CS 6220 Big Data Homework 1 Report – Option 1.1

By: Nipun Chhajed

Problem 1 – Run the word count map-reduce program, and report the runtime for two different sizes of datasets

Introduction:

- For this problem, I ended up running two different word count map-reduce programs. One of the programs was in Python, which I ran through the Hadoop Streaming function using the Hadoop Streaming Jar (*hadoop-streaming-3.3.6.jar*). The other one was the built-in Hadoop Word Count Example Program (*hadoop-mapreduce-examples-3.3.6-sources.jar*).
- The dataset was sourced from the Project Gutenberg website, specifically the Moby Dick book (https://www.gutenberg.org/ebooks/2701) and the Middle March book (https://www.gutenberg.org/ebooks/145). I made a custom dataset "small_dataset.txt" by replicating the text files of these two Ebooks - 500 times for each - resulting in a text file with a size of 1.46 GB and I made a second custom dataset "smaller_dataset.txt" by replicating the text files of these two Ebook – 50 times for each – resulting in a text file with a size of 149 MB. Roughly a 10x size difference between the two datasets. The reason I kept my largest dataset at a smaller size of 1.46 GB is because I only allocated roughly 2 GB of memory for my Hadoop Docker Container.
- The code for the python files was sourced from the following link: https://dev.to/boyu1997/run-python-mapreduce-on-local-docker-hadoop-cluster-1g46.
- The instructions for running this via my Docker Container is given in README.md under the "Running Problem 1" section.

Results:

To see screenshots, terminal results and word count outputs, I have created an "outputs" folder, under which I have separated the outputs and screenshots based on the "smaller_dataset/" and "small_dataset/" designation, as they are their own folders.

Table Data:

File Size	Hadoop Built-in Map Reduce Word Count Runtime	Python Streaming Code Word Count Runtime
Smaller Dataset – 149 MB	85 seconds	175 seconds
Small Dataset – 1.46 GB	273 seconds	1162 seconds

Analysis:

- From the table above, one can immediately see that the built-in word count jar that comes with Hadoop had a significantly faster runtime than the python streaming code, for both of the datasets. For the smaller dataset (149 MB), the Hadoop jar was 2x faster, and for the small dataset (1.46 GB), the Hadoop jar was 4.3x faster.
- This makes sense because Hadoop natively supports java programs the best, and since you can create a jar using the java program, Hadoop can directly run that jar and not have to do any sort of extra processing. For python on the other hand, Hadoop must use the streaming jar to start a separate virtual machine to process the python code, and then the data being streamed must also be copied between the jvm and the python vm, which takes significantly longer. In addition, python in general is just a slower compiled language compared to java because java is statically typed, leading to less time spent overall during compiling.
- The interesting thing I noticed is that for python, a 10x increase in data size led to a roughly 6x increase in runtime, whereas for java the increase was only around 3x, so python was almost double the increase in runtime compared to java.
- Overall, though, the runtime does seem to be linear for both java and python, and it's a direct correlation: as dataset size increases, runtime increases as well.

From this analysis, we can conclude that using a java-based jar for MapReduce programs is going to be performing significantly faster than streaming with a different language such as python. I used this finding to fuel my approach for Problem 2.

Problem 2 – Take Google's LibriSpeech Data and find the top 100 words

Introduction:

- For this problem, I ended up running the built-in Hadoop Word Count Example Program (*hadoop-mapreduce-examples-3.3.6-sources.jar*) to generate the word counts for all the files, and then I extracted the output and ran my own python file (*top100words.py*) on it to extract the top 100 words.
- The dataset for this problem was sourced from the following link: https://www.kaggle.com/datasets/google/userlibri, specifically, the "lm_data" folder within that Kaggle link, as that had 95 text files of data I could use. Using this dataset, I created three different sizes of datasets (the folders can be downloaded from my github repository for this homework with the link – (https://github.com/nchhaj189/BigData-HW1/tree/main/data):
    - One with the first 20 of the 95 text files (16.9 MB)
    - One with the first 40 of the 95 text files (33.7 MB)
    - One with all 95 text files (60.0 MB)
- The instructions for running this via my Docker Container is given in README.md under the "Running Problem 2" section.

