Big Data Top K Words Report 1

1. Introduction

In recent years, data has become very abundant due to the rapid increase of internet. It's consequently getting increasingly challenging to read, store and process large datasets; the input size is one of the key factors in determining how well or efficiently a program works and therefore, poses a problem and reduces efficiency the larger it gets. Being able to read, store and process huge amounts of data is the main problem of Big Data. Traditional database and data processing systems have been in place; however, the datasets are now so large that it's becoming difficult to manage this data efficiently.

Apart from input size, there are various factors that affect how a program executes: the data structures used, the memory available and the algorithm used. This project focuses on analysing how these factors affect the performance for three datasets of different sizes i.e. three input sizes. The final outputs are compared, and the performance analyzed by measuring the execution time taken by each algorithm for each input size. The time for each case and dataset is graphically represented and the results analyzed along with plotting the speedup for each case.

In this project, the main objective was to find the top K words in an input file where K is some integer pertaining to the frequency of each word i.e. how many times each word occurs in the text file and then print the top K most frequent words. Three text files were used as input, each of a different size: 400MB, 8GB and 32GB. Section 1 introduces the project, section 2 is details of the dataset and computer used, section 3 explains the algorithms and data structures used. The graphical results and analysis are showcased in section 4 and section 5 concludes the paper.

2. Experimental Setup

There were three datasets that were given of three different sizes: 400MB, 8GB and 32GB text files. Each text file had English text along with some symbols or numbers. For the experiments, only English text was considered, and rest of the symbols were ignored. The experiments I did were done on my personal laptop, specifications of which are shown in figure 1. This laptop was used for all experiments and the available memory remained the same throughout.

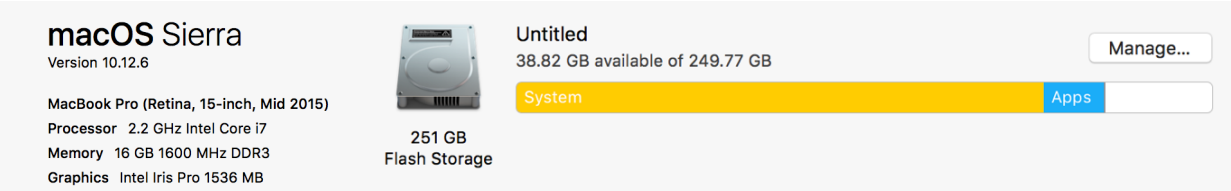

Figure 1: Laptop Specifications

3. Method

For this project, I wrote the program to find top K words in Python. The data structure I used for the experiments was a dictionary. A dictionary, also known as an associative array, consists of key-value pairs where each pair maps the key to its associated value. This works well for this project as the words can be mapped to its frequency. Dictionaries are also easy to use extremely fast look up time making them useful for this project.

3.1 Cases 1 and 2: Read entire file into memory

To begin the experiment, I tried a basic method where I read in the entire text file into memory then for each line, read and processed the words and stored their frequency into a dictionary. Then this dictionary was sorted using a sort method. To find the top K words, I used a for loop. I also tried a similar approach where I again read in the entire file into memory, stored the words and frequency into a dictionary and then used the python Counter function. This counter tool is in place to support rapid tallies which would work well for this project.

3.2 Cases 3 and 4: Read file line by line

The previous method worked however it was very slow and put a lot of pressure on the computer memory as the entire text file was being read into memory. Therefore, the next step was to read the text file line by line. Reading line by line meant only one line would be in memory at any point in time which dramatically eases the strain on memory. Reading the files line by line, I again stored the words and frequency into a dictionary and then used the sorting approach as well as the counter approach.

3.3 Case 5: Read file in chunks and process in parallel

Reading the text files line by line did slightly improve time and also the strain on memory, however, not significantly enough. An important aspect in Big Data that is used is to break files into smaller chunks and process those chunks in parallel. I tried to use this method for the final experiment on the three datasets. Each file was broken into equal sized chunks; each chunk was read and processed for words and frequencies in parallel using dictionary and Python counter as before. I used the counter for this as previous four experiments showed counter worked faster than sorting the dictionary. The final result was stored in a dictionary. This method significantly improved the time and speedup was an average of 3 times over previous cases.

