Introduction to Hadoop – Part One

Contents

[1 Accessing the Hadoop Server 1](#_Toc3231125)

[1.1 VPN Software 1](#_Toc3231126)

[1.2 Hadoop Server 1](#_Toc3231127)

[2 Hadoop Directory Structure 1](#_Toc3231128)

[2.1 Hadoop Commands 1](#_Toc3231129)

[3 Example Programs 2](#_Toc3231130)

[3.1 Pi program 2](#_Toc3231131)

[4 Word Count Version 1 2](#_Toc3231132)

[4.1 Word Count.java 3](#_Toc3231133)

[4.2 Running the Word Count Program 3](#_Toc3231134)

[4.3 Rerunning the Word Count 4](#_Toc3231135)

[4.4 Using a larger dataset 4](#_Toc3231136)

[5 Word Count Version 2 5](#_Toc3231137)

[5.1 Making Word Count Case Insensitive 5](#_Toc3231138)

[5.2 What Changed? 6](#_Toc3231139)

[6 Word Count Version 3 6](#_Toc3231140)

[6.1 Removing Punctuation from Words 7](#_Toc3231141)

[7 Exercises to do 7](#_Toc3231142)

[7.1 Updated Word Count 8](#_Toc3231143)

[7.2 Sample Programs 8](#_Toc3231144)

[7.3 Shell Script 9](#_Toc3231145)

[8 Python Code 10](#_Toc3231146)

[Appendices 12](#_Toc3231147)

[Appendix One: WordCount.java – Version One 12](#_Toc3231148)

[Appendix Two: WordCount.java – Version Two 14](#_Toc3231149)

[Appendix Three: WordCount.java – Version Three 16](#_Toc3231150)

The following workshop introduces the Hadoop server.

Command for you to try out are in blue text:

runThis command

Comments or sample results will be in purple:

Sample results….

Do note, the Hadoop server is a Linux box and so the commands below are case sensitive. Most Linux and Hadoop commands should be typed in lower case.

# Accessing the Hadoop Server

The Hadoop Server can is accessed via a Virtual Private Network (VPN). This is not an issue when accessing the server from a lab in the University, however, if you want to use it from outside the University, such as at home, you will need to setup some additional software first.

## VPN Software

On Canvas there is a file called:

Installing the VPN Client.pdf

This gives you instructions on how to setup the VPN client. You only need to install this for use outside the University.

## Hadoop Server

The server name is hpd-srv.wlv.ac.uk, which can be access via a SSH client, such as putty, which is available on Apps Anywhere. If you are using your own computer, any alternative Secure Shell (ssh) program can be used instead.

Your username will be your University student number. Your password will be your username appended with your initials, in the order of Surname and First name, as recorded on eVision. If you have several first (given) names, only the first one will be used – again in the order show on eVision.

For example, if your student number is 0123456 and your name is John Malcolm Smith, then your password will be 0123456SJ

You are strongly advised to change your password when you first login by using the passwd command. Type:

passwd

It will prompt for your original password, then your new one, which must be 8 characters or more.

# Hadoop Directory Structure

Hadoop is not a conventional database in that you do not need to be setup with a username and password, such as seen with Oracle. Hadoop has its own file system and you do need to have your own directory to store files.

By default your directory will be called:

/user/studentNo

So if your student number is 0123456 then your directory will be called:

/user/0123456/

## Hadoop Commands

When accessing Hadoop most commands will start with hdfs dfs. The most common commands include:

|  |  |
| --- | --- |
| hdfs dfs -mkdir mydir | Create a directory on HDFS |
| hdfs dfs -ls | List files and directories on HDFS |
| hdfs dfs -cat myfile | View a file’s content |
| hdfs dfs -put myfile mydir | Store a file on HDFS |
| hdfs dfs -get myfile filename | Retrieve a file on the HDFS |
| hdfs dfs -rm myfile | Delete a file on HDFS |
| hdfs dfs -touchz myfile | Create an empty file on HDFS |
| hdfs dfs -stat myfile | Check the status of a file (file size, owner, …) |
| hdfs dfs -test -e myfile | Check if file exists on HDFS |
| hdfs dfs -test -z myfile | Check if file is empty on HDFS |
| hdfs dfs -du | Check disk space usage on HDFS |

For example, to view what files you have run the –ls command:

hdfs dfs –ls /user/**studentNo**

substituting studentNo with your own number.

