ELEN4020A: Data Intensive Computing
Laboratory Exercise 3

Lynch Mwaniki 1043475
Madimetja Sethosa 1076467
Teboho Matsheke 1157717

April 14, 2019

1 Using MapReduce Framework

MapReduce is a framework which enables parallel and distributed processing on large data sets in a distributed environment. To improve performance, Mapreduce processes data near the places it is stored and brings processing to the data instead fetching data sets from different nodes and amalgamating the results. As the name suggests, Mapreduce consist of two main tasks; map and reduce.

The data flow of MapReduce is shown in Figure 1. Sets of blocks of data are read and processed to produce key-value pairs in a mapper. The key-value pairs are redistributed by worker nodes. The reducer receives key-value pairs from multiple map jobs then bundles those into a smaller set which is the final output.

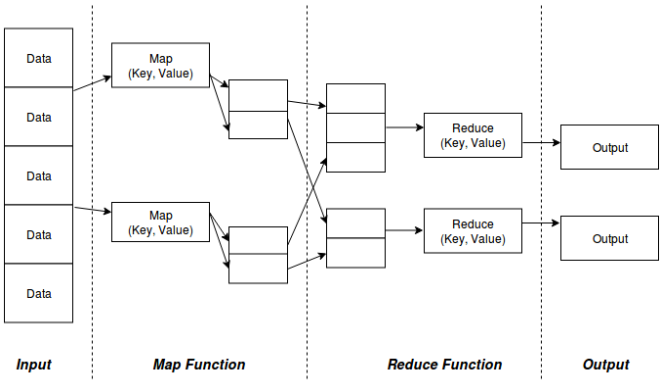

Figure 1: Figure Data flow in Mapreduce

Mapreduce eliminates the challenges associated with the traditional approaches to parallel processing of enormous data sets, that is:

- Critical path: Time taken to finish the job without delaying the next milestone.
- Reliability: A machine fails while working on a data set.
- Equal split: Even distribution of data to the machines so as to not overload or under utilize a machine.
- Single slip failure: If a machine fails to produce an output. Lack of a failure tolerance capability.
- Agregation of results: Mechanism to collect the results generated by each of the machines to produce the final output

This lab showcases the use of two MapReduce frameworks: MRJob and Mrs MapReduce. These are python based implementations and were used to perform the lab exercises. MRJob version 0.6.7 was used and Mrs-MapReduce version 0.9 in the lab exercises.

2 Word Count

This section looks at a simple word count algorithm of a text. This gives the frequencies of currencies of words in a text. The algorithms include a lists stop words which are ignored during the word count. The list of Stop Words that were ignored can be seen in Appendix A, Table 1. Both MR Job's and Mrs MapReduce's algorithm follow the pseudo codes provided in Algorithm 1 and 2.

Algorithm 1: Mapper Algorithm for Word Frequency Count

```
Result: A key and value pair, key equals the word
        and value is one indicating count
Input  : key
Input  : line
Split the words in the line text into a list.
Get Stop words list.
foreach word in line do
    Change word to lowercase.
    if word not in STOP_WORDS and word is
     not a digit then
        return word, 1
```

Algorithm 2: Reducer Algorithm for Word Frequency Count

```
Result: A key and value pair, key equals the word
        and value is the total frequency count
Input  : key
Input  : frequency list
Sum the frequencies.
totalFrquency = sum(frequency)
return word, totalFrquency
```

2.1 MRJob

MRJob implements the word count problem simply by making use of Algorithm 1 and 2. The library reads in the text file and the mapper receives a key which is *None* and a line of text from the text file [1]. The algorithm is run by executing Listing 1 in the command line terminal.

```
python runWordCount.py <filename>
```

Listing 1: Command to execute Word Count using MRJob

The output of the reducer algorithm in the word count script gets printed by the MRJob library to the terminal. The result will simply be the words discovered and their frequency.

