Gregory ISLAS
16/2/17
Assignment #1

Introduction

The following document describes the procedures performed to complete Assignment #1 for the Massive Data Processing course at École Centrale Paris, including java code implemented and Hadoop file system commands used. The system setup for the code below is as follows. All programs were run on a virtual machine (cloudera-quickstart-vm-5.8.0-0 virtualbox) which was running the Red Hat (64-bit) operating system. The virtual machine was setup to use 4096 MB of RAM. The Virtual Hard Disk Controller used the IDE standard, and the Primary Master had a capacity of 64.0 GB. There was no IDE Secondary Master. The host computer was a MacBook Pro running macOS Sierra, with 8.0 GB of RAM and a 2.7 GHz Intel Core i5 processor.

As for the Hadoop setup, the Hadoop version used in the running of all was version 2.6.0-cdh5.8.0 subversion http://github.com/cloudera/hadoop -r 57e7b8556919574d517e874abfb7ebe31a366c2b. Native libraries are as follows:

```
Native library checking:
hadoop:  true /usr/lib/hadoop/lib/native/libhadoop.so.1.0.0
zlib:    true /lib64/libz.so.1
snappy:  true /usr/lib/hadoop/lib/native/libsnappy.so.1
lz4:     true revision:10301
bzip2:   true /lib64/libbz2.so.1
openssl: true /usr/lib64/libcrypto.so
```

The procedure for testing and running the code for each part of the assignment was the same. All programs were first tested on the document Page 31100.txt (the works of Mark Twain – even though this is the largest of the three files performed, it was chosen since it contains a variety of strange words) in the IDE Eclipse in standalone mode. Once adequate results were obtained, the same procedure was repeated on the Hadoop File System in pseudo-distributed mode using Cloudera's pre-installed Apache Hadoop suite. After this was completed, the previous two steps were repeated, this time using all three files mentioned in the assignment description (http://www.gutenberg.org/cache/epub/100/pg100.txt - The Complete Works of William Shakespeare; http://www.gutenberg.org/cache/epub/31100/pg31100.txt - The Complete Works of Mark Twain; and http://www.gutenberg.org/cache/epub/3200/pg3200.txt - The Complete Works of Jane Austen). The files were stored in a directory, which was passed in as an argument to the MapReduce program. The code from the example wordcount Map Reduce program in Assignment 0 was used as a basis template.

Section (a)

In Section (a) of the assignment, the task was to create a MapReduce program that identified stopwords in the aforementioned corpus, e.g. words that appeared at least 4000 times throughout the document corpus. Once MapReduce was completed, the words would be stored in a single CSV file on the Hadoop File System, with each line corresponding to (word, count(word)). The idea behind the implementation was to create a java class that extended the Configured abstract class and Implemented the Tool interface, and overrode the run method in ToolRunner to create unique Map, Combine, and Reduce classes. The class mentioned was named Stopwords.java. Each separate part of Section (a) used the same Stopwords.java class, with small alterations to the run() method depending upon the task to be performed. The OutputKeyClass was set to a Text.class. This key corresponds to the unique word to be counted. The OutputValueClass was set to an IntWritable, which would correspond to the count of the word, or the number of times it appeared in the document corpus. The InputFormatClass was set to TextInputFormat.class, and the OutputFormatClass was set to TextOutputFormat. For the job to run successfully, the user must input the correct input directory, and an output directory that does not already exist.

Furthermore, the run method contained an extra section of code that combines the output from the reducers into one CSV file. Upon completion of the job, this section of code takes the "part-r-XXXX" files that were written into the output directory, and combines them into one file that will be stored on the Hadoop File System home directory.

Finally, a note concerning the definition of a word, which was implemented in the Map class which extends the Mapper abstract class. Since the `map(Longwritable key, Text value, Context context)` method is called for each line in the file, the algorithm for creating the words is as follows. First, the line is split into separate Strings via Java's `String.split()` method, using the regular expression "\\s+|--". In other words, this splits a line into separate Strings based on spaces or double hyphens (it was noticed that double hyphens often did not have spaces between them, but that the words connected by a "--" usually referred to separate words; on the other hand, words separated by a "-" were assumed to refer to one idea/word). Since this did not take care of punctuation, each String was then further processed. All non-alphanumeric characters from the beginning of the String until the first alphanumeric character in the String were deleted. Similarly, all non-alphanumeric characters from the end of a String until the last alphanumeric of the String were deleted. If a non-alphanumeric character occurred in the middle of a String, and was not either the - character or ' character, it was deleted. While this is obviously not a perfect definition of a word, it was felt that this implementation provided a simple yet informative count of the number of words in a document.