Initially your home directory will be empty. You can store any files in your own directory, but will not have access to anyone else’s. You can create subdirectories here too, so typically you will need to create an input folder to store the files Hadoop will work with.

To create a folder called input use the –mkdir command:

hdfs dfs –mkdir input

Run the list directory command to see the new folder:

hdfs dfs –ls /user/**studentNo**

and you should now see your input directory

# Example Programs

Hadoop comes with some example programs.

To see what examples are available:

yarn jar $YARN\_EXAMPLES/hadoop-mapreduce-examples.jar

Note, most of the examples require files to be copied to or from the HDFS to work correctly.

## Pi program

Pi is one of the sample programs, which calculates the value of pi using a quasi-Monte Carlo method and MapReduce:

yarn jar $YARN\_EXAMPLES/hadoop-mapreduce-examples.jar pi 16 1000

Which should give the following output:

Estimated value of Pi is 3.142500

# Word Count Version 1

The first program to try is the word count. A discussion of the Word Count file can be found in Lecture 8.

## Word Count.java

There is a copy of the WordCount.java on Canvas, or can be copied directly on the hdp-srv server. We will be creating different versions of this file, so keep them in separate directories.

Create a directory called wordcount-v1:

mkdir wordcount-v1

Change to this directory:

cd wordcount-v1

Copy the WordCount.java file:

cp /home/6cs030/WordCount-V1/WordCount.java .

The code is also available in Appendix One.

## Running the Word Count Program

There are several steps you need to undertake:

1. Compile the file:

javac -classpath $(hadoop classpath) WordCount.java

1. Produce the Jar file:

jar cf wordcount.jar Word\*.class

1. Create the input directory on the hdfs:

hdfs dfs -mkdir input\_word

1. Create the input files:

echo A long time ago in a galaxy far far away > testfile1

echo Another episode of Star Wars > testfile2

1. Save the files to the input directory:

hdfs dfs -put testfile? input\_word

1. It is important that the output directory does not already exist. If you have run the program before you need to delete the previous output directory:

hdfs dfs -rm -R output\_word

1. Run the Map Reduce program:

hadoop jar wordcount.jar WordCount input\_word output\_word

1. Check what files are in the output directory:

hdfs dfs -ls output\_word

1. See what is in the output file:

hdfs dfs -cat output\_word/part-r-00000

1. If you want to use the output file outside the hdfs you have to retrieve it:

hdfs dfs -get output\_word/part-r-00000 word-results.txt

These steps are generally what you need to run each time you have a new Java file to compile and run. You only need to carry out some of the steps if you have made changes to the Java file (Steps 1 and 2), or want to change the input directory (Step 3) or the files stored in it (Steps 4-5).

## Rerunning the Word Count

If you want to run the same java file with some new data, provided no changes have been made to the java file then you do not need to carry out Steps 1 and 2 again.

If you simply want to run the Word Count program again you can just run steps 6 and 7:

hdfs dfs -rm -R output\_word

hadoop jar wordcount.jar WordCount input\_word output\_word

Then view the output file:

hdfs dfs -cat output\_word/part-r-00000

Do note, if the output directory already exists before you run the Hadoop command you will get an error!

## Using a larger dataset

The above program only uses a very small dataset. In the WordCount program Hadoop will read any file that is stored in the input directory. This time we will use the *Complete Works of Shakespeare*, a text version has been downloaded from here:

<https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt>

This is stored on the hpd-srv in a directory called /home/6cs030, so first copy it to your own account:

cp /home/6cs030/data/shakespeare.txt .