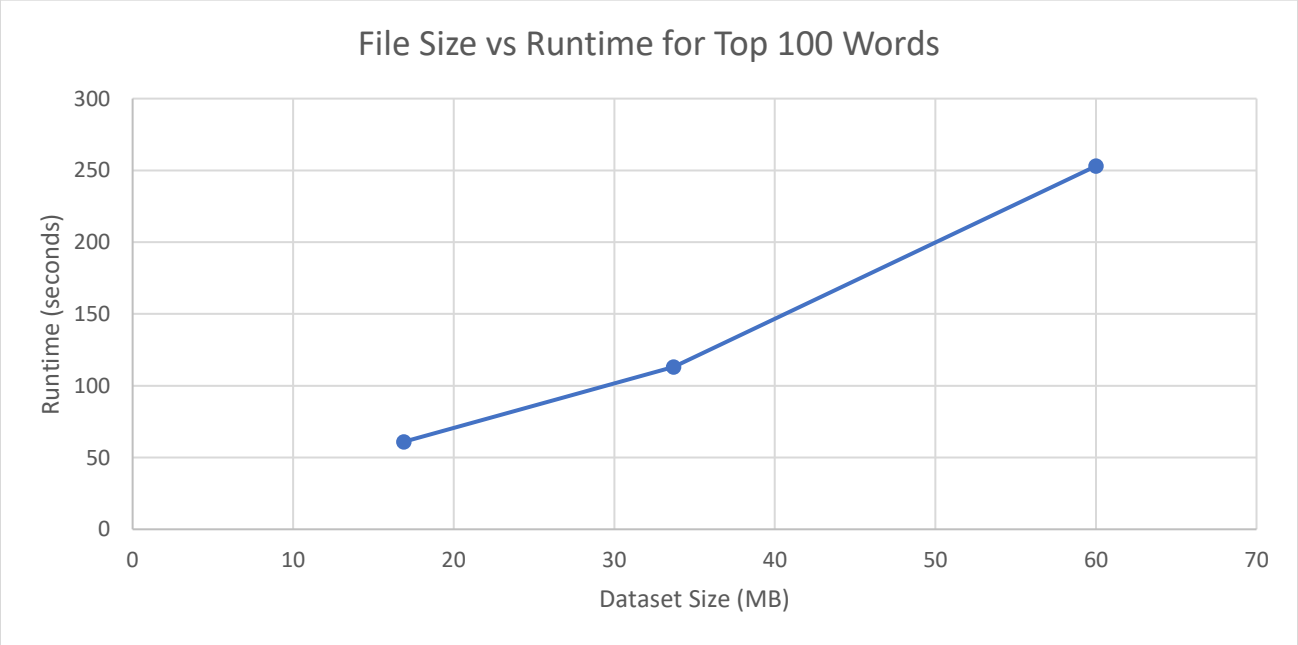
Results:

To see screenshots, terminal results and word count and top 100 data outputs, I have created an "outputs" folder, under which I have separated the outputs and screenshots based on the dataset sizes with the following folder names: "20file_dataset/", "40file_dataset/ and "95file_dataset/. In addition, the word count outputs and the top100outputs for each of the 3 datasets is in the top-level right under the "outputs" folder, as that is where I also stored my python script (*top100words.py*).

Table Data:

Dataset Size (Input Data)	Hadoop Built-in Map Reduce Word Count Runtime	File Size (Word Count outputted files)	Python Script Runtime to output the top 100 words
20 Files – 16.9 MB	61 seconds	865 KB	0.17 seconds
40 Files – 33.7 MB	113 seconds	1.31 MB	0.28 seconds
95 Files – 60.0 MB	253 seconds	1.98 MB	0.43 seconds

Graph (made in Excel):



Analysis

- As expected, as the dataset size increases, the runtime to find the top 100 words, increases almost linearly.
- Since the runtime for the python script was basically negligible (less than 1 second for each of the 3 datasets), I did not include them in the overall runtime.
- What was interesting to notice was that in the terminal output for each of the 20 file, 40 file, and 95 file datasets, the MapReduce program opened the same amount of mappers as the number

of files. For example, for the 20 file it opened 20 mappers, for the 40 file it opened 40 mappers and for the 95 file it opened 95 mappers.

- Example screenshot below (can find terminal outputs in the .md files in the folders under the “output” folder)
-

```
2023-09-09 09:20:58 INFO  FileInputFormat:300 - Total input files to process : 95
2023-09-09 09:20:58 INFO  JobSubmitter:202 - number of splits:95
2023-09-09 09:20:58 INFO  JobSubmitter:298 - Submitting tokens for job: job_1694244890195_0009
```

- The python script I wrote to compute the top 100 words was very simple, it was just looping through the output text file, loading all the word and word count pairs in to a dictionary, sorting that dictionary, and printing the first 100 values of that dictionary. This program overall has a runtime of O(n log n) due to the sorting, however, since the bulk of the computation is happening in the MapReduce itself for getting the word counts, the runtime for that is O(n) roughly and that overtakes because the amount of data being processed.