2.2 Mrs MapReduce

Mrs MapReduce has a similar implementation to MRJob. The text file is read and the words are fed to the mapper, then reduced to form a list of words with their frequency of occurrence. The script is executed using the command in Listing 2.

```
$ python run_word_count.py <filename>
```

Listing 2: Command to execute Word Count (Mrs)

The output of the program is written to a file in the out put directory given as: *"outPut/source_0_split_0_.mtxt"*. The output directory is specified in the *run_word_count.py* script and it is the same everytime when the program is ran.

3 Top K Query

Top-K query algorithm makes use of the word frequency list produced by the word count algorithm, to give out the top K most frequently occurring words, ignoring stop words. The algorithm does the computations for $K = 10$ and 20. Both MR Job's and Mrs MapReduce's algorithms follow the mapper and reducer pseudo codes in Section 2 Algorithms 1 and 2.

Algorithm 3: Sorting Algorithm for Top-K Query

Result: A word list sorted in descending order of frequency
Input : word
Input : frequency
1 Iterate through the list and sort by frequency from highest to lowest.
2 **return** A sorted list

Algorithm 4: Algorithm to return Top-K Words list

Result: A word list with Top-K distinct words
Input : word
Input : frequency
1 Iterate through the list and print the k elements.
2 **return** A list of k items of distinct words

3.1 MRJob

For this task, the output of the reducer is saved to a list in memory and it gets sorted in descending order of frequency. Algorithm 3 is used to achieve the sorting. The sorted list, gets truncated using Algorithm 4 and the Top K distinct words are printed to the terminal. The script for this task can be executed using the command in Listing 3.

```
python runTopKQuery.py <filename>
```

Listing 3: Command to execute Top-K Query using MRJob

3.2 Mrs MapReduce

For Top K Query, the main script runs the word count script first so that the output can be captured. Before the script is ran output directory to make sure that the output is always in the same file. The script will the produce a new output directory and store the results there. The main script will then access the output file, read its content and feed it to a function to sort the list. The sorting of the list is done by Algorithm 3, which sorts them in descending order base on their frequency of occurrence. The list is then truncated using Algorithm 4 to print the top K distinct words. The script is executed using the command in Listing 4.

```
$ python runTopKQuery.py <filename>
```

Listing 4: Command to execute Top K Query (Mrs)

4 Inverted Indexing

The algorithm produces an inverted index of the text. This involves listing, for each word, the line numbers of the text that the words occur. The algorithms only list out about 50 lines of the distinct words as instructed. Both MR Job's and Mrs MapReduce's algorithm follow the pseudo codes provided in Algorithm 5 and 6.

Algorithm 5: Mapper Algorithm for Inverted Indexing

Result: A key and value pair, key equals the word and value is the line number the word appears in.
Input : key
Input : Value = Line text

```
Receive the key and value.
foreach word in line do
    Change word to lowercase.
    if word not in STOP_WORDS and word is not a digit then
        return word, lineNumber
```

Algorithm 6: Reducer Algorithm for Inverted Indexing

Result: A key and value pair, key equals the word and value is the line numbers the word appears in.
Input : key
Input : Line number list

```
Appends all line numbers for a word into one list
return word, line number list
```

4.1 MRJob

The way MRJob reads in a file is different from that in Mrs-MapReduce. The key given as input into the mapper is *None* as stated in the documentation [1]. For the algorithm to work the authors prepended the line numbers to the input file and separated them from the text using a tab character. Algorithm 7 shows the algorithm followed by MRJob's mapper function.

Algorithm 7: Mapper Algorithm for Inverted Indexing in MRJob

Result: A key and value pair, key equals the word and value is the line number it appears in.
Input : key
Input : Value = Line text

```
Receive the key and value.
In MRJob key is None and value will contain line text with line numbers prepended and separated by a tab.
Split the line text into lineNumber and line at the first tab.
foreach word in line do
    Change word to lowercase.
    if word not in STOP_WORDS and word is not a digit then
        return word, lineNumber
```

For this task, Listing5 indicates how to execute the code in the command line terminal.

```
python runInvertedIndexing.py <filename>
```