Assuming you have not made any changes to the java and jar files, carry out the following steps:

1. Remove the previous testfiles:

hdfs dfs -rm input\_word/testfile?

1. Save the new file to the input directory:

hdfs dfs -put shakespeare.txt input\_word

1. By now you will have created an output directory in the previous run of the program, so you need to delete this:

hdfs dfs -rm -R output\_word

1. Run the Map Reduce program:

hadoop jar wordcount.jar WordCount input\_word output\_word

1. Check what files are in the output directory:

hdfs dfs -ls output\_word

1. See what is in the output file:

hdfs dfs -cat output\_word/part-r-00000

1. This time there will be a bigger results set, so to view it properly, copy the file locally:

hdfs dfs -get output\_word/part-r-00000 shakespeare-results.txt

1. Use the Linux command more to view the results file in the Operating System:

more shakespeare-results.txt

# Word Count Version 2

The results file is quite large, so you can use the Linux command grep to search for a particular word:

grep anon shakespeare-results.txt

and also

grep Anon shakespeare-results.txt

The above two commands show that the word count is case sensitive. Anon and anon are counted separately, though they are the same word. Unless there is a good reason for treating them differently the program should count them as the same word.

## Making Word Count Case Insensitive

Some extra code is needed to make the word count not case sensitive.

Return to your home directory and create a new directory to keep the versions separate, for example:

cd

mkdir wordcount-v2

Change to this new directory:

cd wordcount-v2

Then copy the new version here:

cp /home/6cs030/WordCount-V2/WordCount.java .

Compile the file and create a Jar file as seen in steps 1 and 2 of Section 4.2

Remove the output directory:

hdfs dfs -rm -R output\_word

Assuming that the shakespeare.txt file is still stored in the hdfs run the program:

hadoop jar wordcount.jar WordCount input\_word output\_word

If everything has worked correctly, retrieve the results file from the hdfs:

hdfs dfs -get output\_word/part-r-00000 shakespeare-results.txt

Try using grep to search for anon again:

grep anon shakespeare-results.txt

grep Anon shakespeare-results.txt

This time you should get no results for the second command. If you add up the word counts, the totals should still match the previous version, but this time the results are not case sensitive.

## What Changed?

If you examine the code in Appendix Two you will notice the addition of a setup method in the Mapper class:

protected void setup(Mapper.Context context)

throws IOException, InterruptedException {

Configuration config = context.getConfiguration();

this.caseSensitive = config.getBoolean("wordcount.case.sensitive", false);

} // setup method

And an additional variable:

private boolean caseSensitive = false;

The setup method is called automatically when a job is submitted. It instantiates a Configuration object and then sets the class caseSensitive variable to the value of the wordcount.case.sensitive system variable set from the command line. The above code will set this to false by default.

A further change is to test if the caseSensitive variable is true or not, if false it will convert all words to lower case:

String line = value.toString();

// check if job is case sensitive or not

if (!caseSensitive)

line= line.toLowerCase();

After that the mapper class carries on as before. No changes were needed for the reducer class.

# Word Count Version 3

The Word Count program is still not perfect. For example, when searching for anon, a sample of the output shows it still has duplicate versions of the word:

'anon 1

'anon!' 1

'anon, 1

anon! 3

anon, 30

anon. 43

anon.- 1

anon; 5

anon? 1

This is because punctuation marks such as commas, questions marks, etc. have made the word appear to have different values. If the aim is to just count words, then really we want to treat all of the above as being instances of the word anon.

The Word Count program is not very sophisticated in checking for any punctuation, null words, or anything else that might crop up.

The above examples use a StringTokenizer, which according to the Java manual [[1]](#footnote-1)”*is now a legacy class that is retained for compatibility reasons and its use is discouraged in new code”*. Instead the manual recommends the use the split method of String or the java.util.regex package instead.