4. Results and Analysis

In this section, I evaluate the performance of the cases. The performance measure that I used was execution time for each case measured in seconds. Figures 2, 3 and 4 showcase the execution times for each case for all three file sizes respectively. As can be seen in the results, case 1 and 2 were the slowest in terms of execution time which proves reading entire file into memory is not efficient. Cases 1 and 2 also showcase the advantage of using Counter over sorting the dictionary in all three charts. Cases 3 and 4 did improve the execution time when reading line by line. Case 5 for all three files significantly reduced the execution time proving that reading in chunks and processing in parallel works well for large data files.

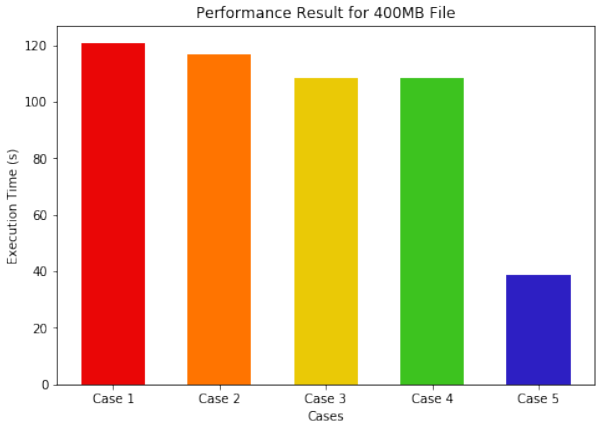

Figure 2: Performance for 400 MB File

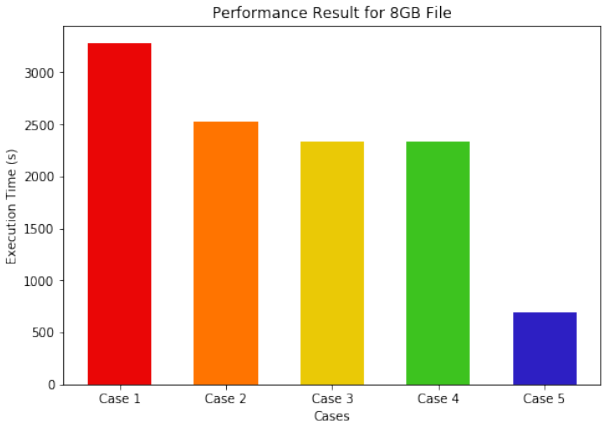

Figure 3: Performance for 8 GB File

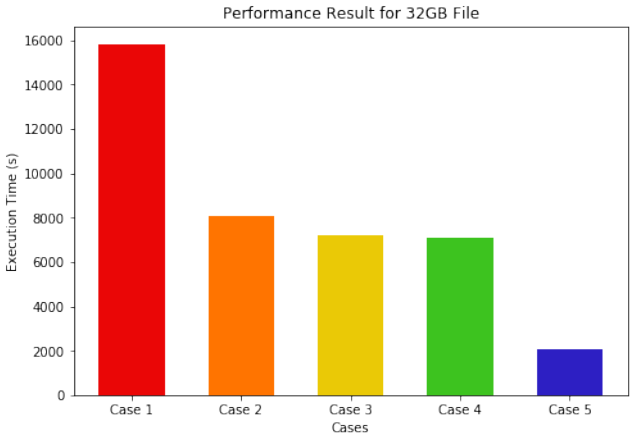

Figure 4: Performance for 32 GB File

Case 1: Read the entire text file into memory, store the words and their frequency into a dictionary then sort the dictionary. Using this sorted dictionary, print out the top K words.

Case 2: Read Entire File into memory, store the words and their frequency into a dictionary then use the Python Counter function to sort and count the top K words.