Listing 5: Command to execute Inverted Indexing using MRJob

4.2 Mrs MapReduce

The Inverted Indexing operates is a similar as the Mrs MapReduce's Top K Query script. The script starts by deleting the output directory if it exists, then instructs the operating system to run the *invertedIndexing.py* script using the filename and the default output directory (named "outPut"). The *invertedIndexing.py* script is a modified word count which reads the distinct words and the list line on which they occur. The main algorithms for mapping the words with the lines and then reducing them are shown in Algorithm 5 and 6 respectively. The Inverted Indexing script can be executed by using the command in Listing 6.

```
python runInvertedIndexing.py <filename>
```

Listing 6: Command to execute Inverted Indexing (Mrs)

5 Analysis of Results

To analyze the performance of the MapReduce frameworks implemented, the authors ran the python scripts on a desktop computer with the following specs: an Intel Core i7-6700 CPU with 8 cores running at 3.40 GHz and 8 GB DDR3 RAM and on the Wits EIE Hornet01 Linux computer which has 8 cores running at 3.40 GHz and 16 GB RAM. Both computers were running Ubuntu with the only difference that the desktop was using version 16.04 LTS and the Hornet01 was running Ubuntu 18.04.2 LTS.

The text files generated for testing were *File1ForLab3.txt* which contained only 34 lines of text and *File2ForLab3.txt* which contained 1407 lines of text.

5.1 MRJob

The MRJob's jobs were tested in two different configurations for the runners. They were executed with an *inline* runner and a *local* runner. An inline runner runs MRJob in the same process thus parallelism isn't utilized entirely [2], while a local runner spawns multiple sub-processes for each task[3].

The average time taken to execute the jobs were taken down in seconds and the results for the inline runner can be seen in Appendix B Table 2 and those for the local runner can be seen in Appendix B Table 3. The local

runner configuration can be achieved by executing the python scripts in the terminal as shown in Listing 7.

```
python <py script> -r local <filename>
```

Listing 7: Command to execute MRJob python script in Local runner configuration

The scripts were executed three times and the average of the results were taken down.

From the results, it an be seen that the local runner configuration performs better on Hornet01 than it does on the Dekstop PC for both input files. The speed differences might be because Hornet01 is configured to run workloads in parallel and MRJob would try to run the workloads across the Desktop PC's cores. The opposite trend is seen when running the inline runner configuration. The Desktop PC performs better in this configuration. The variance in results may be due to the configuration of the Hornet01 computer.

5.2 Mrs MapReduce

Unlike MrJob, Mrs MapReduce was only tested as a local runner. This allowed for the jobs to be split into sub-processes. All the Mrs MapReduce scripts had a runner script because the Mrs' reduce function has an exit script command which closes the script without return to where it was called. The runner script was used to time the main algorithm's performance. The scripts were ran at least five times and the average time in seconds was recorded. The results can be observe in Appendix B, Table 4.

From the results, it can be seen that scripts took longer to compute on the Desktop PC as compared to when they were ran on Hornet01. This is mainly because Hornet01 is configured to run workloads in parallel.

5.3 MRJob vs Mrs-MapReduce

When processing *File1ForLab3.txt* on Hornet01, MRJob in Local runner configuration and Mrs-MapReduce have similar execution times. This is not the same when processing the larger text in *File2ForLab3.txt*, where it can be seen that MRJob in Local runner configuration is faster than Mrs-MapReduce by as much as approximately 400 ms and a minimum of approximately 200 ms.

When processing *File1ForLab3.txt* on Hornet01, MRJob in inline runner configuration and Mrs-MapReduce have different execution times. Mrs-MapReduce can be seen to perform better in this case. Same can be said when processing *File2ForLab3.txt*. The performance gap widens even further and MRJob is slower by as much as 2500 ms for the Word Count task.

On the Desktop PC, MRJob in both Inline and Local runner configuration performed better than Mrs-MapReduce on both files in most cases. The differences in processing times might be due to the differences in how the libraries were coded.