## Removing Punctuation from Words

This final version of Word Count will use the Pattern class from java.util.regex

Return to your home directory:

cd

Create another new directory to keep this version separate:

mkdir wordcount-v3

Change to this new directory:

cd wordcount-v3

Then copy the new version here:

cp /home/6cs030/WordCount-V3/WordCount.java .

Compile the file and create a Jar file as seen in steps 1 and 2 of Section 4.2

Remove the output directory:

hdfs dfs -rm -R output\_word

Assuming that the shakespeare.txt file is still stored in the hdfs run the program:

hadoop jar wordcount.jar WordCount input\_word output\_word

If everything has worked correctly, retrieve the results file from the hdfs:

hdfs dfs -get output\_word/part-r-00000 shakespeare-results.txt

Try using grep to search for anon again:

grep anon shakespeare-results.txt

This time anon will only appear once:

anon 128

anonymous 1

canon 6

canoniz 2

canonize 1

canonized 2

canons 1

The results are still case insensitive too, this should produce no results:

grep Anon shakespeare-results.txt

# Exercises to do

The results are still not perfect. If you use more to list the whole file:

more shakespeare-results.txt

You will see output such as this:

161

' 1

[ 3

[ 13

' 1

[ 5

[ 10

This is because the code is still relatively unsophisticated in that it is not removing the punctuation or spaces from the output.

## Updated Word Count

* An exercise for you to try is to amend Version 3 of the code to remove some (or all!) of the punctuation.

Hiint: the following code checks if the current word is empty and skips to the next word:

if (word.isEmpty())

continue;

Similar code could be written to check if the current word contains any punctuation characters. The String class’s method contain could be used for this.

## Sample Programs

Section 3 Introduced the sample programs that come with Hadoop. As a reminder to see the full list type:

yarn jar $YARN\_EXAMPLES/hadoop-mapreduce-examples.jar

Which produces a list including:

* **aggregatewordcount**: An Aggregate based map/reduce program that counts the words in the input files.
* **aggregatewordhist**: An Aggregate based map/reduce program that computes the histogram of the words in the input files.
* **bbp**: A map/reduce program that uses Bailey-Borwein-Plouffe to compute exact digits of Pi.
* **dbcount**: An example job that count the pageview counts from a database.
* **distbbp**: A map/reduce program that uses a BBP-type formula to compute exact bits of Pi.
* **grep**: A map/reduce program that counts the matches of a regex in the input.
* **join**: A job that effects a join over sorted, equally partitioned datasets
* **multifilewc**: A job that counts words from several files.
* **pentomino**: A map/reduce tile laying program to find solutions to pentomino problems.
* **pi**: A map/reduce program that estimates Pi using a quasi-Monte Carlo method.
* **randomtextwriter**: A map/reduce program that writes 10GB of random textual data per node.
* **randomwriter**: A map/reduce program that writes 10GB of random data per node.
* **secondarysort**: An example defining a secondary sort to the reduce.
* **sort**: A map/reduce program that sorts the data written by the random writer.
* **sudoku**: A sudoku solver.
* **teragen**: Generate data for the terasort
* **terasort**: Run the terasort
* **teravalidate**: Checking results of terasort
* **wordcount**: A map/reduce program that counts the words in the input files.
* **wordmean**: A map/reduce program that counts the average length of the words in the input files.
* **wordmedian**: A map/reduce program that counts the median length of the words in the input files.
* **wordstandarddeviation**: A map/reduce program that counts the standard deviation of the length of the words in the input files.

To find out what options are required, add one of the above to the command, for example, if you are a Sudoku fan:

yarn jar $YARN\_EXAMPLES/hadoop-mapreduce-examples.jar Sudoku

Which tells you to: Include a puzzle on the command line.