Thus, once the String processing method in the Map class was performed, it would write the String (first checking to see if the String was empty, i.e. prior to processing it had contained only non-alphanumeric characters) along with the IntWritable constant value of "1". In the Reducer class, the value of "1" was summed for every occurrence of the word, resulting in the final count of the number of appearances of the word. The stopword criteria check was performed in the Reducer class, by checking to see if this value was greater than or equal to 4000. If so, the word, along with its count, would be written to the Context. This sums up the basic functionality of the Stopwords.java class – the specific tweaks to answer each subpart of Section (a) will now be examined.

<u>Section (a) – i.</u>

In this subsection, the Stopwords.java program was run on the entire document corpus with 10 reducers. As instructed, no Combiner was used. To set the number of reducers to 10, the following line of code was added (the `myjob` variable is an instance of the Job class that was used to run the program):

```
myjob.setNumReduceTasks(10); //set the number of reducers to 10
```

To run the program (which was exported to a .jar file stopwords_10nc.jar), the following command was used:

```
hadoop jar stopwords_10nc.jar stop.words.StopWords  Pages
output_10nc
```

On the Hadoop File System with the aforementioned setup, the program's total time was 2 minutes and 22 seconds, and the CPU time was 32920 milliseconds (approximately 32.9 seconds).

Screenshot from Command Prompt:

```
        Map-Reduce Framework
                Map input records=507535
                Map output records=4506876
                Map output bytes=41887748
                Map output materialized bytes=50901680
                Input split bytes=381
                Combine input records=0
                Combine output records=0
                Reduce input groups=74804
                Reduce shuffle bytes=50901680
                Reduce input records=4506876
                Reduce output records=134
                Spilled Records=9013752
                Shuffled Maps =30
                Failed Shuffles=0
                Merged Map outputs=30
                GC time elapsed (ms)=2502
                CPU time spent (ms)=32920
                Physical memory (bytes) snapshot=1807994880
                Virtual memory (bytes) snapshot=19570429952
                Total committed heap usage (bytes)=1104359424
```

Screenshot from Logs:

Job Overview	
Job Name:	stopwords_10nc.jar
User Name:	cloudera
Queue:	root.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Mon Feb 13 07:01:56 PST 2017
Started:	Mon Feb 13 07:02:19 PST 2017
Finished:	Mon Feb 13 07:04:42 PST 2017
Elapsed:	2mins, 22sec
Diagnostics:	
Average Map Time	48sec
Average Shuffle Time	36sec
Average Merge Time	4sec
Average Reduce Time	7sec

Section (a) – ii.

In this subsection, the program was run again with the same number of reducers, while also using a combiner. Here, the combiner class was set to the reducer class with the following line of code:

```
myjob.setCombinerClass(Reduce.class);
```

The combiner class was set to the same class as the reducer class since it is just performing the sum operation, and the sum(a, b, c, d, …) = sum(sum(a, b), sum(c, d), …).

The program's total runtime was 2 minutes and 7 seconds, while the CPU time was 22190 milliseconds (approximately 22.2 seconds).

Screenshot from Job Logs:

Job Overview	
Job Name:	stopwords_10c.jar
User Name:	cloudera
Queue:	root.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Mon Feb 13 07:49:27 PST 2017
Started:	Mon Feb 13 07:49:43 PST 2017
Finished:	Mon Feb 13 07:51:51 PST 2017
Elapsed:	2mins, 7sec
Diagnostics:	
Average Map Time	46sec
Average Shuffle Time	38sec
Average Merge Time	0sec
Average Reduce Time	3sec

The decreased runtime can be attributed to the fact that the combiner class greatly reduced the amount of data transferred over the network from the map phase to the reduce phase. For a better look, here are screenshots from each of the outputs for the two jobs:

Output from Job w/No Combiner:

	Name	Map	Reduce	Total
Map-Reduce Framework	Combine input records	0	0	0
	Combine output records	0	0	0
	CPU time spent (ms)	13600	19320	32920
	Failed Shuffles	0	0	0
	GC time elapsed (ms)	867	1635	2502
	Input split bytes	381	0	381
	Map input records	507535	0	507535
	Map output bytes	41887748	0	41887748
	Map output materialized bytes	50901680	0	50901680
	Map output records	4506876	0	4506876
	Merged Map outputs	0	30	30
	Physical memory (bytes) snapshot	608976896	1199017984	1807994880
	Reduce input groups	0	74804	74804
	Reduce input records	0	4506876	4506876
	Reduce output records	0	134	134
	Reduce shuffle bytes	0	50901680	50901680
	Shuffled Maps	0	30	30
	Spilled Records	4506876	4506876	9013752
	Total committed heap usage (bytes)	496840704	607518720	1104359424
	Virtual memory (bytes) snapshot	4500639744	15069790208	19570429952

Output from Job w/Combiner:

	Name	Map	Reduce	Total
Map-Reduce Framework	Combine input records	4506876	0	4506876
	Combine output records	146	0	146
	CPU time spent (ms)	13460	8730	22190
	Failed Shuffles	0	0	0
	GC time elapsed (ms)	1014	1584	2598
	Input split bytes	381	0	381
	Map input records	507535	0	507535
	Map output bytes	41887748	0	41887748
	Map output materialized bytes	1643	0	1643
	Map output records	4506876	0	4506876
	Merged Map outputs	0	30	30
	Physical memory (bytes) snapshot	618651648	1083330560	1701982208
	Reduce input groups	0	89	89
	Reduce input records	0	146	146
	Reduce output records	0	89	89
	Reduce shuffle bytes	0	1643	1643
	Shuffled Maps	0	30	30
	Spilled Records	146	146	292
	Total committed heap usage (bytes)	496840704	607518720	1104359424
	Virtual memory (bytes) snapshot	4500652032	15071641600	19572293632

As can been seen, the number of map output materialized bytes, reduce input records, and reduce shuffle bytes is much lower in the job with the combiner. Hence, the reduce phase finished quicker, and less time was taken to transfer the map output to the reducers.

<u>Section (a) – iii.</u>

In this subsection, the program was run again with the same number of reducers, while also using a combiner and compressing the intermediate map output. The compression instructions were implemented by the following lines:

```
getConf().setBoolean(Job.MAP_OUTPUT_COMPRESS, true);
getConf().setClass(Job.MAP_OUTPUT_COMPRESS_CODEC,
BZip2Codec.class, CompressionCodec.class);
```

This code was obtained from the Hadoop Definitive Guide 4th Edition. The program's total runtime was 2 minutes and 29 seconds, and the total CPU time was 26980 (approximately 27 seconds).

Screenshot from Command Prompt:

```
        Map-Reduce Framework
                Map input records=507535
                Map output records=4506876
                Map output bytes=41887748
                Map output materialized bytes=2789
                Input split bytes=381
                Combine input records=4506876
                Combine output records=146
                Reduce input groups=89
                Reduce shuffle bytes=2789
                Reduce input records=146
                Reduce output records=89
                Spilled Records=292
                Shuffled Maps =30
                Failed Shuffles=0
                Merged Map outputs=30
                GC time elapsed (ms)=4104
                CPU time spent (ms)=26980
                Physical memory (bytes) snapshot=1767190528
                Virtual memory (bytes) snapshot=19594764288
                Total committed heap usage (bytes)=1104359424
```

Screenshot from Job Logs:

	Job Overview
Job Name:	stopwords_10cc.jar
User Name:	cloudera
Queue:	root.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Mon Feb 13 10:42:27 PST 2017
Started:	Mon Feb 13 10:42:47 PST 2017
Finished:	Mon Feb 13 10:45:16 PST 2017
Elapsed:	2mins, 29sec
Diagnostics:	
Average Map Time	1mins, 2sec
Average Shuffle Time	42sec
Average Merge Time	0sec
Average Reduce Time	4sec

The fact that the runtime increased is likely due to that the combiner had already reduced the size of the output from the mapper enough that further compression did not lead to much less data being transferred over the network. In fact, it may have been that the time it took to compress and decompress the data led to a small increase in runtime.

<u>Section (a) – iv.</u>

The final part of Section (a) consists of running the program (note: the program was run without compression) using 50 reducers. This was implemented by the following line:

```
myjob.setNumReduceTasks(50);
```

The program's total runtime was 6 minutes and 44 seconds, and the total CPU time was 55650 (approximately 55.7 seconds).

