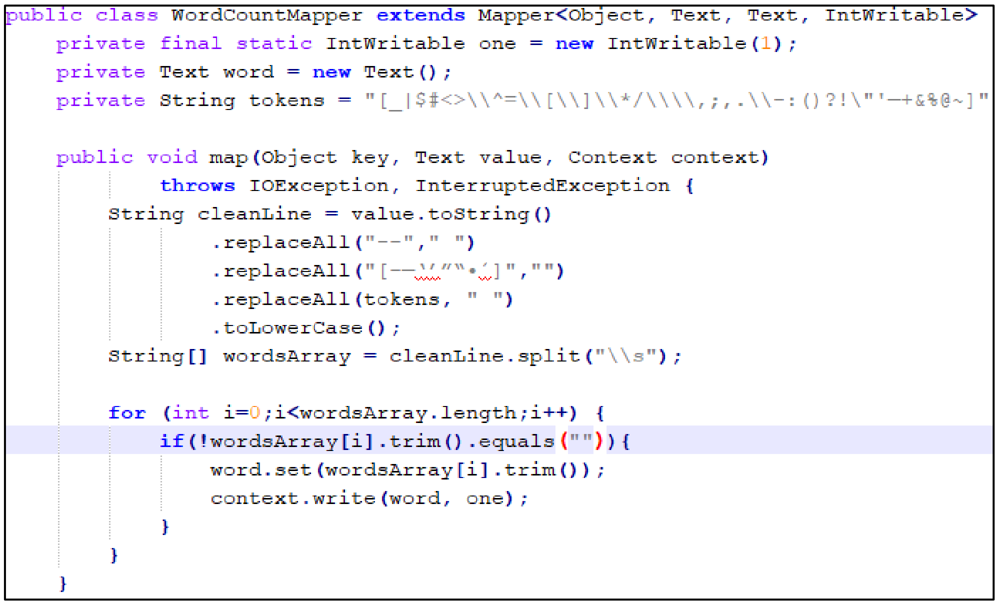
Step 4: Download the books (Same as the links provided)
The books will be downloaded from a file named as url.txt. It will read the URLs from the file and download them in the input-data folder. Once downloaded, they will be copied to hdfs.

Kindly Note: *The provided URLs were Cache URLs and some of them were not working. Those which were giving junk value have been replaced by their original URL. If you get garbage value for files, download files manually* [A1]

Step 5: Tokenize the words (Common Step)
Text from the downloaded books is now used to create words token in the similar fashion we did before in cli bash. Once the tokens are ready, a set of key value pairs can be generated by the mapper class. Same words in lowercase or upper case are considered same. Instead of Java inbuilt string tokenizer, a set of custom rules have been used to generate tokens.

The set of rules for valid token in Map phase (common to all) are: Replace "--" with a space, To make it consistent with the bash script (symbols different than regular punctuation marks) remove [-—‘’”“•´], remove punctuations with space and convert the words to lower case.