A sample puzzle could be:

8 5 ? 3 9 ? ? ? ?  
? ? 2 ? ? ? ? ? ?  
? ? 6 ? 1 ? ? ? 2  
? ? 4 ? ? 3 ? 5 9  
? ? 8 9 ? 1 4 ? ?  
3 2 ? 4 ? ? 8 ? ?  
9 ? ? ? 8 ? 5 ? ?  
? ? ? ? ? ? 2 ? ?  
? ? ? ? 4 5 ? 7 8

Which Sudoku fans will know is a typical 9x9 matrix containing either a number between 1 and 9, or a question mark for the missing numbers.

A copy of the above puzzle can be found on the server:

cp /home/6cs030/misc/puzzle.txt .

To run Sudoku with this puzzle:

yarn jar $YARN\_EXAMPLES/hadoop-mapreduce-examples.jar Sudoku puzzle.txt

Did it produce the right answer? A number should not appear more than once in any single row, column or 3x3 matrix.

To do:

* If you are Sudoku fan try the above with a different puzzle and see if it gets the right results
* Pick one of the other Hadoop programs and see if you can get it to work.
  + If an input or output directory is required you can create a new one, to avoid conflict with the Word Count examples.

## Shell Script

As you will now appreciate, running a Java program involves a lot of steps.

A Shell script can be written to create a batch file to run a set of commands. Such an example can be found in the samples 6cs030 directory. To copy this:

cp /home/6cs030/misc/runProg .

This has the following code:

javac -classpath $(hadoop classpath) WordCount.java

jar cf wordcount.jar Word\*.class

# test if output file exists

if hdfs dfs -test -e output\_word;

then hdfs dfs -rm -R output\_word;

fi

hadoop jar wordcount.jar WordCount input\_word output\_word

hdfs dfs -ls output\_word

# uncomment the next line if want to show the output

#hdfs dfs -cat output\_word/part-r-00000

if [ -f word-results.txt ];

then rm word-results.txt;

fi

hdfs dfs -get output\_word/part-r-00000 word-results.txt

The file needs to be executable, so change the permissions to:

chmod u+x runProg

Then to run it in future type:

./runProg

This script is tailored to the Word Count program. You will need to amend it if you create a different java file, or want to change the input or output directories. This can be done using the Linux editors vi or nano. Or if you are not familiar with Linux editors, use a sftp program, such as FileZilla to transfer the program to/from hdp-srv to your local computer and then use a text editor such as NotePad or equivalent to amend it.

# Python Code

Python can be used with Hadoop too. Note if you wish to use Python code the process to run it is different to Java.

Assuming that the input directory exists with the required input files, the following steps are needed (replace anything beginning with my with your own student number or file name):

hdfs dfs -rm -*R myHadoop\_output\_directory* # remove the output directory if it exists

# need the full path name for the Python files

hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-3.1.2.jar \

-input *myHadoop\_input\_directory* \

-output *myHadoop\_output\_directory* \

-mapper /home/*myStudentNo/my\_mapper\_program.py* \

-reducer /home/*myStudentNo/my\_reducer\_program.py*

The Python files must be readable and executable by Hadoop:

chmod 755 *myPython\_program.py*

The output files will be stored the myHadoop\_output\_directory:

hdfs dfs -ls *myHadoop\_output\_directory*

This time the output files produced are slightly different. For example:

Found 2 items

-rw-r--r-- 1 0123456 hadoop 0 2019-03-09 16:16 myHadoop\_output\_directory/\_SUCCESS

-rw-r--r-- 1 0123456 hadoop 348 2019-03-09 16:16 myHadoop\_output\_directory/part-00000

# Appendices

There are many versions of the Word Count program available online. The following code is based on: [https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Inputs\_and\_Outputs](https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html) and <https://www.cloudera.com/documentation/other/tutorial/CDH5/topics/Hadoop-Tutorial.html>

## Appendix One: WordCount.java – Version One

import java.io.IOException;

import java.util.StringTokenizer;