Screenshot from Command Prompt:

```
                Total megabyte-seconds taken by all reduce tasks=1976136704
        Map-Reduce Framework
                Map input records=507535
                Map output records=4506876
                Map output bytes=41887748
                Map output materialized bytes=2363
                Input split bytes=381
                Combine input records=4506876
                Combine output records=146
                Reduce input groups=89
                Reduce shuffle bytes=2363
                Reduce input records=146
                Reduce output records=89
                Spilled Records=292
                Shuffled Maps =150
                Failed Shuffles=0
                Merged Map outputs=150
                GC time elapsed (ms)=8811
                CPU time spent (ms)=55650
                Physical memory (bytes) snapshot=6174023680
                Virtual memory (bytes) snapshot=79837536256
```

Screenshot from Job Logs:

Job Overview	
Job Name:	stopwords_50c.jar
User Name:	cloudera
Queue:	root.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Mon Feb 13 12:04:22 PST 2017
Started:	Mon Feb 13 12:04:39 PST 2017
Finished:	Mon Feb 13 12:11:24 PST 2017
Elapsed:	6mins, 44sec
Diagnostics:	
Average Map Time	56sec
Average Shuffle Time	34sec
Average Merge Time	0sec
Average Reduce Time	3sec

The likely reason that the program's runtime increased so much is due to the fact that the mapper-to-reducer ratio was so low. Rather than improving the runtime, the system likely spent too much time splitting the output to send to the reducers. Furthermore, since the system isn't fully distributed, it is likely that the Reducers had to wait for other Reducers to finish, since not all of them could run at once due to the system's architecture.

Section (b)

In this section, the task was to implement an inverted index for the corpus, while skipping the words in stopwords.csv. To accomplish this end, the assumption was made that a file with the name stopwords.csv existed in the home directory of the Hadoop file system. This assumption was also made when running the program in standalone mode in Eclipse. Thus, to remove the stopwords, the method setup(Context context) was overridden in the Map class used in the program. As opposed to the last program, the key and values were both changed to the Text class. Before going into the map(LongWritable key, Text value, Context context) method, the setup(Context context) method would be called. This method simply attempts to open a file named "stopwords.csv" in the directory mentioned previously, and places each word into a global class instance of Java datatype HashSet. HashSet was chosen because it allows for fast lookup, since each time in the map method it will be checked to see if the current word already exists in the HashSet. Although HashSet uses more memory than a List, it was assumed that there would not be an inordinately large amount of stopwords, and thus that no problems with memory exceptions should be seen. As for defining a word, the same logic as the previous Section was used. For the Map class, the key was defined as the word, whereas the value was defined as the name of the file. The name of the file was obtained using this code:

```
String fileName = ((FileSplit)
context.getInputSplit()).getPath().getName();
```

As for the Reducer class, the use of the Java datatype LinkedList was used to store all the values of the filenames. If the filename did not already exist in the List, it was added to the list, and a String consisting of all the filenames was updated as the values were iterated through. Finally, once the loop was completed, the key/value pair was written to the context. The output of the program can be seen in the Github repository.

Section (c)

The number of unique words in the document corpus can be obtained from the Map Reduce counter number Reduce output records in the previous section, which for that program was 74715 (the total number of words for the document corpus can be obtained from the counter Map output records, which was 2211986). To obtain the number of total and unique words per document, counters were defined dynamically (ultimately two per each file name). The first of these two counters was defined in the Mapper class – one would be defined for each filename encountered, and would be incremented for each (non empty) word seen per file. The second was defined in the Reducer class – it would be incremented once for each (unique) word seen per file. After running the program again with these counters implemented, the following output was obtained:

```
pg100.txt
     pg100.txttotal_words=486167
     pg100.txtunique_words=29308
```

```
pg31100.txt
     pg31100.txttotal_words=368409
     pg31100.txtunique_words=16115
pg3200.txt
     pg3200.txttotal_words=1357410
     pg3200.txtunique_words=58690
```

Section (d)

In part d of Assignment 1, the inverted index was modified to also return the number of occurrences of each word in each document. The final output would be returned in a similar format to the Section (b), with the list of documents having a "**#x**" appended to the end of the document, where **x** represented the number of occurrences of the word in the document. To obtain this format, the Mapper from the previous format was modified to output (key, value) pairs of the format (word, "filename#1"). A Combiner was also implemented to take input of this form and output a format of (key, value) = (word, "filename#count(filename)\t…."). Since the Combiner was of this format, the Reducer class could be implemented as the same as the Combiner class. In other words, the only difference in the input of the Combiner class and Reducer class was that each value in the Combiner input corresponded to one filename and counter, whereas the Reducer input might have multiple files/counts in an entry.