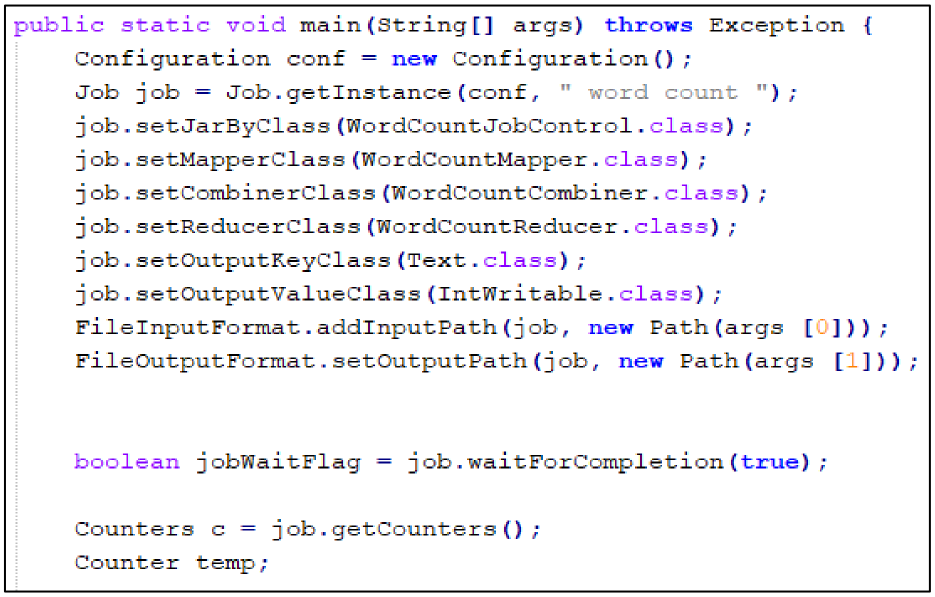
Code Snippet: WordCountMapper.java


```
public class WordCountMapper extends Mapper<Object, Text, Text, IntWritable>
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    private String tokens = "[_|$#<>\\^=\\[\\]\\*/\\\\,;,.\\-:()?!\"'—+&%@~]"

    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {
        String cleanLine = value.toString()
                .replaceAll("--"," ")
                .replaceAll("[–—\\‘’“”•´]","")
                .replaceAll(tokens, " ")
                .toLowerCase();
        String[] wordsArray = cleanLine.split("\\s");

        for (int i=0;i<wordsArray.length;i++) {
            if(!wordsArray[i].trim().equals("")){
                word.set(wordsArray[i].trim());
                context.write(word, one);
            }
        }
    }
```

Explanation:

The Mapper class is responsible for generating key, value pairs. Mapper does not perform any aggregation task but simple generates key, value pairs.
The idea is to divide large chunks into smaller groups so that they can be processed individually.
Here, we are generating word tokens as explained before.

Code Snippet: WordCountJobControl.java


```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, " word count ");
    job.setJarByClass(WordCountJobControl.class);
    job.setMapperClass(WordCountMapper.class);
    job.setCombinerClass(WordCountCombiner.class);
    job.setReducerClass(WordCountReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args [0]));
    FileOutputFormat.setOutputPath(job, new Path(args [1]));


    boolean jobWaitFlag = job.waitForCompletion(true);

    Counters c = job.getCounters();
    Counter temp;
```

Explanation:

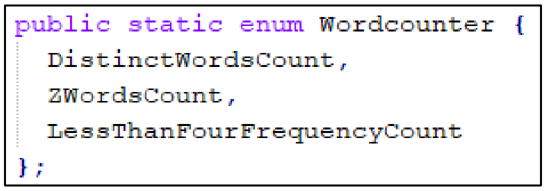
This is the main class and the flow of code starts here, we set the Configuration object, job object here.
After setting the job object we assign mapper, combiner and reducer to it. Type of key and value is also assigned to it.

Now other parameters are set such as Input Path and Output Path

1.2.1. Question 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?

Emit: (Key-Word<Text>, Value- 1<IntWritable>)
As explained before it takes two arguments. It also initializes a static enum counter containing: *DistinctWordsCount*, *ZWordsCount*, *LessThanFourFrequencyCount* for maintaining counts.
In the mapper phase, the class will generate the (key, value) pairs after reading the files line by line. The map phase will generate the (key, value) pairs based on the previously mentioned token generation logic.

Code Snippet: Declaration of counters


```
public static enum Wordcounter {
    DistinctWordsCount,
    ZWordsCount,
    LessThanFourFrequencyCount
};
```

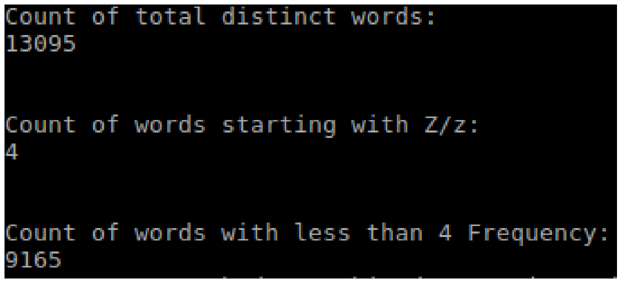
WordCountCombiner.java receives input in the form (key, <value1,value2,value3...>) for each mapper. In the combiner phase, the class will act as reducer but locally and adds up the values corresponding to a key. It emits new modified (key, value) pairs. It is same as reducer without incrementing the counters.