Case 3: Read the file line by line into memory so that only a single line is stored in memory at any given time, store the words and their frequency into a dictionary then sort the dictionary. Using this sorted dictionary, print out the top K words.

Case 4: Read the file line by line into memory so that only a single line is stored in memory at any given time, store the words and their frequency into a dictionary then use the Python Counter function to sort and count the top K words.

Case 5: Break each file into chunks, then read and process each chunk in parallel.

Case 5 divides each file into equal sized chunks however the size of the chunks could be varied. To be able to discern which chunk size works best, I tried three experiments for three different chunk sizes: 50MB, 100MB and 200MB. Figures 5, 6 and 7 showcase the results for these experiments and as can be seen for all three file sizes, 100 MB worked best (least execution time).

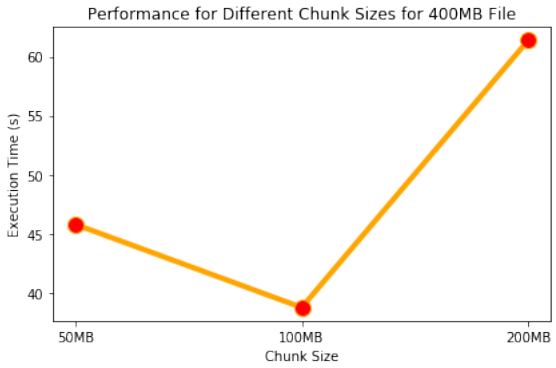

Figure 5: Performance for varied chunk sizes: 400 MB File

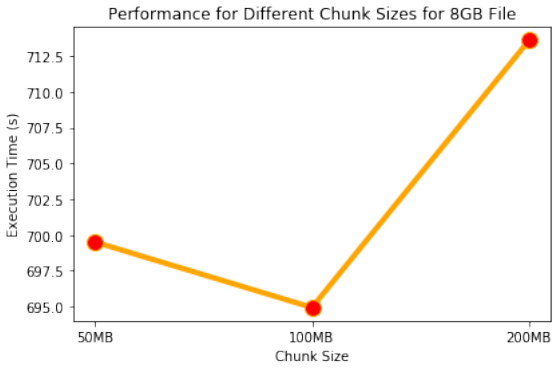

Figure 6: Performance for varied chunk sizes: 8 GB File

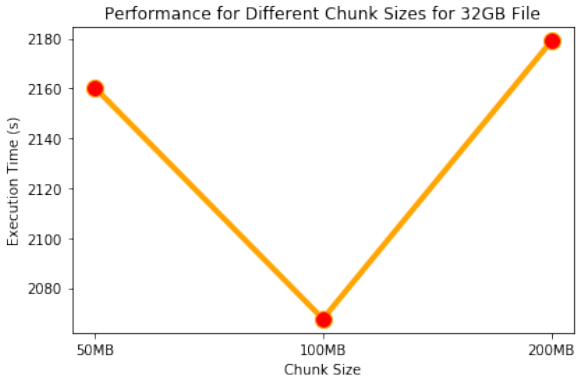

Figure 7: Performance for varied chunk sizes: 32 GB File

To be able to better showcase the improvement of each case, I calculated the speedup ratio and plotted this as shown in figures 7, 8 and 9. The speedup ratio was ratio of old and new execution time and was calculated as in equation (1) where $S(n\ to\ n+1)$ is the speedup from case n to $n+1$ and $E_T(n)$ is the execution time for a certain case n in seconds.