To handle this input/output, the Java datatype LinkedList was used. In particular, one LinkedList stored the names of the files, whereas another stored the count for each file, such that the indexes were aligned. Although LinkedList provides a lookup time of O(n) in the worst case scenario, compared to a HashMap, which provides a lookup time of O(1), the idea was to save memory in case of massive document corpus. In the Reducer/Combiner class, the input values were looped through, with hitherto unseen filenames being added to the filename LinkedList, and a value of 1 added to the counter LinkedList (added to the end for both cases to maintain order). If the filename already existed in the LinkedList, the counter at the corresponding position in the counter LinkedList was incremented. In this way, the output seen in the github repository was obtained.

Bonus Section

Part (a) – Relative Frequency w/Stripes Approach

In Part (a), a MapReduce program was written to compute and display the top 100 word relative frequency pairs based on co-line occurrences. Part (a) used the Stripes approach. In this approach, (key, value) pairs are as such: the key corresponds to a word, and the value corresponds to a list of all the other words in the line (note: this list excluded co-occurrences of the same word, but if the same word occurred in the same line, this would result in the same key value combinations for however many times the word occurred in the line) (Source: https://chandramanitiwary.wordpress.com/2012/08/19/map-reduce-design-patterns-pairs-stripes/).

Thus, in the Map class, this corresponded to (key, value) (Text, Text) pairs of the following format. A key would be created for each word, whereas the value would correspond to a String with all the other words in the line (as noted above, this String excluded occurrences of the key word) separated by tab characters. The tab character was chosen because in the Mapper class, it was not possible for a word to contain the tab character, and thus words could be split later in the by the tab character in the Reducer class. The Reducer input was thus of the format (key, value) = (word, [“wordx	\twordy…”, “wordz\t…”, ….]). Since sorted output was required, only one reducer was used. To compute the frequencies, a HashMap of String, Integer was used to store the co-occurences of every word with the original key (i.e. count(A, B)). The key for the HashMap corresponded to all words other than key. To keep track of the top 100 pairs, two global (global in the ReduceStripes class) variables were created: a double array and a string array. The double array would store the top 100 seen frequency pairs seen up to that point in sorted ascending order (the String array would store the pairs corresponding to the frequency seen in the double array, e.g. their indices were aligned). For each pair, if its frequency was greater than the first element of the double array (note: since the double array was initialized at all zeros, the logic doesn’t change for the first 100 iterations), the index was found at which to insert the element, and then all values were shifted to the left.

After all keys were processed, the cleanup(Context context) method was overwritten. All the cleanup method did was loop through the two arrays, writing the String and the frequency to the context. A minimum denominator of 50 was used to avoid very infrequent words, and the output can be seen in the github repository. One other note is that only 100 (even in case of ties) relative frequencies will be written to the output.

Part (b) – Relative Frequency w/Pairs Approach

In Part(b) of the Bonus section, the Pairs approach was used. In this approach, the keys correspond to each word pair (similar to Part (a), a key would never contain two of the same word), whereas the values correspond to the count of the pairs. A separate key value mapping was used to also track the denominator – namely, a key without a pair was used to track the denominator. Therefore, in implementing the Pairs approach, the Mapper class from above was changed to output a Text, and DoubleWriter (Double was chosen due to the fact that decimal frequencies would eventually need to be computed). The Mapper class thus outputted (key, values) of the from (“worda \t wordb”, 1).

The task in the Reducer class thus was to add up both the numerators and denominators of the frequencies. To accomplish this, the sorted input of the keys was exploited. Since the keys were sorted, it was possible to keep track of the denominator for each input, so the sorting based on frequencies could take place during the usual reduce phase. This was because it was guaranteed that a key of word by itself would appear directly before all of its word, wordx (key, value) combinations. In addition, two arrays fulfilling the same function as in Part (a) of this section were created to store the frequencies. Also similar to Part (a), only one reducer was used, and the cleanup method was used to print the frequencies to the context. The results can be seen in the Github repository.