WordCountReducer.java class will find the sum of values for all keys. It receives the values in the form (key, <value1,value2,value3…>) pairs from all the mapper via combiners and adds up the values corresponding to a key to find total occurrences. Since one reducer works per distinct key, the following code snippet shows how to find the required counts. For every key increment *DistinctWordCount* by 1, if the key starts with 'z' or 'Z' increment ZWordsCount by 1 and if the total frequency is less than 4, increment LessThanFourFrequencyCount by 1.

Code Snippet: WordCountReducer.java

```java
public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
        sum += val.get();
    }
    context.getCounter(WordCountJobControl.Wordcounter.DistinctWordsCount).increment(1);
    if(key.toString().startsWith("Z") || key.toString().startsWith("z")){
        context.getCounter(WordCountJobControl.Wordcounter.ZWordsCount).increment(1);
    }
    if(sum<4){
        context.getCounter(WordCountJobControl.Wordcounter.LessThanFourFrequencyCount).increment(1);
    }
    result.set(sum);
    context.write(key, result);
}
```

Output:



```
Count of total distinct words:
13095


Count of words starting with Z/z:
4


Count of words with less than 4 Frequency:
9165
```

Code files for Hadoop Question1:
./data/source-code/Q1/src/wordcount

1. WordCountJobControl.java
2. WordCountMapper.java
3. WordCountCombiner.java
4. WordCountReducer.java

1.2.2. Question 2 - how many terms appear in only one single document?

WordCountJobControl.java accepts two arguments: *Input files Path in hdfs and Output folder path to save the output* as shown above. It also initializes the static enum counter containing: *UniqueWordsInOneDoc*.

Code Snippet: Counter Declaration (Job Control)

```java
public static enum Wordcounter {
        UniqueWordsInOneDoc
    };
```

WordCountMapper.java class will generate the (key, value) pairs after reading the files line by line. The map phase will generate the (key, value) pairs based on the previously mentioned token generation logic but with document id as value.

Code Snippet: Emitting (Key-Word<Text>, Value- DocID<Text>)

```java
for (int i=0;i<wordsArray.length;i++) {
    if(!wordsArray[i].trim().equals("")){
        word.set(wordsArray[i].trim());
        context.write(word, new Text(fileName));
    }
}
```

The setup function will execute before the mappers start executing and in this method, we have set a static variable to find the filename. We will then pass this value with the key while emitting key, value pairs.

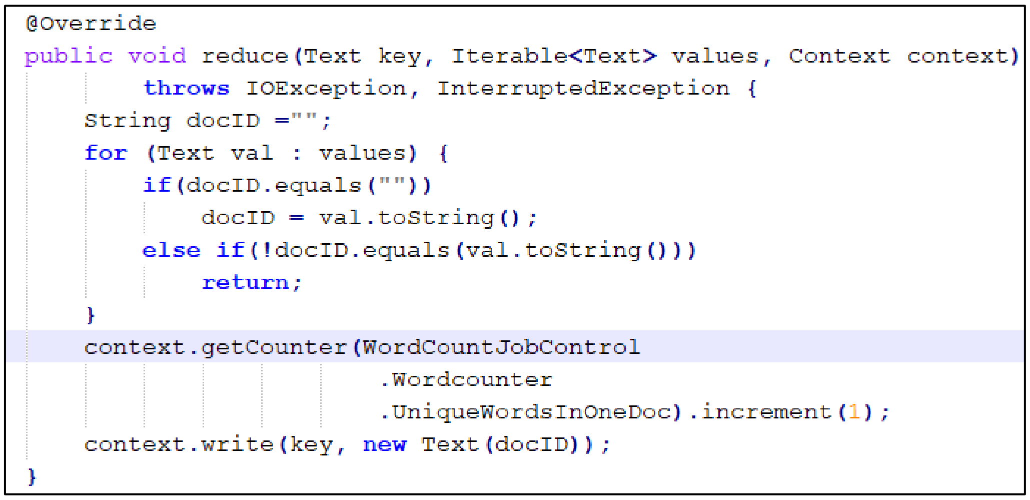
Code Snippet: Setup Method (Mapper)

```java
public static String fileName = new String();
protected void setup(Context context)
            throws java.io.IOException, java.lang.InterruptedException{
    fileName = ((FileSplit) context.getInputSplit()).getPath().getName();
}
```