$$S(n\ to\ n+1) = \frac{E_T(n)}{E_T(n+1)} \quad (1)$$

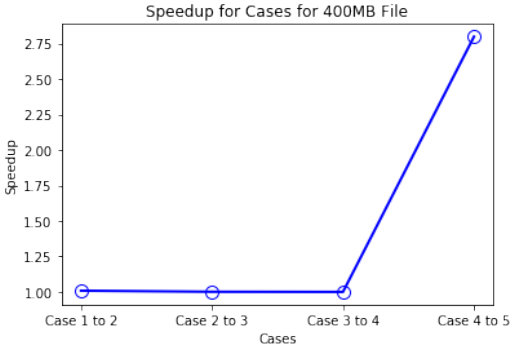

Figure 8: Speedup for the cases for 400MB File

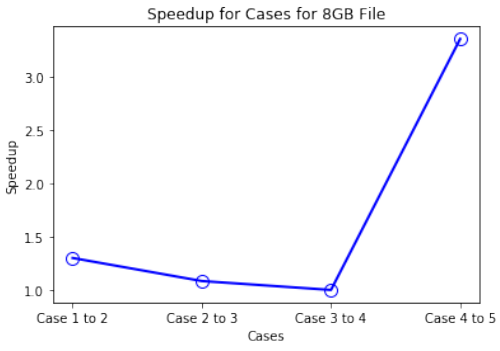

Figure 9: Speedup for the cases for 8 GB File

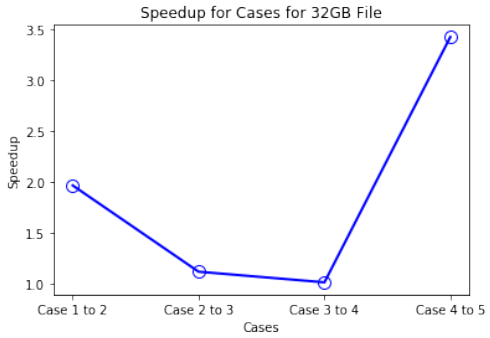

Figure 10: Speedup for the cases for 32 GB File

The line plots for speedup showcase the advantage of case 5 which has provided an average speed up of 3 times over the previous case.

The final outputs of the experiment i.e. the top K words are shown in figures 11, 12 and 13 where K = 6. These are represented as pie chart where the percentage represents the frequency of the word.