// set up the standard libraries

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

// Mapper class

public static class TokenizerMapper

extends Mapper<Object, Text, Text, IntWritable>{

// one is simply a variable to store a word count of 1

private final static IntWritable one = new IntWritable(1);

private Text word = new Text(); // use Text for character data

// context will be used to emit the output values

public void map(Object key, Text value, Context context

) throws IOException, InterruptedException {

StringTokenizer itr = new StringTokenizer(value.toString());

// iterate through the input files

while (itr.hasMoreTokens()) {

word.set(itr.nextToken());

context.write(word, one); // this is what is output by mapping stage

} // while loop

} // map method

} // TokenizeMapper class

// Reducer class

public static class IntSumReducer

extends Reducer<Text,IntWritable,Text,IntWritable> {

private IntWritable result = new IntWritable();

// this will read in the output generated by the map function

public void reduce(Text key, Iterable<IntWritable> values,

Context context

) throws IOException, InterruptedException {

int sum = 0;

// use IntWriteable for numeric values

// this will add up all the words with the same name

for (IntWritable val : values) {

sum += val.get();

} // for loop

result.set(sum);

// this outputs the final result

context.write(key, result);

} // reduce method

} // IntSumReducer class

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

// give the job a name

Job job = Job.getInstance(conf, "word count");

// tell Hadoop which jar file to use

job.setJarByClass(WordCount.class);

// tell Hadoop which Mapper class to use

job.setMapperClass(TokenizerMapper.class);

// tell Hadoop which Combination class to use

// used to summarise map output with the same key

// Combiner is also known as a semi-reducer

job.setCombinerClass(IntSumReducer.class);

// tell Hadoop which Reducer class to use

job.setReducerClass(IntSumReducer.class);

// set the type of the key and value

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

// defines how many arguments are expected

// can have more than one input directory

// args[0] will be the input directory and args[1] the output

FileInputFormat.addInputPath(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

// will stop when the job completes

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

## Appendix Two: WordCount.java – Version Two

Changes from Version One are highlighted in bold.

// Version 2 - makes the word count case insensitive

// set up the standard libraries

import java.io.IOException;

import java.util.StringTokenizer;

// set up the Hadoop libraries

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

// Mapper class

public static class TokenizerMapper

extends Mapper<Object, Text, Text, IntWritable>{

// one is simply a variable to store a word count of 1

private final static IntWritable one = new IntWritable(1);

private Text word = new Text(); // use Text for character data

// context will be used to emit the output values

**private boolean caseSensitive = false;**

**// added to make Word Count case insensitive**

**// Hadoop will call this method automatically when a job is submitted**

**protected void setup(Mapper.Context context)**

**throws IOException, InterruptedException {**

**Configuration config = context.getConfiguration();**

**this.caseSensitive = config.getBoolean("wordcount.case.sensitive", false);**

**} // setup method**

public void map(Object key, Text value, Context context

) throws IOException, InterruptedException {

**String line = value.toString();**

**// check if job is case sensitive or not**

**if (!caseSensitive)**

**line= line.toLowerCase();**

StringTokenizer itr = new StringTokenizer(line);

// iterate through the input files

while (itr.hasMoreTokens()) {

word.set(itr.nextToken());

context.write(word, one); // this is what is output by mapping stage

} // while loop

} // map method

} // TokenizeMapper class

// Reducer class

public static class IntSumReducer

extends Reducer<Text,IntWritable,Text,IntWritable> {

private IntWritable result = new IntWritable();

// this will read in the output generated by the map function

public void reduce(Text key, Iterable<IntWritable> values,

Context context

) throws IOException, InterruptedException {

int sum = 0;

// use IntWriteable for numeric values

// this will add up all the words with the same name

for (IntWritable val : values) {

sum += val.get();

} // for loop

result.set(sum);

// this outputs the final result

context.write(key, result);