WordCountCombiner.java receives input in the form (key, <value1,value2,value3…>) for each mapper. In the combiner phase, the class will act as reducer but locally and checks the values corresponding to a key. It emits new modified (key, value) pairs where keys will be having only unique docID. It is same as reducer but do not have the counter.

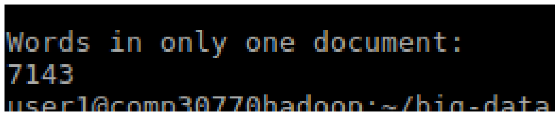
WordCountReducer.java class will check the values for all keys. It receives the values in the form (key, <value1,value2,value3…>) pairs from all the mapper via combiners and if it does not have single value, then it returns without increment the counter else it increments the counter by 1 and write the (Word, docID) in context object.

Code Snippet(Reducer):



```
@Override
public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
    String docID ="";
    for (Text val : values) {
        if(docID.equals(""))
            docID = val.toString();
        else if(!docID.equals(val.toString()))
            return;
    }
    context.getCounter(WordCountJobControl
                        .Wordcounter
                        .UniqueWordsInOneDoc).increment(1);
    context.write(key, new Text(docID));
}
```

Output:



```
Words in only one document:
7143
user1@comp30770hadoop:~/big-data
```

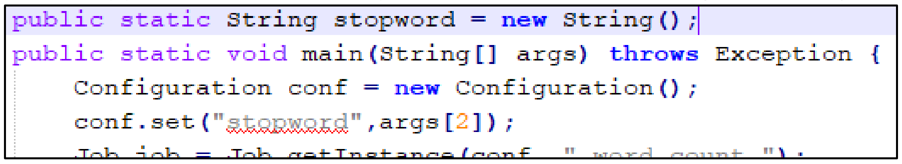
Code files for Hadoop Question2:
./data/source-code/Q2/src/wordcount

1. WordCountJobControl.java
2. WordCountMapper.java
3. WordCountCombiner.java
4. WordCountReducer.java

1.2.3. Question 3 - Compute the five words that appear the most after a stopword

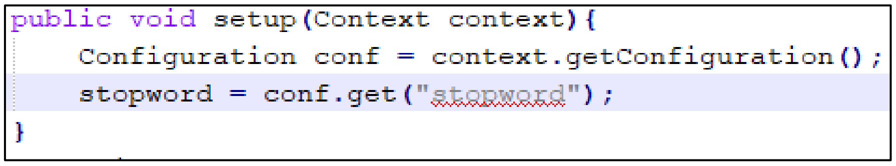
WordCountJobControl.java is now accepting three arguments: *Input files Path in hdfs*(Arg 1)*, Output folder path to save the output*(Arg 2) and the third one is the stop word and set in configuration object so that it could be used later. WordCountMapper.java will generate the (key, value) pairs after reading the files line by line. The map phase will generate the (key, value) pairs based on the previously mentioned token generation logic but instead it will check if the word matches the stop word then it produces the next string as token.

Code Snippet: Read additional parameter and set in configuration object (Job Control)



```
public static String stopword = new String();
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    conf.set("stopword",args[2]);
    Job job = Job.getInstance(conf, " word count ");
```

Code Snippet: Get the value of variable from configuration object in setup method (Mapper)



```
public void setup(Context context){
    Configuration conf = context.getConfiguration();
    stopword = conf.get("stopword");
}
```

Code Snippet: Get the next tokens if word matches stopword (Mapper)

```
for (int i=0;i<(wordsArray.length-1);i++) {
     if(wordsArray[i].trim().equalsIgnoreCase(stopword))
     {
          word.set(wordsArray[i+1].trim());
          context.write(word, one);
     }
}
```

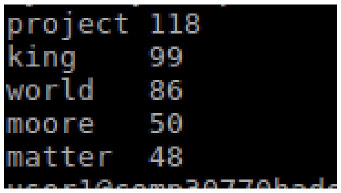
WordCountCombiner.java receives input in the form (key, <value1,value2,value3…>) for each mapper. In the combiner phase, the class will act as reducer but locally and adds up the values corresponding to a key. It emits new modified (key, value) pairs.