Part (c) – Comparison of the Two Approaches

The runtime of the two programs is shown here on the Hadoop file system for the input consisting of all three documents:

Runtime from Stripes Approach:

Job Overview	
Job Name:	relative_freq_stripes.jar
User Name:	cloudera
Queue:	root.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Thu Feb 16 13:36:33 PST 2017
Started:	Thu Feb 16 13:36:46 PST 2017
Finished:	Thu Feb 16 13:38:20 PST 2017
Elapsed:	1mins, 33sec
Diagnostics:	
Average Map Time	50sec
Average Shuffle Time	21sec
Average Merge Time	3sec
Average Reduce Time	19sec

Runtime from Pairs Approach:

Job Overview	
Job Name:	relative_freq_pairs.jar
User Name:	cloudera
Queue:	root.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Thu Feb 16 13:56:36 PST 2017
Started:	Thu Feb 16 13:56:44 PST 2017
Finished:	Thu Feb 16 13:58:36 PST 2017
Elapsed:	1mins, 51sec
Diagnostics:	
Average Map Time	1mins, 0sec
Average Shuffle Time	45sec
Average Merge Time	4sec
Average Reduce Time	10sec

As seen from these runtimes, the Stripes runtime was faster than the Pairs. This can be attributed to a few different reasons. For one, the amount of (key, value) pairs in the Stripes approach is much fewer than in the Pairs approach. This can be seen in the screenshots below:

Selected Counters from Stripes Approach:

	Name	Map	Reduce	Total
Map-Reduce Framework	Combine input records	18054400	0	18054400
	Combine output records	12153179	0	12153179
	CPU time spent (ms)	40230	14570	54800
	Failed Shuffles	0	0	0
	GC time elapsed (ms)	1348	431	1779
	Input split bytes	381	0	381
	Map input records	507535	0	507535
	Map output bytes	261851828	0	261851828
	Map output materialized bytes	165679682	0	165679682
	Map output records	12718716	0	12718716
	Merged Map outputs	0	3	3
	Physical memory (bytes) snapshot	679415808	315547648	994963456
	Reduce input groups	0	6220423	6220423
	Reduce input records	0	6817495	6817495
	Reduce output records	0	100	100
	Reduce shuffle bytes	0	165679682	165679682
	Shuffled Maps	0	3	3
	Spilled Records	12153179	6817495	18970674
	Total committed heap usage (bytes)	496840704	263376896	760217600
	Virtual memory (bytes) snapshot	4503797760	1507098624	6010896384

<u>Selected Counters from Pairs Approach:</u>

	Name	Map	Reduce	Total
Map-Reduce Framework	Combine input records	18054400	0	18054400
	Combine output records	12153179	0	12153179
	CPU time spent (ms)	40230	14570	54800
	Failed Shuffles	0	0	0
	GC time elapsed (ms)	1348	431	1779
	Input split bytes	381	0	381
	Map input records	507535	0	507535
	Map output bytes	261851828	0	261851828
	Map output materialized bytes	165679682	0	165679682
	Map output records	12718716	0	12718716
	Merged Map outputs	0	3	3
	Physical memory (bytes) snapshot	679415808	315547648	994963456
	Reduce input groups	0	6220423	6220423
	Reduce input records	0	6817495	6817495
	Reduce output records	0	100	100
	Reduce shuffle bytes	0	165679682	165679682
	Shuffled Maps	0	3	3
	Spilled Records	12153179	6817495	18970674
	Total committed heap usage (bytes)	496840704	263376896	760217600
	Virtual memory (bytes) snapshot	4503797760	1507098624	6010896384

Thus, more data is sent across the data in the Pairs approach, and the shuffle and sort phase also takes much longer in the Pairs approach (as demonstrated by the first two screenshots). Of course, also as a result the Map phase takes more time to complete.

<u>Conclusion</u>

In this project, the power of Hadoop's Map Reduce was demonstrated, along with the value of using Combiners when possible (Section a), and the necessity of thinking about the way the Mappers and Reducers are designed (Bonus Section). Although the true power of Hadoop was not realized due to the small size of the datasets, it would be interesting to compare Hadoop versus standard text reading operations on large datasets. Nevertheless, much insight was gained during this project about the inner workings of a basic Hadoop Map Reduce program.