6 Conclusion

This report detailed the use of MRJob and Mrs-MapReduce, in Python, to solve simple problems using a MapReduce approach. It was observed that MRJob in local runner configuration generally performed better on all tasks than Mrs-MapReduce on both Hornet01 and on the Desktop PC for both large and small texts. Mrs-MapReduce was faster on Hornet01 when MRJob was using Inline runner configuration for both texts. Both frameworks provided an easy way to perform MapReduce tasks and are documented well.

References

[1] (2019) mrjob documentation. [Online]. Available: https://buildmedia.readthedocs.org/media/pdf/mrjob/stable/mrjob.pdfr

[2] (2019) mrjob.local - simulate hadoop locally with subprocesses. [Online]. Available: https://pythonhosted.org/mrjob/runners-local.html#mrjob.local.LocalMRJobRunner

[3] (2019) mrjob.inline - debugger-friendly local testing. [Online]. Available: https://pythonhosted.org/mrjob/runners-inline.html#mrjob.inline.InlineMRJobRunner

A Stop Words

Table 1: Table showing a list of stop words used.

Stop Words
"a","about","above","after","again",
"against", "all", "am", "an", "and", "any", "are",
"as", "at", "be", "because", "been", "before", "being",
"below", "between", "both", "but", "by", "could", "did",
"do", "does", "doing", "down", "during", "each", "few",
"for", "from", "further", "had", "has", "have", "having",
"he", "he'd", "he'll", "he's", "her", "here", "here's",
"hers", "herself", "him", "himself", "his", "how", "how's",
"i", "i'd", "i'll", "i'm", "i've", "if", "in", "into", "is",
"it", "it's", "its", "itself", "let's", "me", "more", "most",
"my", "myself", "nor", "of", "on", "once", "only", "or",
"other", "ought", "our", "ours", "ourselves", "out", "over",
"own", "same", "she", "she'd", "she'll", "she's", "should",
"so", "some", "such", "than", "that", "that's", "the",
"their", "theirs", "them", "themselves", "then", "there",
"there's", "these", "they", "they'd", "they'll", "they're",
"they've", "this", "those", "through", "to", "too", "under",
"until", "up", "very", "was", "we", "we'd", "we'll", "we're",
"we've", "were", "what", "what's", "when", "when's", "where",
"where's", "which", "while", "who", "who's", "whom", "why",
"why's", "with", "would", "you", "you'd", "you'll", "you're",
"you've", "your", "yours", "yourself", "yourselves"

B Results

B.1 MRJob

Table 2: Table showing time taken in seconds to execute MRJob with an inline runner

MRJob's Runner in Inline Mode						
	Word Count		**Top K Query**		**Inverted Indexing**	
	PC	**Hornet01**	**PC**	**Hornet01**	**PC**	**Hornet01**
File1ForLab3.txt	0.1614866860	0.234113037	0.206561392	0.237100641	0.155400192	0.167444410
File2ForLab3.txt	0.3390905024	0.478173682	0.348368527	0.475915451	0.252612733	0.306580266

Table 3: Table showing time taken in seconds to execute MRJob with a local runner

MRJob's Runner in Local Mode						
	Word Count		**Top K Query**		**Inverted Indexing**	
	PC	**Hornet01**	**PC**	**Hornet01**	**PC**	**Hornet01**
File1ForLab3.txt	0.349227458	0.145003454	0.292195913	0.151581794	0.292010370	0.129592853
File2ForLab3.txt	0.321756097	0.185552867	0.332126871	0.169163441	0.302988557	0.164811396

B.2 Mrs MapReduce

Table 4: Table showing time taken in seconds to execute Mrs Map Reduce.

Mrs Map Reduce						
	Word Count		**Top K Query**		**Inverted Indexing**	
	PC	**Hornet01**	**PC**	**Hornet01**	**PC**	**Hornet01**
File1ForLab3.txt	0.334859609	0.157367229	0.363074302	0.159703254	0.415356397	0.157572984
File2ForLab3.txt	0.397320747	0.220757007	0.436202526	0.220881700	0.436696052	0.220025539