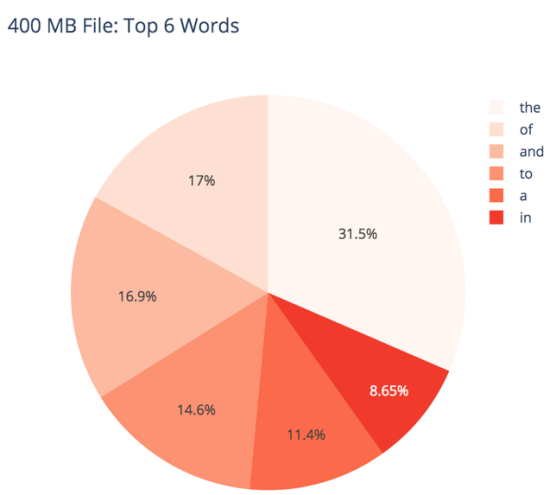

Figure 11: Top K Words (K = 6) for 400 MB File

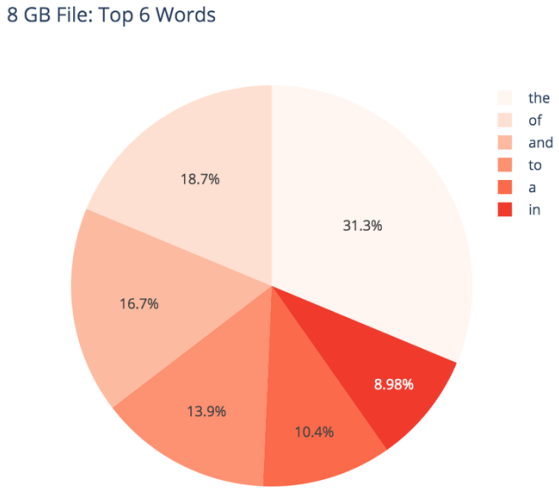

Figure 12: Top K Words (K = 6) for 8 GB File

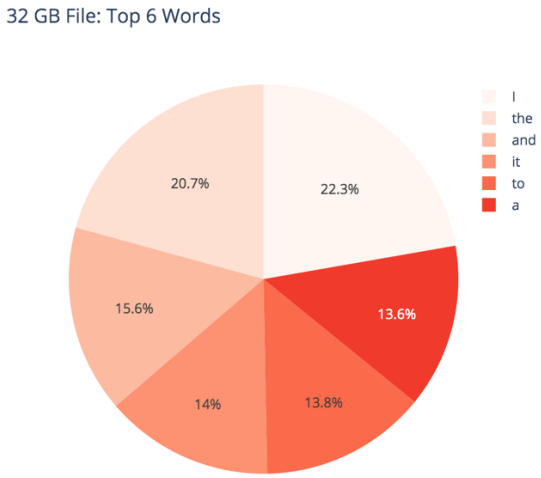

Figure 13: Top K Words (K = 6) for 32 GB File

5. Conclusion

The crux of Big Data is being able to efficiently read and process large files; this project's main objective was to analyze the various factors affecting the performance of an application and understand their impact on efficiency. The main task was to implement an efficient code to find the top K most frequent words in a text file with least possible execution time. I tried 5 different cases and analyzed the performance of the algorithms I used using execution time as a performance metric and also showcased the speedup of the various cases.

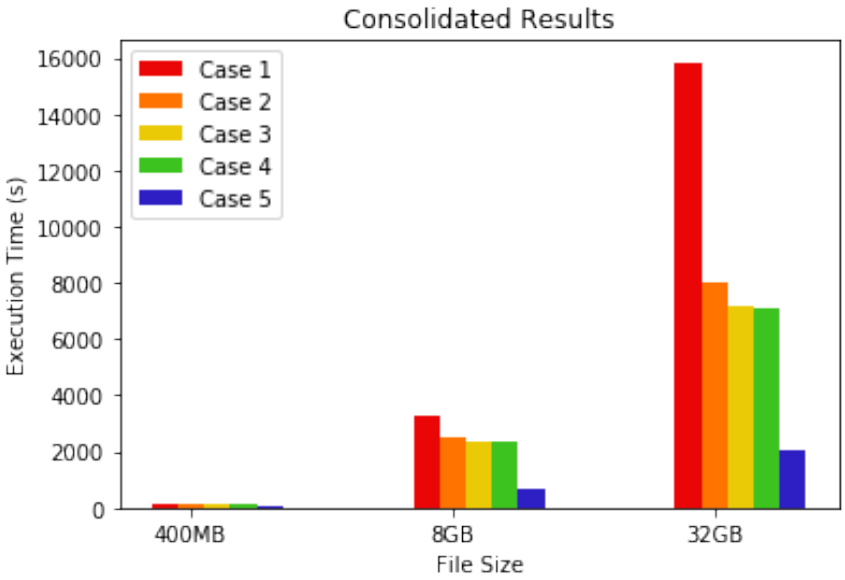

Figure 14: Consolidated Resutls for all File Sizes

Figure 14 showcases the consolidated results for all the cases for all the datasets. These results have shown that reading entire files into memory, as in cases 1 and 2, is slow in execution and also a strain on memory. Case 2's improvement over 1 shows the advantage of using Counter over sorting the dictionary. Reading each file line by line, as in cases 3 and 4, doesn't use up entire memory, however it's still not a vast improvement from cases 1 and 2.

The best performance is of case 5 for all three file sizes where each file is broken into chunks and, reading and processing is done on the chunks in parallel. This supports the reasoning behind Big Data tools storing large files into separate nodes and making use of parallel processing for efficient performance. Big Data is the field of being able to deal with large input files that are too complex for traditional data-processing software and processing this data efficiently; this project and the results help showcase one such way to deal with large files efficiently.

References

1. Pusukuri, Kishore. COEN 242 Class Slides. Santa Clara University, 2020
2. Mehta, Anukrati. "What is a Python Dictionary and How does it work?" https://www.digitalvidya.com/blog/python-dictionary/, 2019
3. Matplotlib tutorial: https://matplotlib.org/
4. Hennessy, John L.; David A., Patterson (2012). *Computer Architecture: A Quantitive Approach*. Waltham, MA: Morgan Kaufmann. pp. 46–47