} // reduce method

} // IntSumReducer class

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

// give the job a name

Job job = Job.getInstance(conf, "word count");

// tell Hadoop which jar file to use

job.setJarByClass(WordCount.class);

// tell Hadoop which Mapper class to use

job.setMapperClass(TokenizerMapper.class);

// tell Hadoop which Combination class to use

// used to summarise map output with the same key

// Combiner is also known as a semi-reducer

job.setCombinerClass(IntSumReducer.class);

// tell Hadoop which Reducer class to use

job.setReducerClass(IntSumReducer.class);

// set the type of the key and value

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

// defines how many arguments are expected

// can have more than one input directory

// args[0] will be the input directory and args[1] the output

FileInputFormat.addInputPath(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

// will stop when the job completes

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

## Appendix Three: WordCount.java – Version Three

Changes in bold:

// Version 3 - removes punctuation around words

// set up the standard libraries

import java.io.IOException;

**// replace StringTokenizer with regex Pattern**

**import java.util.regex.Pattern;**

// set up the Hadoop libraries

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

// Mapper class

public static class TokenizerMapper

extends Mapper<Object, Text, Text, IntWritable>{

// one is simply a variable to store a word count of 1

private final static IntWritable one = new IntWritable(1);

private Text word = new Text(); // use Text for character data

// context will be used to emit the output values

private boolean caseSensitive = false;

**// set up a regular expression for word boundaries**

**// these include spaces, tabs and punctuation**

**private static final Pattern WORD\_BOUNDARY = Pattern.compile("\\s\*\\b\\s\*");**

// added to make Word Count case insensitive

// Hadoop will call this method automatically when a job is submitted

protected void setup(Mapper.Context context)

throws IOException, InterruptedException {

Configuration config = context.getConfiguration();

this.caseSensitive = config.getBoolean("wordcount.case.sensitive", false);

} // setup method

public void map(Object key, Text value, Context context

) throws IOException, InterruptedException {

String line = value.toString();

// check if job is case sensitive or not

if (!caseSensitive)

line= line.toLowerCase()

**Text currentWord = new Text();**

**// split the line into individual words based on word boundaries**

**// if the word is empty then move on**

**for (String word : WORD\_BOUNDARY.split(line)) {**

**if (word.isEmpty()) {**

**continue;**

**}**

**currentWord = new Text(word);**

**context.write(currentWord,one); // this is what is output by mapping stage**

**} // for loop**

} // map method

} // TokenizeMapper class

public static class IntSumReducer

extends Reducer<Text,IntWritable,Text,IntWritable> {

private IntWritable result = new IntWritable();

// this will read in the output generated by the map function

public void reduce(Text key, Iterable<IntWritable> values,

Context context

) throws IOException, InterruptedException {

int sum = 0;

// use IntWriteable for numeric values

// this will add up all the words with the same name

for (IntWritable val : values) {

sum += val.get();

} // for loop

result.set(sum);

// this outputs the final result

context.write(key, result);

} // reduce method

} // IntSumReducer class

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

// give the job a name

Job job = Job.getInstance(conf, "word count");

// tell Hadoop which jar file to use

job.setJarByClass(WordCount.class);

// tell Hadoop which Mapper class to use

job.setMapperClass(TokenizerMapper.class);

// tell Hadoop which Combination class to use

// used to summarise map output with the same key

// Combiner is also known as a semi-reducer

job.setCombinerClass(IntSumReducer.class);

// tell Hadoop which Reducer class to use

job.setReducerClass(IntSumReducer.class);

// set the type of the key and value

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

// defines how many arguments are expected

// can have more than one input directory

// args[0] will be the input directory and args[1] the output

FileInputFormat.addInputPath(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

// will stop when the job completes

System.exit(job.waitForCompletion(true) ? 0 : 1);

} // main

} // WordCount class

1. https://docs.oracle.com/javase/7/docs/api/java/util/StringTokenizer.html [↑](#footnote-ref-1)