In the reducer phase, the reduce method will sum up the count for each key and add them to a static map. It receives the values in the form (key, <value1,value2,value3…>) pairs from all the mapper via combiners. Once all the reducers have completed working, the cleanup method will execute. This method works after all the reducers complete and will sort the keys based on values. It will then write only the first five records.(word, frequency)

Code Snippet(Reducer):

```
public static Map<String, Integer> countMap = new HashMap<>();
@Override
public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
        sum += val.get();
    }
    countMap.put(key.toString(), sum);
}

@Override
protected void cleanup(Context context) throws IOException, InterruptedException {
    Map<String, Integer> sortedMap = new LinkedHashMap();
    countMap.entrySet()
            .stream().sorted(Map.Entry.<String, Integer>comparingByValue().reversed())
            .forEachOrdered(x -> sortedMap.put(x.getKey(), x.getValue()));
    int counter = 0;
    for (String key: sortedMap.keySet()) {
        context.write(new Text(key), new IntWritable(sortedMap.get(key)));
        counter ++;
        if(counter == 5) return;
    }
}
```

Output:



```
project 118
king    99
world   86
moore   50
matter  48
```

Code files for Hadoop Question3:
./data/source-code/Q3/src/wordcount

1. WordCountJobControl.java
2. WordCountMapper.java
3. WordCountCombiner.java
4. WordCountReducer.java

2. CONCLUSION

There were few points that could be easily learnt from this project.

- Hadoop Jobs are slower for smaller files where as bash can give good speed for small data
- The count for both the output matches since the similar methodology/ steps to tokenize is used for both.
- For large processes, Map Reduce tasks can be optimized with the use if combiner (may reduce the total number of keys for reducer). But if not used properly, combiner may affect results. Also, for some problems combiner cannot be used

APPENDIX

[A1] URL LINKS

In case of incorrect output, download URLS manually and keep in folders.
./data/books/ - Folder for bash scripts (CLI or Hadoop both share same set of books)

1. http://www.gutenberg.org/files/1524/1524-0.txt
2. http://www.gutenberg.org/cache/epub/1112/pg1112.txt
3. http://www.gutenberg.org/cache/epub/2267/pg2267.txt
4. http://www.gutenberg.org/cache/epub/2253/pg2253.txt
5. http://www.gutenberg.org/files/1513/1513-0.txt
6. http://www.gutenberg.org/cache/epub/1120/pg1120.txt

[A2] Common Operations on files in hdfs used:

hdfs dfs -test -d <hdfs_path>	Checks if directory exists, return 0 if not exist
hdfs dfs -mkdir <hdfs_path>	Creates directory
hdfs dfs -copyFromLocal -f <localpath> <hdfs_path>	Copy files from localpath to hdfs path
hdfs dfs -rm -r <hdfs_path>	Recursively removes folder
hdfs dfs -cat <filename>	Reads file

[A3] Run a Hadoop task

```
hadoop jar <jar_name> <argument list>
```

Example:

```
hadoop jar Q3.jar /17200430/books/*.txt /output-folder/
```

[A4] Start dfs, yarn and jobhostory server

start-dfs.sh -- To start distributed file system (NameNode, DataNode, SecondaryNameNode)
start-yarn.sh – To start yarn (Resource Manager and Nodemanager)
mr-jobhistory-deamon.sh start historyserver – To start job history server

[A5] Stop dfs, yarn and jobhostory server

stop-dfs.sh -- To stop distributed file system (NameNode, DataNode, SecondaryNameNode)
stop-yarn.sh – To stop yarn (Resource Manager and Nodemanager)
mr-jobhistory-deamon.sh stop historyserver – To stop job history server

Code Files Attached. Please refer ./data/source-code folder to view detailed logic/code.